A Method for Consistent Non-Local Configuration of Component Interfaces

Pavel Zaichenkov

January 2017

A thesis submitted to the
University of Hertfordshire
in partial fulfilment of the requirements of
the degree of

Doctor of Philosophy

in Computer Science

Abstract

Service-oriented computing is a popular technology that facilitates the development of large-scale distributed systems. However, the modular composition and flexible coordination of such applications still remains challenging for the following reasons: 1) the services are provided as loosely coupled black boxes that only expose their interfaces to the environment; 2) interacting services are not usually known in advance: web services are dynamically chosen to fulfil certain roles and are often replaced by services with a similar functionality; 3) the nature of the service-based application is decentralised. Loose coupling of web services is often lost when it comes to the construction of an application from services. The reason is that the object-oriented paradigm, which is widely used in the implementation of web services, does not provide a mechanism for service interface self-tuning. As a result, it negatively impacts upon the interoperability of web services.

In this dissertation we present a formal method for automatic service configuration in the presence of subtyping, polymorphism, and flow inheritance. This is a challenging problem. On the one hand, the interface description language must be flexible enough to maintain service compatibility in various contexts without any modification to the service itself. On the other hand, the composition of interfaces in a distributed environment must be provably consistent.

Our method is based on constraint satisfaction and Boolean satisfiability. First, we define a language for specifying service interfaces in a generic form, which is compatible with a variety of contexts. The language provides support for parametric polymorphism, Boolean variables, which are used to control dependencies between any elements of interface collections, and flow inheritance using extensible records and variants. We implemented the method as a constraint satisfaction solver.

In addition to this, we present a protocol for interface configuration. It specifies a sequence of steps that leads to the generation of context-specific service libraries from generic services. Furthermore, we developed a toolchain that performs a complete interface configuration for services written in C++. We integrated support for flexible interface objects (i.e. objects that can be transferred in the application along with their

structural description). Although the protocol relies solely on interfaces and does not take behaviour concerns into account, it is capable of finding discrepancies between input and output interfaces for simple stateful services, which only perform message synchronisation. Two running examples (a three buyers use-case and an image processing application) are used along the way to illustrate our approach.

Our results seem to be useful for service providers that run their services in the cloud. The reason is twofold. Firstly, interfaces and the code behind them can be generic as long as they are sufficiently configurable. No communication between service designers is necessary in order to ensure consistency in the design. Instead, the interface correspondence in the application is ensured by the constraint satisfaction algorithm, which we have already designed. Secondly, the configuration and compilation of every service are separated from the rest of the application. This prevents source code leaks in proprietary software which is running in the cloud.

To my wife, Ziliya.
Your love knows no distance.

Acknowledgments

First and foremost, I would like to thank my first supervisor, *Olga Tveretina*, for spending numerous hours discussing my work. It has been an honour to be her first PhD student. She has taught me a great deal about formal logic and has greatly influenced my mathematical thinking. Her continuous advice both on a professional and personal level has helped me immensely.

Next to the excellent mentoring by Olga, I have been guided by *Alex Shafarenko*, who is a superb research advisor. His ideas laid the foundation for my work. His patience, attention to details, and insightful comments have had a positive influence on my work. I would especially like to thank him for educating me in technical writing.

Furthermore, I would like to thank *Raimund Kirner*, who provided guidance when it was essentially needed. Moreover, he taught me the useful skill of critical thinking and argument.

This dissertation would also have been impossible without my special friend and colleague, *Artem Shinkarov*, who inspired me to start a fascinating journey in Computer Science and gave me the confidence to continue on the doctoral programme. It would be remiss if I did not thank *Sven-Bodo Scholz*, who has provided excellent feedback on an early stage of this work.

I also owe a debt of gratitude to my friends and colleagues. They have been a source of friendship, collaborative work, and enjoyable times. Special thanks must also go to my family members who have been providing me with their support throughout the years, whether financially, practically, or morally.

Contents

Notations and Definitions

b, g	a Boolean expression called a guard, p. 32
$\mathcal{B}$	the set of all Boolean expressions, p. 32
f	a Boolean variable, p. 55
t, s	an MDL term, p. 33
v	an MDL term variable, p. 32
$v^{\downarrow}$	a down-coerced MDL term variable, p. 32
$v^{\uparrow}$	an up-coerced MDL term variable, p. 32
$\mathsf{l_i}$	the label for a record or choice element, p. 33
$\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$	the label for a record or choice element, p. 34
$x, y, z, w \ldots$	a guard for a record or choice element, p. 33
$\mathsf{V}^b(g)$	the set of Boolean variables that occur in the guard g, p. 56
$\mathsf{V}^{\uparrow}(t)$	the set of up-coerced term variables that occur in the term t, p. 56
$\mathsf{V}^{\downarrow}(t)$	the set of down-coerced term variables that occur in the term t, p. 56
$\mathsf{V}^b(t)$	the set of Boolean variables that occur in the term t, p. 34
d, e	a number of elements in a tuple, a record or a choice, p. 34
nil	the empty record, p. 35
none	the empty choice, p. 35
$\sqsubseteq$	the seniority relation, p. 35
$\mathcal{T}$	the set of all ground terms, p. 35
$\mathcal{T}^{\uparrow}$	the set of all up-coerced ground terms, p. 35
$\mathcal{T}^{\downarrow}$	the set of all down-coerced ground terms, p. 35
$\mathcal{T}_m^{\uparrow}$	the set of all vectors of up-coerced ground terms, which have m elements, p. 35
$\mathcal{T}_n^{\downarrow}$	the set of all vectors of down-coerced ground terms, which have n elements, p. 35

$(\mathcal{T}, \sqsubseteq)$	the pair of meet and join semilattices for down-coerced and up-coerced terms, p. 36
$(\mathcal{T}_m^{\downarrow}, \sqsubseteq)$	the meet semilattice for vectors of down-coerced terms, which have m elements, p. 36
$(\mathcal{T}_n^{\uparrow}, \sqsubseteq)$	the join semilattice for vectors of up-coerced terms, which have n elements, p. 36
$=$	the equality relation on terms, p. 36
l_p	the name for the service port p, p. 45
(l_p, t_p)	the pair, which consists of the name and the term, and represents the interface of the service port p, p. 45
s	a service, p. 45
I_s	the number of input ports in the service s, p. 45
O_s	the number of output ports in the service s, p. 45
N	a service network, p. 46
$\mathcal{I}_N$	the set of input ports of the network N, p. 46
$\mathcal{O}_N$	the set of output ports of the network N, p. 46
$\mathcal{C}$	a set of seniority constraints, p. 46
$(\mathcal{I}_N, \mathcal{O}_N, \mathcal{C})$	the type of the network N, p. 46
$V(\mathcal{C})$	the set of variables in the network $\mathcal{C}$, p. 50
$\models$	the logical entailment, p. 50
$\mathcal{V}$	a set of graph vertices, p. 51
$g[\vec{f}/\vec{b}]$	substitution of Boolean variables $\vec{f}$ with Boolean values $\vec{b}$ in a Boolean expression g, p. 55
$t[\vec{v}/\vec{t'}]$	substitution of term variables $\vec{v}$ with the values $\vec{t'}$ in the term t, p. 55
$\mathsf{V}^b(\mathcal{C})$	the set of Boolean variables in the set of constraints $\mathcal{C}$, p. 56
$\mathsf{V}^{\downarrow}(\mathcal{C})$	the set of down-coerced variables in the set of constraints $\mathcal{C}$, p. 56
$\mathsf{V}^{\uparrow}(\mathcal{C})$	the set of up-coerced variables in the set of constraints $\mathcal{C}$, p. 56
$(\widetilde{\mathcal{T}}_m^{\downarrow}, \sqsubseteq)$	the lattice of down-coerced terms, p. 63
$(\widetilde{\mathcal{T}}_n^{\uparrow}, \sqsubseteq)$	the lattice of up-coerced terms, p. 63
$(\bot, \overline{\top})$	the pair, which consists of the meet and the join lattice elements, that represents Unsat, p. 64
$\overline{\mathsf{IF}}(t_1 \sqsubseteq t_2, \vec{a}^{\downarrow}, \vec{a}^{\uparrow})$	the iterated function for a serniorityconstraint $t_1 \sqsubseteq t_2$ given approximations $\vec{a}^{\downarrow}$ and $\vec{a}^{\uparrow}$, p. 64

$\overline{\mathsf{IF}}_{\mathcal{C}}(\vec{a}^{\downarrow}, \vec{a}^{\uparrow})$	the iterated function for the set of seniority constraints $\mathcal{C}$ given approximations $\vec{a}^{\downarrow}$ and $\vec{a}^{\uparrow}$, p. 71
$\mathsf{GIF}(t_1 \sqsubseteq t_2, \mathsf{B}, b, \vec{a}^{\downarrow}, \vec{a}^{\uparrow})$	the generalised interated function for a seniority constraint $t_1 \sqsubseteq t_2$ and given a Boolean constraint b, and approximations $\vec{a}^{\downarrow}$ and $\vec{a}^{\uparrow}$, p. 84
$\vec{a}$	a vector of terms representing an approximation for variable values in the CSP-WS, p. 64
$\vec{a}^{\downarrow}$	a vector of down-coerced terms representing an approximation for down-coerced variable values in the CSP-WS, p. 64
$\vec{a}^{\uparrow}$	a vector of up-coerced terms representing an approximation for up-coerced variable values in the CSP-WS, p. 64
$\mathsf{SAT}(S)$	the set of Boolean vectors satisfying a set of Boolean constraints S, p. 80
$\mathsf{WFC}(t)$	the set of well-formedness constraints for the term t, p. 80
$\mathsf{SC}(t_1 \sqsubseteq t_2)$	the set of seniority constraints for $t_1 \sqsubseteq t_2$, p. 82
$\mathbb{P}(\mathcal{B})$	the powerset of Boolean expressions, p. 83

Chapter 1

Introduction

Web services provide an efficient and productive way of building large-scale distributed applications, which spread through businesses all over the internet. The number of APIs that are available for anyone via the web has been increasing rapidly for the last decade. 'ProgrammableWeb' is an online repository of public web services. In 2012, it hit the milestone of 8,000 publicly available APIs [Pro13], which increased to 16,000 APIs at the end of 2016. [1] This has been achieved thanks to the flexibility that the service-oriented architecture [PL03] provides. Services can be implemented both as proprietary and as open-source software components; the source code for the services is typically hidden from API consumers, separating concerns about independent application modules. Furthermore, the exposed API interface is fully available to third parties to use and extend as they need. Consumers can combine APIs from various providers to construct and offer completely new products which, in their turn, expose new APIs with very little development or financial cost. Providers and consumers benefit from exposing APIs while a high degree of separation and independence from each other is maintained [JRSS14].

However, creating a flexible and easy-to-use API is difficult. The API that is available to third parties is a source of potential communication errors. A service can often exhibit behaviour that is not expected by other parties, because those parties rely only on the informal functional specification being offered by the service provider. Various research approaches have addressed this issue:

- Carbone, Honda, and Yoshida [CHY07, HYC08] present work on *multiparty session types*. Specifically, they answer the question: **given a global behaviour protocol, how does it project into local services so that those services interact precisely according to the original global protocol?** This approach requires a global description of service interactions, which is provided by a description language such as WS-CDL [G+02]. The research establishes a mapping from global behaviour

[1] http://www.programmableweb.com/category/all/apis.

protocol to local behaviour for each participant. This is useful for programmers who develop services that must conform to a global protocol. Using this methodology, they can check whether the code is functionally correct and does not lead to communication errors.

- Castagna, Gesbert, and Padovani [CGP09] provide a foundation for behaviour subtyping in web services. This facilitates software reuse. Specifically, the research provides **a methodology to use web services exposing 'larger' contracts in place where 'smaller' contracts are required**. Contracts that are specified in contract description languages, such as the Web Service Conversation Language (WSCL) [BBB+02] or the Web Services Business Process Execution Language (WS-BPEL) [JEA+07], are too concrete to be used as contracts. In these languages, compatible contracts must be syntactically equal. Castagna et al. relax this requirement. They formalise what exactly 'larger' and 'smaller' means by introducing a *subcontract preorder* relation [LP07]. They develop a subcontracting formalism that 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 frameworks and language-independent mechanism for assisting service developers to check service behaviour compatibility [ABP12].
- Honda, Yoshida, and Carbone [HYC08] developed a framework that is based on session types. The framework **analyses service contracts to guarantee an absence of potential deadlocks and races in an application**. These are the most common bugs in programming that involves communication [LPSZ08]. In large systems these bugs are hard to detect statically without executing the application. The bug detection problem is also addressed by another formalism called *interface automata* [LNW07].

WS-CDL [G+02], session types [HYC08] and interface automata [HM05] attempt to solve the behavioural consistency problem in web services. In general, they model a behaviour protocol as a state transition system that must be compatible with other protocols in the application. These 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, which 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, a problem that is related to service compatibility can be addressed from the perspective of the interfaces. A service interface provides the specification for a data format and the service's functionality. The interfaces are more generic (i.e. contain less specification) than behaviour protocols. Matching the service interface to the service implementation is easier than matching protocols to the code. For example, incorporation of session types in a Java-like language requires syntax extensions and specification rules that are used to check behaviour conformance [ABB+16, KDPG16]. In contrast, the interfaces do not specify any information about the service behaviour and can be uniformly mapped to function or class declarations. As a result, the interfaces can be derived from the code or can first be specified in an *Interface Description Language* (IDL) and then provided as an API to the service.

An example of an IDL is Protocol Buffers from Google [Goo08]. This is a mechanism for serialising and storing structured data. A structure of the data is defined in the protocol buffers language (PBL)[2]. A developer specifies a structure for data exposed to the environment. Afterwards, the specification is generated as a library in a target language. The library provides an API for serialising the structured data. It can be used by the developer to read and write data to or from a variety of communication streams, even if communicating services are written in different languages.

A message in Protocol Buffers consists of one or more uniquely labelled fields. Each field has a name and a value type. The type is either an integer, a floating-point number, a Boolean, a string, raw bytes, or some other form of a message; the latter introduces a hierarchical message structuring. In addition to this, each field has one of three specifiers: compulsory, optional, or repeated (the latter allows arrays of fields to be created).

Using Protocol Buffers, the development of a service-oriented application is done as follows: a developer specifies the data structuring offered by a service to other services in the PBL. Afterwards, the specification is compiled into libraries, which are generated in all possible languages. The interacting services serialise or deserialise the data using the generated library before sending or after receiving the data in a message.

Support for subtyping in Protocol Buffers facilitates service reusability. Fields that are declared as being optional can either be present or can be omitted in the message. Therefore, a producer must provide only values for those fields that are marked as required. As a result, the service is compatible with a variety of possible inputs. On the other hand, all optional fields must be specified before using the service. If a service receives a field that is not listed in its Protocol Buffers specification, a run-time error occurs.

Apart from the limited support for subtyping, Protocol Buffers do not have features

[2] https://developers.google.com/protocol-buffers/docs/proto.

for software reuse. Therefore, existing services should be modified frequently before interacting with new services. The only way to guarantee consistency in data formats in a pair of interacting services is to use the same Protocol Buffers specification. This is impossible in applications that have their development spread across various businesses. A technology similar to Protocol Buffers is Apache Thrift [SAK07] from Facebook, but it fails to support features that facilitate software reuse, such as inheritance, polymorphism, or overloading. [3]

The importance of generic and reusable interfaces in web services is typically underestimated when comparing them with other concerns, such as performance, scalability, fault-tolerance, etc. The reason is that mismatching interfaces can always be fixed through manual configuration and service fine-tuning [AGR13b]. However, configuring service interfaces is often not a local process because any modification of the API for a service may trigger modifications in other services over the entire communication graph. As explained above, a similar problem arises while trying to achieve behavioural consistency in web services: if the behaviour of a service changes, how does one guarantee that it still conforms a global protocol? However, in the case of interfaces, establishing correspondence between interfaces and the code is an easier problem, one which is solved in this thesis.

Specifically, we address the following challenges regarding compatibility in service interfaces and data formats:

- **Achieving a proper level of abstraction in the interface.** The diverse requirements of customers are, typically, not known in advance. Therefore, the interfaces must be relatively abstract. On the other hand, too abstract interfaces may lead to communication errors due to imprecise service functionality and data format specification. The specification may allow excessive data formats that are not safe to use and may break the behaviour.
- **Providing a reusable interface.** Fixed interfaces may hinder their adoption in various contexts. Customers may have different needs, which change over time. Adjusting the interfaces for a particular context or breaking backwards compatibility is expensive and leads to poor productivity. Moreover, it can also make existing customers abandon the API and damage the reputation of the provider.
- **Given a set of services, how should they be composed in an application that exhibits the expected behaviour and which is also free of communication errors, such as a data format/functionality mismatch?** Typically, as with other errors, the number of communication errors increases with the size and complexity of the

[3] https://thrift.apache.org/docs/features.

application. An automated composition mechanism is required. In other words, we are looking for a mechanism that finds an errorless service configuration given a set of arbitrary services.

- **Providing flexible interfaces and configurable composition for proprietary services.** There are many use scenarios in which the code for a service must remain confidential. These are services that carry out financial transactions, user authorisation, encryption, etc. A composition mechanism that needs access to the source code of a service cannot be applied to proprietary services.

  This is a serious limitation for some composition approaches. For example, C++ templates, which introduce parametric polymorphism into the language, cannot be used with software that is provided as a closed-source library. Functions that use templates must be defined in exportable and externally visible header files, so that they can be specialised during compilation [Sta09, cF16].

However, the transition from the behaviour reconciliation problem to the interface reconciliation problem loses the state-dependence of the communication. Indeed, behaviour may depend upon its state, but information about that state is not preserved in a type signature.

1.1 Contribution

This thesis shows that *it is possible to contextualise generic interfaces that support subtyping, polymorphism, flow inheritance and configuration parameters, and to guarantee that data formats correspond* using constraint satisfaction and SAT techniques. The main contribution of this research is as follows:

1. We discover that none of the existing interface description languages provides support for flexible and configurable interfaces. As an alternative we propose the Message Definition Language (MDL), which is a term algebra for specifying data formats. The MDL is designed to be used for specifying interfaces in services in service-oriented applications. A term is basic block for constructing a representation of data formats in the MDL. Term variables are used for supporting polymorphism and flow inheritance. The latter is a mechanism for implicit data propagation in a computational pipeline [GSS08, GSS10]. Furthermore, we introduce Boolean variables as part of a term. They are used to specify dependencies between elements of the interfaces. We define the seniority relation for terms that corresponds to a relation between a data producer and a data consumer in web services. Term variables facilitate the reusability of services by enabling generic interfaces. On the

other hand, we must ensure that the interfaces are compatible in the given context. In other words, the seniority relation must be valid for all communicating services. We represent this problem as a constraint satisfaction problem (CSP).

2. We propose a fixed-point algorithm that finds a solution to the CSP. The algorithm computes a sequence of iterative approximations for term and Boolean variable values. A solution is found once an approximation makes all seniority relations valid. In order to be able to solve this problem, we solve Boolean satisfiability as a subproblem.

3. To demonstrate the use of the MDL in services, we developed a configuration mechanism for web services [Zai17]. The mechanism automatically derives the interfaces from the services, constructs the seniority constraints, runs the solver, configures the services using the solution produced by the solver and, finally, generates a library for each service. The library contains the implementation of services specific to data formats in the given environment. Being able to provide solution to the CSP guarantees that the data formats, which are sent and received from the library, are compatible with data formats in other services. Although our implementation is proof of a concept for services written in C++, the mechanism can easily be used in services that have been written in other languages, and which are commonly used for service development.

4. A major advantage of the configuration mechanism is its support for flexible and configurable classes (abstract data types) using the MDL. We enable flow inheritance for class fields and methods. Furthermore, we provide a mechanism for transforming method definitions in the pipeline.

5. Many stateful services have at least two input ports: one for receiving input messages and another one for receiving an immediate state of the service. It is often required to merge an input message with the immediate state and produce the result as an output message. We propose an MDL term called a *union* that automatically maps the message formats in input ports to the message format in an output port. The union is a construct that is automatically expanded in the correct message format. As a result, a service designer does not need to update the output interface if the input interface gets modified.

1.1.1 Publications

During the work on this thesis, the following publications have been produced (listed in chronological order):

1. Pavel Zaichenkov, Bert Gijsbers, Clemens Grelck, Olga Tveretina, and Alex Shafarenko. A Case Study in Coordination Programming: Performance Evaluation of S-Net vs Intel's Concurrent Collections. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, IPDPSW 2014*, Phoenix, USA, pages 1059–1067, 2014.
2. Pavel Zaichenkov, Olga Tveretina, and Alex Shafarenko. Interface Reconciliation in Kahn Process Networks using CSP and SAT. In *Fifth International Workshop on the Cross-Fertilization Between CSP and SAT, CSPSAT 2015*, Cork, Ireland, 2015.
3. Pavel Zaichenkov, Olga Tveretina, and Alex Shafarenko. A Constraint Satisfaction Method for Configuring Non-local Service Interfaces. In *Integrated Formal Methods — 12th International Conference, IFM 2016*, Reykjavik, Iceland, pages 474–488, 2016.
4. Pavel Zaichenkov, Olga Tveretina, and Alex Shafarenko. Configuring Cloud-Service Interfaces Using Flow Inheritance. In *1st International Workshop on Formal Methods for and on the Cloud, iFMCloud 2016*, Reykjavik, Iceland, 2016.
5. Pavel Zaichenkov, Bert Gijsbers, Clemens Grelck, Olga Tveretina, and Alex Shafarenko. The Cost and Benefits of Coordination Programming: Two Case Studies in Concurrent Collections and S-Net. In *Parallel Processing Letters* journal, 2016.
6. Pavel Zaichenkov, Olga Tveretina, and Alex Shafarenko. Constraint Programming for Non-local Web-service Interface Reconciliation. To be submitted to *TPLP Special Issue: Past and Present (and Future) of Parallel and Distributed Computation in (Constraint) Logic Programming* journal, 2017.

1.2 Thesis Structure

First of all, in Chapter 2 we provide a brief overview of object-oriented programming, aspect-oriented programming, component-based software engineering, service-oriented programming, and coordination programming — the technologies that facilitate modularity, decontextualisation, and software reuse. Although service-based applications ease the development of large-scale systems, automatic composition of services with support for interoperability is a key challenge today. We accept the challenge and focus on the interoperability problem from the interface perspective. We signify the importance of subtyping, polymorphism, and flow inheritance (a mechanism for propagating data and functionality along service pipelines). However, the form of structural subtyping and polymorphism that is required in service-oriented computing is different from the mechanism which are provided in languages that are commonly used for implementation of web services, such as object-oriented ones.

In Chapter 3 we defined a domain-specific language for describing interfaces which was called the *Message Definition Language* (MDL). The major novelty of the language is its support for (multiple) flow inheritance and configuration Boolean parameters. The language is essentially a term algebra for specifying data formats. We introduce the *seniority relation*, which is a partial ordering on terms, for comparing and coercing data formats. At the end of the section we use a three-buyer use case to demonstrate the capabilities of the MDL.

Furthermore, a formal approach for reconciling web-service requires a formal description of service-oriented networks. In Chapter 4 we introduce such a description in the form of a language of combinators. Combinators wire services into a network. We introduce a simple type system on top of the language that helps us to aggregate relations between service interfaces into a set of communication constraints.

In Chapter 5 we present a constraint satisfaction problem for web services, called CSP-WS, which is the key problem to be solved in this thesis. In a nutshell, the goal is to find values for term and Boolean variables, which are present in the seniority constraints. We tried to employ existing SMT solvers to solve the problem, but have not achieved this due to the specific properties of the constraints. We decided to develop an iterative fixed-point algorithm, which finds a solution for the problem.

In Chapter 6 we present the fixed-point algorithm that solves a modified version of the CSP-WS. As the first step, we consider the CSP-WS when all Boolean variables are instantiated. This allows us to focus on resolution of term variables only. For each term variable, the algorithm iteratively finds an approximation of the solution in the semilattice. This is more efficient than an exhaustive solution search, because we use a *branch-and-bound* approach to eliminate infeasible search spaces until a solution is found. In Chapter 7 we present the algorithm for solving the original CSP-WS, which is a modification of the algorithm from Chapter 6. We extend the algorithm with solution of Boolean satisfiability using a SAT solver.

The contribution of this thesis is not limited to finding a solution for the theoretical problem at hand. Using our mechanism, we also demonstrate the configuration of interfaces in services written in C++ (see Chapter 8). We designed and implemented a protocol for a complete reconciliation process, starting from the code for generic services and a network topology to the compiled mutually compatible libraries. The configuration process consists of:

1. annotation of a service source code with macros;
2. the derivation of terms representing the interfaces from the code;
3. constraint construction given the terms and network topology;

4. a CSP-WS solution;
5. the derivation of macro values from the CSP-WS solution; and
6. a separate compilation of annotated services libraries.

Furthermore, we demonstrate that our configuration protocol supports services that are provided as non-analysable closed-source services. The latter is essential for proprietary libraries provided as services in the cloud.

All services are either stateless or stateful. The former are 'query' services that provide some information to clients without modifying an internal state of the service. The latter are the rest of services, which have a persistent state. *Service statelessness* is a design principle of the service-oriented architecture which states that scalable services must be separated their state as much as possible [Erl05, RDL+09]. This is typically achieved by exporting a state of a service to an external component, which exists outside of the service implementation boundary (for example, a dedicated database). In the context of service interfaces, the service statelessness principle leads to the fact that each service has at least two input ports, one for receiving an input message and another one for importing an immediate service state. An output message is typically produced as a combination of these two messages, which needs to be reflected in the interface. Specifically for stateful services, in Chapter 3 we introduce an MDL term called *union* that declares 'a service interface that is produced by merging any two service interface', and in Chapter 9 we discuss support for message synchronisation in our interface configuration protoocol.

Finally, in Chapter 10, we illustrate the configuration protocol using a simple image processing use case. We implemented a k-means clustering algorithm as a service-based application to demonstrate the overall mechanism and support for subtyping, overloading, and flow inheritance in the MDL. The source code of the service and auxiliary transformations and derivations for the use case are provided in Appendix A.

Chapter 11 concludes the thesis with a summary and a discussion of potential directions for further research.

Chapter 2

Background

> *'the separation of concerns' . . . it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant.*
>
> – E. W. Dijkstra [Dij82]

A conventional centralised software engineering approach results in inadequate complexity and costs to the design, deployment, maintenance and evolution of modern software systems. Software intensive systems have reached an unprecedented scale in every measure: the number of lines of code; the number of people involved in the development; the number of dependencies between software components; the amount of data being stored, accessed, manipulated, and refined; and the number and diversity of hardware elements composing the system. Each user group which is involved in the development process has different goals and tools which leads to conflicting, unknown, and diverse requirements; continuous evolution and deployment; heterogeneous and changing elements; erosion of the people/system boundary, and failures in parts of the system. Therefore, the systems must be configurable and adaptable so that they are able to meet the requirements of all user groups who are involved in the development process. [FSW+06]

In general, the problem can be addressed from different perspectives. First, we can try to analyse the protocols of components for compatibility and reusability. This is the most advanced and yet the most challenging approach. Session types [Hon93, THK94], interface automata [DAH01], and contract automata [BHM05, BCZ15] attempt to solve the problem, but no solution that guarantees the compatibility and reusability of software components exists so far.

In this thesis, we solve a simpler problem. We ensure compatibility and reusability on the interface level. This approach provides weaker guarantees, because it does not take the behaviour of components into account. On the other hand, the solution presented here can be integrated into a framework for solving the original problem of protocol

compatibility.

During the last four decades there have been a good many attempts to design a methodology for managing the complexity of software systems. In this chapter we provide an overview of the seminal works in software systems engineering:

1. data encapsulation and the principle of the separation of concerns;
2. inheritance versus composition in object-oriented programming;
3. the separation of components and global dependencies in aspect-oriented programming;
4. component-based software engineering;
5. the choreography of distributed and decontextualised components in web services;
6. separation between computation and communication concerns in coordination programming.

We discuss the advantages and disadvantages of these approaches. As a result, we demonstrate that modularity, decontextualisation and hierarchical structuring is the key to managing the complexity of software systems. Furthermore, we introduce flow inheritance and show its importance for pipelined data processing. Across the following chapters we design a generic mechanism for adaptable service configuration that incorporates these software composition properties.

2.1 Engineering Modular Software Systems

Segmenting an application into *modules* reduces the complexity of software design. Inputs and outputs for each module are well-defined at implementation time and a module can be designed with little knowledge of the code in another module. As a result, development time decreases. Furthermore, the correct behaviour of the module can be tested independently and runtime errors can quickly be resolved because the scope for error searching is limited to the specific module. It is possible to study one module at a time, and in general the system is better understood and designed. Finally, the modular system is flexible, because the modules can be replaced or drastically changed without significant modification of the rest of the system.

That is the theory. In practice, an application can be segmented into modules in various ways and it is obscure to design a *modularisation* (a specific segmentation of an application) that includes all the advantages described above. Some modularisations lack flexibility, because changing some design decisions, such as input format, would result in changes inside every module. David Parnas formulated an 'information hiding'

criterion (which is more or less the *'data encapsulation'* principle as we know it today) for guiding the process of creating modules. He suggests that a modular system needs to be designed in a way which ensures that all interfaces between modules are fairly abstract. Major design decisions, such as data structures, core formats and conventions, should be structured as separate modules and hidden from other modules [Par71, Par72].

In order to achieve a complete modularisation, decontextualisation is also required. Shortly after Parnas formulated the principles of encapsulation and modularisation, Edsger W. Dijkstra explicitly formulated the principle of the *separation of concerns*. In order to control the growing complexity of the systems, a software engineering problem needs to be segmented into such modularisation, so that different concerns, which relate to the problem, could be treated in an isolated manner without any requirement for knowledge about other parts, and composing a solution out of solutions to the subproblems. In order to achieve a complete separation of concerns, it is necessary not only to segment the application into modules, but also to decontextualise them from each other in order to avoid context that can be shared between the modules. [Dij82]

2.1.1 Object-Oriented Programming

It may seem that *classes* in *object-oriented programming (OOP)* [Ing81] can play the same role as modules, because a class restricts the visibility of abstract data types (which means fields and methods) that it contains. Furthermore, the class supports other features, such as inheritance (subclassing). It is a valuable mechanism that facilitates code reuse and is used for specifying hierarchical relations between objects; however, it breaks the encapsulation provided by the data types: the classes that belong to the same hierarchy cannot be considered as being self-contained modules, because changes in the class affect the behaviour of other classes in the hierarchy [MS98]. Therefore, the hierarchical classes can be regarded as being one module only, and their development should be assigned to one team.

Inheritance

For a long time inheritance was thought to be productive for code reuse [JF88]: the code that is shared by several classes can be placed in their common superclass, and new classes that rely on this code can start off by being inherited from the superclass. Every subtype of the superclass can be created through *incremental inheritance*; that is, by only adding new methods to the class definition [Cus91]. However, at some point a problem known as *inheritance anomaly* was identified [MY93, CRR98].

The anomaly typically occurs in *concurrent object-oriented programming (COOP)* lan-

guages [YT86] where synchronisation constructs, such as method guards, locks, etc, are used inside class methods. The idea of the anomaly is as follows. Assume that there exists a class, C, that implements some behaviour, B. If a programmer decides to implement a behaviour B' that is derived from B, then they need to implement a class C' that implements B'. Ideally, they should be able to reuse the code in C; however, they are forced to redefine much of the C behaviour while implementing C'. This rewriting is called the anomaly and it often occurs when concurrency is involved [Wis06]. The reader is encouraged to refer to [MS04] for typical examples of the inheritance anomaly. There has been a good deal of effort to solve the inheritance anomaly problem during the last two decades ([Wis06] provides a comprehensive survey on papers related to the inheritance anomaly); however, none of the proposals have been implemented as practical solutions [GS14].

It is dangerous to use class inheritance in concurrent programming. In sequential languages, class methods can be invoked in any order. On the other hand, in concurrent languages, methods may not execute in arbitrary order; the order is preserved by synchronisation constraints. If we incrementally inherit C' from C, we expect methods that are present both in C and C' to respond identically to input messages. Such behaviour is known as *behaviour preserving*. It was proved by Crnogorac et al [CRR98] that languages with the support for behaviour which preserves incremental inheritance are not free from the inheritance anomaly. Informally, it means that there is no perfect inheritance mechanism. Instead, there are two options: either we allow classes that are not behaviour preserving, or allow the inheritance anomaly to occur.

In other words, the problem is caused by the fact that a class methods encode behaviour protocols rather than relations on types and data, which are specified by the inheritance mechanism. Inheritance ignores the order in which methods may be executed, whereas some ordering may lead to a violation of the global protocol that must be fixed by modifying the inherited methods. Session types attempt to solve the problem by incorporating protocols into type signatures and verifying the correctness of the global protocol when inheritance is used [GGRV15].

Having said that, the contradiction between anomaly freedom and behaviour preservation was identified only for COOP languages, which were known at that time, and potentially there may exist a COOP language that satisfies both properties. This question was recently addressed in [GS14]: it was proved that checking behaviour preservation in a language that is anomaly free is PSPACE hard. Therefore, verification of a program in an anomaly-free language that ensures substitutability may be expensive and humanly infeasible.

Composition

Since inheritance does not facilitate code reuse, an often-stated OOP principle today is that composition should be favoured over inheritance in order to achieve polymorphic behaviour and code reuse [GHJV94, Kno02]. Following this principle, a software design begins with the declaration of interfaces that represent the certain behaviours of the system. The use of the interfaces allows polymorphic behaviour to be supported. For each interface there exists at least one implementation; an alternative system's behaviour is accomplished by providing an alternative class that implements the same interface. Thanks to this, code reuse is achieved without inheritance. Notably, some languages, such as Go, provide type composition exclusively.

On the other hand, composition requires the implementation of all methods specified in a class interface, even if some method behaviour is identical in various classes that are implementing one interface. The disadvantage can be avoided by using mixins [FKF98] and traits [SDNB03]. Furthermore, Tempero et al [TYN13] found out that many open-source Java software projects use inheritance in non-trivial scenarios. Further work in this direction would help to identify whether the use of inheritance in modern large-scale projects is justified or if replacing it with composition would improve modularity, flexibility and code reuse in the projects.

Support for inheritance or composition in the environment with replaceable modules require the verification of module behaviour compliance. In addition to session types [Hon93, THK94], various approaches that model the behaviour as a state transition system exist (interface automata [DAH01], contract automata) [BHM05, BCZ15]. They build formal frameworks to statically address better dynamic guaranties, such as communication safety, deadlock-freedom, protocol fidelity, etc. On the other hand, behaviour protocols cannot automatically be derived from the code today and have to be provided explicitly.

2.1.2 Aspect-Oriented Programming

Hierarchical modular mechanisms that are based on inheritance in object-oriented languages are unable to separate concerns in a complex system. *Aspect-oriented programming (AOP)* [KLM+97, KHH+01] has been specifically proposed as a technique for improving the separation of concerns as it was formulated by Parnas in [Par72]. In particular, AOP introduces two kinds of entities: 1) *components* are units of the system's functional decomposition, which can be cleanly encapsulated in a well-localised generalised structure (for example, object, method, procedure, API), and 2) *aspects* are entities that cannot be cleanly encapsulated in a generalised structure. The latter tend not to be units of the system's

functional decomposition, but rather properties that affect the performance or semantics of the components in a systematic way [KLM+97]. Typically the aspects relate to development process functionality, such as testing, tracing, logging, and contract checking. The goal of AOP is to provide facilities to separate components and aspects from each other, as opposed to general-purpose programming language that provide facilities to separate only components from each other. The aspects are defined in a separate *aspect language* that contains a specification which needs to be executed before and after the component code. Scattered functionality, such as logging, exception handling, or authentication, are represented as aspects and are passed to components as dependencies.

However, data dependency between the aspect and the component fails to solve the problem of the separation of concerns [Ste06]. If the dependency is moved to an explicit interface, the Parnas principle, which states that the interfaces in modular systems must be abstract, is violated. Alternatively, the implicit interface allows the aspect to access the data encapsulated in the model. A general problem of AOP is that when the code is moved out of the component's context to the aspects, it must carry a reference to the context upon which it depends. Supporting modularity does not lead to a complete separation of concerns, because the context is shared across modules.

2.1.3 Component-Based Software Engineering

Component-based software engineering (CBSE) [SBW99, HC01] is a software engineering approach that facilitates the separation of concerns. It relies on the decomposition principle in which semantically related components should be placed in one module and the remaining components are independent. Structurally the components are relatively abstract: they can contain functions, data, classes and even class hierarchies. The components expose their interfaces to the environment and communicate only by sending messages that other components can process, which is checked using interface specification. The messages must be self-contained (without references or pointers), otherwise, the decontextualisation principle would be violated.

Some surveys [B+81, Jon97, Kos03] show that, on average, about 70–80% of a software project's lifetime is spent on maintenance. Therefore, reusability is an essential property of software components. The CBSE itself does not offer any features that would help to improve reusability and, as a result, a programmer relies only on mechanisms provided by the programming language, such as polymorphism, higher-order functions, etc. As a result, the programmer needs to come up with a trade-off between reusability and usability: the more generic the interface is, the more reusable is the component, and yet implementation becomes ever more complex and, therefore, the component becomes less

usable and maintainable.

Component adaptation and reusability is recognised as being one of the most important problems in CBSE [HO98, CJ99, MA03, BCP06]. The possibility of software designers to integrate off-the-shelf components into the application they develop opens up a field of opportunities for the development of a component marketplace and component deployment in overall terms [BW98, DSWdS09]. At the signature level, component-based applications address interoperability by means of an Interface Description Language (IDL). IDLs specify the functionality of distributed components in a uniform language, such as an XML description for instance. More specifically, the IDL interface defines the compatible data formats and the signature of the methods being offered by the component. For correct communication to occur, the interfaces for the communicating components must exactly match [AGR13a], which is a problem otherwise. The mismatch between data formats and method signatures is a common scenario, providing that components are developed by distributed teams which are unaware of each other's activity. Typically, in order to achieve the compatibility of the components, hand-crafting is required. The challenge is to lift component integration from rigorous labour to a software engineering activity.

Some existing works focus on checking the compatibility of component protocols, which define the behaviour of the components [BPR02, BBC05, BCP06]. This, however, requires the extension of IDL interfaces with a formal description of the behaviour of the components, which explicitly declares interaction protocols. Technically, this is achieved by using a process algebra, such as π-calculus [SBS06], interface automata, or session types [VVR06], which allow behaviour to be effectively described by means of types instead of processes. [BBC05, BCP04, BCP06] present a formal methodology for the adaptation of components which are presenting mismatching interaction behaviour. Given the interfaces of two components with mismatching behaviour, an adaptor that allows the components to interoperate can automatically be generated. If an adaptor that strictly satisfies the given specification does not exist, in many situations an adaptor is nevertheless deployed by weakening some of the requirements stated in the specification.

Unfortunately, in the existing works, issues that regard the incompatibility of data representations in different components have been ignored. Even if we ignore the behavioural aspects, two components may be unable to interact if their data format requirements are unfulfilled.

Another issue is caused by inflexibility in the IDL interfaces. Although CBSE components are completely decontextualised from each other, developing an application independently in dispersed teams is laborious due to the inflexible interfaces involved. If modifications of the server code affect methods or data formats specified in the interface,

the IDL compiler needs to regenerate interface files that are provided to all clients who interact with the server. This can trigger a chain reaction of modifications in the application and each conflict between interface specification and implementation needs to be resolved manually.

2.1.4 Service-Oriented Architecture and Web Services

Service-oriented architecture (SOA) [BEK+00, PL03] is a software design approach in which an application is composed of heterogeneous, loosely-coupled web services, which communicate via standard internet protocols (HTTP or SMTP). SOA offers a middleware that enables client-server communication outside the local network without requiring sophisticated configuration, any uniformity of protocols, or homogeneous hardware.

There is no clear dividing line between SOA and CBSE. In general, SOA can be approached as the enhancement of components. Similarly to CBSE, a service has a clearly defined interface that conforms to a behaviour exhibited by service. Unlike the object-oriented approach, SOA is constructed from loosely joined, highly interoperable business services which are implemented using different development technologies. In this way, SOA shares a lot of fundamental similarities with CBSE.

The most significant property of web services is their loose coupling. Since web services are accessed by a network, it does not make any difference where services are located and whether they use the same language or operating system. Moreover, services can be publicly discoverable (using technologies such as UDDI), and accessible from the internet. Therefore, during service development it is not known how the service will be used. Usage for a variety of purposes is possible only thanks to loose coupling being involved.

Another advantage of SOA that leads to a looser coupling between services is the lack of a standard protocol. Component-based architectures, such as DCOM [BK98], CORBA [Sie00], or EJB [Inc03], rely on their specific protocols instead of reusing existing ones. Thereby network administrators had to explicitly open a new port in the firewall for every communication partner [Pet06]. Moreover, the component-based architectures encode messages in a binary format, which cannot be analysed by company security systems in a reasonable way. This introduces the risk of malware package delivery in the software heart of the company. In contrast, web services use an established protocol (HTTP or HTTPS) and an open data format (XML or JSON). As a result, SOA has advantages over DCOM, CORBA and EJB, such as firewall friendliness, universal acceptance, and platform and language independence.

For the last decade web services have offered a promising technology that facilitates

the development of large-scale distributed systems. According to the 'ProgrammableWeb' web service publication website [Pro13], the number of publicly available web services has exponentially increased since 2005. Businesses use them to expose their internal business systems as services that are available via the internet. On the other hand, clients can combine services and reuse them for developing their own applications or constructing more complex services. In addition to this, the rapid development of cloud computing, social networks and the Internet of Things accelerate the growth of public web services [BSD13, DYV12, MASM13].

Although web services continue to play an important role in modern software development, service *composition* is still a key challenge for service-oriented computing and web services. Web service composition empowers organisations to build inter-enterprise software, to outsource software modules, and to provide an easily accessible functionality for their customers. Furthermore, service composition reduces both the cost and risks involved in new software development, because the software elements that are represented as web services can be reused repeatedly [SQV+14].

Achieving interoperability in web services is more challenging when compared to interoperability support in distributed object-oriented frameworks, such as CORBA and Remote Method Invacation (RMI) [WRW96]. Consider the substitution principle, which is the essence of subtyping and is a key requirement for interoperability. RMI follows the Java object model and, therefore, the substitution principle in RMI is valid, to an extent to which it is valid in Java. Similarly, the CORBA interfaces, which are object-oriented, support a subtyping relation that ensures substitutability. However, neither RMI nor CORBA include the translation of objects (in CORBA the objects are passed by references), because the substitution principle failure in object-oriented frameworks for web services arises from the required data binding between the object layer and the service layer [AGR13b].

Several surveys on web service composition have been published recently. [DS05] provides overview of the available composition approaches based on classification criteria, such as manual/automatic and static/dynamic composition. [RS05] provides an overview of automatic web service composition approaches. [SK03, MM04, MG06] all focus on web service composition standards. [SQV+14] is one of the most recent surveys, which presents the last decade's developments in web service composition and the current issues and activities involved in terms of the research topic.

In general, current efforts in web services composition can be grouped into three categories: *manual*, *automatic*, and *semi-automatic* [MM04]. Manual composition involves an application designer having to handcraft the composition of interacting services using one of the existing composition languages, such as BPEL [ACD+03] or OWL-

S [MBH+04]. This is a non-trivial and error-prone field of work that does not guarantee correct execution. Automated service composition models usually exploit the semantic web [BLHL+01] and artificial intelligence techniques. The challenge is to automatically construct a composition of web services, given a selection of web services and specified user requirements. Unfortunately, implementing a completely automated service composition is still impossible because the web services do not share a full understanding of their semantics, which affects the automatic selection and configuration of web services [BCDGM05, MBE03]. A semi-automatic approach leverages manual and automated service composition by providing the designer with tools that assist them at each step of the composition. The existing automatic and semi-automatic composition approaches focus on behavioural communication consistency between the services (the behaviour specification is defined by means of finite state machines [HNT08], interface automata [HM05], process calculi [WDW07], or session types [CHY07]) and does not attempt to establish consistency on a data format level.

This dissertation is the first attempt to present a mechanism for consistent service configuration that not at the behavioural level, but on the level of interfaces instead. The IDL, which is presented in Chapter 3, can be used to specify interfaces of arbitrary complexity with the support for structural subtyping, inheritance and polymorphism. Although the presented approach cannot be used to argue about the behaviour of a service, in contrast to behavioural protocols, such interfaces can automatically be inferred from the code of a service as presented in Chapter 8. Furthermore, our approach does not raise any security issues in closed-source services when service behaviour is publicly exposed in the form of a contract.

There are two ways to compose a service-based application: service orchestration and service choreography [Pel03]. The orchestration refers to an executable business process that coordinates the interaction amongst different services. This differs from service choreography, which is more collaborative and allows each party to describe its role in the interaction. In contrast, choreography represents a global description of observable behaviour that is specified by service interaction rules and agreements between multiple web services. This dissertation addresses the following issues in choreography composition.[1]

Dependable service composition. Services in a service-based application are distributed and provided autonomously by various organisations. Reliable and dependable service composition is a significant challenge today [DS05, SQV+14, SMY+14]. Developers of applications, particularly safety-critical applications, such as health

[1] Since web services have evolved from component-based architectures, the issues in web services that we raise in this dissertation are also present in component-based architectures.

care, stock trading, or nuclear systems, must be able to check the soundness and completeness of service composition at the early stages. Therefore, model checking and the verification of fault-tolerant web services is being actively researched today [BSS12, ZL13, SMY+14].

Web Services Choreography Description Language (WS-CDL) [G+02] and Web Service Choreography Interface (WSCI) [AAF+02] are choreography standards. They provide the means for tools to validate conformance to choreography descriptions in order to ensure interoperability between web services, and these are used to describe the observable collaborative behaviour of multiple services from a global perspective.

On the other hand, the choreography is wired to specific interfaces defined in Web Service Description Language (WSDL) [CCM+01] or Web Application Description Language (WADL) [Had06], which is too restrictive for dependable service composition. The choreography is statically bounded to specific operation names and types, which impedes the reusability of compound services and their interaction descriptions [BDO05]. Choreography descriptions that rely on abstract interfaces would allow services to be specialised within the context based on the requirements of the customers, rather than restricting communication to specific data formats and operations.

Adaptable and autonomous service composition. Today the environment in which services are developed and executed has become more open, changing, and dynamic. An adaptable and flexible approach to service composition is required. Self-configuration and self-adaptation are promising recent research topics [CDM09, SBMN09]. Self-configuration allows a service automatically to discover and select services that meet its requirements. Self-adaptation helps services to adjust their behaviour to changes in other services without requiring a programmer's intervention. Typically, web services that are developed by different organisations have incompatible interfaces and cannot interact directly. One way to deal with the problem is to provide automatic or semi-automatic service mediation [KMNB+09, LFMS10, XFZ10, MNXB10]. In practice, however, interaction between services often cannot be mediated due to irreconcilable mismatches.

The issues mentioned above shift the research focus from static composition methods to dynamic ones, which are more flexible and allow choreography configuration and adaptation at runtime. However, for obvious reasons it is more difficult to establish compositional correctness in the case of dynamic methods.

2.2 Coordination Programming

Structuring applications as a set of decontextualised components not only facilitates productivity and eases software development, but can also improve performance too. In coordination languages, such as Concurrent Collections (CnC) [BCK+09, BBC+10] and S-NET [GSS08, GSS10, ZGG+14], the separation of concerns is achieved by employing a coordination language operating on top of concurrency-agnostic domain-specific components, written in a conventional language. The separation makes individual components not only oblivious to the detailed network composition, but also concurrency agnostic.

A coordination layer provides a glue that connects the components in a complete application. Two kinds of glue exist for connecting components together: a tuple space model and a streaming model. These models provide different mechanisms for communication for decoupled components, which are covered below.

2.2.1 Tuple Space Model

In the tuple space model, the components interact by sharing access to the memory that holds tuples. Such shared memory is called a *tuple space*. Occupants of the tuple space are accessed by logical name; therefore the only information that the processes share is the discipline of occupants' tag use. Linda is the first coordination language to be based on the tuple space model [ACG86, GC92]. Many further languages (for example, Lime [PMR99], NetWorkSpaces [BCS+09], or Swift [WHW+11]) adapted Linda's tuple space model.

Following the coordination approach, the CnC model decouples computation from the expression of its parallelism. A domain expert determines the design of the algorithm, and a tuning expert can be called upon to deal with parallelism, communication, scheduling, and distribution issues. The CnC components are required to be pure. According to their semantics, the components are composed from three sections: 1) the input section that is the opening part of the component where it reads data from its statically-known tuple spaces; 2) pure functions which represent side-effect-free computations that work out the result based solely on the read data; and 3) an output section that optionally places the result from the previous section in the form of newly computed data in the statically-known tuple space.

Given the components, the communication graph and data dependencies between the components (the latter can be supplied by the programmer in the form of *dependency functions*), optimum scheduling and resource management can be achieved at runtime.

Self-sufficiency and a purity of components allows the runtime to schedule them when their input becomes available or to re-launch them later if it is impossible to precompute some data dependencies before task scheduling.

2.2.2 Streaming Model

S-NET and some other languages and technologies [MKD93, AHS93, Arb04] use streams to glue components into a single application. In contrast to CnC, S-NET does not share a contract with a programmer in regard to the structure of a component. On the other hand, the components are not allowed to hold an internal state and consequently cannot be used to synchronise messages (it would need to hold on to one while expecting another). It means that messages come to the component pre-synchronised by the coordinator and also that the component can be transparently wound down and reinstated between messages, which makes it especially flexible in a distributed computing context. On the other hand, the coordination program presents the coordination compiler with structures that can be analysed to learn behaviour modulo the behaviour of the components. All of this offers attractive benefits for large-scale applications. There is one obstacle, though, in getting the coordination approach to work in practice: as it espouses a higher level of abstraction (which stems from its characteristic separation of concerns), practitioners tend to be wary of a possible loss of performance, especially in applications of the 'no expense spared' type, where performance is the primary (if not the only) concern.

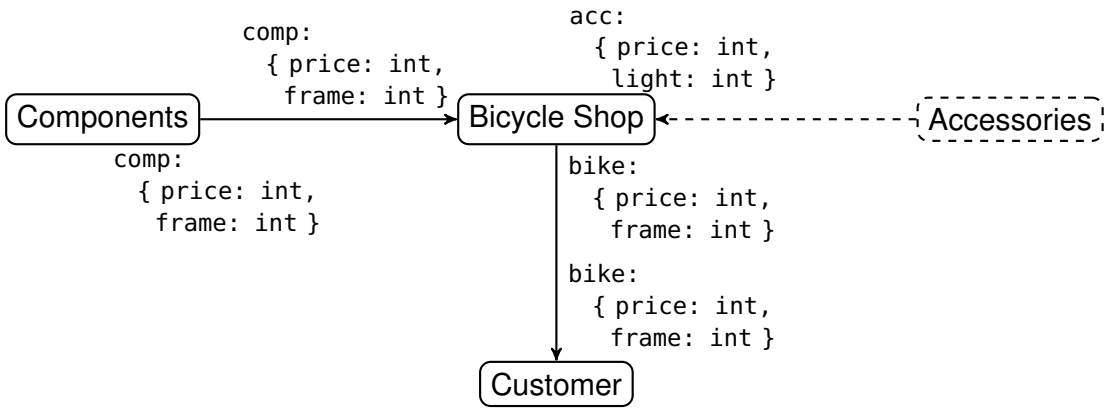
However, a higher level of abstraction *and* high performance can live happily together. Our experiments presented in [ZGG+14] demonstrate that structuring an application into a set of self-contained components in CnC and S-NET does not negatively affect the performance. Both models use dataflow synchronisation and single-assignment semantics for components, but S-NET is a fully fledged coordination language, which promotes automatic component configuration and reuse, while CnC requires a more significant integration of the component and runtime system codes.

Our experiments show that providing mechanisms which serve to ease the development of applications that are structured as a set of decontextualised components may be useful not only for service-oriented computing, but also for a high-performance computing application. On the other hand, modularity and the reuse of software are not as important for performance engineering as they are for the design of service-based applications, because the components in a performance-oriented application are typically tightly-coupled and are developed within one organisation for the sake of achieving high performance and power efficiency.

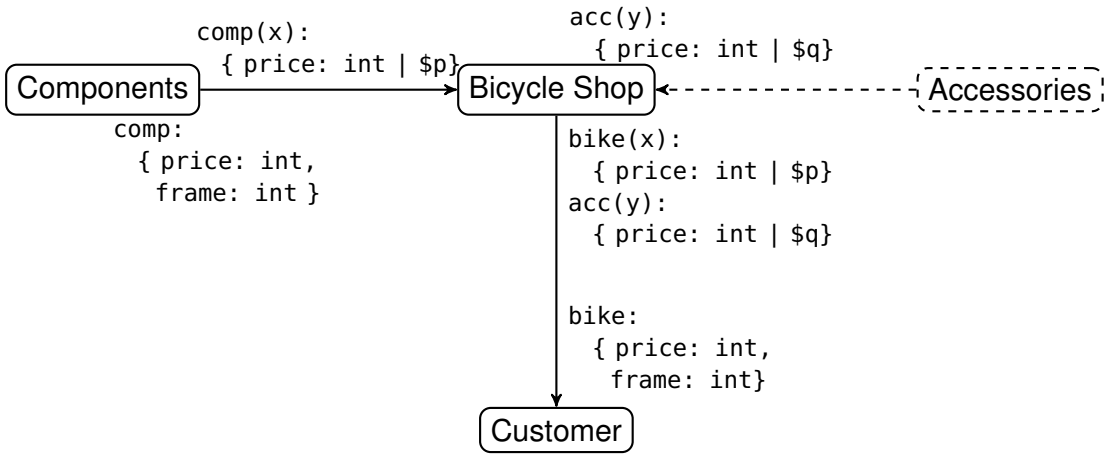
2.3 Subtyping, Polymorphism and Flow Inheritance

The web services are typically developed in a decentralised manner: an organisation that 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. 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 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 by an example. Consider the service-based network in Figure 2.1.



(a) Interfaces specified in one of the existing IDLs



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

Figure 2.1: A service-based application that illustrates advantages of introducing variables to the existing IDLs

This example illustrates a purchasing system scenario in which an online bicycle shop sells bicycles or accessories with configuration provided from the component and accessory suppliers. The suppliers provide the shop with available configurations. After computing the final price, the shop sends a quotation to the customer. In Figure 2.1a we provide a simplification of service choreography with interfaces specified in WSDL or any other of the existing IDLs, such as Google Protocol Buffers [Goo08] or Apache Thrift [SAK07, Apa11].

The interfaces in a pair of communicating services are identical: a message that the consumer expects must have precisely the same format as the one declared by the producer [AGR13a]. 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 this way it can be seen that web services are tightly coupled, which contradicts the concept of SOA. This behaviour is caused by an *impedance mismatch* problem between objects and tree structures in XML and JSON [LM07]. 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 nevertheless shows that *subtyping* would be useful for web service interoperability [CL06, LC10, Men09]. [AGR13a] demonstrated that existing web service models, such as SOAP and REST, fail to preserve the subtyping property while respecting the loose coupling principle.

In the dissertation we introduce an IDL with support for structural subtyping (both depth and width subtyping). When two services are connected by a unidirectional communication channel (a source is called a producer and a sink is called a consumer), a consumer agrees to accept any input that can *safely* be used instead of the expected input. For example, in Figure 2.1b the bicycle shop is allowed to deliver any input to the customer as soon as it contains information about the `price` and the `frame`.

Furthermore, in our approach the reusability of services is improved by introducing *term variables*. The term 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 the variable. A variable can also be present in a record in place of a 'tail'. This is a form of *row polymorphism* that implements *flow inheritance*, which is essential for pipelined data processing (see Section 2.3.1, Section 3.5, and Chapter 8 for flow inheritance details). Flow inheritance is a novel mechanism for automatic data propagation across the services. It cannot be supported by the existing frameworks at all due to global network analysis that is required for flow inheritance support. In the

example, the variable `$p` is present in all input and output interfaces of the Bicycle Shop service. The Components service adds an element `frame: int` to its output messages. Although the Bicycle Shop service may not require the element, when using flow inheritance the element is forwarded directly to the Customer service if it requires the element.

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 Figure 2.1 are inconsistent, because the Bicycle Shop service cannot provide accessories to the customer if the latter wishes to buy them. However, in Figure 2.1b 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 Figure 2.1b can infer which operations are not used in the context, while such analysis cannot be applied to the network in Figure 2.1a.

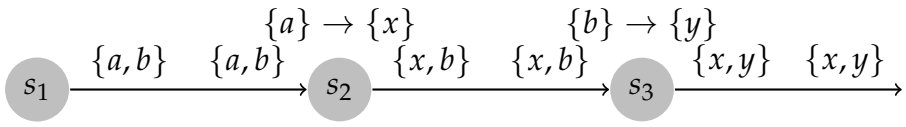
Interface variables provide facilities similar to C++ templates. Services can specify a generic behaviour that is compatible with multiple contexts and input/output data formats. Given the context, the compiler then configures the interfaces based on the requirements and capabilities of other services.

2.3.1 Flow Inheritance

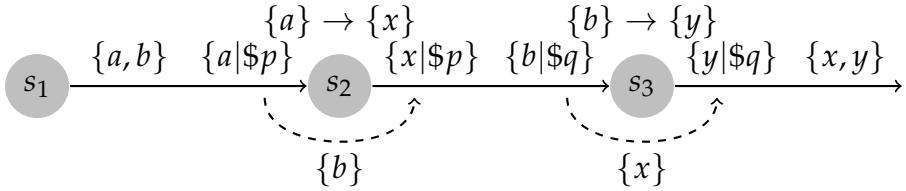
The flow inheritance [GSS08, GSS10] is a novel mechanism for forwarding data from producers to indirect consumers (i.e. consumers of the data that are not connected to the producers by communication channels) in computational pipelines. Decontextualised services that produce data do not have information about services that demand this data in the context. Similarly, decontextualised consumers are not aware of services which can provide the data that they demand. Flow inheritance is a mechanism that analyses the application network and 'connects' the producers with the consumers allowing the former to bypass the data to the former. The existing mechanisms for implementing web services do not support flow inheritance, because it requires global interface analysis. The existing mechanisms can only check pairwise relation between interfaces of connected services.

Informally, flow inheritance can be illustrated using the following example. Assume services s_1, s_2 and s_3 are connected in a pipeline (see Figure 2.2a).[2] The service s_1

[2] For simplicity, we assume that services called s_i have one-step semantics: having received the input message, they compute the result and return to the initial state without storing the state. We discuss services with a persistent state in Chapter 9.



(a) A pipeline without support for flow inheritance



(b) A pipeline with support for flow inheritance

Figure 2.2: A pipeline of services. $\{\ldots\}$ denotes a record as a message type

produces a record that contains fields of type a and b[3], which can be processed by s_2 and s_3 respectively. Although b is not required by s_2, it must be included in the input and output interfaces of s_2, so that it could be forwarded directly to s_3. Without flow inheritance, if a programmer wants data to be implicitly bypassed through the services, they are forced to explicitly modify services, or apply additional wiring with splitting, storing and resynchronising data.

Consider a pipeline with support for flow inheritance (Figure 2.2b). Flow inheritance automatically propagates b from s_1 to s_3 and x from s_2 to the output. We declare the interface of s_2 service as $\{a|\$p\} \rightarrow \{x|\$p\}$, where a is the type of an element in an input message, x is the type of an element in an output message and $\$p$ is a variable that matches part of the input message that is not matched by the type a and is used to redirect the data to the output. Furthermore, flow inheritance can be disabled by modifying the service interface. For example, if the variable $\$p$ is removed from the output interface of s_2 (that is to say the interface becomes $\{a|\$p\} \rightarrow \{x\}$), the service can no longer inherit data to the output of the service.

So far we have illustrated the inheritance mechanism for services with only one input and one output (SISO) channel. On the other hand, the inheritance mechanism has more complex semantics in services with multiple input and output (MIMO) channels. Inheritance in services with MIMO channels can be implemented in different ways and row polymorphism is a flexible mechanism for supporting various forms of the inheritance. In the examples in Figure 2.3a and Figure 2.3b, a service s_4 has one input channel and two output channels. The service inherits record elements that are not matched by the pattern a and matches them using a variable $\$p$. By modifying the interface, a designer can enable or disable inheritance in output channels. Similarly,

[3] In examples in this section we use letters to denote elements of records; the particular structure of elements is not important in this context.

in Figure 2.3c and Figure 2.3d, s_5 is a service with one output channel and two input ones. Inheritance from several input channels (Figure 2.3c) raises a multiple inheritance problem: if input channels can provide data of various formats, it is unclear which data should be inherited to the output. Forbidding multiple inheritance (Figure 2.3d) is a potential solution, but service flexibility deteriorates. A more generic solution allows inheritance from all input channels: in Figure 2.3e every channel has a different tail variable, where some elements in $\$r$ are taken from $\$p$ and some from $\$q$. Finally, the example in Figure 2.3f demonstrates the fact that for services with accumulating facility flow inheritance works in a special way. r is a reductor that produces a message in response to a set of input messages. Every input message may contain data that has to be inherited and there is only one output message. Since the data from various input messages cannot be inherited in one output message, inheritance in services with accumulating facility must be disabled.

In this thesis, we provide a generic mechanism for specifying all of the aforementioned forms of flow inheritance for a stateless service (Figures 2.3a to 2.3e). As far as services with the accumulating facility are concerned, we present an idea of a domain-specific language for message combination in Chapter 9. In this way, accumulation in service-based applications can be externalised from the services. It should positively affect run-time properties, such as performance, fault-tolerance and reliability.

2.3.2 Dependable and Adaptable Service Composition

The existing technologies for service choreography are wired to inflexible interfaces as defined in WSDL or WADL and do not allow dependable service composition. Furthermore, self-adaptation to changes in the interfaces of other services without a programmer's intervention is impossible.

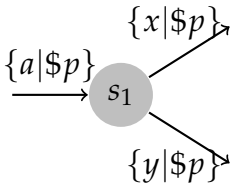
In order to be able to solve these problems we need a mechanism that links the input/output interfaces of individual services and tracks dependencies between them. This can naturally be achieved using logical expressions as illustrated in Figure 2.4. s_6 is a service with dependent interfaces. Expressions $x \wedge \neg y$ and $y \wedge \neg x$ are logical predicates called *guards*, where x and y are Boolean variables that are shared across the interfaces of one service. Depending on the values of variables, the predicates evaluate to true or false. The former requires the corresponding element to be present in the message and the latter forbids it. As a result, we can use Boolean variables and predicates to specify arbitrary relation between entries for input and output interfaces.

In this particular example, s_6 transforms coloured geometrical shapes: a circle (with a radius property) is transformed into a square (with an sside property denoting the length

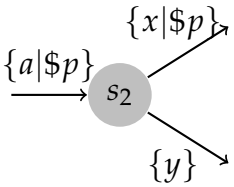
of a side) and a square is transformed into a equilateral triangle (with a tside property denoting the length of a side). The values of x and y define the shape into which the service transforms in the particular context.

Basically speaking, the use of Boolean variables is a compact way of representing *intersection types* [Pie97], which increase the expressiveness of function signatures: the type $(a \rightarrow c) \wedge (b \rightarrow d)$ maps particular input types to particular output types in an overloaded function. Similarly, in the example in Figure 2.4, each element of the input interfaces may be mapped to each element in the output interface: an element bike is present in the output interface for Bicycle Shop service only if the comp element is present in one of its input interfaces, and an element acc is present in the service's output interface only if acc is present in another input interface. In general case, the intersection type that specifies all possible mappings can be too excessive. Each service is allowed to have multiple input and output ports, each with its own interface, which increases the relation complexity between interface elements.

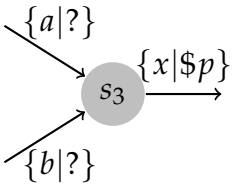
To sum up, configurable service composition is essential for service orchestration. In the next section we introduce the Message Definition Language: an interface specification language with support for subtyping, polymorphism, flow inheritance and dependable interfaces, the last supported via Boolean variables.



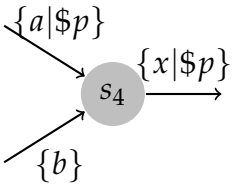
(a) A service with multiple output channels. Input data is propagated to all output channels



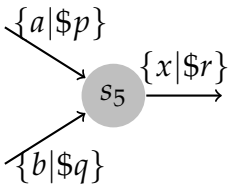
(b) A service with multiple output channels. Input data is propagated to some of output channels



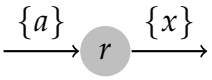
(c) A service with multiple input channels. Data is inherited from various channels. A problem of multiple inheritance has to be solved



(d) A service with multiple input channels. The input data is inherited from only one channel



(e) A service with multiple input channels. The input data is inherited from all input channels if some elements in $\$r$ are present in $\$p$ and the rest in $\$q$



(f) A service with accumulating facility. Inheritance must be disabled

Figure 2.3: Examples of MIMO services with support for flow inheritance

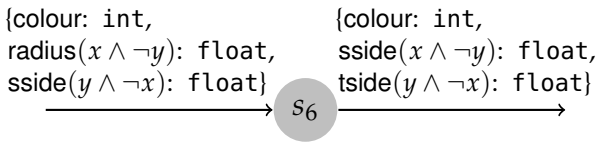


Figure 2.4: Interfaces with support for logical expressions

Chapter 3

Interface Definition Language for Service-Based Application

In this chapter we define a term algebra called Message Definition Language (MDL). The purpose of the MDL is to describe flexible service interfaces. WSDL [CCM+01] 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, which is introduced in Chapter 4. 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, which is associated with the given combinator.

In our approach, a message is a collection of data entities, each specified by a corresponding *term*. The intention of the term is to represent

1. a standard atomic type such as `int`, `string`, etc;
2. inextensible data collections such as tuples;
3. extensible data records [GJ96, Lei05], where additional named fields can be introduced without breaking the match between the producer and the consumer and where fields can also be inherited from input to output records by lowering the output type, which is always safe;
4. data-record variants, where in general 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 — hence contravariance — again, by raising the input type, which is also always safe.

3.1 Terms

Each term is either atomic or a collection in its own right. Atomic terms are *symbols*, which are identifiers used to represent standard types such as `int`, `string`, etc. To account for subtyping[1] we include three categories of collections: *tuples* that are demanded 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).

We introduce Boolean variables (called *b-variables* below) in the term interfaces for the following purposes: 1) B-variables allow to define configurable interfaces; 2) B-variables are used to specify dependencies between input and output data formats. They provide functionality similar to intersection types [Pie97], which increase the expressiveness of function signatures.

A Boolean expression $g \in \mathcal{B}$ ($\mathcal{B}$ denotes a set of Boolean expressions) called a guard is defined by the following grammar:

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

Selection of the Boolean operators (conjunction, disjunction, implication and negation) is justified by the fact that 1) an arbitrary Boolean formula can be present in a term; 2) implication is a common operation that is used in Boolean constraints (see Figures 7.1 and 7.2), so for brevity we introduce it as a separate operator.

In order to support parametric polymorphism and inheritance in interfaces, we introduce term variables (called later *t-variables*), which are similar to type variables. For coercion of interfaces it is important to distinguish between two variable categories: *down-coerced* and *up-coerced* ones. The former can be instantiated with symbols, tuples and records (terms of these three categories are called down-coerced terms), and the latter can only be instantiated with choices (up-coerced terms). 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. We use the notation $v^{\downarrow}$ and $v^{\uparrow}$ for down-coerced and up-coerced variables respectively, and v when its coercion sort is unimportant. Explicit sort annotation on variables is useful for simplifying partial order definitions on terms.

[1] From now on we use 'subtyping' to refer to a relation on algebraic terms and not to subtyping on types that is defined in the previous chapter.

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

⟨*term*⟩ ::= ⟨*symbol*⟩ | ⟨*tuple*⟩ | ⟨*record*⟩ | ⟨*choice*⟩ | t-variable

⟨*tuple*⟩ ::= **(** ⟨*term*⟩ [⟨*term*⟩]* **)**

⟨*record*⟩ ::= **{**[⟨*element*⟩[**,** ⟨*element*⟩]*[**|**down-coerced t-variable]]**}**

⟨*choice*⟩ ::= **(:**[⟨*element*⟩[**,** ⟨*element*⟩]*[**|**up-coerced t-variable]]**:)**

⟨*element*⟩ ::= ⟨*label*⟩ **(** ⟨*guard*⟩ **):** ⟨*term*⟩

⟨*label*⟩ ::= ⟨*symbol*⟩

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, which are unique within a single record. A *choice* is a collection of alternative terms. The syntax of choice is the same as that of record except for the delimiters. The difference between records and choices is in width subtyping [BBDCL97] and will become clear below when we define seniority on terms. Briefly, the subtyping on records is defined that a record with *more* elements is a subtype of one with *fewer* elements, and, similarly, a choice with *fewer* elements is a subtype of a choice with *more* elements.

We use choices to represent polymorphic messages and service interfaces at the top level. Records and choices are defined in *tail form*. 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 labels $\mathsf{l}_i : \mathsf{l}_i \neq \mathsf{l}_1, \ldots \mathsf{l}_i \neq \mathsf{l}_d$.

We extend the basic set of the MDL terms with a switch term. The switch is an auxiliary construct intended for building conditional terms, which is specified as a set of unlabelled (by contrast to a choice) guarded alternatives. Formally, it is defined as

⟨*switch*⟩ ::= **<**⟨*guard*⟩**:**⟨*term*⟩[**,** ⟨*guard*⟩**:**⟨*term*⟩]***>**

Exactly one guard must be true for any well-formed switch (see Definition 3.2.2), that is the switch is substitutionally equivalent to the term marked by the true guard:

$$\langle(\mathsf{false})\colon t_1, \ldots, (\mathsf{true})\colon t_i, \ldots, (\mathsf{false})\colon t_d\rangle = \langle(\mathsf{true})\colon t_i\rangle = t_i.$$

For example, $\langle(x)\colon \texttt{int}, (\neg x)\colon \texttt{string}\rangle$ represents the symbol `int` if $x = \mathsf{true}$, and the symbol `string` otherwise. A switch is a construct that allows to define an arbitrary

interface as a function of Boolean values. Furthermore, a switch is an essential construct for a constraint resolution algorithm, which is presented in Section 7.5.

3.2 Seniority Relation

For a guard g, we denote as $\mathsf{V}^b(g)$ the set of b-variables that occur in g. For a term t, we denote as $\mathsf{V}^{\uparrow}(t)$ the set of up-coerced t-variables that occur in t, and as $\mathsf{V}^{\downarrow}(t)$ the set of down-coerced ones; and finally $\mathsf{V}^b(t)$ is the set of b-variables in t.

Definition 3.2.1 (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, a record $\{\mathsf{a}\colon \mathtt{int}, \mathsf{b}\colon \mathtt{string}\}$ is a ground term, $\{\mathsf{a}\colon \mathtt{int}, \mathsf{b}(x)\colon \mathtt{string}\}$ is a semi-ground term, and $\{\mathsf{a}\colon \mathtt{int}, \mathsf{b}(x)\colon v_1 \mid v_2^{\downarrow}\}$ is neither ground nor semi-ground term.

Well-formedness for terms is defined next. It ensures that a term is not a switch with multiple true guards, neither the term contains such switch as a subterm.

Definition 3.2.2 (Well-formed terms)**.** A term t is well-formed if it is ground (i.e. does not contain variables) and exactly one of the following holds:

1. t is a symbol;
2. t is a tuple
$$(t_1 \dots t_d),$$
$d > 0$, where all t_i, $1 \le i \le d$, are well-formed;
3. t is a record
$$\{\mathsf{l}_1(g_1)\colon t_1, \dots, \mathsf{l}_\mathsf{d}(g_d)\colon t_d\}$$
or a choice
$$(\!:\mathsf{l}_1(g_1)\colon t_1, \dots, \mathsf{l}_\mathsf{d}(g_d)\colon t_d\!:),$$
$d \ge 0$, where for all $1 \le i \ne j \le d$, $g_i \wedge g_j \implies l_i \ne l_j$ and all t_i for which g_i are true are well-formed;
4. t is a switch
$$\langle (g_1)\colon t_1, \dots, (g_n)\colon t_d\rangle,$$
$d > 0$, where for some $1 \le i \le d$, $g_i = \mathsf{true}$ and t_i is well-formed and where $g_j = \mathsf{false}$ for all $j \ne i$.

If an element of a record, choice or switch has a guard that 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 an element of a record or a choice has a guard that is true, the guard can be syntactically omitted, for example,

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

We define the *canonical form* of a well-formed collection as a representation that does not include false guards, and we omit 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. Hereafter in the thesis we specify terms in the canonical form when it is possible.

Next we introduce a seniority relation $\sqsubseteq$ on terms for the purpose of structural subtyping. 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 $t' \sqsubseteq t$. In the sequel we use nil to denote the empty record $\{\ \}$, which has the meaning of unit type and represents a message without any data. Similarly, we use none to denote the empty choice $(:\ :)$. We use an empty choice for specifying service interfaces that cannot send or receive any messages.

Definition 3.2.3 (Seniority relation)**.** The seniority relation $\sqsubseteq$ on well-formed terms in canonical form is defined as follows:

1. $\mathsf{none} \sqsubseteq t$ if t is a choice;
2. $t \sqsubseteq \mathsf{nil}$ if t is a symbol, a tuple or a record;
3. $t \sqsubseteq t$;
4. $t_1 \sqsubseteq t_2$, 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 $t_1^i \sqsubseteq t_2^i$ for each $1 \le i \le d$;
   (b) $t_1 = \{\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_1^d\colon t_1^d\}$ and $t_2 = \{\mathsf{l}_2^1\colon t_2^1, \ldots, \mathsf{l}_2^e\colon t_2^e\}$, where $d \ge e$ and for each $j \le e$ there is $i \le d$ such that $l_1^i = l_2^j$ and $t_1^i \sqsubseteq t_2^j$;
   (c) $t_1 = (:\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_1^d\colon t_1^d:)$ and $t_2 = (:\mathsf{l}_2^1\colon t_2^1, \ldots, \mathsf{l}_2^e\colon t_2^e:)$, where $d \le e$ and for each $i \le d$ there is $j \le e$ such that $l_1^i = l_2^j$ and $t_1^i \sqsubseteq t_2^j$.

Similarly to the t-variables, terms are classified into two categories: symbols, tuples and records are down-coerced terms and choices are up-coerced terms. The seniority relation defines a symmetric relation on down-coerced and up-coerced terms: an element nil is the maximum element for down-coerced terms; on the other hand, none is the minimal element for up-coerced terms. $\mathcal{T}^{\downarrow}$ denotes the set of all down-coerced ground terms, $\mathcal{T}^{\uparrow}$ denotes the set of all up-coerced ground terms and $\mathcal{T} = \mathcal{T}^{\downarrow} \cup \mathcal{T}^{\uparrow}$ is the set of all ground terms. Similarly, $\mathcal{T}_m^{\downarrow}$ denotes the set of all vectors of down-coerced ground

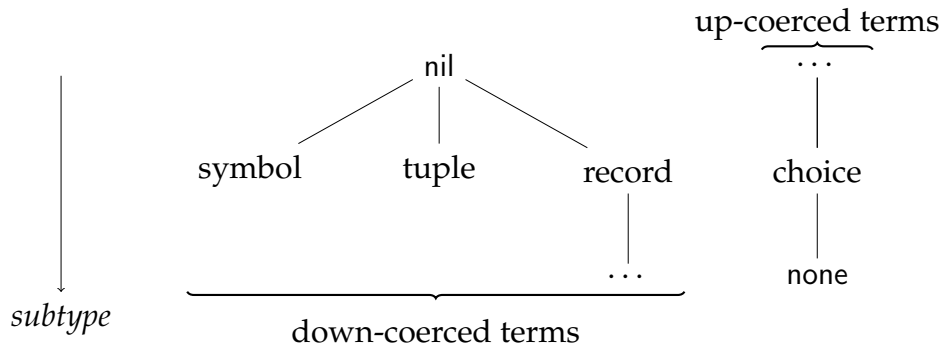


Figure 3.1: Two semilattices $(\mathcal{T}^{\downarrow}, \sqsubseteq)$ and $(\mathcal{T}^{\uparrow}, \sqsubseteq)$ representing the seniority relation for terms of different categories. The lower terms are the subtypes of the upper ones

terms of length m and $\mathcal{T}_n^{\uparrow}$ denotes the set of all vectors of up-coerced ground terms of length n.

If $\vec{t_1}$ and $\vec{t_2}$ are vectors of terms $(t_1^1, \ldots, t_d^1)$ and $(t_1^2, \ldots, t_d^2)$ of size d, then $\vec{t_1} \sqsubseteq \vec{t_2}$ denotes the seniority relation for all pairs $t_i^1 \sqsubseteq t_i^2$ $(1 \leq i \leq d)$.

Proposition 3.2.4. The seniority relation $\sqsubseteq$ is a partial order, and $(\mathcal{T}, \sqsubseteq)$ is a pair of meet and join semilattices (illustrated in Figure 3.1):

$$\forall t_1, t_2 \in \mathcal{T}^{\downarrow}, t_1 \sqsubseteq t_2 \text{ iff } t_1 \sqcap t_2 = t_1$$
$$\forall t_1, t_2 \in \mathcal{T}^{\uparrow}, t_1 \sqsubseteq t_2 \text{ iff } t_1 \sqcup t_2 = t_2.$$

Proposition 3.2.5. $(\mathcal{T}_m^{\downarrow}, \sqsubseteq)$ and $(\mathcal{T}_n^{\uparrow}, \sqsubseteq)$ is a pair of meet and join semilattices:

$$\forall \vec{t_1}, \vec{t_2} \in \mathcal{T}_m^{\downarrow}, \vec{t_1} \sqsubseteq \vec{t_2} \text{ iff } \vec{t_1} \sqcap \vec{t_2} = \vec{t_1}$$
$$\forall \vec{t_1}, \vec{t_2} \in \mathcal{T}_n^{\uparrow}, t_1 \sqsubseteq t_2 \text{ iff } \vec{t_1} \sqcup \vec{t_2} = \vec{t_2}.$$

Definition 3.2.6. A relation $=$ denotes an equality relation on terms and is defined as follows.

$$t_1 = t_2 \iff t_1 \sqsubseteq t_2 \wedge t_2 \sqsubseteq t_1 \tag{3.1}$$

In other words, $=$ is a reflexive, a symmetric relation and a transitive relation.

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 such ground term values for the t-variables and such Boolean values for the b-variables that the seniority relation holds represents a CSP problem, which is formally introduced in Chapter 5.

3.3 Configuration Parameters

We introduce b-variables to the MDL as interface configuration parameters that enable or disable some functionality provided by a service within the context. In Chapter 8 we define a fixed interface format for a service: in each input interface the service exposes the name of and arguments for a function that processes input messages from a given port. Depending on the context, the clients use a subset of the provided functions. In our approach, clients are allowed to use the functions if they are connected by statically-defined channels to the corresponding service ports. Our interface configuration protocol (see Chapter 8) automatically disables unused functions by setting the b-variables in the interface MDL term to false. In this thesis we present an algorithm (see Chapter 7) finds processing functions in the application which are never triggered in the given context.

Our mechanism shows that *a processing function is exposed in a configured service if and only if it can provide data that is demanded by other services*. Otherwise, any inability to provide the required data leads to communication errors. In this mechanism the b-variables are used as follows: we assume that service interfaces are represented as a choice term at the top level (see Chapter 8, where we discuss the format of interfaces for services). Each element of a choice in the input interface corresponds to a service functionality; similarly, each element of a choice in the output interface corresponds to an output message of a particular kind.

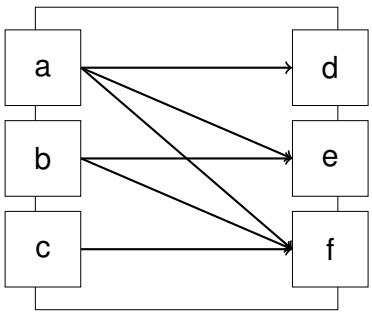


Figure 3.2: Relation between input message formats and output message formats. d is produced in response to a, e is produce in response to a and b, and f is produced in response to a, b and c

Consider an example in Figure 3.2. The example illustrates a service that can accept messages of three kinds: a, b and c, each of them corresponding to some functionality of the service. Similarly, the service produces messages of three kinds: d is produced as a response to a, e is produced as a response to a or b, and f is produced as a response to a, b or c.

In the existing IDLs such as WSDL it is impossible to maintain a relation between input and output interfaces. In these IDLs the interfaces are local and, therefore, it is

impossible to verify correspondence between interfaces globally, taking a topology of a complete application into account.

In our approach, we use b-variables to maintain a relation between interfaces. The input interface of the service in Figure 3.2 is defined by the following choice term:

$$(:\mathsf{a}(x):\ldots,\mathsf{b}(y):\ldots,\mathsf{c}(z):\ldots:),$$

where x, y and z are guards associated with a, b and c. Similarly, the output interface is defined by the choice term

$$(:\mathsf{d}(x):\ldots,\mathsf{e}(x\vee y):\ldots,\mathsf{f}(x\vee y\vee z):\ldots:).$$

The guards in the output interface specify conditions for producing output messages. A message with the tag e can be produced if and only if messages with tags a and b can be received. As a result, the mechanism removes e from the output interface if and only if messages with the tag a and b can never be received.

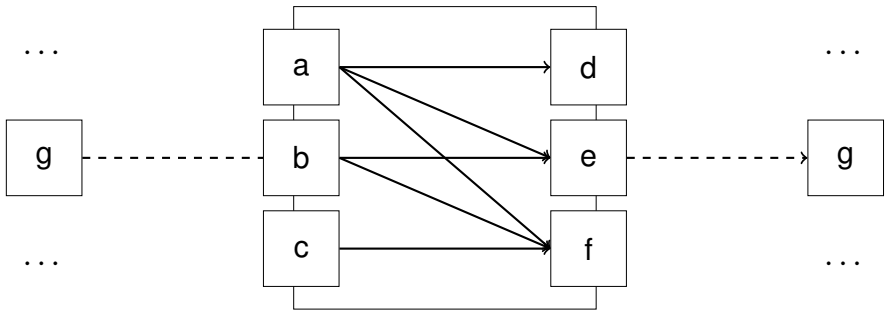
3.4 Flow Inheritance

Web services are experiencing a transition from batch to stream processing [ABB+13]. Under stream processing, services often form long pipelines, in which a service processes only part of its input message, with the rest of it being passed down the pipeline without modification. A mechanism that implicitly redirects part of a message from the input to the output of a service is called *flow inheritance* [GSS10]. Although existing technologies for service composition, such as session types, enhance the flexibility and the reuse of services with subtyping and polymorphism, they lack the analysis and configuration capacity for flow inheritance (the advantages of flow inheritance are explained in Chapter 8).

Flow inheritance is enabled using tail variables, which are present in a record and a choice term. Consider the interface format introduced in Section 3.3, in which the interfaces are represented as choice terms at the top level. By inserting the same tail variable in both input and output interface, we enable the support for flow inheritance at the interface level.

Consider the example in Figure 3.3. The input interface of the service is a choice that contains elements labelled a, b, c and a tail variable $v^{\uparrow}$. The output interface of the service contains elements labelled d, e, f and the tail variable $v^{\uparrow}$. Assume that the service is a consumer for a service that provides g. Since g cannot be processed by the service, g can

be automatically bypassed to the output using flow inheritance.



$$(:\mathsf{a}(x){:}\ldots,\mathsf{b}(y){:}\ldots,\mathsf{c}(z){:}\ldots\,|v^{\uparrow}:) \qquad (:\mathsf{d}(x){:}\ldots,\mathsf{e}(x\vee y){:}\ldots,\mathsf{f}(x\vee y\vee z){:}\ldots\,|v^{\uparrow}:)$$
$$v^{\uparrow} = (:\mathsf{g}{:}\ldots:)$$

Figure 3.3: The service that automatically bypasses a message with the label g using flow inheritance

Two seniority constraints follow from the interconnection:

$$(:\ldots,\mathsf{g}{:}\ldots,\ldots:) \sqsubseteq (:\mathsf{a}(x){:}\ldots,\mathsf{b}(y){:}\ldots,\mathsf{c}(z){:}\ldots\,|v^{\uparrow}:)$$
$$(:\mathsf{d}(x){:}\ldots,\mathsf{e}(x\vee y){:}\ldots,\mathsf{f}(x\vee y\vee z){:}\ldots\,|v^{\uparrow}:) \sqsubseteq (:\ldots,\mathsf{g}{:}\ldots,\ldots:).$$

The first constraint constraint is satisfied only if $(:\mathsf{g}{:}\ldots:) \sqsubseteq v^{\uparrow}$. In other words, $v^{\uparrow}$ must contain an element with the label g. Given this, the second constraint is automatically satisfied, because the left term contains g, which is required by the right term.

In Chapter 8 we describe flow inheritance provided by the interface configuration mechanism.

3.5 Multiple Flow Inheritance

In Chapter 9 we discuss the message synchronisation mechanism, which allows messages received from multiple input ports to be merged and merged messages to be sent to the output. It is impossible to support such behaviour in the interfaces using only the MDL terms described previously. Specifically, given that record terms represent message interfaces, we want to be able to define the output interface as a record that contains all elements from the input interfaces (provided that element labels are disjointed). We call such behaviour *multiple flow inheritance*.

Consider the following example. Let input messages be specified by two records $\{\mathsf{a}\colon \mathtt{int}, \mathsf{b}\colon \mathtt{bool}\}$ and $\{\mathsf{b}\colon \mathtt{int}, \mathsf{c}\colon \mathtt{int}\}$. A message that is composed by merging these records is ambiguously specified by $\{\mathsf{a}\colon \mathtt{int}, \mathsf{b}\colon \mathtt{bool}, \mathsf{c}\colon \mathtt{int}\}$ or $\{\mathsf{a}\colon \mathtt{int}, \mathsf{b}\colon \mathtt{int}, \mathsf{c}\colon \mathtt{int}\}$,

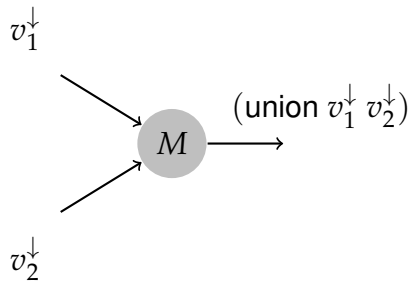


Figure 3.4: A simple merger

depending on the context. Consequently, merging records that contain elements with the same label is something that must be forbidden.

We introduce a special term called a union. This represents a record that is constructed from two records. The algorithm presented in Section 7.6 guarantees that: 1) the union record contains all elements from both records; and 2) no two elements with the same label are present in records that constitute the union term.

In order to illustrate multiple flow inheritance, we discuss the following example. Consider a simple service called a *merger*. It receives two records from one or two input channels, merges them, and sends the merged record as the output message.

The input interfaces of the merger are arbitrary records represented by down-coerced variables $v_1^{\downarrow}$ and $v_2^{\downarrow}$. The output interface of the merger is defined by the union term (union $v_1^{\downarrow}$ $v_2^{\downarrow}$) (see Figure 3.4).[2]

(union $v_1^{\downarrow}$ $v_2^{\downarrow}$) must be replaced by a record that contains all elements from records $v_1^{\downarrow}$ and $v_2^{\downarrow}$, but not from both. If $v_1^{\downarrow}$ and $v_2^{\downarrow}$ contain elements with the same labels under all instantiations of b-variables, then it is impossible to merge records unambiguously.

The semantics of the union is defined as follows. Let the union be a record of the form $\{\tilde{\mathsf{l}}_1(\tilde{g}_1)\colon \tilde{t}_1, \ldots, \tilde{\mathsf{l}}_d(\tilde{g}_d)\colon \tilde{t}_d\}$. Then

$$(\text{union } v_1^{\downarrow}\ v_2^{\downarrow}) \iff \begin{cases} v_1^{\downarrow} = & \{\tilde{\mathsf{l}}_1(\tilde{g}_1 \wedge g_1')\colon t_1, \ldots, \tilde{\mathsf{l}}_d(\tilde{g}_d \wedge g_d')\colon \tilde{v}_d\} \\ v_2^{\downarrow} = & \{\tilde{\mathsf{l}}_1(\tilde{g}_1 \wedge \neg g_1')\colon t_2, \ldots, \tilde{\mathsf{l}}_d(\tilde{g}_d \wedge \neg g_d')\colon \tilde{v}_d\}, \end{cases} \tag{3.2}$$

where $\iff$ is a logical equivalence and $g_1', \ldots, g_d'$ are generated free b-variables. In other words, the union is obtained by combining disjoint elements from $v_1^{\downarrow}$ and $v_2^{\downarrow}$. It is important that no elements can be present in both $v_1^{\downarrow}$ and $v_2^{\downarrow}$. Presence of b-variables $g_1', \ldots, g_d'$ forces elements with the same label to not be present in both $v_1^{\downarrow}$ and $v_2^{\downarrow}$.

[2] The union term has no relation to a union data type in C language.

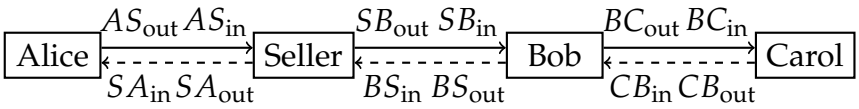


Figure 3.5: Service composition in a Three Buyer use case

3.6 Motivating Example: Three Buyer Use Case

Our approach for configuring web services is motivated by rapid development of Cloud computing, social networks and Internet of Things, which accelerate the growth and complexity of service choreographies [BSD13, DYV12]. Accordingly, we chose a simple but non-trivial example from one of those areas to illustrate our approach. The same 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 [HYC08].

Consider a system involving buyers called Alice, Bob and Carol that cooperate in order to buy a book from a Seller. Each buyer is specified as an independent service that is connected with other services via a channel-based communication. 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. The interfaces are defined in the MDL. Figure 3.5 depicts composition of the application where Alice is connected to Seller only and can interact with Bob and Carol indirectly. AS, SB, BC, CB, BS, SA denote interfaces that are associated with service input/output ports. For brevity, we only provide AS, SB and BC (the rest of the interfaces are defined in the same manner), which are specified in the MDL in the following way:

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

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

Collection elements contain guards x, y and z. A guard instantiated to false excludes the element from the collection. This is the main 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 that are connected by a communication channel 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.

AS_{out} declares an output interface of Alice, which declares functionality and a 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 a title of type string is only acceptable, which means that $tv^{\downarrow}$ must be instantiated to string.
- Furthermore, Alice can provide a payment for a book. In addition to the title and the required amount of money, Alice provides her id in the message. Although 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 (the example illustrates a composition where a service communicates with services that the service is not directly connected to), our mechanism detects that Alice can send a message with 'share' label to Bob by bypassing it implicitly through Seller. In order to enable flow inheritance in Seller's service, the mechanism sets a tail variable $ct1^{\uparrow}$ to $(:\text{share: } \{\text{title: string}, \text{money: int}\}:)$. 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 configuration mechanism proposed in this thesis analyses the interfaces of services Seller, Bob and Carol in the same manner. The presence of $ct1^{\uparrow}$ variable in both input and output interfaces of Bob enables support for data inheritance on the interface level. Furthermore, the Boolean variable z behaves as an intersection type: Bob

has 'purchase sharing' functionality declared as an element share(z): $\{\dots\}$ in its input interface SB_{in} (used by Seller). The element is related to the element 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. In our example, z is true, because Carol declares that it can receive messages with the label 'share'. Note that there could be any Boolean formula in place of z, which wires any input and output interfaces of a single service in an arbitrary way. The existing interface description languages (WSDL, WS-CDL, etc.) do not support such interface wiring capabilities.

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

The problem being solved is similar to type inference problem; however, it has large combinatorial complexity and, therefore, direct search of a solution is impractical. Furthermore, additional complexity 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. In our approach, we define our problem as a constraint satisfaction problem. Then we employ a constraint solver, which was specifically developed to solve this problem, to find correct instantiations of the variables.

Chapter 4

Description of Service-Based Application in Language of Combinators

Definition of service composition using the 'algebraic' style facilitates formal reasoning about the services [Fer04, SBS06]. 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 see a *service* as a module that contains functions for input message processing. The service has a set of named *input* and *output ports*. The former are the ports where the incoming messages arrive; the latter are the ports where services send produced messages to. The ports are required for service functional decomposition: for example, messages that relate to error tracing, logging, authorisation can naturally be read from and sent to different ports.

Each port is a pair (l_p, t_p), where l_p is the name of the port and $t_p \in \mathcal{T}$ is the MDL term (see Chapter 3) specifying the service interface associated with the port.

With each service s we associate the function $\tau(s) = (\mathcal{I}_s, \mathcal{O}_s)$, where

$$\mathcal{I}_s = \{(l_p, t_p) \mid 1 \leq p \leq I_s\},$$
$$\mathcal{O}_s = \{(l_p, t_p) \mid 1 \leq p \leq O_s\},$$

and I_s and O_s are the number of s's input and output ports, respectively. The tuple $\tau(s)$ represents the service properties related to its interface. They can either be provided explicitly or, as we demonstrate in Chapter 8, automatically be derived from the service.

4.1 Wiring

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

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

A label is a basic building block of a network, which represents a single service. The label is unique name of the service. **..** , || , \ are called *wiring patterns*.

.. 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. The result of the wiring is that the input (output) ports of one service are combined with the input (output) ports of another service.

\ denotes the wrap-around connection for a network. It creates a cycling connection by wiring output ports to input ports in the network.

Furthermore, parenthesis are introduced for grouping subnetworks together. For example, (`A` **..** `B`) || `C` and `A` **..** (`B` || `C`) specify different networks due to different subnetwork grouping.

4.2 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

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

where

$$\mathcal{I}_N = \{(l_p, t_p) \mid 1 \leq p \leq I_N\},$$
$$\mathcal{O}_N = \{(l_p, t_p) \mid 1 \leq p \leq O_N\},$$

and I_N and O_N are the number of N's input and output ports, respectively.

$\mathcal{C}$ is a set of the seniority constraints, which guarantees communication safety. Each constraint is a relation

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

on the wired ports p and p'. The relation $\sqsubseteq$ is the seniority relation, which is defined in Definition 3.2.3. It specifies conditions on the service interface wired with a communication channel.

4.2.1 Typing Rules

The four typing rules, which we propose in this section, specify a mechanism for aggregating the wiring constraints from the network.

$$(\text{SING})\ \frac{\mathsf{L}}{\mathsf{L}\colon (\mathcal{I}_{\mathsf{L}}, \mathcal{O}_{\mathsf{L}}, \emptyset)}$$

is a typing rule for a single service also referred as a *singleton network*, where L is a service label and $(\mathcal{I}_{\mathsf{L}}, \mathcal{O}_{\mathsf{L}}) = \tau(\mathsf{L})$. Sets of port descriptions (port names and the MDL term associated with the port) $\mathcal{I}_{\mathsf{L}}$ and $\mathcal{O}_{\mathsf{L}}$ can be specified in a separate service configuration file or, as we demonstrate in Chapter 8, can automatically be derived from the service.

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

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

where

$$\begin{aligned}
\mathcal{I}' &= \mathcal{I}_{N_1} \cup \{(l_p, t_p) \mid (l_p, t_p) \in \mathcal{I}_{N_2} \nexists (l_{p'}, t_{p'}) \in \mathcal{O}_{N_1} : l_p = l_{p'}\} \\
\mathcal{O}' &= \mathcal{O}_{N_2} \cup \{(l_p, t_p) \mid (l_p, t_p) \in \mathcal{O}_{N_1} \nexists (l_{p'}, t_{p'}) \in \mathcal{I}_{N_2} : l_p = l_{p'}\} \\
\mathcal{C}' &= \mathcal{C}_{N_1} \cup \mathcal{C}_{N_2} \cup \{t_p \sqsubseteq t_{p'} \mid \exists l_p : (l_p, t_p) \in \mathcal{O}_{N_1}, \exists l_{p'} : (l_{p'}, t_{p'}) \in \mathcal{I}_{N_2} \wedge l_p = l_{p'}\}.
\end{aligned}$$

The constructed type is a tuple that consists of N_1's input ports, N_2's output ports. Furthermore, the seniority constraints are constructed as a union of $\mathcal{C}_{N_1}$, $\mathcal{C}_{N_2}$ and constraints that represent data relations on newly constructed channels.

$l_p = l_{p'}$ is called an *identity condition*: the channels always wire the identically named ports. Consequently, the wiring of the services in the network depends on service port names. On the other hand, 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. Consequently, the wiring relation is completely generic: it can lead to one-to-one, one-to-many, many-to-one or many-to-many connections.

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

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

The constructed type is a tuple that consists of a union of N_1's and N_2's input ports, a union of N_1's and N_2's output ports and a union of sets $\mathcal{C}_{N_1} \cup \mathcal{C}_{N_2}$. 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, $\mathcal{I}_{N_1}$ and $\mathcal{I}_{N_2}$, as well as $\mathcal{O}_{N_1}$ and $\mathcal{O}_{N_2}$ may contain ports with the same name without having compatibility issues. Consider the following example.

Example 4.2.1. Assume a network

$$N_1 \,\bullet\!\bullet\, (N_2 \,||\, N_3)$$

is given, where N_1, N_2 and N_3 are the following services:

$$\begin{aligned} N_1 &: (\{\}, \{(a : t_1)\}, \emptyset) \\ N_2 &: (\{(a : t_2)\}, \{\}, \emptyset) \\ N_3 &: (\{(a : t_3)\}, \{\}, \emptyset). \end{aligned}$$

Having applied the parallel connection to the services N_2 and N_3, a network

$$N_2 \,||\, N_3 : (\{(a : t_2, a : t_3)\}, \{\}, \emptyset),$$

which contains ports with the same name, is obtained. The serial connection produces a set of two seniority constraints as part of the network type: $\mathcal{C} = \{t_1 \sqsubseteq t_2, t_1 \sqsubseteq t_3\}$. The constraints are satisfied only when a join term for t_2 and t_3 exists. Similarly, if ports with the same name are present in the output interface, a meet term must exist. Essentially, this demonstrates how one-to-many and many-to-one connections are implemented. $\triangle$

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

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

where

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

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 $\mathcal{I}_N$ and $\mathcal{O}_N$.

Although a service network may have various representations in the language of combinators, the type that corresponds to the service network is unique and does not depend on the representation. This is due to equivalence of ($\mathbf{..}$) and ($\backslash$) rules in cyclic networks. For instance, $(N_1 \mathbf{..} N_2)\backslash$ and $(N_2 \mathbf{..} N_1)\backslash$ represent the same network and application of the rule ($\backslash$) to $N_1 \mathbf{..} N_2$ derives the same type as application of the rule ($\backslash$) to $(N_2 \mathbf{..} N_1)\backslash$.

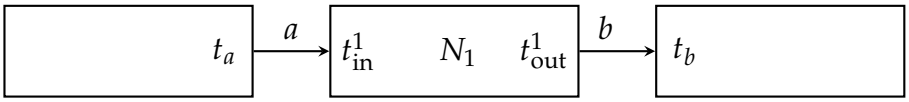
The type of the top-level network contains a set of communication constraints that serves as an input for the CSP-WS problem presented in Chapter 5. The network topology is safe for communication if the constraints can be satisfied and unsafe otherwise.

4.3 Subtyping

Next we introduce subtyping on types $(\mathcal{I}, \mathcal{O}, \mathcal{C})$. It defines the hierarchy of networks and hierarchy of partial solutions to the constraints $\mathcal{C}$. Intuitively, a network $N_2 : (\mathcal{I}_{N_2}, \mathcal{O}_{N_2}, \mathcal{C}_{N_2})$ is a supertype of $N_1 : (\mathcal{I}_{N_1}, \mathcal{O}_{N_1}, \mathcal{C}_{N_1})$ if N_1 is less generic network. Specifically, the sets $\mathcal{I}_{N_1}$ and $\mathcal{O}_{N_1}$ may contain more ports than the sets $\mathcal{I}_{N_2}$ and $\mathcal{O}_{N_2}$ and a number of solutions to $\mathcal{C}_{N_1}$ is less than a number of solutions to $\mathcal{C}_{N_2}$. Furthermore, for input ports with the same labels (l_p, t^1_p) and (l_p, t^2_p) in $\mathcal{I}_{N_1}$ and $\mathcal{I}_{N_2}$ 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 $\mathcal{O}_{N_1}$ and $\mathcal{O}_{N_2}$ respectively, $\widetilde{t}^2_p$ must be a subtype of $\widetilde{t}^1_p$. We illustrate this property in Figure 4.1.

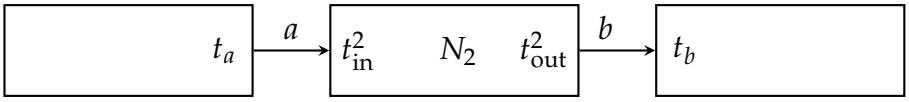
The formal rule for subtyping is

$$\text{(S-SUB)}\ \frac{\begin{array}{c}\forall (l^2_p, t^2_p) \in \mathcal{I}_{N_2} \exists (l^1_p, t^1_p) \in \mathcal{I}_{N_1} : l^1_p = l^2_p \wedge t^1_p \sqsubseteq t^2_p \\ \forall (l^2_p, t^2_p) \in \mathcal{O}_{N_2} \exists (l^1_p, t^1_p) \in \mathcal{O}_{N_1} : l^1_p = l^2_p \wedge t^2_p \sqsubseteq t^1_p \\ \mathcal{C}_{N_1} \neq \bot \qquad V(\mathcal{C}_{N_1}) \supset V(\mathcal{C}_{N_2}) \vee \mathcal{C}_{N_1} \equiv \mathcal{C}_{N_2} \qquad \mathcal{C}_{N_1} \models \mathcal{C}_{N_2}\end{array}}{(\mathcal{I}_{N_1}, \mathcal{O}_{N_2}, \mathcal{C}_{N_1}) \leq (\mathcal{I}_{N_1}, \mathcal{O}_{N_2}, \mathcal{C}_{N_2})}$$



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

(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. As a result, the constraints $t_a \sqsubseteq t^1_{\text{in}}$ and $t^1_{\text{out}} \sqsubseteq t_b$ are produced



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

(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. As a result, the constraints $t_a \sqsubseteq t^2_{\text{in}}$ and $t^2_{\text{out}} \sqsubseteq t_b$ are produced

Figure 4.1: 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 $t_a \sqsubseteq t^1_{\text{in}}$ and $t^1_{\text{out}} \sqsubseteq t_b$ implies $t^2_{\text{out}} \sqsubseteq t_b$

Here $V(\mathcal{C})$ denotes the set of all variables that occur in $\mathcal{C}$. $\mathcal{C}_{N_1} \equiv \mathcal{C}_{N_2}$ declares that $\mathcal{C}_{N_2}$ is a satisfiable set of constraints if and only if $\mathcal{C}_{N_1}$ is a satisfiable set of constraints too. $\models$ is a logical entailment, which defines the following relation: if $\mathcal{C}_{N_1} \equiv \mathcal{C}_{N_2}$, then the solution to $\mathcal{C}_{N_1}$ is also a solution to $\mathcal{C}_{N_2}$. The subtyping relation is reflexive, transitive and antisymmetric. There exists a unique type $(\mathcal{I}_{\text{top}}, \mathcal{O}_{\text{top}}, \mathcal{C}_{\text{top}}) = (\emptyset, \emptyset, \emptyset)$ such that $(\mathcal{I}, \mathcal{O}, \mathcal{C}) \leq (\mathcal{I}_{\text{top}}, \mathcal{O}_{\text{top}}, \mathcal{C}_{\text{top}})$ for any $(\mathcal{I}, \mathcal{O}, \mathcal{C})$, where $\mathcal{C} = \emptyset$ denotes a set of tautological constraints. It leads us to the fact that subtyping relation is a semilattice with $\mathcal{C}_{\text{top}}$ as the top element.

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

Assume that we are looking for a solution to $\mathcal{C}_{N_2}$. The *tightest* solution is a vector of values to variables $V(\mathcal{C}_{N_2})$ such that the constraints $\mathcal{C}_{N_2}$ are satisfied and the vector is not a solution to $\mathcal{C}_{N_1}$, where N_1 is any subtype of N_2 and $\mathcal{C}_{N_1} \neq \mathcal{C}_{N_2}$. A typical set of constraints may have infinitely many solutions. In Chapters 6 and 7 we present an algorithm that always finds a tight solution, which may not be unique though.

4.4 Arbitrary Topology

The three wiring patterns defined above are sufficient to specify an arbitrary topology of the network. The algorithm that transforms the network N into an algebraic expression

that uses the three combinators as operators and names of services as operands is the following:

1. If N is a cyclic graph, find an acyclic subgraph N' [Len73, BS90].[1] The edges that are removed from the cyclic graph form the *feedback edge set*. Break the edges and replace them by a pair of identically named ports in the network, an input port p and an output port p. Include the input port p to the set of input portss and the output port p to the set of output ports. Continue breaking the edges until the graph becomes acyclic.
2. The graph is acyclic now, therefore, there are edges that do not have incoming edges. Call them root vertices. Introduce a function

$$\gamma : \mathcal{V} \to \mathbb{N},$$

   such that for any $v \in \mathcal{V}$ ($\mathcal{V}$ is a set of graph vertices), $\gamma(v)$ is the length of the longest path from a graph input to v. Assuming that the graph is connected, such path must exist. The function can be constructed by sorting graph vertices [Kah62] topologically. Tag each vertex with its value of γ. Finally, remove the vertices v_{in} and v_{out}. The result is a set of vertices tagged with values from γ.
3. Construct specification in the language of combinators as follows:

$$\left((v_1^{[0]} \mid\mid v_2^{[0]} \mid\mid \ldots \mid\mid v_{k_0}^{[0]}) \bullet\bullet (v_1^{[1]} \mid\mid \ldots \mid\mid v_{k_1}^{[1]}) \bullet\bullet \ldots \bullet\bullet (v_1^{[d]} \mid\mid v_2^{[d]} \mid\mid \ldots \mid\mid v_{k_d}^{[d]})\right) \setminus,$$

   where $v_i^{[j]}$ are the vertices that have the tag j. Here we assume that a vertex represents the name of a particular service.

The algorithm breaks ties arbitrarily in the first step, which means that representation of the service network in the language of combinators may not be unique.

Example 4.4.1. Consider the service-based application in Figure 2.2b. 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
```

[1] In our work it is not important whether the subgraph is a maximum acyclic subgraph or not. However, in order to obtain a simple and clear algebraic representation of the network, the minimal number of wrap-around edges should be preferred.

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:

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

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 (`Comp`, `Acc`, `Shop` and `Cust` are used as shorthands for `Components`, `Accessories`, `Bicycle Shop` and `Customer` services, respectively):

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{Comp}}{\texttt{Comp} : (\{\}, \{a : t_{a_{\text{out}}}\}, \emptyset)} \quad \dfrac{\texttt{Acc}}{\texttt{Acc} : (\{\}, \{b : t_{b_{\text{out}}}\}, \emptyset)}}{\texttt{Comp} \parallel \texttt{Acc} : (\{\}, \{a : t_{a_{\text{out}}}, b : t_{b_{\text{out}}}\}, \emptyset)} \quad \dfrac{\texttt{Shop}}{\texttt{Shop} : (\{a : t_{a_{\text{in}}}, b : t_{b_{\text{in}}}\}, \{c : t_{c_{\text{out}}}\}, \emptyset)}}{(\texttt{Comp} \parallel \texttt{Acc}) \mathbin{..} \texttt{Shop} : (\{\}, \{c : t_{c_{\text{out}}}\}, \{t_{a_{\text{out}}} \sqsubseteq t_{a_{\text{in}}}, t_{b_{\text{out}}} \sqsubseteq t_{b_{\text{in}}}\})} \quad \dfrac{\texttt{Cust}}{\texttt{Cust} : (\{c : t_{c_{\text{in}}}\}, \{\}, \emptyset)}}{((\texttt{Comp} \parallel \texttt{Acc}) \mathbin{..} \texttt{Shop}) \mathbin{..} \texttt{Cust} : (\{\}, \{\}, \{t_{a_{\text{out}}} \sqsubseteq t_{a_{\text{in}}}, t_{b_{\text{out}}} \sqsubseteq t_{b_{\text{in}}}, t_{c_{\text{out}}} \sqsubseteq t_{c_{\text{in}}}\})}$$

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

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

△

4.5 Subnetworks

The language that specifies service networks is recursively closed. Indeed, a network that is constructed using the combinators has the same form as the networks that it is

composed from. Using this property we can naturally introduce an encapsulation mechanism to the service networks. Not only individual services can be used for constructing a network, but also networks can be used for constructing more complex networks.

Furthermore, when a network designer uses combinators to construct a new network, they do not need to be aware of structure of the networks that are used as building blocks. As a result, different programmers can develop various subnetworks without interfering their concerns.

Also, a runtime job scheduler can benefit from hierarchical structuring of the network while it maps subnetworks to the nodes of a distributed platform. There has been some research on topology-aware communication on distributed platforms [RGB+11, SPK+12, HS11]. Typically, the scheduler uses various heuristics for allocating tasks that belong to one subnetwork to a single core. As a result, time and memory locality are improved.

Providing a mechanism for supporting subnetworks in the language of combinators is not a topic of particular interest in this thesis. On the other hand, those who are interested in the language extension can be referred to S-Net language [GSS10], which provides support for hierarchical networks. Hierarchical networks in the language of combinators can be supported in the same way.

Chapter 5

Constraint Satisfaction Problem for Web Services

We defined the language of combinators that specifies the service network in an algebraic form. Furthermore, we defined a type system on top of the language. The type of a network contains a set of communication constraints (one constraint for each communication channel). The constraints must ensure the compatibility of data formats. In this chapter we formally introduce a constraint satisfaction problem for web services (CSP-WS), which is the main contribution of this thesis. The algorithm for solving the CSP-WS is discussed in the following sections.

5.1 CSP-WS Definition

The constraint is a relation on the MDL terms. We define a substitution as a syntactic transformation that replaces b-variables with Boolean values and t-variables with ground or semi-ground terms in the MDL terms.

Definition 5.1.1 (Substitution)**.** Let g be a guard, t be a term, $k = |\mathsf{V}^b(g) \cup \mathsf{V}^b(t)|$, and $\vec{f} = (f_1, \ldots, f_k)$ be a vector of b-variables contained in g and t, and $\vec{v} = (v_1, \ldots, v_k)$ be a vector of term variables contained in t. Then for any vector of Boolean values $\vec{b} = (b_1, \ldots, b_k)$ and a vector of terms $\vec{s} = (s_1, \ldots, s_k)$

1. $g[\vec{f}/\vec{b}]$ denotes a Boolean value (true or false), which is obtained as a result of the simultaneous replacement and evaluation of f_i with b_i for each $1 \leq i \leq k$;
2. $t[\vec{f}/\vec{b}]$ denotes the vector obtained as a result of the simultaneous replacement of f_i with b_i for each $1 \leq i \leq k$;
3. $t[\vec{v}/\vec{s}]$ denotes the vector obtained as a result of the simultaneous replacement of v_i with s_i for each $1 \leq i \leq k$;

4. $t[\vec{f}/\vec{b}, \vec{v}/\vec{s}]$ is a shortcut for $t[\vec{f}/\vec{b}][\vec{v}/\vec{s}]$.

In the following we regard a service-based application as a network N, and $\mathcal{C}$ is the set of constraints that is contained in the type for N.

Given the set of constraints $\mathcal{C}$, we define the set of b-variables as

$$\mathsf{V}^b(\mathcal{C}) = \bigcup_{t \sqsubseteq t' \in \mathcal{C}} \mathsf{V}^b(t),$$

the sets of of down-coerced and up-coerced t-variables as

$$\mathsf{V}^{\downarrow}(\mathcal{C}) = \bigcup_{t \sqsubseteq t' \in \mathcal{C}} \mathsf{V}^{\downarrow}(t) \cup \quad \text{and} \quad \mathsf{V}^{\uparrow}(\mathcal{C}) = \bigcup_{t \sqsubseteq t' \in \mathcal{C}} \mathsf{V}^{\uparrow}(t).$$

In the following for each set of constraints S such that $|\mathsf{V}^b(S)| = \ell$, $|\mathsf{V}^{\uparrow}(S)| = m$ and $|\mathsf{V}^{\downarrow}(S)| = n$ we use $\vec{f} = (f_1, \ldots, f_\ell)$ to denote the vector of b-variables contained in S, $\vec{v}^{\uparrow} = (v_1^{\uparrow}, \ldots, v_m^{\uparrow})$ to denote the vector of up-coerced t-variables and $\vec{v}^{\downarrow} = (v_1^{\downarrow}, \ldots, v_n^{\downarrow})$ to denote the vector of down-coerced t-variables.

Let $\mathcal{C}$ be a set of constraints such that $|\mathsf{V}^b(\mathcal{C})| = \ell$, $|\mathsf{V}^{\downarrow}(\mathcal{C})| = m$, $|\mathsf{V}^{\uparrow}(S)| = n$ and for some $\ell, m, n \geq 0$. Now we can define a CSP-WS formally as follows.

Definition 5.1.2 (CSP-WS)**.** Find a vector of Boolean values $\vec{b} = (b_1, \ldots, b_\ell)$ and vectors of ground terms $\vec{t}^{\downarrow} = (t_1^{\downarrow}, \ldots, t_m^{\downarrow})$, $\vec{t}^{\uparrow} = (t_1^{\uparrow}, \ldots, t_n^{\uparrow})$, such that for each $t_1 \sqsubseteq t_2 \in \mathcal{C}$

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

The tuple $(\vec{b}, \vec{t}^{\downarrow}, \vec{t}^{\uparrow})$ is called a solution.

A solution to the CSP-WS is not unique. In the context of web services, multiple solutions correspond to multiple interface configurations in a service-based application.

5.2 CSP-WS Solution Discussion

Definition 5.1.2 brings us to the most challenging problem that is solved in this thesis. We need a mechanism that solves the CSP-WS in order to demonstrate automatic reconciliation of the MDL interfaces. Below we discuss potential approaches to solving the problem.

The CSP-WS is solved if for all b-variables and t-variables we find values that satisfy constraints in $\mathcal{C}$. The domain of b-variables is finite, so we can apply brute-force search (in the worst case) or SAT solver (in the best case) to find values for the b-variables. On

the other hand, the domain of t-variables is infinite. Therefore, we can't use exhaustive search, which does not terminate over an infinite number of cases.

No universal approach for solving CSPs exist [Kum92]. Many of the problems can be solved using SMT solvers, which support a set of (typically decidable) theories. We considered applicability of some of the supported theories for solving the CSP-WS:

Propositional logic. Clearly, due to presence of b-variables in the MDL, propositional logic can be used to solve constraints for b-variables. Boolean SAT is an NP-complete problem. This a lower complexity bound of an algorithm that solves the CSP-WS providing that no other knowledge about the structure of the input constraints is given.

Equality logic and uninterpreted functions. We found out that equality logic and uninterpreted functions cannot be used for representing the MDL terms. We can use the uninterpreted functions and constants 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 fails to specify the seniority relation, which encodes an inequality.

Bit vectors. A bit vector be can used to represent a finite set of elements: a 1-bit indicates that an element from the finite domain is present in the set and a 0-bit indicates that the set lacks the element. In the MDL, presence of an element with a certain label in a record or choice can be represented as a bit in a bit vector. However, bit vectors cannot encode all properties of records and choices. First, all elements of a term are parametrised by a guard (a Boolean expression), which is impossible to represent in a bit vector. Moreover, hierarchical structuring of bit vectors is not supported. Therefore, support for records and choices that may contain each other as subterms is impossible.

To sum up, theories that are commonly supported in SMT solvers fail to encode the MDL terms. To overcome the limitations, we could extend SMT-LIB [BFT10][1]. However, this requires too much efforts dictated by non-trivial definition of well-formedness and the seniority relation for records, choices and switches, in particular. This would require a similar amount of efforts as designing an algorithm and a solver that is indented and tuned to solve specifically the CSP-WS. Finally, we decided to design a new algorithm and develop a solver that solves the CSP-WS by taking advantage of a particular structure of the problem.

[1] A language used for formal specification of constraints in SMT solvers (such as Z3, for instance [DMB08]).

[Kil73] presented an algorithm for global analysis of program entities that are structured in a lattice. The algorithm is widely used for data-flow analysis and optimisations in compilers. The algorithm finds values for terms (expressions, registers, basic blocks, etc.) that satisfy a set of given constraints. The algorithm starts with the most general approximation of term variable values, which are the top or the bottom elements of the lattice. By iteratively applying an iterated function to approximations that are obtained in previous steps, the algorithm converges to a solution for the CSP.

Unfortunately, we cannot directly apply the algorithm from [Kil73] to solve the CSP-WS due to two challenges that arise from the structure of the MDL:

1. The terms are structured in two semilattices, the top one and the bottom one, instead of a single lattice. Moreover, the terms of both semilattices may contain each other as subterms. This makes application of the fixed-point algorithm non-obvious. Furthermore, it is unclear whether such algorithm can lead to a correct solution or not.

   For example, consider the following constraint:

   $$\{\mathsf{a}\colon p^{\uparrow}\} \sqsubseteq \{\mathsf{a}\colon (:\mathsf{x}\colon q^{\downarrow} \,|r^{\uparrow}:) \,|s^{\downarrow}\}.$$

   $p^{\uparrow}$ and $r^{\uparrow}$ are up-coerced variables that are bounded by the bottom term none in the meet semilattice. The fixed-point algorithm can iteratively and monotonically refine a solution taking none as initial and the most general approximation. On the other hand, $q^{\downarrow}$ and $s^{\downarrow}$ are down-coerced variables that are bounded by the top term nil in the join semilattice. The algorithm should refine a solution starting from nil, which is the most general approximation for down-coerced terms. In this example, down-corced terms contain up-coerced terms as subterms and vice versa. Therefore, the algorithm must be carefully designed in a way that the values of the down-coerced variables are never coerced up and the values of the up-coerced variables are never coerced down (hence the name of the term categories).
2. The terms may contain b-variables. In addition to values for t-variables, values of b-variables must be found as part of a solution, which brings additional computational complexity to the problem.

   For example, in a relation

   $$p^{\downarrow} \sqsubseteq \{\mathsf{a}(x)\colon \mathtt{int}, \mathsf{b}(y)\colon \mathtt{float}\}$$

   the value of $p^{\downarrow}$ may be equal to nil, $\{\mathsf{a}\colon \mathtt{int}\}$, $\{\mathsf{b}\colon \mathtt{int}\}$ or $\{\mathsf{a}\colon \mathtt{int}, \mathsf{b}\colon \mathtt{float}\}$ depending on instantiations of x and y. In general, there are 2^{ℓ} ways to instantiate

b-variables, where ℓ is the total number of b-variables. Solving the CSP-WS for each instantiation is inefficient.

In Chapter 6 we present a fixed-point algorithm that solves the CSP-WS assuming that all b-variables are instantiated. We extend the meet and join semilattices to complete lattices by introducing the bottom element to the meet semilattice and the top element to the join semilattice. This allows us to apply the algorithm from [Kil73] to our problem. In Chapter 7 we improve the algorithm and make it applicable to the CSP-WS problem that is allowed to contain b-variables.

This is a simplified version of the algorithm presented in Chapter 7, which finds values for b-variables using a SAT solver.

Chapter 6

Solving the CSP-WS Without Boolean Variables

In the previous chapter we formally defined the constraint satisfaction problem for web services and then discussed approaches for solving the problem. In this chapter we design a modification of the fixed-point algorithm from [Kil73] for solving the CSP-WS. As the first step, we consider the CSP-WS when all b-variables are instantiated. This allows us to focus on the resolution of t-variables. Furthermore, we extend the semilattices to complete lattices in Section 6.2, because the fixed-point algorithm requires terms to be structured in a lattice or a bounded semilattice. In Section 6.3, we define an iterated function. In Section 6.5 we present the solution algorithm and prove that it converges to a solution for the CSP-WS.

6.1 Idea of the Iterative Algorithm

The algorithm presented in Section 6.5 finds a solution for the CSP-WS under an assumption that all b-variables in the constraints are instantiated. The algorithm monotonically traverses the join semilattice $(\mathcal{T}_m^{\downarrow}, \sqsubseteq)$ and the meet semilattice $(\mathcal{T}_n^{\uparrow}, \sqsubseteq)$, where $m = |\mathsf{V}^{\downarrow}(\mathcal{C})|$ and $n = |\mathsf{V}^{\uparrow}(\mathcal{C})|$. At the end, it either converges to the solution or returns Unsat if no solution exists.

As the first step, we extend the semilattices with two additional elements $\underline{\bot}$ and $\overline{\top}$, each of them representing lack of a solution (see Section 6.2). Such elements are often used in data-flow algorithms, such as algorithm for constraint propagation [CCKT86], for example. The algorithm operates on the extended semilattices. It performs the following steps:

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

$$(\vec{a}_i^{\downarrow}, \vec{a}_i^{\uparrow}) = ((\mathsf{nil}, \ldots, \mathsf{nil}), (\mathsf{none}, \ldots, \mathsf{none})).$$

   $(\mathsf{nil}, \ldots, \mathsf{nil})$ is the top element of the join semilattice $(\mathcal{T}_m^{\downarrow}, \sqsubseteq)$. Similarly, $(\mathsf{none}, \ldots, \mathsf{none})$ is the bottom element of the meet semilattice $(\mathcal{T}_n^{\uparrow}, \sqsubseteq)$.

2. Compute the next approximation $(\vec{a}_{i+1}^{\downarrow}, \vec{a}_{i+1}^{\uparrow})$ (using an iterated function $\overline{\mathsf{IF}}_{\mathcal{C}}$ introduced in Section 6.3), such that

$$\vec{a}_{i+1}^{\downarrow} \sqsubseteq \vec{a}_i^{\downarrow} \text{ and } \vec{a}_i^{\uparrow} \sqsubseteq \vec{a}_{i+1}^{\uparrow}, \tag{6.1}$$

   and for every constraint $t_1 \sqsubseteq t_2 \in \mathcal{C}$

$$t_1[\vec{v}^{\downarrow}/\vec{a}_2^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}_1^{\uparrow}] \sqsubseteq t_2[\vec{v}^{\downarrow}/\vec{a}_1^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}_2^{\uparrow}]. \tag{6.2}$$

   As we show in Lemma 6.4.1, these conditions are enough to show monotonicity of the approximations.

3. Repeat the step 2 if the chain of approximations has not converged to the solution (see termination proof inTheorem 6.5.1). The solution is found if

$$(\vec{a}_{i+1}^{\downarrow}, \vec{a}_{i+1}^{\uparrow}) = (\vec{a}_i^{\downarrow}, \vec{a}_i^{\uparrow})$$

   and

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

   The iterated function $\overline{\mathsf{IF}}_{\mathcal{C}}$ returns Unsat if a given constraint does not match one of the predefined constraints from Sections 6.3.1, 6.3.2 and 6.3.3. Otherwise, if constraints are satisfiable, the last approximation $(\vec{a}_i^{\downarrow}, \vec{a}_i^{\uparrow})$ is a solution.

6.2 Extension of the Semilattices

The original algorithm in [Kil73] terminates only if terms are structured in a lattice or a bounded semilattice. Since the MDL semilattices are unbounded, we extend them with two terms, which transform the semilattices to complete lattices.

Lemma 6.2.1 (Finite height of the semilattices)**.** If the set of record and choice labels in $\mathcal{T}$ is finite, then the semilattices $(\mathcal{T}^{\downarrow}, \sqsubseteq)$ and $(\mathcal{T}^{\uparrow}, \sqsubseteq)$ have a finite height.

Proof. Proof by contradiction.

Assume that the semilattice $(\mathcal{T}^{\downarrow}, \sqsubseteq)$ (the proof for $(\mathcal{T}^{\uparrow}, \sqsubseteq)$ is equivalent) has an infinite height if the set of labels in $\mathcal{T}$ is finite. Then there exists an infinite sequence of terms $\mathcal{T}^{\downarrow} \ni t_1, t_2, \ldots$, such that $t_1 \sqsupseteq t_2 \sqsupseteq \ldots$.

For each category of terms we show that it cannot infinitely expand.

Symbols. According to Definition 3.2.3, for any symbol s no other term t ($t \not\equiv s$) exists, such that $t \sqsubseteq s$. Therefore, s is the bottom term of the semilattice.

Tuples. According to Definition 3.2.3, a tuple can expand only if at least one of its subterms expands. The symbols cannot inifinitely expand, therefore, the tuples can infinitely expand only if records or choices can infinitely expand.

Records. According to Definition 3.2.3, a record can expand in one of two cases: 1) either elements with new labels are added to the records; or 2) at least one of record's subterms expands. Infinite expansion of the record with elements is impossible, because the set of labels is finite. Therefore, infinite expansion of records is possible only if choices can infinitely expand.

Choices. Similarly to records, we can show that infinite expansion of choices is impossible.

□

Since both semilattices have a finite height (given that the number of labels is finite), we can convert the semilattices to the lattices by adding the top and the bottom elements.

We extend the set $\mathcal{T}_m^{\downarrow}$ with an element $\underline{\bot}$:

$$\widetilde{\mathcal{T}}_m^{\downarrow} = \mathcal{T}_m^{\downarrow} \cup \{\underline{\bot}\}.$$

Symmetrically, we extend the set $\mathcal{T}_n^{\uparrow}$ with an element $\overline{\top}$:

$$\widetilde{\mathcal{T}}_n^{\uparrow} = \mathcal{T}_n^{\uparrow} \cup \{\overline{\top}\}.$$

$\underline{\bot}$ is defined as the bottom element of the meet semilattice:

$$\underline{\bot} \sqsubseteq \vec{a}^{\downarrow} \text{ for any } \vec{a}^{\downarrow} \in \widetilde{\mathcal{T}}_m^{\downarrow}.$$

$\overline{\top}$ is defined as the top element of the join semilattice, that is

$$\vec{a}^{\uparrow} \sqsubseteq \overline{\top} \text{ for any } \vec{a}^{\uparrow} \in \widetilde{\mathcal{T}}_n^{\uparrow}.$$

Introduction of new elements $\underline{\bot}$ and $\overline{\top}$ extends the join and the meet semilattices to complete lattices $(\widetilde{\mathcal{T}}_m^{\downarrow}, \sqsubseteq)$ and $(\widetilde{\mathcal{T}}_n^{\uparrow}, \sqsubseteq)$.

$\underline{\bot}$ or $\overline{\top}$ are used to encode Unsat. Theorem 6.5.5 proves that $\vec{a}^{\downarrow} = \underline{\bot}$ or $\vec{a}^{\uparrow} = \overline{\top}$ only if no solution for the CSP-WS exists.

6.3 Iterated Function

We introduce the *iterated function* $\overline{\mathsf{IF}} : \mathcal{C} \times \widetilde{\mathcal{T}}_m^{\downarrow} \times \widetilde{\mathcal{T}}_n^{\uparrow} \to \widetilde{\mathcal{T}}_m^{\downarrow} \times \widetilde{\mathcal{T}}_n^{\uparrow}$. Given a set of constraints and an approximation at some iteration i, the iterated function computes the new approximation that satisfies (6.1) and (6.2).

According to [Kil73], the iterated function has the homomorphism property if for any constraint $t_1 \sqsubseteq t_2$ and approximations $\vec{a}_1^{\downarrow}, \vec{a}_1^{\uparrow}, \vec{a}_2^{\downarrow}, \vec{a}_2^{\uparrow}, \vec{a}_1'^{\downarrow}, \vec{a}_1'^{\uparrow}, \vec{a}_2'^{\downarrow}$ and $\vec{a}_2'^{\uparrow}$, such that

$$\overline{\mathsf{IF}}(t_1 \sqsubseteq t_2, \vec{a}_1^{\downarrow}, \vec{a}_1^{\uparrow}) = (\vec{a}_1'^{\downarrow}, \vec{a}_1'^{\uparrow}),$$

$$\overline{\mathsf{IF}}(t_1 \sqsubseteq t_2, \vec{a}_2^{\downarrow}, \vec{a}_2^{\uparrow}) = (\vec{a}_2'^{\downarrow}, \vec{a}_2'^{\uparrow}),$$

the following equation holds:

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

In order to show the homomorphism property it is enough to show that $\overline{\mathsf{IF}}$ is monotonic (Lemma 6.4.1) and returns the tightest possible approximation (Lemma 6.5.3). The former is important for showing termination of the algorithm (Theorem 6.5.1); the latter is important for proving correctness of the algorithm (Lemma 6.5.4 and Theorem 6.5.5).

We define $\overline{\mathsf{IF}}$ for all categories of terms, except for a switch. In the absence of b-variables, a switch can always be converted to a term of different category in the canonical form (see the definition on page 35).

Based on Definition 3.2.3, we split constraints into two groups: satisfiable and unsatisfiable. If a constraint is unsatisfiable, the algorithm returns a pair of the lattices' elements $(\underline{\bot}, \overline{\top})$, which represents Unsat. Otherwise, we classify satisfiable constraints into four categories. From Definition 3.2.3 it follows that a constraint is satisfiable iff both terms are eiher symbols, tuples, records or choices. No solution exists for constraints on terms that belong two different categories (unless one the terms is nil). Consequently, we define $\overline{\mathsf{IF}}$ for each of the constraint categories in Sections 6.3.1, 6.3.2 and 6.3.3. Other constraints, which are not covered in the subsections, are considered unsatisfiable.

We use the following notation in the rest of the chapter. Let

$$\begin{aligned}\vec{v}^{\downarrow} &= (\overline{v}_1, \ldots, \overline{v}_m),\\ \vec{v}^{\uparrow} &= (\overline{\overline{v}}_1, \ldots, \overline{\overline{v}}_n),\\ \vec{a}^{\downarrow} &= (\overline{a}_1, \ldots, \overline{a}_m)\\ \vec{a}^{\uparrow} &= (\overline{\overline{a}}_1, \ldots, \overline{\overline{a}}_n).\end{aligned}$$

6.3.1 Iterated Function for Constraints on Atomic Terms and Variables

In this section we define the iterated function for the basic constraints. In such constraints both terms are either symbols, t-variables, nil or none. Constraints that are presented in other sections are always recursively reduced to the basic constraints.

- Assume a constraint of the form $t \sqsubseteq \mathsf{nil}$, where t is a down-coerced term, is given.

  The given approximation $(\vec{a}^{\downarrow}, \vec{a}^{\uparrow})$ already satisfies the constraint:

  $$\overline{\mathsf{IF}}(t \sqsubseteq \mathsf{nil}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\vec{a}^{\downarrow}, \vec{a}^{\uparrow}).$$

- Assume a constraint of the form $\mathsf{none} \sqsubseteq t$, where t is an up-coerced term, is given.

  The given approximation $(\vec{a}^{\downarrow}, \vec{a}^{\uparrow})$ already satisfies the constraint:

  $$\overline{\mathsf{IF}}(\mathsf{none} \sqsubseteq t, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\vec{a}^{\downarrow}, \vec{a}^{\uparrow}).$$

- Assume a constraint of the form $t \sqsubseteq t$ is given.

  The given approximation $(\vec{a}^{\downarrow}, \vec{a}^{\uparrow})$ already satisfies the constraint:

  $$\overline{\mathsf{IF}}(t \sqsubseteq t, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\vec{a}^{\downarrow}, \vec{a}^{\uparrow}).$$

- Assume a constraint of the form $t \sqsubseteq \overline{v}_\ell$, where t is a down-coerced term and $\overline{v}_\ell$ is a down-coerced variable, is given. Let $\overline{a}_\ell$ be the approximation for $\overline{v}_\ell$.

  The constraint can be reduced to the constraint with a ground term instead of $\overline{v}_\ell$. This is achieved by applying the substitution $\overline{v}_\ell[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}]$:

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

- Assume a constraint of the form $\overline{\overline{v}}_\ell \sqsubseteq t$, where t is an up-coerced term, is given.

This case is dual to the previous one. Specifically, the constraint can be reduced to the constraint with a ground term instead of $\overline{\overline{v}}_\ell$:

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

- Assume a constraint $\overline{v}_\ell \sqsubseteq t$, where $\overline{v}_\ell$ is a down-coerced variable and t is a down-coerced term, is given.

  In this case, the value for $\overline{v}_\ell$ is bounded by the term $t[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}]$. In other words, the value is below the term that is obtained by substituting variables in t with the values from the current approximation. Furthermore, the value for $\overline{v}_\ell$ that is the closest to $t[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}]$ is preferred. As a result, the new approximation is computed as the greatest lower bound of $t[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}]$ and the current approximation $\overline{a}_\ell$:[1]

$$\overline{\mathsf{IF}}(\overline{v}_\ell \sqsubseteq t, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = ((\overline{a}_1, \ldots, \overline{a}_\ell \sqcap t[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}], \ldots, \overline{a}_m), \vec{a}^{\uparrow}). \tag{6.3}$$

- Assume a constraint $t \sqsubseteq v_\ell^{\uparrow}$, where t is an up-coerced term and $v_\ell^{\uparrow}$ is an up-coerced variable, are given.

  This case is symmetric to the previous one. The new approximation for $v_\ell^{\uparrow}$ is computed as the least upper bound of the current approximation $\overline{\overline{a}}_\ell$ and $t[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}]$:

$$\overline{\mathsf{IF}}(t \sqsubseteq v_\ell^{\uparrow}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\vec{a}^{\downarrow}, (\overline{\overline{a}}_1, \ldots, \overline{\overline{a}}_\ell \sqcup t[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}], \ldots, \overline{\overline{a}}_n)). \tag{6.4}$$

Example 6.3.1. In this example, the current approximation for $\vec{v}^{\downarrow} = (\overline{v}_1)$ is $\vec{a}_i^{\downarrow} = (\texttt{int})$ and the set of values for up-coerced variables is empty. nil in the right part of the constraint declares that the consumer can accept a message of any format. Therefore, the existing approximation `int` for the variable $v_1^{\downarrow}$ already satisfies the constraint:

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

△

Example 6.3.2. A constraint, where the left part and the right part is the same symbol, is always satisfied regardless of the current approximation, like in this example:

$$\overline{\mathsf{IF}}(\texttt{string} \sqsubseteq \texttt{string}, (\texttt{int}), ()) = ((\texttt{int}), ()).$$

△

[1] The correctness proof of the algorithm (Theorem 6.5.5) relies on this property.

Example 6.3.3. 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:

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

△

Example 6.3.4. Consider the example of a constraint for up-coerced terms with up-coerced variable in the left part. The current approximation for the vector $\vec{v}^{\downarrow} = (\overline{v}_1)$ is $\vec{a}^{\downarrow} = (\mathtt{int})$ and the approximation for the vector $\vec{v}^{\uparrow} = (\overline{\overline{v}}_1)$ is $\vec{a}^{\uparrow} = ((\mathsf{:a}\colon \mathtt{int}\mathsf{:}))$. The example is symmetric to Example 6.3.3. By substitution, the constraint is reduced to the constraint with ground term in the left part (note that the right part may contain down-coerced variables as subterms):

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

△

Example 6.3.5. 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 $\{\mathsf{a}\colon \mathtt{int}\}$:

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

△

Example 6.3.6. 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:

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

△

6.3.2 Iterated Function for Constraint on Tuples

In the second category of constraints, both terms are tuples. The constraints on tuples are satisfiable only if the constraints on the tuple subterms are satisfiable, so these constraints are always reduced to simpler constraints.

Assume a constraint $t_1 \sqsubseteq t_2$, where $t_1 = (t_1^1 \dots t_k^1)$ and $t_2 = (t_1^2 \dots t_k^2)$, is given.

The constraint holds when the constraints for the corresponding nested terms hold as well:

$$\overline{\mathsf{IF}}((t_1^1 \dots t_k^1) \sqsubseteq (t_1^2 \dots t_k^2), \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\prod_{1 \le i \le k} \vec{a_i}^{\downarrow}, \bigsqcup_{1 \le i \le k} \vec{a_i}^{\uparrow}), \tag{6.5}$$

where $(\vec{a_i}^{\downarrow}, \vec{a_i}^{\uparrow}) = \overline{\mathsf{IF}}(t_i^1 \sqsubseteq t_i^2, \vec{a}^{\downarrow}, \vec{a}^{\uparrow})$.

Example 6.3.7. Consider a constraint on tuples $(\texttt{int}\ \overline{v}_1) \sqsubseteq (\overline{v}_2\ \texttt{string})$ and approximations $a^{\downarrow} = (\mathsf{nil}, \mathsf{nil})$ and $a^{\uparrow} = ()$ for vectors $v^{\downarrow} = (\overline{v}_1, \overline{v}_2)$ and $v^{\uparrow} = ()$, respectively, are given. Then the next approximation is computed in the following way:

$$\overline{\mathsf{IF}}((\texttt{int}\ \overline{v}_1) \sqsubseteq (\overline{v}_2\ \texttt{string}), (\mathsf{nil}, \mathsf{nil}), ()) = ((\mathsf{nil}, \texttt{string}), ())$$

△

6.3.3 Iterated Function for Constraints on Records

This subsection presents the iterated function for constraints where records are present.

- Assume a constraint $t_1 \sqsubseteq t_2$, where $t_1 = \{\mathsf{l}_1^1 \colon t_1^1, \dots, \mathsf{l}_\mathsf{d}^1 \colon t_d^1\}$ and $t_2 = \{\mathsf{l}_1^2 \colon t_1^2, \dots, \mathsf{l}_\mathsf{e}^2 \colon t_e^2\}$ (no tail variables), is given.

  The constraint holds if for all j ($1 \le j \le e$), there exists i ($1 \le i \le d$) such that $\mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2$ and $t_i^1 \sqsubseteq t_j^2$. Therefore, $\overline{\mathsf{IF}}$ for the records is defined as follows:

$$\overline{\mathsf{IF}}(\{\mathsf{l}_1^1 \colon t_1^1, \dots, \mathsf{l}_\mathsf{d}^1 \colon t_d^1\} \sqsubseteq \{\mathsf{l}_1^2 \colon t_1^2, \dots, \mathsf{l}_\mathsf{e}^2 \colon t_e^2\}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\prod_{1 \le k \le e} \vec{a_k}^{\downarrow}, \bigsqcup_{1 \le k \le e} \vec{a_k}^{\uparrow}), \tag{6.6}$$

  where

$$(\vec{a_k}^{\downarrow}, \vec{a_k}^{\uparrow}) = \begin{cases} \overline{\mathsf{IF}}(t_i^1 \sqsubseteq t_j^2, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) & \text{if } \exists i : \mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2 \\ (\bot, \overline{\top}) & \text{otherwise.} \end{cases}$$

- Consider a constraint $t_1 \sqsubseteq t_2$, where $t_1 = \{\mathsf{l}_1^1 \colon t_1^1, \dots, \mathsf{l}_\mathsf{d}^1 \colon t_d^1 \,|\overline{v}_\ell\}$, $t_2 = \{\mathsf{l}_1^2 \colon t_1^2, \dots, \mathsf{l}_\mathsf{e}^2 \colon t_e^2\}$, $\overline{v}_\ell$ is a down-coerced variable and $\overline{a}_\ell$ is an approximation of $\overline{v}_\ell$.

The constraint holds if for every element for every element $\mathsf{l}_\mathsf{j}^2 \colon t_j^2$ of the record t_2 one of the following holds:

1. there exists an element $\mathsf{l}_\mathsf{i}^1 \colon t_i^1$ in t_1 such that $\mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2$ and $t_i^1 \sqsubseteq t_j^2$;
2. a coercion $\overline{a}_\ell \sqcap \{\mathsf{l}_\mathsf{j}^2 \colon t_j^2\}$ exists.

Formally, $\overline{\mathsf{IF}}$ is defined as follows:

$$\overline{\mathsf{IF}}(\{\mathsf{l}_1^1 \colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1 \colon t_d^1 \,|\overline{v}_\ell\} \sqsubseteq \{\mathsf{l}_1^2 \colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2 \colon t_e^2\}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\prod_{1 \le k \le e} \vec{a_k}^{\downarrow}, \bigsqcup_{1 \le k \le e} \vec{a_k}^{\uparrow}), \quad (6.7)$$

where

$$(\vec{a_k}^{\downarrow}, \vec{a_k}^{\uparrow}) = \begin{cases} \overline{\mathsf{IF}}(t_i^1 \sqsubseteq t_j^2, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) & \text{if } \exists i : \mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2 \\ ((\overline{a}_1, \ldots, \overline{a}_\ell \sqcap t_j^2[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}], \overline{a}_{\ell+1}, \ldots, \overline{a}_m), \vec{a}^{\uparrow}) & \text{otherwise.} \end{cases}$$

- Consider a constraint $t_1 \sqsubseteq t_2$, where t_1 is a record $\{\mathsf{l}_1^1 \colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1 \colon t_d^1\}$ or a record $\{\mathsf{l}_1^1 \colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1 \colon t_d^1 \,|\overline{v}_\ell\}$, and t_2 is a record $\{\mathsf{l}_1^2 \colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2 \colon t_e^2 \,|\overline{v}_r\}$.

  Using substitution the constraint can be reduces to one of the previous constraints as follows:

$$\overline{\mathsf{IF}}(t_1 \sqsubseteq t_2, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = \overline{\mathsf{IF}}(t_1 \sqsubseteq t_2[\overline{v}_r/\overline{a}_r], \vec{a}^{\downarrow}, \vec{a}^{\uparrow}), \quad (6.8)$$

  where $\overline{a}_r$ is an approximation of $\overline{v}_r$.

Example 6.3.8. Consider a constraint for records and the approximations $v^{\downarrow} = (\overline{a}_1) = (\mathsf{nil})$ and $v^{\uparrow} = ()$ for the variables $v^{\downarrow} = (\overline{v}_1)$ and $v^{\uparrow} = ()$, respectively:

$$\begin{aligned} \overline{\mathsf{IF}}(\{\mathsf{a}\colon \mathtt{int}, \mathsf{b}\colon \overline{v}_1, \mathsf{c}\colon \mathtt{int}\} \sqsubseteq \{\mathsf{a}\colon \mathsf{nil}, \mathsf{b}\colon \mathtt{string}\}, (\mathsf{nil}), ()) &= ((\mathsf{nil} \sqcap \mathtt{string}), ()) \\ &= ((\mathtt{string}), ()) \quad (6.9) \end{aligned}$$

△

Example 6.3.9. Consider the constraint for records $\{\mathsf{a}\colon \overline{v}_1 \,|\overline{v}_2\} \sqsubseteq \{\mathsf{a}\colon \mathtt{string}, \mathsf{b}\colon \mathtt{int}\}$, given the approximation $a^{\downarrow} = (\mathsf{nil}, \{\mathsf{c}\colon \mathtt{int}\})$ and $a^{\uparrow} = ()$ for variables $v^{\downarrow} = (\overline{v}_1, \overline{v}_2)$ and $v^{\uparrow} = ()$, respectively:

$$\begin{aligned} \overline{\mathsf{IF}}(\{\mathsf{a}\colon \overline{v}_1 \,|\overline{v}_2\} \sqsubseteq \{\mathsf{a}\colon \mathtt{string}, \mathsf{b}\colon \mathtt{int}\}, (\mathsf{nil}, \{\mathsf{c}\colon \mathtt{int}\}), ()) &= \\ ((\mathtt{string}, \{\mathsf{c}\colon \mathtt{int}\} \sqcap \{\mathsf{b}\colon \mathtt{int}\}), ()) &= \\ ((\mathtt{string}, \{\mathsf{b}\colon \mathtt{int}, \mathsf{c}\colon \mathtt{int}\}), ()) & \end{aligned}$$

Note that $\overline{v}_2$ cannot contain an element with the label a. Otherwise, the well-formed term for $\{\mathsf{a}\colon \overline{v}_1 \,|\overline{v}_2\}$ does not exist and, therefore, the constraint is unsatisfiable. △

6.3.4 Iterated Function for Constraints on Choices

The seniority relation for choices is symmetric to the seniority relation for records. Therefore, we define $\overline{\mathsf{IF}}$ for choices similarly to $\overline{\mathsf{IF}}$ for records.

- Assume a constraint $t_1 \sqsubseteq t_2$, where $t_1 = (:\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1\colon t_d^1:)$ and $t_2 = (:\mathsf{l}_1^2\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2\colon t_e^2:)$ (no tail variables), is given.

  The constraint holds if for all i ($1 \leq i \leq d$), there exists j ($1 \leq j \leq e$) such that $\mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2$ and $t_i^1 \sqsubseteq t_j^2$. Therefore, $\overline{\mathsf{IF}}$ for the records is defined as follows:

$$\overline{\mathsf{IF}}((:\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1\colon t_d^1:) \sqsubseteq (:\mathsf{l}_1^2\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2\colon t_e^2:), \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\bigsqcap_{1\leq k\leq d} \vec{a}_k{}^{\downarrow}, \bigsqcup_{1\leq k\leq d} \vec{a}_k{}^{\uparrow}), \quad (6.10)$$

  where

$$(\vec{a}_k{}^{\downarrow}, \vec{a}_k{}^{\uparrow}) = \begin{cases} \overline{\mathsf{IF}}(t_i^1 \sqsubseteq t_j^2, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) & \text{if } \exists j : \mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2 \\ (\bot, \top) & \text{otherwise.} \end{cases}$$

- Consider a constraint $t_1 \sqsubseteq t_2$, where $t_1 = (:\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1\colon t_d^1:)$, $t_2 = (:\mathsf{l}_1^2\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2\colon t_e^2 \,|\overline{\overline{v}}_\ell:)$, and $\overline{\overline{a}}_\ell$ is an approximation of $\overline{\overline{v}}_\ell$.

  The constraint holds if for every element for every element $\mathsf{l}_\mathsf{i}^1\colon t_i^1$ of the choice t_1 one of the following holds:

  1. there exists an element $\mathsf{l}_\mathsf{j}^2\colon t_j^2$ in t_2 such that $\mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2$ and $t_i^1 \sqsubseteq t_j^2$;
  2. a coercion $\overline{\overline{a}}_\ell \sqcup \{\mathsf{l}_\mathsf{i}^1\colon t_i^1\}$ exists.

  Formally, $\overline{\mathsf{IF}}$ is defined as follows:

$$\overline{\mathsf{IF}}((:\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1\colon t_d^1 \,|\overline{\overline{v}}_\ell:) \sqsubseteq (:\mathsf{l}_1^2\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2\colon t_e^2:), \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\bigsqcap_{1\leq k\leq d} \vec{a}_k{}^{\downarrow}, \bigsqcup_{1\leq k\leq d} \vec{a}_k{}^{\uparrow}), \quad (6.11)$$

  where

$$(\vec{a}_k{}^{\downarrow}, \vec{a}_k{}^{\uparrow}) = \begin{cases} \overline{\mathsf{IF}}(t_i^1 \sqsubseteq t_j^2, \vec{a}_k{}^{\downarrow}, \vec{a}_k{}^{\uparrow}) & \text{if } \exists i : \mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2 \\ (\vec{a}^{\downarrow}, (\overline{\overline{a}}_1, \ldots, \overline{\overline{a}}_\ell \sqcup t_i^1[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}], \overline{\overline{a}}_{\ell+1}, \ldots, \overline{\overline{a}}_m)) & \text{otherwise.} \end{cases}$$

- Consider a constraint $t_1 \sqsubseteq t_2$, where t_1 is a choice $(:\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1\colon t_d^1 \,|\overline{\overline{v}}_\ell:)$ and t_2 is a choice $(:\mathsf{l}_1^2\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2\colon t_e^2:)$ or $(:\mathsf{l}_1^2\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2\colon t_e^2 \,|\overline{\overline{v}}_r:)$.

Using substitution the constraint can be reduces to one of the previous constraints as follows:

$$\overline{\mathsf{IF}}(t_1 \sqsubseteq t_2, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = \overline{\mathsf{IF}}(t_1[\overline{\overline{v}}_\ell / \overline{\overline{a}}_\ell] \sqsubseteq t_2, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}),$$

where $\overline{\overline{a}}_\ell$ is an approximation for $\overline{\overline{v}}_\ell$.

Example 6.3.10. $\overline{\mathsf{IF}}$ can be used to find a solution even for non-trivial constraints, for instance, those that contain interleaving down-coerced and up-coerced terms. For example, consider the constraint where the top-level terms are records that contain choices as nested subterms:

$$\{\mathsf{f}\colon (\mathsf{:c}\colon \mathtt{int}\mathtt{:})\,|v_1^{\downarrow}\} \sqsubseteq \{\mathsf{f}\colon v^{\uparrow}\,|v_2^{\downarrow}\},$$

and the initial approximations for variables $\vec{v}^{\downarrow} = (v_1^{\downarrow}, v_2^{\downarrow})$ and $\vec{v}^{\uparrow} = (v^{\uparrow})$ are $\vec{a}^{\downarrow} = (\mathsf{nil}, \mathsf{nil})$ and $\vec{a}^{\uparrow} = (\mathsf{none})$, respectively. Then, according to (6.8)

$$\overline{\mathsf{IF}}(\{\mathsf{f}\colon (\mathsf{:c}\colon \mathtt{int}\mathtt{:})\,|v_1^{\downarrow}\} \sqsubseteq \{\mathsf{f}\colon v^{\uparrow}\,|v_2^{\downarrow}\}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = \overline{\mathsf{IF}}(\{\mathsf{f}\colon (\mathsf{:c}\colon \mathtt{int}\mathtt{:})\,|v_1^{\downarrow}\} \sqsubseteq \{\mathsf{f}\colon v^{\uparrow}\}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}).$$

Following (6.7), the 'nested' constraint $(\mathsf{:c}\colon \mathtt{int}\mathtt{:}) \sqsubseteq v^{\uparrow}$ holds:

$$\overline{\mathsf{IF}}((\mathsf{:c}\colon \mathtt{int}\mathtt{:}) \sqsubseteq v^{\uparrow}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\vec{a}^{\downarrow}, (\mathsf{none} \sqcup (\mathsf{:c}\colon \mathtt{int}\mathtt{:}))) = ((\mathsf{nil}, \mathsf{nil}), ((\mathsf{:c}\colon \mathtt{int}\mathtt{:})).$$

△

6.4 Monotonicity of the Iterated Function

In this section we show that the function $\overline{\mathsf{IF}}$ is monotonic.

Lemma 6.4.1 (Monotonicity)**.** Let $\overline{\mathsf{IF}}(t_1 \sqsubseteq t_2, \vec{a}_1^{\downarrow}, \vec{a}_1^{\uparrow}) = (\vec{\tilde{a}}_1^{\downarrow}, \vec{\tilde{a}}_1^{\uparrow})$ and $\overline{\mathsf{IF}}(t_1 \sqsubseteq t_2, \vec{a}_2^{\downarrow}, \vec{a}_2^{\uparrow}) = (\vec{\tilde{a}}_2^{\downarrow}, \vec{\tilde{a}}_2^{\uparrow})$.

$$\text{If } \vec{a}_1^{\downarrow} \sqsubseteq \vec{a}_2^{\downarrow} \text{ and } \vec{a}_2^{\uparrow} \sqsubseteq \vec{a}_1^{\uparrow}, \text{ then } \vec{\tilde{a}}_1^{\downarrow} \sqsubseteq \vec{\tilde{a}}_2^{\downarrow} \text{ and } \vec{\tilde{a}}_2^{\uparrow} \sqsubseteq \vec{\tilde{a}}_1^{\uparrow}.$$

Monotonicity of $\overline{\mathsf{IF}}$ follows from definition of the function. Indeed, in Eqs. (6.3), (6.5), (6.6), (6.7), (6.10) and (6.11), the approximations for down-coerced variables can only coerce down in the lattice $\widetilde{\mathcal{T}}_m^{\downarrow}$. Similarly, in Eqs. (6.4), (6.5), (6.6), (6.7), (6.10) and (6.11) the approximations for up-coerced variables can only coerce up in the lattice $\widetilde{\mathcal{T}}_n^{\uparrow}$.

We define the function $\overline{\mathsf{IF}}_{\mathcal{C}}$ as a composition of $\overline{\mathsf{IF}}$ functions that are sequentially applied to all constraints in $\mathcal{C}$ (the order in which $\overline{\mathsf{IF}}$ is applied to the constraints is not important due to distributivity of the seniority relation):

$$\overline{\mathsf{IF}}_{\mathcal{C}}(\vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = \overline{\mathsf{IF}}(t_1^{|\mathcal{C}|} \sqsubseteq t_2^{|\mathcal{C}|}, \overline{\mathsf{IF}}(t_1^{|\mathcal{C}|-1} \sqsubseteq t_2^{|\mathcal{C}|-1}, \ldots, \overline{\mathsf{IF}}(t_1^1 \sqsubseteq t_2^1, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) \ldots)) \tag{6.12}$$

The sequential composition preserves monotonicity for $\overline{\mathsf{IF}}_{\mathcal{C}}$. In Section 7.5 we tacitly assume that for arbitrary terms (including those that contain b-variables) the function $\overline{\mathsf{IF}}_{\mathcal{C}}$ is defined in a similar way.

6.5 Fixed-Point Algorithm

We present Algorithm 1 that computes a chain of approximations using the function $\overline{\mathsf{IF}}_{\mathcal{C}}$. The input for the algorithm is the set of constraints $\mathcal{C}$ such that $\mathsf{V}^b(\mathcal{C}) = \emptyset$ (in other words, terms in $\mathcal{C}$ must not contain b-variables). In addition to this, in this section we formally prove that the algorithm returns the solution if one exists.

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

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

We already provided an informal description of Algorithm 1 on page 61. The algorithm sets the initial approximation in Line 2. The algorithm uses $\overline{\mathsf{IF}}_{\mathcal{C}}$ to compute next approximation that satisfies properties (6.1) and (6.2) in Line 5. After each iteration the termination condition in Line 6 is checked. The algorithm terminates if the series of approximations has converged to a value (in this case, the value is returned as as a solution). In fact, $\overline{\mathsf{IF}}_{\mathcal{C}}$ always returns the approximation, even if the set of constraints is unsatisfiable (in case of the latter, $(\overline{\top}, \underline{\bot})$ is returned). This allowed us to create a generic algorithm for all possible sets of constraints regardless of whether a solution exists or not.

Theorem 6.5.1 (Termination)**.** *For any set of constraints $\mathcal{C}$ such that $\mathsf{V}^b(\mathcal{C}) = \emptyset$, Algorithm 1 terminates after a finite number of steps.*

Proof. $\overline{\mathsf{IF}}_{\mathcal{C}}$ is a monotonic function (Lemma 6.4.1) that maps $\vec{a}_{i-1}^{\downarrow} \in \widetilde{\mathcal{T}}_m^{\downarrow}$ to $\vec{a}_i^{\downarrow} \in \widetilde{\mathcal{T}}_m^{\downarrow}$ and $\vec{a}_{i-1}^{\uparrow} \in \widetilde{\mathcal{T}}_n^{\uparrow}$ to $\vec{a}_i^{\uparrow} \in \widetilde{\mathcal{T}}_n^{\uparrow}$, where $\vec{a}_{i-1}^{\downarrow}$ and $\vec{a}_i^{\downarrow}$ are elements of the lattice $(\widetilde{\mathcal{T}}_m^{\downarrow}, \sqsubseteq)$ such that $\vec{a}_i^{\downarrow} \sqsubseteq \vec{a}_{i-1}^{\downarrow}$, and $\vec{a}_{i-1}^{\uparrow}$ and $\vec{a}_i^{\uparrow}$ are elements of the lattice $(\widetilde{\mathcal{T}}_n^{\uparrow}, \sqsubseteq)$ such that $\vec{a}_{i-1}^{\uparrow} \sqsubseteq \vec{a}_i^{\uparrow}$. Below we prove that the algorithm iteratively calls $\overline{\mathsf{IF}}_{\mathcal{C}}$ until the fixed-point is reached. As a

result, it terminates after a finite number of steps if both lattices have a finite height (Lemma 6.2.1).

We prove it by induction on term depth. $(\widetilde{\mathcal{T}}_m^{\downarrow}, \sqsubseteq)$ and $(\widetilde{\mathcal{T}}_n^{\uparrow}, \sqsubseteq)$ have a finite height if the semilattices $(\mathcal{T}^{\downarrow}, \sqsubseteq)$ and $(\mathcal{T}^{\uparrow}, \sqsubseteq)$ have a finite height too.

Consider the semilattice $(\mathcal{T}^{\downarrow}, \sqsubseteq)$ (the proof for $(\mathcal{T}^{\uparrow}, \sqsubseteq)$ 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 3.2.3 and illustration in Figure 3.1): the term category is constant and cannot be changed unless the term is nil.

Symbol. The semilattice $(\mathcal{T}^{\downarrow}, \sqsubseteq)$ 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 $\mathcal{C}$ 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 $\mathcal{C}$ 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 is finite.

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

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

Next we introduce lemmas that are required for showing the correctness of Algorithm 1.

Substitution of variables with ground terms is a monotonic function. Below we prove that the substitution of down-coerced variables is a decreasing function. Similarly we can prove that the substitution of up-coerced variables is an increasing function.

Lemma 6.5.2 (Substitution monotonicity)**.** Let t be a term such that $|\mathrm{V}^b(t)| = \emptyset$, $\vec{v}^{\downarrow} = (v_1, \ldots, v_k)$ be a vector of down-coerced variables in t, and $\vec{s_1}^{\downarrow} = (s_1^1, \ldots, s_k^1)$ and $\vec{s_2}^{\downarrow} = (s_1^2, \ldots, s_k^2)$ be vectors of down-coerced ground terms such that $\vec{s_1}^{\downarrow} \sqsubseteq \vec{s_2}^{\downarrow}$. Then

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

Proof. Substitution monotonicity follows from the structure of the seniority relation. Any term is covariant with respect to its subterms (see Definition 3.2.3). □

In general, subsitution monotonicity is not guaranteed by a term algebra. Assume that we extend the MDL with a *function* term $t_1 \to t_2$, and the seniority relation for function terms is defined as follows:

$$t_1^1 \to t_2^1 \sqsubseteq t_1^2 \to t_2^2 \text{ if } t_1^2 \sqsubseteq t_1^1 \text{ and } t_2^1 \sqsubseteq t_2^2.$$

Due to contravariance of t_1^1 and t_1^2, the substitution monotonicity for function terms does not hold.

The next lemma states that the function $\overline{\mathsf{IF}}$ produces the tightest possible approximation of a solution. Furthermore, such approximation is unique.

Lemma 6.5.3. Assume a constraint $t_1 \sqsubseteq t_2$ and approximations $(\vec{a}_1^{\downarrow}, \vec{a}_1^{\uparrow})$ and $(\vec{a}_2^{\downarrow}, \vec{a}_2^{\uparrow})$ such that $\overline{\mathsf{IF}}(t_1 \sqsubseteq t_2, \vec{a}_1^{\downarrow}, \vec{a}_1^{\uparrow}) = (\vec{a}_2^{\downarrow}, \vec{a}_2^{\uparrow})$ are given. If

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

then:

1. (Tightness) No approximation $(\vec{a}_3^{\downarrow}, \vec{a}_3^{\uparrow})$ exists such that $(\vec{a}_3^{\downarrow}, \vec{a}_3^{\uparrow}) \neq (\vec{a}_2^{\downarrow}, \vec{a}_2^{\uparrow})$, $\vec{a}_2^{\downarrow} \sqsubseteq \vec{a}_3^{\downarrow}$, $\vec{a}_3^{\uparrow} \sqsubseteq \vec{a}_2^{\uparrow}$ and
$$t_1[\vec{v}^{\downarrow}/\vec{a}_3^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}_1^{\uparrow}] \sqsubseteq t_2[\vec{v}^{\downarrow}/\vec{a}_1^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}_3^{\uparrow}]. \tag{6.13}$$
2. (Uniqueness) For any other approximation $(\vec{\tilde{a}}_2^{\downarrow}, \vec{\tilde{a}}_2^{\uparrow})$ such that
$$t_1[\vec{v}^{\downarrow}/\vec{\tilde{a}}_2^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}_1^{\uparrow}] \sqsubseteq t_2[\vec{v}^{\downarrow}/\vec{a}_1^{\downarrow}, \vec{v}^{\uparrow}/\vec{\tilde{a}}_2^{\uparrow}], \tag{6.14}$$
there exists $(\vec{\tilde{a}}_3^{\downarrow}, \vec{\tilde{a}}_3^{\uparrow})$ such that $(\vec{\tilde{a}}_2^{\downarrow}, \vec{\tilde{a}}_2^{\uparrow}) \neq (\vec{\tilde{a}}_3^{\downarrow}, \vec{\tilde{a}}_3^{\uparrow})$ and
$$t_1[\vec{v}^{\downarrow}/\vec{\tilde{a}}_3^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}_1^{\uparrow}] \sqsubseteq t_2[\vec{v}^{\downarrow}/\vec{a}_1^{\downarrow}, \vec{v}^{\uparrow}/\vec{\tilde{a}}_3^{\uparrow}].$$

Proof.
1. By the definition of the iterated function in Section 6.3, the function $\overline{\mathsf{IF}}$ makes a coercion of the approximation $(\vec{a}_1^{\downarrow}, \vec{a}_1^{\uparrow})$ only if $(\vec{a}_1^{\downarrow}, \vec{a}_1^{\uparrow})$ 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 $(\vec{a}_2^{\downarrow}, \vec{a}_2^{\uparrow})$. The approximation $(\vec{a}_3^{\downarrow}, \vec{a}_3^{\uparrow})$, such that (6.13) holds, would exist only if $\overline{\mathsf{IF}}$ performed excessive coercions, which we avoid in the definition of $\overline{\mathsf{IF}}$.
2. The uniqueness of $(\vec{a}_2^{\downarrow}, \vec{a}_2^{\uparrow})$ follows from the definition of the seniority relation.

□

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

Figure 6.1: $(\vec{s}_1^{\downarrow}, \vec{s}_1^{\uparrow})$ and $(\vec{s}_2^{\downarrow}, \vec{s}_2^{\uparrow})$ are fixed points of $\overline{\mathsf{IF}}_{\mathcal{C}}$ in the lattices $(\widetilde{\mathcal{T}}_m^{\downarrow}, \sqsubseteq)$ and $(\widetilde{\mathcal{T}}_n^{\uparrow}, \sqsubseteq)$. By Knaster-Tarski theorem, the set of the fixed points form lattices too. Lemma 6.5.4 states that the chain of approximations (the dashed path) converges to the greatest fixed point $\vec{s}^{\downarrow}$ in $(\widetilde{\mathcal{T}}_m^{\downarrow}, \sqsubseteq)$ and the least fixed point $\vec{s}^{\uparrow}$ in $(\widetilde{\mathcal{T}}_n^{\uparrow}, \sqsubseteq)$.

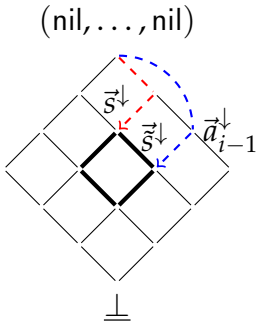


Figure 6.2: Illustration of the reachability proof in Lemma 6.5.4. It is proved by contradiction that a chain of approximations converges to the greatest fixed point $\vec{s}^{\downarrow}$. The chain of approximations would converge to some other fixed point $\vec{\tilde{s}}^{\downarrow}$ only if $\overline{\mathsf{IF}}_{\mathcal{C}}$ generated an approximation, which is not the 'tightest'. The latter contradicts Lemma 6.5.3.

Since $\overline{\mathsf{IF}}$ produces the tightest approximation, $\overline{\mathsf{IF}}_{\mathcal{C}}$ produces the tightest approximation too, which follows from the structure of $\overline{\mathsf{IF}}_{\mathcal{C}}$ (Eq. (6.12)).

Lemma 6.5.4. Assume a set of constraints $\mathcal{C}$, $\mathsf{V}^b(\mathcal{C}) = \emptyset$, is given. Let for $k > 0$

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

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

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

Proof. The proof consists of two parts. First, we prove that a fixed-point $(\vec{s}^{\downarrow}, \vec{s}^{\uparrow})$ exists, such that for any fixed point $(\vec{\tilde{s}}^{\downarrow}, \vec{\tilde{s}}^{\uparrow})$, $\vec{s}^{\downarrow} \sqsubseteq \vec{\tilde{s}}^{\downarrow}$ and $\vec{s}^{\uparrow} \sqsubseteq \vec{\tilde{s}}^{\uparrow}$. Then we show that $\overline{\mathsf{IF}}_{\mathcal{C}}$

converges to $(\vec{s}^{\downarrow}, \vec{s}^{\uparrow})$, that is $(\vec{a}_k^{\downarrow}, \vec{a}_k^{\uparrow}) = (\vec{s}^{\downarrow}, \vec{s}^{\uparrow})$.

Existence $(\widetilde{\mathcal{T}}_m^{\downarrow}, \sqsubseteq)$ and $(\widetilde{\mathcal{T}}_n^{\uparrow}, \sqsubseteq)$ are complete lattices and $\overline{\mathsf{IF}}_{\mathcal{C}}$ is an order-preserving function. By Knaster-Tarski theorem [T+55], the sets of fixed points of $\overline{\mathsf{IF}}_{\mathcal{C}}$ in $(\widetilde{\mathcal{T}}_m^{\downarrow}, \sqsubseteq)$ and $(\widetilde{\mathcal{T}}_n^{\uparrow}, \sqsubseteq)$ are complete lattices too. Therefore, there exists the fixed-point $(\vec{s}^{\downarrow}, \vec{s}^{\uparrow})$ such that for any fixed-point $(\vec{\tilde{s}}^{\downarrow}, \vec{\tilde{s}}^{\uparrow})$, $\vec{\tilde{s}}^{\downarrow} \sqsubseteq \vec{s}^{\downarrow}$ and $\vec{s}^{\uparrow} \sqsubseteq \vec{\tilde{s}}^{\uparrow}$ (see Figure 6.1).

Reachability Proof by contradiction (see illustration of the proof in Figure 6.2). Assume that $\overline{\mathsf{IF}}_{\mathcal{C}}$ does not converge to $(\vec{s}^{\downarrow}, \vec{s}^{\uparrow})$, that is $(\vec{a}_k^{\downarrow}, \vec{a}_k^{\uparrow}) = (\vec{\tilde{s}}^{\downarrow}, \vec{\tilde{s}}^{\uparrow})$, where $(\vec{\tilde{s}}^{\downarrow}, \vec{\tilde{s}}^{\uparrow}) \neq (\vec{s}^{\downarrow}, \vec{s}^{\uparrow})$, and $\vec{\tilde{s}}^{\downarrow} \sqsubseteq \vec{s}^{\downarrow}$ or $\vec{s}^{\uparrow} \sqsubseteq \vec{\tilde{s}}^{\uparrow}$. Assume that $\vec{\tilde{s}}^{\downarrow} \sqsubseteq \vec{s}^{\downarrow}$ (the case $\vec{s}^{\uparrow} \sqsubseteq \vec{\tilde{s}}^{\uparrow}$ is considered similarly).

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

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

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

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

On the other hand, $\vec{\tilde{s}}^{\downarrow} \sqsubseteq \vec{s}^{\downarrow}$. Due to substitution monotonicity (Lemma 6.5.2),

$$t_1[\vec{v}^{\downarrow}/\vec{\tilde{s}}^{\downarrow}, \vec{v}^{\uparrow}/\vec{\tilde{a}}_{i-1}^{\uparrow}] \sqsubseteq t_1[\vec{v}^{\downarrow}/\vec{s}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}_{i-1}^{\uparrow}]. \tag{6.15}$$

Lemma 6.5.3 states that $\overline{\mathsf{IF}}_{\mathcal{C}}$ produces the 'tightest' approximation $\vec{\tilde{s}}^{\downarrow}$. On the other hand, from (6.15) it follows that $\vec{s}^{\downarrow}$ is 'tighter' than $\vec{\tilde{s}}^{\downarrow}$. Contradiction found.

Therefore, $\overline{\mathsf{IF}}_{\mathcal{C}}$ converges to the fixed point $\vec{s}^{\downarrow}$ and no $\vec{\tilde{s}}^{\downarrow}$ exists such that $\vec{s}^{\downarrow} \sqsubseteq \vec{\tilde{s}}^{\downarrow}$.

□

Theorem 6.5.5 (Correctness)**.** *For any set of constraints $\mathcal{C}$ such that $\mathsf{V}^b(\mathcal{C}) = \emptyset$,* CSP-WS *for $\mathcal{C}$ is unsatisfiable iff Algorithm 1 returns* Unsat.

Proof. Proof by contradiction.

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

($\Leftarrow$) Let Algorithm 1 return Unsat and $\mathcal{C}$ has a solution. In this case the chain of approximations in Algorithm 1 returns $(\bot, \overline{\top})$. This is the fixed point and by Lemma 6.5.4

no other fixed point $(\vec{\vec{s}}^{\downarrow}, \vec{\vec{s}}^{\uparrow})$ exists such that $\underline{\bot} \sqsubseteq \vec{\vec{s}}^{\downarrow}$ or $\vec{\vec{s}}^{\uparrow} \sqsubseteq \overline{\top}$, which means that no fixed points apart from $(\underline{\bot}, \overline{\top})$ exists. This contradicts the initial hypothesis. Therefore, $\mathcal{C}$ is unsatisfiable if Algorithm 1 returns Unsat. □

Chapter 7

CSP-WS Algorithm

In Chapter 6 we presented the algorithm that solves the CSP-WS for the set of constraints without b-variables. The algorithm iteratively calls the approximation function $\overline{\mathsf{IF}}_{\mathcal{C}}$ and traverses the extended semilattices $(\widetilde{\mathcal{T}}_m^{\downarrow})$ and $(\widetilde{\mathcal{T}}_n^{\uparrow})$ until the fixed point is reached. 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 from the initial problem, where l is the number of b-variables. Then, each subproblem can be solved using Algorithm 1. The solution for any of the subproblems is a solution for the original problem. If all of the subproblems do not have a solution, the original problem does not have a solution either.

Solving the subproblems independently is inefficient. Instead, we propose to find a solution to all subproblems simultaneously. For this purpose, we introduce two improvements to Algorithm 1:

- We introduce a set of Boolean constraints that specifies Boolean instantiations that can potentially lead to a solution. If the set contains a contradiction, then all instantiations are unsatisfiable. As a result, the algorithm returns Unsat.
- For each t-variable, we store multiple values as terms (in Algorithm 1 only one value is associated with each t-variable). Each value is associated with a Boolean constraint that specifies a set of Boolean instantiations. As a result, the algorithm solves all subproblems simultaneously.

In Section 7.1 we explain how the set of Boolean constraints is generated. Then we provide the details of the algorithm. We conclude this chapter by providing a sub-algorithm that transforms a term $(\mathsf{union}\ v_1^{\downarrow}\ v_2^{\downarrow})$ to a record that contains elements from both records $v_1^{\downarrow}$ and $v_2^{\downarrow}$. A union is a special MDL term that we use for implementing multiple flow inheritance (see Section 3.5). Multiple flow inheritance allows multiple messages to be merged into one. Informally, a union is a macro with two records given as

parameters. It is replaced with a record that is a composition of the parameters after the CSP-WS is solved.

7.1 Boolean Constraints for CSP-WS

The fixed-point algorithm for solving the CSP-WS iterates over a set of constraints until the constraints are satisfied or no satisfiable Boolean assignments are left. At each iteration, the algorithm generates Boolean constraints that eliminate unsatisfiable assignment. We denote such set of the Boolean constraints as B, $\mathsf{B} \subseteq \mathcal{B}$. Let $\mathsf{SAT}(\mathsf{B})$ denote the set of Boolean vectors satisfying B.

If $t_1^1 \sqsubseteq t_2^1 \in \mathcal{C}$ and $t_1^2 \sqsubseteq t_2^2$ are some constraints and B_1 and B_2 are the sets of Boolean constraints that specify Boolean assignments satisfying $t_1^1 \sqsubseteq t_2^1$ and $t_1^2 \sqsubseteq t_2^2$ respectively, then the set of Boolean constraints that satisfies both $t_1^1 \sqsubseteq t_2^1$ and $t_1^2 \sqsubseteq t_2^2$ is produced as $\mathsf{B}_1 \cup \mathsf{B}_2$.

At each iteration constraints are added to B. A Boolean assignment that is a solution belongs to an intersection of individual solutions to each of the constraints. As a result, when new constraints are added to B, the number of potential solutions is not increasing.

7.1.1 Well-Formedness Constraints

The first set of Boolean constraints is called *well-formedness constraints*. Well-formedness constraints specify conditions for a term t, $\mathsf{V}(t) = \emptyset$, expressed as Boolean constraints $\mathsf{WFC}(t)$, which guarantee that t is well-formed if $\mathsf{WFC}(t)$ evaluates to true. The well-formedness constraints are given in Figure 7.1. The constraints directly follow from definition of the well-formed term in Definition 3.2.2.

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

Figure 7.1: The set of Boolean constraints that ensures well-formedness of a term t

A symbol is a well-formed term and, therefore, no Boolean constraints if t is a symbol 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, must be produced. In addition to this, well-formedness constraints for nested terms must be taken into account, which is guaranteed by a constraint $g_i \rightarrow g$, 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 instantiating to true. Furthermore, $g_i \rightarrow g$, 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.

7.1.2 Seniority Constraints

Another set of Boolean constraints is called *seniority constraints*. The Boolean seniority constraints naturally follow from the definition of the seniority relation. 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 must include Boolean seniority constraints for nested terms.

Seniority constraints are defined as $\mathsf{SC}(t_1 \sqsubseteq t_2)$ for a constraint $t_1 \sqsubseteq t_2$ ($\mathsf{V}(t_1) = \mathsf{V}(t_2) = \emptyset$). $\mathsf{SC}(t_1 \sqsubseteq t_2)$ specifies a set of constraints that ensures the seniority relation $t_1 \sqsubseteq t_2$, where t_1 and t_2 are known to be well-formed terms, that is $\mathsf{SAT}(\mathsf{WFC}(t_1)) \neq \emptyset$ and $\mathsf{SAT}(\mathsf{WFC}(t_2)) \neq \emptyset$. In other words, $\mathsf{SAT}(\mathsf{WFC}(t_1) \cup \mathsf{WFC}(t_2) \cup \mathsf{SC}(t_1 \sqsubseteq t_2)) \neq \emptyset$ guarantees that the seniority relation $t_1 \sqsubseteq t_2$ holds.

If t_1 and t_2 are records, then for every element $\mathsf{l}^2_j(g^2_j)$: t^2_j in t_2 one of the following sets of Boolean constraints must be produced:

1. if there exists an element $\mathsf{l}^1_i(g^1_i)$: t^1_i in t_1 such that l^1_i is the same as l^2_j, then the Boolean seniority constraints must include the seniority constraints for nested subterms providing that $g^1_i \wedge g^2_j$;
2. otherwise, an element $\mathsf{l}^2_j(g^2_j)$: t^2_j must be excluded from the collection: the constraint $\neg g^2_j$ is generated.

The Boolean seniority constraints for choices are generated in a dual way.

The Boolean seniority constraints for switches (t_1 or t_2 is a switch) ensure only the Boolean seniority constraints for nested terms.

If $t_1 \sqsubseteq t_2$ is not matched by one of the above mentioned cases, then the constraint is unsatisfiable and false is generated.

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. $\mathsf{SC}(t_1 \sqsubseteq t_2) = \{g_i^1 \to g \mid 1 \le i \le d \text{ and } g \in \mathsf{SC}(t_i^1 \sqsubseteq t_i^2)\}$, 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. $\mathsf{SC}(t_1 \sqsubseteq t_2) = \{g_i^2 \to g \mid 1 \le i \le d \text{ and } g \in \mathsf{SC}(t_1 \sqsubseteq t_i^2)\}$, 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.

Figure 7.2: The set of Boolean constraints that ensures the seniority relation $t_1 \sqsubseteq t_2$

7.2 Iterative method

The CSP-WS algorithm takes advantage of the order-theoretical structure of the MDL and iteratively converges to a solution if one exists.

Definition 7.2.1 (Conditional approximation)**.** Let B be a set of Boolean constraints, and $\vec{a}^{\downarrow}$ and $\vec{a}^{\uparrow}$ be vectors of semi-ground terms such that $|\vec{a}^{\downarrow}| = |\mathsf{V}^{\downarrow}(\mathcal{C})|$ and $|\vec{a}^{\uparrow}| = |\mathsf{V}^{\uparrow}(\mathcal{C})|$. A tuple $(\mathsf{B}_i, \vec{a}_i^{\downarrow}, \vec{a}_i^{\uparrow})$ is called a *conditional approximation* and

$$(\mathsf{B}_0, \vec{a}_0^{\downarrow}, \vec{a}_0^{\uparrow}), \dots, (\mathsf{B}_{h-1}, \vec{a}_{h-1}^{\downarrow}, \vec{a}_{h-1}^{\uparrow}), (\mathsf{B}_s, \vec{a}_h^{\downarrow}, \vec{a}_h^{\uparrow}),$$

is called a *a series of conditional approximations* of a CSP-WS solution if for every $1 \le k \le h$,

$$\mathsf{B}_0 \subseteq \mathsf{B}_1 \subseteq \cdots \subseteq \mathsf{B}_k \subseteq \cdots \subseteq \mathsf{B}_h \tag{7.1}$$

and a vector of Boolean values $\vec{b}$ that is a solution to $\mathsf{SAT}(\mathsf{B}_k)$:

$$\vec{a}_k^{\downarrow}[\vec{f}/\vec{b}] \sqsubseteq \vec{a}_{k-1}^{\downarrow}[\vec{f}/\vec{b}] \qquad \text{and} \qquad \vec{a}_{k-1}^{\uparrow}[\vec{f}/\vec{b}] \sqsubseteq \vec{a}_k^{\uparrow}[\vec{f}/\vec{b}]. \tag{7.2}$$

The conditional approximation $(\mathsf{B}_k, \vec{a}_k^{\downarrow}, \vec{a}_k^{\uparrow})$ is called a *closer conditional approximation* of a solution to CSP-WS than $(\mathsf{B}_{k-1}, \vec{a}_{k-1}^{\downarrow}, \vec{a}_{k-1}^{\uparrow})$ if $\mathsf{B}_{k-1} \subseteq \mathsf{B}_k$, $\vec{a}_k^{\downarrow}[\vec{f}/\vec{b}] \sqsubseteq \vec{a}_{k-1}^{\downarrow}[\vec{f}/\vec{b}]$ and $\vec{a}_{k-1}^{\uparrow}[\vec{f}/\vec{b}] \sqsubseteq \vec{a}_k^{\uparrow}[\vec{f}/\vec{b}]$.

Similarly to the algorithm presented in Section 6.5, the core of the CSP-WS solution algorithm is the approximating function $\mathsf{IF} : \mathcal{C} \times \mathbb{P}(\mathcal{B}) \times \widetilde{\mathcal{T}}_m^{\downarrow} \times \widetilde{\mathcal{T}}_n^{\uparrow} \to \mathbb{P}(\mathcal{B}) \times \widetilde{\mathcal{T}}_m^{\downarrow} \times \widetilde{\mathcal{T}}_n^{\uparrow}$, which maps a constraint and the current approximation to the new approximation. The approximation is a tuple $(\mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow})$, where $\mathsf{B} \subseteq \mathcal{B}$ is a set of Boolean formulas, and $\vec{a}^{\downarrow} \in \widetilde{\mathcal{T}}_m^{\downarrow}$ and $\vec{a}^{\uparrow} \in \widetilde{\mathcal{T}}_n^{\uparrow}$ are vectors of down-coerced and up-coerced terms, $|\mathsf{V}^{\downarrow}(\mathcal{C})| = m$ and $|\mathsf{V}^{\uparrow}(\mathcal{C})| = n$, respectively. $(\emptyset, (\mathsf{nil}, \ldots, \mathsf{nil}), (\mathsf{none}, \ldots, \mathsf{none}))$ is the initial approximation. In contrast to the approximating function $\overline{\mathsf{IF}}$ from Section 6.3, IF maintains a set of approximations for all t-variables in the form of semi-ground terms (as opposed to $\overline{\mathsf{IF}}$, which stores t-variable values in the form of ground terms). The semi-ground term represent values for all instantiations of b-variables. The ground values are obtained by Boolean substitution. Following (7.2), IF must be a monotonic function.

7.2.1 Boolean Satisfiability

The set of Boolean constraints B potentially expands at every iteration of the algorithm by inclusion of further logic formulas called *assertions* into its conjunction as the algorithm processes constraints $\mathcal{C}$. The expansion may, in particular, be due to the term variables becoming bound to potentially ill-formed expressions but for the fact that those ill-formed expressions would be prohibited (in the sense of Eq. (7.2)) by the expanded SAT. Whether the set of Boolean constraints actually expands or not can be determined by checking the satisfiability of $\mathsf{SAT}(\mathsf{B}_k) \neq \mathsf{SAT}(\mathsf{B}_{k-1})$ for the current iteration k.

We argue below that if the original CSP-WS is satisfiable then so is $\mathsf{SAT}(\mathsf{B}_h)$. Furthermore, the tuple of vectors $(\vec{b_h}, \vec{a}_h^{\downarrow}[\vec{f}/\vec{b_h}], \vec{a}_h^{\uparrow}[\vec{f}/\vec{b_h}])$ is a solution to the former problem, where $\vec{b_h}$ is a solution for $\mathsf{SAT}(\mathsf{B}_h)$. In other words, the iterations terminate when the conditional approximation limits the term variables, and when the SAT constrains the b-variables enough to ensure the satisfaction of all CSP-WS constraints. In general, the set $\mathsf{SAT}(\mathsf{B}_h)$ can have more than one solution. The algorithm is flexible enough to generate all solutions that correspond to all $\mathsf{SAT}(\mathsf{B}_h)$ solutions. Alternatively, the algorithm can return one solution, which is obtained arbitrarily or using some heuristics (for example, a solution to $\mathsf{SAT}(\mathsf{B}_h)$ that maximises the number of false values could be preferred).

The presence of the SAT makes the algorithm NP-complete. On the other hand, it is possible to reduce the complexity of solving instances $\mathsf{B}_1, \ldots, \mathsf{B}_h$ by using an incremental SAT solver [ES03], because $\mathsf{B}_1 \subseteq \cdots \subseteq \mathsf{B}_h$.

7.3 Iterated function

In Sections 7.3.1 and 7.3.2, we specify the iterated function IF for all categories of terms. We use our conventional notation as follows. Let

$$\begin{aligned}\vec{v}^{\downarrow} &= (\overline{v}_1, \ldots, \overline{v}_m),\\ \vec{v}^{\uparrow} &= (\overline{\overline{v}}_1, \ldots, \overline{\overline{v}}_n),\\ \vec{a}^{\downarrow} &= (\overline{a}_1, \ldots, \overline{a}_m),\\ \vec{a}^{\uparrow} &= (\overline{\overline{a}}_1, \ldots, \overline{\overline{a}}_n).\end{aligned}$$

We introduce a *generalised iterated function*

$$\mathsf{GIF}(t_1 \sqsubseteq t_2, \mathsf{B}, b, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) : \mathcal{C} \times \mathbb{P}(\mathcal{B}) \times \widetilde{\mathcal{T}}_m^{\downarrow} \times \widetilde{\mathcal{T}}_n^{\uparrow} \times \mathcal{B} \to \mathbb{P}(\mathcal{B}) \times \widetilde{\mathcal{T}}_m^{\downarrow} \times \widetilde{\mathcal{T}}_n^{\uparrow}.$$

It is a more generic version of the iterated function $\overline{\mathsf{IF}}$. GIF is defined as follows:

$$\mathsf{GIF}(t_1 \sqsubseteq t_2, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (b \implies \mathsf{IF}(t_1 \sqsubseteq t_2, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow})).$$

The definition states that the new approximation given $(\mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow})$ for a constraint $t_1 \sqsubseteq t_2$ is constructed only under assumption b. For example,

$$\mathsf{GIF}(\{\mathsf{a}(x)\colon v_1^{\downarrow}\} \sqsubseteq \{\mathsf{a}(y)\colon v_2^{\downarrow}\}, \{\mathsf{true}\}, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow})$$

calls the helper approximation function for nested terms

$$\mathsf{GIF}(v_1^{\downarrow} \sqsubseteq v_2^{\downarrow}, \{x \wedge y\}, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}),$$

because $v_1^{\downarrow} \sqsubseteq v_2^{\downarrow}$ arises only if $x \wedge y$.

Furthermore, we can define IF using GIF:

$$\mathsf{IF}(t_1 \sqsubseteq t_2, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = \mathsf{GIF}(t_1 \sqsubseteq t_2, \{\mathsf{true}\}, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}).$$

Below we only define GIF. The definition of IF naturally arises from GIF.

7.3.1 Iterated Function for Constraints on Basic Terms and Tuples

B-variables are not present in a symbol or a tuple (for tuples, b-variables are present in nested terms though). Therefore, the definition of GIF for symbols and tuples is similar to the $\overline{\mathsf{IF}}$ definition for symbols and tuples (see Sections 6.3.1 and 6.3.2).

If t is a down-coerced term, then

$$\mathsf{GIF}(t \sqsubseteq \mathsf{nil}, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}).$$

If t is a symbol, then

$$\mathsf{GIF}(t \sqsubseteq t, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}).$$

If t is a down-coerced term and $\overline{v}_\ell$ is a down-coerced variable, then

$$\mathsf{GIF}(t \sqsubseteq \overline{v}_\ell, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = \mathsf{GIF}(t \sqsubseteq \overline{v}_\ell[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}], b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}).$$

If $\overline{\overline{v}}_\ell$ is an up-coerced variable and t is an up-coerced term, then

$$\mathsf{GIF}(\overline{\overline{v}}_\ell \sqsubseteq t, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = \mathsf{GIF}(\overline{\overline{v}}_\ell[\vec{v}^{\uparrow}/\vec{a}^{\uparrow}] \sqsubseteq t, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}).$$

If $\overline{v}_\ell$ is a down-coerced variable and t is a down-coerced term, then

$$\mathsf{GIF}(\overline{v}_\ell \sqsubseteq t, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\mathsf{B}, (\overline{a}_1, \ldots, \overline{a}_\ell \sqcap \langle (b)\colon t[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}], (\neg b)\colon \mathsf{nil}\rangle, \ldots, \overline{a}_m), \vec{a}^{\uparrow}).$$

If t is an up-coerced term and $\overline{\overline{v}}_\ell$ is an up-coerced variable, then

$$\mathsf{GIF}(t \sqsubseteq \overline{\overline{v}}_\ell, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\mathsf{B}, \vec{a}^{\downarrow}, (\overline{\overline{a}}_1, \ldots, \overline{\overline{a}}_\ell \sqcup \langle (b)\colon t[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}], (\neg b)\colon \mathsf{none}\rangle, \ldots, \overline{\overline{a}}_n)).$$

If t_1 is a tuple $(t_1^1 \ \ldots \ t_d^1)$ and t_2 is a tuple $(t_1^2 \ \ldots \ t_d^2)$, then

$$\mathsf{GIF}(t_1 \sqsubseteq t_2, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\bigcup_{1 \le i \le d} \mathsf{B}_i, \prod_{1 \le i \le d} \vec{a}_i^{\downarrow}, \bigsqcup_{1 \le i \le d} \vec{a}_i^{\uparrow}),$$

where

$$(\mathsf{B}_i, \vec{a}_i^{\downarrow}, \vec{a}_i^{\uparrow}) = \mathsf{GIF}(t_1 \sqsubseteq t_2, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}).$$

If t_1 or t_2 is a symbol or a tuple that is not matched by one of these cases, then $t_1 \sqsubseteq t_2$ does not have a solution.

7.3.2 Iterated Function for Constraints on Records

If t_1 and t_2 are records $\{\mathsf{l}_1^1(b_1^1)\colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1(b_d^1)\colon t_d^1\}$ and $\{\mathsf{l}_1^2(b_1^2)\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2(b_e^2)\colon t_e^2\}$ (no tail variables are present) respectively, then for all i ($1 \le i \le e$), four cases must be considered:

1. if there exists j, such that $\mathsf{l}_\mathsf{j}^1 = \mathsf{l}_\mathsf{i}^2$, and

(a) $b_j^1 \wedge b_i^2 = \mathsf{true}$, then $t_j^1 \sqsubseteq t_i^2$ must hold;

(b) $b_j^1 = \mathsf{false}$ and $b_i^2 = \mathsf{true}$, then the solution does not exist;

(c) $b_i^2 = \mathsf{false}$, then the solution exists and no further approximation is needed.

2. otherwise, the solution does not exist.

Therefore, GIF is defined as follows:

$$\mathsf{GIF}(\{\mathsf{l}_1^1(b_1^1)\colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1(b_d^1)\colon t_d^1\} \sqsubseteq \{\mathsf{l}_1^2(b_1^2)\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2(b_e^2)\colon t_e^2\}, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = \\ = (\bigcup_{1\le i\le e} \mathsf{B}_i, \prod_{1\le i\le e} \vec{a}_i^{\downarrow}, \bigsqcup_{1\le i\le e} \vec{a}_i^{\uparrow}),$$

where

$$(\mathsf{B}_i, \vec{a}_i^{\downarrow}, \vec{a}_i^{\uparrow}) = \begin{cases} \mathsf{GIF}(t_j^1 \sqsubseteq \langle (b_j^1 \wedge b_i^2)\colon t_i^2, (\neg(b_j^1 \wedge b_i^2))\colon t_j^1\rangle, b, \mathsf{B}', \vec{a}_i^{\downarrow}, \vec{a}_i^{\uparrow}) & \text{if } \exists j : \mathsf{l}_\mathsf{j}^1 = \mathsf{l}_\mathsf{i}^2 \quad (7.3) \\ (\mathsf{B} \cup \{\neg b\}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) & \text{otherwise.} \quad (7.4) \end{cases}$$

and

$$\mathsf{B}' = \mathsf{B} \cup \{b \to (b_i^2 \to b_j^1)\}.$$

(7.3) covers all three cases that correspond to $\mathsf{l}_\mathsf{j}^1 = \mathsf{l}_\mathsf{i}^2$. This is achieved by reduction to a constraint that contains a switch term:

1. if $b_j^1 \wedge b_i^2$, then the constraint $t_j^1 \sqsubseteq t_i^2$ is solved;
2. if $\neg b_j^1 \wedge b_i^2$, then $b_i^2 \to b_j^1$ evaluates to false given b. In this case the constraint does not have a solution;
3. otherwise, the constraint $t_j^1 \sqsubseteq t_j^1$ is solved. The constraint always is satisfied and is introduced only to acknowledge satisfiability of the original constraint.

If there is no element in t_1 that has the label equal to l_i^2 from t_2, then the constraint is unsatisfiable and the assertion false is added to B (see (7.4)).

Example 7.3.1.

$$\mathsf{GIF}(\{\mathsf{a}(x)\colon \mathtt{int}, \mathsf{b}(y)\colon \mathtt{int}\} \sqsubseteq \{\mathsf{b}(z)\colon \mathtt{int}, \mathsf{c}(w)\colon \mathtt{int}\}, u, B, (), ()) = \\ (B \cup \{u \to (z \to y), u \to \neg w\}, (), ())$$

△

If $\overline{v}_\ell$ is a down-coerced variable, and t_1 and t_2 are records $\{\mathsf{l}_1^1(b_1^1)\colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1(b_d^1)\colon t_d^1 \,|\overline{v}_\ell\}$ and $\{\mathsf{l}_1^2(b_1^2)\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2(b_e^2)\colon t_e^2\}$ respectively, then for every element $\mathsf{l}_\mathsf{i}^2(b_i^2)\colon t_i^2$ from t_2 one of the following must hold:

1. there exists an element $\mathsf{l}^1_{\mathsf{j}}(b^1_j)\colon t^1_j$ in t_1 such that $l^1_j = l^2_i$ and
   (a) if $b^1_j \wedge b^2_i = \mathsf{true}$, then $t^1_j \sqsubseteq t^2_i$ must hold;
   (b) if $b^1_j = \mathsf{false}$ and $b^2_i = \mathsf{true}$, then the seniority relation $\overline{v}_\ell \sqsubseteq \{\mathsf{l}^2_{\mathsf{i}}\colon t^2_i\}$ must hold, and therefore, $\overline{v}$ must be coerced to $\{\mathsf{l}^2_{\mathsf{i}}(\neg b^1_j \wedge b^2_i)\colon t^2_i\}$ (if it is impossible, the solution does not exist);
   (c) if $b^2_i = \mathsf{false}$, then the solution exists and no further approximation is needed.
2. otherwise, $\overline{v}_\ell$ must be coerced to $\{\mathsf{l}^2_{\mathsf{i}}(b^2_i)\colon t^2_i\}$.

Therefore, for these cases GIF is defined as follows:

$$\mathsf{GIF}(\{\mathsf{l}^1_1(b^1_1)\colon t^1_1, \ldots, \mathsf{l}^1_{\mathsf{d}}(b^1_d)\colon t^1_d \,|\overline{v}_\ell\} \sqsubseteq \{\mathsf{l}^2_1(b^2_1)\colon t^2_1, \ldots, \mathsf{l}^2_{\mathsf{e}}(b^2_e)\colon t^2_e\}, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = \\ (\bigcup_{1\le i\le e} \mathsf{B}_i, \prod_{1\le i\le e} \vec{a}_i^{\downarrow}, \bigsqcup_{1\le i\le e} \vec{a}_i^{\uparrow}),$$

where

$$(\mathsf{B}_i, \vec{a}_i^{\downarrow}, \vec{a}_i^{\uparrow}) =$$

$$= \begin{cases} \mathsf{GIF}(t^1_j \sqsubseteq \langle (b^1_j \wedge b^2_i)\colon t^2_i, (\neg(b^1_j \wedge b^2_i))\colon t^1_j\rangle, b, \mathsf{B}, \vec{a}_i'^{\downarrow}, \vec{a}_i^{\uparrow}) & \text{if } \exists j : \mathsf{l}^1_{\mathsf{j}} = \mathsf{l}^2_{\mathsf{i}} \quad (7.5) \\ (\mathsf{B}, (\overline{a}_1, \ldots, \overline{a}_\ell \sqcup \{\mathsf{l}^2_{\mathsf{i}}(b \wedge b^2_i)\colon t^2_i[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}]\}, \ldots, \overline{a}_n), \vec{a}^{\uparrow}) & \text{otherwise,} \quad (7.6) \end{cases}$$

and

$$\vec{a}_i'^{\downarrow} = (\overline{a}_1, \ldots, \overline{a}_\ell \sqcup \{\mathsf{l}^2_{\mathsf{i}}(b \wedge \neg b^1_j \wedge b^2_i)\colon t^2_i[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}]\}, \ldots, \overline{a}_n).$$

Example 7.3.2.

$$\mathsf{GIF}(\{\mathsf{a}(x)\colon \mathtt{int} \,|v^{\downarrow}\} \sqsubseteq \{\mathsf{a}(y)\colon \mathtt{int}\}, z, B, (\mathsf{nil}), ()) = (B, (\{\mathsf{a}(z \to (y \to \neg x))\colon \mathtt{int}\}), ())$$

Here, nil and $\{\mathsf{a}(z \to (y \to \neg x))\colon \mathtt{int}\}$ are the old and the new approximations for $v^{\downarrow}$, respectively. △

If t_1 is a record $\{\mathsf{l}^1_1(b^1_1)\colon t^1_1, \ldots, \mathsf{l}^1_{\mathsf{d}}(b^1_d)\colon t^1_d\}$ or $\{\mathsf{l}^1_1(b^1_1)\colon t^1_1, \ldots, \mathsf{l}^1_{\mathsf{d}}(b^1_d)\colon t^1_d \,|\overline{v}_\ell\}$ and t_2 is a record $\{\mathsf{l}^2_1(b^2_1)\colon t^2_1, \ldots, \mathsf{l}^2_{\mathsf{e}}(b^2_e)\colon t^2_e \,|\overline{v}_r\}$, then the constraint can by substitution be reduced to the previous cases for records:

$$\mathsf{GIF}(t_1 \sqsubseteq t_2, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = \mathsf{GIF}(t_1 \sqsubseteq t_2[\overline{v}_r/\overline{a}_r], b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}).$$

7.3.3 Iterated Function for Constraints on Choices

Definition of GIF for choices is dual to the definition of GIF for records.

If t_1 and t_2 are choices $(:\mathsf{l}_1^1(b_1^1)\colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1(b_d^1)\colon t_d^1:)$ and $(:\mathsf{l}_1^2(b_1^2)\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2(b_e^2)\colon t_e^2:)$ (no tail variables are present) respectively, then for all i $(1 \leq i \leq d)$, four cases must be considered:

1. if there exists j, such that $\mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2$, and
   (a) $b_i^1 \wedge b_j^2 = \mathsf{true}$, then $t_i^1 \sqsubseteq t_j^2$ must hold;
   (b) $b_i^1 = \mathsf{false}$ and $b_j^2 = \mathsf{true}$, then the solution does not exist;
   (c) $b_i^1 = \mathsf{true}$, then the solution exists and no further approximation is needed.
2. otherwise, the solution does not exist.

Therefore, GIF is defined as follows:

$$\mathsf{GIF}((:\mathsf{l}_1^1(b_1^1)\colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1(b_d^1)\colon t_d^1:) \sqsubseteq (:\mathsf{l}_1^2(b_1^2)\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2(b_e^2)\colon t_e^2:), b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = \\ = (\bigcup_{1\leq i\leq d} \mathsf{B}_i, \prod_{1\leq i\leq d} \vec{a}_i^{\downarrow}, \bigsqcup_{1\leq i\leq d} \vec{a}_i^{\uparrow}),$$

where

$$(\mathsf{B}_i, \vec{a}_i^{\downarrow}, \vec{a}_i^{\uparrow}) =$$

$$\begin{cases} \mathsf{GIF}(t_i^1 \sqsubseteq \langle (b_i^1 \wedge b_j^2)\colon t_j^2, (\neg(b_i^1 \wedge b_j^2))\colon t_i^1 \rangle, b, \mathsf{B}', \vec{a}_j^{\downarrow}, \vec{a}_j^{\uparrow}) & \text{if } \exists i : \mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2 \quad (7.7) \\ (\mathsf{B} \cup \{\neg b\}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) & \text{otherwise.} \quad (7.8) \end{cases}$$

and

$$\mathsf{B}' = \mathsf{B} \cup \{b \to (b_i^1 \to b_j^2)\}.$$

(7.7) covers all three cases that correspond to $\mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2$. This is achieved by reduction to a constraint that contains a switch term:

1. if $b_i^1 \wedge b_j^2$, then the constraint $t_i^1 \sqsubseteq t_j^2$ is solved;
2. if $\neg b_j^2 \wedge b_i^1$, then $b_i^1 \to b_j^2$ evaluates to false given b. In this case the constraint does not have a solution;
3. otherwise, the constraint $t_i^1 \sqsubseteq t_i^1$ is solved. The constraint always is satisfied and is introduced only to acknowledge satisfiability of the original constraint.

If there is no element in t_2 that has the label equal to l_i^1 from t_1, then the constraint is unsatisfiable and the assertion false is added to B (see (7.8)).

Example 7.3.3.

$$\mathsf{GIF}((:\mathtt{a}(x)\colon \mathtt{int}:) \sqsubseteq (:\mathtt{a}(y)\colon \mathtt{int}, \mathtt{b}(z)\colon \mathtt{int}:), u, B, (), ()) = \\ (B \cup \{u \to (x \to y), u \to \neg z\}, (), ())$$

△

If $\overline{\overline{v}}_\ell$ is an up-coerced variable, and t_1 and t_2 are choices $(:\mathsf{l}_1^1(b_1^1)\colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1(b_d^1)\colon t_d^1:)$ and $(:\mathsf{l}_1^2(b_1^2)\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2(b_e^2)\colon t_e^2 \,|\overline{\overline{v}}_\ell:)$ respectively, then for every element $\mathsf{l}_\mathsf{i}^1(b_i^1)\colon t_i^1$ from t_1 one of the following must hold:

1. there exists an element $\mathsf{l}_\mathsf{j}^2(b_j^2)\colon t_j^2$ in t_2 such that $l_i^1 = l_j^2$ and
   (a) if $b_i^1 \wedge b_j^2 = \mathsf{true}$, then $t_i^1 \sqsubseteq t_j^2$ must hold;
   (b) if $b_j^2 = \mathsf{false}$ and $b_i^1 = \mathsf{true}$, then the seniority relation $\{\mathsf{l}_\mathsf{i}^1\colon t_i^1\}\overline{\overline{v}}_\ell$ must hold, and therefore, $\overline{\overline{v}}$ must be coerced to $\{\mathsf{l}_\mathsf{i}^1(\neg b_j^2 \wedge b_i^1)\colon t_i^1\}$ (if it is impossible, the solution does not exist);
   (c) if $b_i^1 = \mathsf{false}$, then the solution exists and no further approximation is needed.
2. otherwise, $\overline{\overline{v}}_\ell$ must be coerced to $\{\mathsf{l}_\mathsf{i}^1(b_i^1)\colon t_i^1\}$.

Therefore, for these cases GIF is defined as follows:

$$\mathsf{GIF}((:\mathsf{l}_1^1(b_1^1)\colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1(b_d^1)\colon t_d^1:) \sqsubseteq (:\mathsf{l}_1^2(b_1^2)\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2(b_e^2)\colon t_e^2 \,|\overline{\overline{v}}_\ell:), b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = \\ (\bigcup_{1\le i\le p} \mathsf{B}_i, \prod_{1\le i\le p} \vec{a}_i^{\downarrow}, \bigsqcup_{1\le i\le p} \vec{a}_i^{\uparrow}),$$

where

$$(\mathsf{B}_i, \vec{a}_i^{\downarrow}, \vec{a}_i^{\uparrow}) =$$

$$\begin{cases} \mathsf{GIF}(\langle (b_i^1 \wedge b_j^2)\colon t_i^1, (\neg(b_j^2 \wedge b_i^1))\colon t_j^2\rangle \sqsubseteq t_j^2, b, \mathsf{B}, \vec{a}_i^{\downarrow}, \vec{a}_i'^{\uparrow}) & \text{if } \exists j : \mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2 \quad (7.9) \\ (\mathsf{B}, \vec{a}^{\downarrow}, (\overline{\overline{a}}_1, \ldots, \overline{\overline{a}}_\ell \sqcap (:\mathsf{l}_\mathsf{i}^1(b \wedge b_i^1)\colon t_i^1[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}]:), \ldots, \overline{\overline{a}}_n)) & \text{otherwise,} \quad (7.10) \end{cases}$$

and

$$\vec{a}_i'^{\uparrow} = (\overline{\overline{a}}_1, \ldots, \overline{\overline{a}}_\ell \sqcap (:\mathsf{l}_\mathsf{i}^1(b \wedge \neg b_j^2 \wedge b_i^1)\colon t_i^1[\vec{v}^{\downarrow}/\vec{a}^{\downarrow}, \vec{v}^{\uparrow}/\vec{a}^{\uparrow}]:), \ldots, \overline{\overline{a}}_n).$$

Example 7.3.4.

$$\mathsf{GIF}((:\mathsf{a}(x)\colon \mathtt{int}:) \sqsubseteq (:\mathsf{a}(y)\colon \mathtt{int} \,|v^{\uparrow}:), z, B, (), (\mathsf{none})) = \\ (B, (), ((:\mathsf{a}(z \to (x \to \neg y))\colon \mathtt{int}:)))$$

Here, none and $(:\mathsf{a}(z \to (x \to \neg y))\colon \mathtt{int}:)$ are the old and the new approximations for $v^{\uparrow}$, respectively. △

If t_1 is a choice $(:\mathsf{l}_1^1(b_1^1)\colon t_1^1, \ldots, \mathsf{l}_\mathsf{d}^1(b_d^1)\colon t_d^1 \,|\overline{\overline{u}}_\ell:)$ and t_2 is a choice $(:\mathsf{l}_1^2(b_1^2)\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2(b_e^2)\colon t_e^2:)$ or $(:\mathsf{l}_1^2(b_1^2)\colon t_1^2, \ldots, \mathsf{l}_\mathsf{e}^2(b_e^2)\colon t_e^2 \,|\overline{\overline{v}}_r:)$, then the constraint can by substitution be reduced to the previous cases for choices:

$$\mathsf{GIF}(t_1 \sqsubseteq t_2, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = \mathsf{GIF}(t_1[\overline{\overline{u}}_\ell/\overline{\overline{a}}_\ell] \sqsubseteq t_2, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}).$$

7.3.4 Iterated Function for Constraints on Switches

If t_1 is a switch $\langle (b_1^1)\colon t_1^1, \ldots, (b_d^1)\colon t_d^1 \rangle$, then the constraint is naturally reduced to a set of constraints without a switch term:

$$\mathsf{GIF}(t_1 \sqsubseteq t_2, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\bigcup_{1 \le i \le d} \mathsf{B}_i, \prod_{1 \le i \le d} \vec{a}_i^{\downarrow}, \bigsqcup_{1 \le i \le d} \vec{a}_i^{\uparrow}),$$

where

$$(\mathsf{B}_i, \vec{a}_i^{\downarrow}, \vec{a}_i^{\uparrow}) = \mathsf{GIF}(t_i^1 \sqsubseteq t_2, b \wedge b_i^1, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}).$$

Similarly, if t_2 is a switch $\langle (b_1^2)\colon t_1^2, \ldots, (b_e^2)\colon t_e^2 \rangle$, then the constraint is naturally reduced to a set of constraints without a switch term:

$$\mathsf{GIF}(t_1 \sqsubseteq t_2, b, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) = (\bigcup_{1 \le i \le e} \mathsf{B}_i, \prod_{1 \le i \le e} \vec{a}_i^{\downarrow}, \bigsqcup_{1 \le i \le e} \vec{a}_i^{\uparrow}),$$

where

$$(\mathsf{B}_i, \vec{a}_i^{\downarrow}, \vec{a}_i^{\uparrow}) = \mathsf{GIF}(t^1 \sqsubseteq t_i^2, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b \wedge b_i^2).$$

7.4 Algorithm Decomposition

The algorithm that recursively solves the constraints in the CSP-WS consists of functions that can be represented in a dependency graph as follows (see Figure 7.3):

1. CSP-WS($\mathcal{C}$) is the top-level function. Formally, the function is defined in Algorithm 2. It receives a set of constraints $\mathcal{C}$ as an input and then performs the following steps:

   (a) it initialises $(\mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow})$ with the most general approximation as specified in Section 7.2;

   (b) it iteratively calls SOLVE$(\ldots)$ function (which essentially represents GIF function from Section 7.3) that computes the next conditional approximation until the fixed point is reached;

   (c) it solves a SAT problem for constraints in B, substitutes b-variables with corresponding Boolean values in term variables and returns ground terms and Boolean values as a solution to $\mathcal{C}$.

2. Given a conditional approximation $(\mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow})$, a Boolean condition b, and terms t_1 and t_2, the function SOLVE$(t_1, t_2, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b)$ (Algorithm 3) computes a new conditional approximation that satisfies the constraint $t_1 \sqsubseteq t_2$ under the condition

b. The algorithm finds a match for (t_1, t_2) over all pairs of term categories and recursively calls SOLVE$(\dots)$ for the subterms of t_1 and t_2. Table 7.1 refers to the lines in Algorithm 3 where the solution algorithms for particular category pairs are explained.

3. We define functions that solve the constraint $t_1 \sqsubseteq t_2$ for specific categories of terms. The functions have identical signature: they receive the current approximation, a Boolean condition b, a pair of terms as an input, and produce an approximation that satisfies the constraint $t_1 \sqsubseteq t_2$ if one exists. If the constraint cannot be satisfied, the algorithm adds the Boolean constraint $\neg b$ to B.

   - If t_1 is an arbitrary term and t_2 is a switch, SOLVETERMSWITCH$(\dots)$ is called (Algorithm 7).
   - If t_1 is a switch and t_2 is an arbitrary term, SOLVESWITCHTERM$(\dots)$ is called (Algorithm 8).
   - If t_1 and t_2 are both tuples, SOLVETT$(\dots)$ is called (Algorithm 4).
   - If t_1 and t_2 are both records, SOLVERR$(\dots)$ is called (Algorithm 5).
   - If t_1 and t_2 are both choices, SOLVECC$(\dots)$ is called (Algorithm 6).

4. COMBINERECORDS (Algorithm 9) and COMBINECHOICES are functions that merge two records or two choices. The functions combine elements with non-intersecting labels into a new collection (either a record or a choice). If elements with equal labels are present in both collections, the functions produce two alternative collections: in one collection a guard and a term for an element with the conflicting label is taken from the first collection and in another one — from the second collection. As a result, the functions produce a switch term that contains alternative collections of merged elements.
5. SET$(v, t, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b)$ is an auxiliary function (Algorithm 11) that is called from the above given functions. It produces a new approximation from the current approximation $(\mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow})$. t is a value as a semi-ground term for a variable v that must hold when b is true.
6. SGROUND$(v, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b)$ is a function (Algorithm 10) that retrieves a value for a variable v from vectors $\vec{a}^{\downarrow}$ and $\vec{a}^{\uparrow}$ given a condition b.

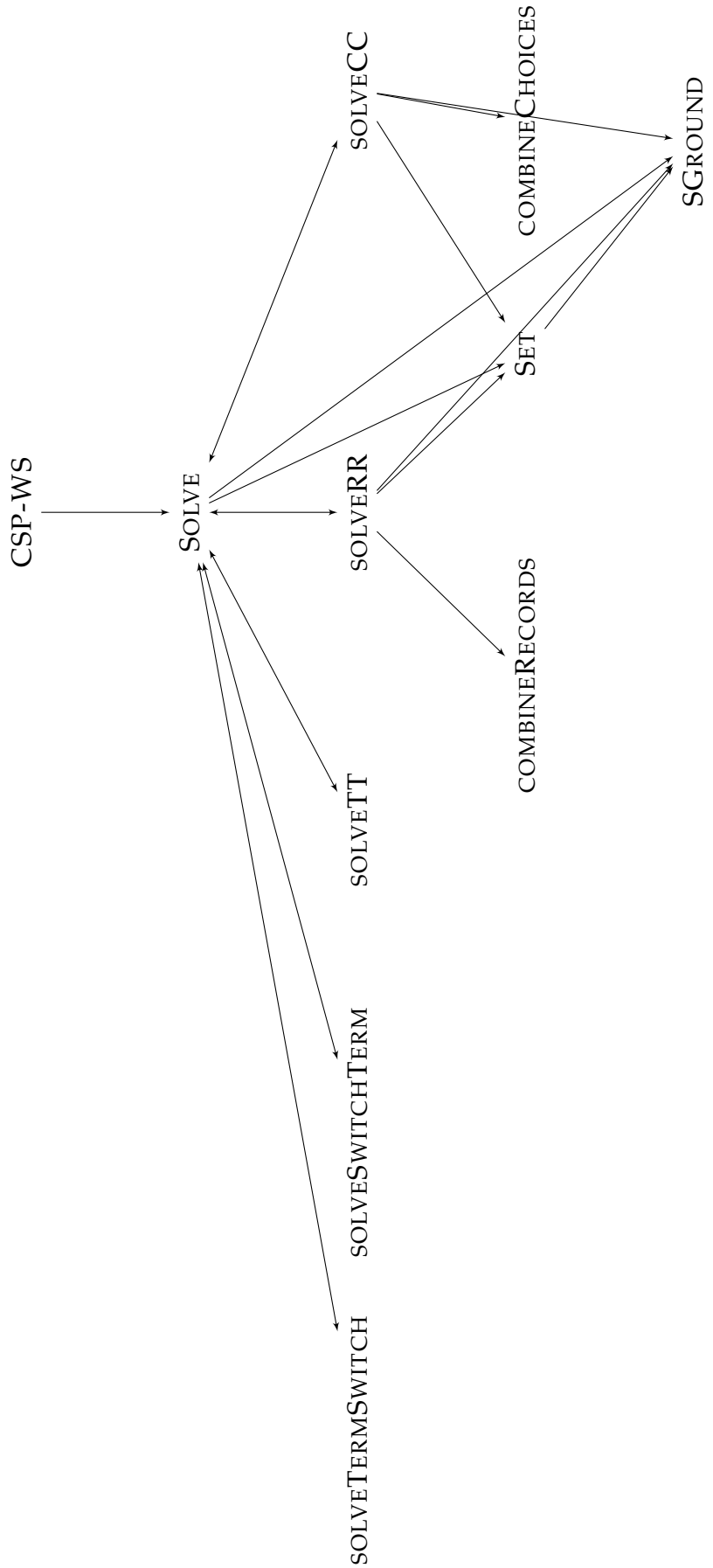


Figure 7.3: A dependency graph of functions that are used in the CSP-WS algorithm

7.5 CSP-WS Algorithm

The top-level Algorithm 2 iteratively finds conditional approximations for a set of constraints $\mathcal{C}$ until the fixed point is reached. It performs the following steps.

Algorithm 2 CSP-WS($\mathcal{C}$)

1: $c \leftarrow |\mathcal{C}|$
2: $i \leftarrow 0$
3: $\mathsf{B}_0 \leftarrow \emptyset$
4: $\vec{a}_0^{\downarrow} \leftarrow (\mathsf{nil}, \ldots, \mathsf{nil})$
5: $\vec{a}_0^{\uparrow} \leftarrow (\mathsf{none}, \ldots, \mathsf{none})$
6: **repeat**
7: **for** $1 \leq j \leq c :\ t_1^j \sqsubseteq t_2^j \in \mathcal{C}$ **do**
8: $(\mathsf{B}_{i\cdot c+j}, \vec{a}_{i\cdot c+j}^{\downarrow}, \vec{a}_{i\cdot c+j}^{\uparrow}) \leftarrow \textsc{Solve}(t_1^j, t_2^j, \mathsf{B}_{i\cdot c+j-1}, \vec{a}_{i\cdot c+j-1}^{\downarrow}, \vec{a}_{i\cdot c+j-1}^{\uparrow}, \mathsf{true})$
9: **end for**
10: $i \leftarrow i + 1$
11: **until** $(\mathsf{SAT}(\mathsf{B}_{i\cdot c}), \vec{a}_{i\cdot c}^{\downarrow}, \vec{a}_{i\cdot c}^{\uparrow}) = (\mathsf{SAT}(\mathsf{B}_{(i-1)\cdot c}), \vec{a}_{(i-1)\cdot c}^{\downarrow}, \vec{a}_{(i-1)\cdot c}^{\uparrow})$
12: **if** $\mathsf{B}_{i\cdot c}$ is unsatisfiable **then**
13: **return** Unsat
14: **else**
15: **return** $(\mathsf{SATSol}(\mathsf{B}_{i\cdot c}), \vec{a}_{i\cdot c}^{\downarrow}[\vec{f}/\vec{b}], \vec{a}_{i\cdot c}^{\uparrow}[\vec{f}/\vec{b}])$
16: **end if**

First, the algorithm initialises $(\mathsf{B}_0, \vec{a}_0^{\downarrow}, \vec{a}_0^{\uparrow})$ with the most general approximation $(\emptyset, (\mathsf{nil}, \ldots, \mathsf{nil}), (\mathsf{none}, \ldots, \mathsf{none}))$. Next, the algorithm iterates over the constraints in $\mathcal{C}$ and constructs a new tuple $(\mathsf{B}_i, \vec{a}_i^{\downarrow}, \vec{a}_i^{\uparrow})$ that is a closer approximation of the solution than $(\mathsf{B}_{i-1}, \vec{a}_{i-1}^{\downarrow}, \vec{a}_{i-1}^{\uparrow})$.

The function $\textsc{Solve}(\ldots)$ solves the constraint $t_1^j \sqsubseteq t_2^j$ and updates the vectors $\vec{a}_{i\cdot c+j}^{\downarrow}$ and $\vec{a}_{i\cdot c+j}^{\uparrow}$ with new values. Furthermore, it adds Boolean assertions that ensure 1) satisfaction of the constraint for any $\vec{b} \in \mathsf{SAT}(\mathsf{B}_{i\cdot c+j})$, and 2) well-formedness of the terms occurring in it (as described in Section 7.1). Some approximations require new Boolean variables to be generated, however, only a finite number of the variables can be generated (which is explained in more details in Section 7.6). They are kept tracked using a hash table: a new variable is generated only if another variable for the same condition was not generated before. The algorithm terminates if $\mathsf{B}_{i\cdot c} \equiv \mathsf{B}_{(i-1)\cdot c}$, $\vec{a}_{i\cdot c}^{\downarrow} = \vec{a}_{(i-1)\cdot c}^{\downarrow}$ and $\vec{a}_{i\cdot c}^{\uparrow} = \vec{a}_{(i-1)\cdot c}^{\uparrow}$.

The SOLVE(...) function

The SOLVE(...) function is called from the top-level function (Line 8 in Algorithm 2) of the algorithm. SOLVE(...) finds the next conditional approximation given the current approximation $(\mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow})$ and a Boolean condition b. b specifies assignments for b-variables when the constraint holds. If b is true, then the constraint holds for any assignment.

SOLVE(...) matches the given constraint against all possible term category pairs (see Algorithm 3). Table 7.1 demonstrates which pairs of term categories may or may not have a solution. In the case a solution does not exist, the constraint $\neg b$ is added to B (Line 26 of Algorithm 3).

The cases in Lines 2 to 14 cover the basic constraints for symbols and tuples (see Section 7.3.1). Lines 16 to 17 represent the constraint for records from Section 7.3.2 and Lines 18 to 19 represent the constraint for choices from Section 7.3.3. Finally, Lines 20 to 25 cover the constraints that include switch terms (see Section 7.3.4). Next we discuss these cases individually.

	N	**DV**	**S**	**T**	**R**	**UV**	**C**	**SW**
N	2	6	✗	14	16	✗	✗	23
DV	2	8	8	8	8	✗	✗	23
S	2	6	4	✗	✗	✗	✗	23
T	2	6	✗	14	✗	✗	✗	23
R	2	6	✗	✗	16	✗	✗	23
UV	✗	✗	✗	✗	✗	12	10	23
C	✗	✗	✗	✗	✗	12	18	23
SW	20	20	20	20	20	20	20	20

(**UV**) up-coerced variable (**N**) nil (**R**) record (**T**) tuple
(**DV**) down-coerced variable (**S**) symbol (**C**) choice (**SW**) switch
relation for down-coerced terms relation for up-coerced terms

Table 7.1: Combinations of all term categories that can constitute a constraint $t \sqsubseteq t'$. A number in the table refers to the line in Algorithm 3 that matches the constraint. Invalid combinations are marked with ✗.

SOLVETT(...) function is presented in Algorithm 4. It solves the CSP-WS for tuples as described in Section 7.3.1. The function recursively calls SOLVE(...) for tuple elements.

Algorithm 5 describes the SOLVERR(...) function. It solves the CSP-WS for records as described in Section 7.3.2. In the most general case, records, which are provided as arguments, may contain the tail variables $v_1^{\downarrow}$ and $v_2^{\downarrow}$. First, the function computes the tightest approximation for $v_2^{\downarrow}$ (Lines 1 to 6) in the form of a switch term s. All elements $\tilde{t}_i$ of s must be records as well. Otherwise, a Boolean constraint is added to B

Algorithm 3 SOLVE$(t_1, t_2, \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b)$

1: **match** (t_1, t_2) **with**
2: **case** (t_1, nil), where t_1 is a down-coerced variable, a symbol, a tuple or a record
3: **return** $(\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow})$
4: **case** (s, s'), where s and s' are symbols and $s = s'$
5: **return** $(\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow})$
6: **case** $(t, v^{\downarrow})$, where t is a symbol, a tuple or a record
7: **return** SOLVE$(t, \text{SGROUND}(v^{\downarrow}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b), \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b)$
8: **case** $(v^{\downarrow}, t)$, where t is a down-coerced variable, a symbol, a tuple or a record
9: **return** SET$(v^{\downarrow}, \text{SGROUND}(t, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b), \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b)$
10: **case** $(v^{\uparrow}, t)$, where t is a choice
11: **return** SOLVE$(\text{SGROUND}(\vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b, v^{\uparrow}), t, \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b)$
12: **case** $(t, v^{\uparrow})$, where t is an up-coerced variable or a choice
13: **return** SET$(v^{\uparrow}, \text{SGROUND}(t, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b), \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b)$
14: **case** $((t_1^1 \ \dots \ t_1^n), (t_2^1 \ \dots \ t_2^n))$
15: **return** SOLVETT$((t_1^1 \ \dots \ t_1^n), (t_2^1 \ \dots \ t_2^n), \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b)$
16: **case** $(\{\mathsf{l}_1^1(g_1^1)\colon t_1^1, \dots, \mathsf{l}_1^n(g_1^n)\colon t_1^n \mid v_1^{\downarrow}\}, \{\mathsf{l}_2^1(g_2^1)\colon t_2^1, \dots, \mathsf{l}_2^m(g_2^m)\colon t_2^m \mid v_2^{\downarrow}\})$, where tail variables $v_1^{\downarrow}$ or $v_2^{\downarrow}$ may be absent
17: **return** SOLVERR$(t_1, t_2, \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b)$
18: **case** $((:\mathsf{l}_1^1(g_1^1)\colon t_1^1, \dots, \mathsf{l}_1^n(g_1^n)\colon t_1^n \mid v_1^{\downarrow}:), (:\mathsf{l}_2^1(g_2^1)\colon t_2^1, \dots, \mathsf{l}_2^m(g_2^m)\colon t_2^m \mid v_2^{\downarrow}:))$, where tail variables $v_1^{\uparrow}$ or $v_2^{\uparrow}$ may be absent
19: **return** SOLVECC$(t_1, t_2, \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b)$
20: **case** $(\langle (g_1^1)\colon t_1^1, \dots, (g_1^n)\colon t_1^n \rangle, t_2)$
21: $\mathsf{B} \leftarrow \mathsf{B} \cup \{b \to (g_1^1 \vee \dots \vee g_1^n)\} \cup \{b \to \neg(g_1^i \wedge g_1^j) | 1 \le i, j \le n, i \ne j\}$
22: **return** SOLVETERMSWITCH$(\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b, \langle (g_1^1)\colon t_1^1, \dots, (g_1^n)\colon t_1^n \rangle, t_2)$
23: **case** $(t_1, \langle (g_2^1)\colon t_2^1, \dots, (g_2^n)\colon t_2^n \rangle)$
24: $\mathsf{B} \leftarrow \mathsf{B} \cup \{b \to (g_2^1 \vee \dots \vee g_2^n)\} \cup \{b \to \neg(g_2^i \wedge g_2^j) | 1 \le i, j \le n, i \ne j\}$
25: **return** SOLVESWITCHTERM$(\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b, t, \langle (g_2^1)\colon t_2^1, \dots, (g_2^n)\colon t_2^n \rangle)$
26: **return** $(\mathsf{B} \cup \{\neg b\}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow})$

(see Line 28). Next, the elements of each valid record $\tilde{t}_i$ in s are merged with elements $\mathsf{l}_2^1(g_2^1)\colon t_2^1, \dots, \mathsf{l}_2^m(g_2^m)\colon t_2^m$. The result is produced as a switch of records (similarly to Line 10). The steps above find a value for the tail variable from the record in the right part of the constraint. The final step reduces the CSP-WS for records to a problem for subterms and approximates the value for $v_1^{\downarrow}$ as follows: if, for each element $\hat{\mathsf{l}}_{\mathsf{xy}}(\hat{g}_{xy})\colon \hat{t}_{xy}$, no element with the same label $\hat{l}_{xy}$ exists in the left record $\{\mathsf{l}_1^1(g_1^1)\colon t_1^1, \dots, \mathsf{l}_1^n(g_1^n)\colon t_1^n \mid v_1^{\downarrow}\}$, then $v_1^{\downarrow}$ must be down-coerced to $\{\hat{\mathsf{l}}_{\mathsf{xy}}(\hat{g}_{xy})\colon \hat{t}_{xy}\}$. If the element with the same label does not exist in the left record and the record lacks the tail variable $v_1^{\downarrow}$, then the constraint is unsatisfiable and the Boolean constraint in Line 22 is generated.

Algorithm 6 describes the function SOLVECC$(\dots)$, which is structured similarly to

Algorithm 4 SOLVETT$\left(\left(t_1^1,\ldots,t_1^n\right),\left(t_2^1,\ldots,t_2^n\right)\mathsf{B},\vec{a}^{\uparrow},\vec{a}^{\downarrow},b\right)$

1: $(\mathsf{B}_0,\vec{a}_0^{\uparrow},\vec{a}_0^{\downarrow}) \leftarrow (\mathsf{B},\vec{a}^{\uparrow},\vec{a}^{\downarrow})$
2: **for** $1 \leq i \leq n$ **do**
3: $\quad(\mathsf{B}_i,\vec{a}_i^{\uparrow},\vec{a}_i^{\downarrow}) \leftarrow \textsc{Solve}(t_1^i,t_2^i,\mathsf{B}_{i-1},\vec{a}_{i-1}^{\uparrow},\vec{a}_{i-1}^{\downarrow},b)$
4: **end for**
5: **return** $(\mathsf{B}_n,\vec{a}_n^{\uparrow},\vec{a}_n^{\downarrow})$

Algorithm 5 (see Section 7.3.3). First, the function obtains the $v_1^{\uparrow}$'s tightest approximation (Line 1alg:solve-choice-choice2) in the form of a switch term s. Next, the elements of a record $\tilde{t}_i$, which is an element of s, are united with elements $\mathsf{l}_1^1(g_1^1)\colon t_1^1,\ldots,\mathsf{l}_1^n(g_1^n)\colon t_1^n$. Due to potential label duplicates, the result of a union in general case is a switch term (Line 10). Finally, the function reduces the problem for choices to the problem for its subterms.

SOLVETERMSWITCH$(\ldots)$ and SOLVESWITCHTERM$(\ldots)$ are discussed in Algorithm 7 and Algorithm 8. The function specify solution for the CSP-WS for a constraint $t_1 \sqsubseteq t_2$, where either t_1 or t_2 is a switch term. In Section 7.3.4 we demonstrate that a constraint that involves a switch term is naturally reduced to constraints for its subterms. In the rest of this section we discuss other functions that are used in the solution algorithm.

Algorithm 9 specifies the COMBINERECORDS$(\ldots)$ function. The function recursively merges records and produces a result in the form of a switch term.

Example 7.5.1.

$$\begin{gathered}\textsc{CombineRecords}(\varnothing,\mathsf{true},\{\mathsf{a}(x)\colon\mathtt{int},\mathsf{b}(y)\colon\mathtt{int}\},\{\mathsf{b}(z)\colon\mathtt{double},\mathsf{c}(u)\colon\mathtt{int}\}) = \\ (\{\neg y \vee \neg z\}, \\ \langle(\neg z)\colon\{\mathsf{a}(x)\colon\mathtt{int},\mathsf{b}(y)\colon\mathtt{int},\mathsf{c}(u)\colon\mathtt{int}\},(\neg y)\colon\{\mathsf{a}(x)\colon\mathtt{int},\mathsf{b}(z)\colon\mathtt{double},\mathsf{c}(u)\colon\mathtt{int}\}\rangle)\end{gathered} \tag{7.11}$$

△

SGROUND$(\ldots)$ is presented in Algorithm 10. The function takes an arbitrary term t and replaces all t-variables in t with their values from conditional approximation $\vec{a}^{\downarrow}$ and $\vec{a}^{\uparrow}$ given a Boolean condition b. In other words, it transforms t to a semi-ground term. Since each variable may have multiple values (each corresponding to some instantiation of b-variables), the result is produced in the form of a switch term. SGROUND$(\ldots)$ is a recursive function. It calls itself to find semi-ground values for t subterms, which are returned as switch terms too. Next, we bring switch terms to the top-level. Specifically, we produce the result as switch terms that do not contain switch terms as its subterms. For this purpose, we perform a combinatorial term construction in Lines 12 and 18.

Algorithm 5 SOLVERR($\{\mathsf{l}_1^1(g_1^1)\colon t_1^1, \ldots, \mathsf{l}_1^n(g_1^n)\colon t_1^n \mid v_1^{\downarrow}\}, \{\mathsf{l}_2^1(g_2^1)\colon t_2^1, \ldots, \mathsf{l}_2^m(g_2^m)\colon t_2^m \mid v_2^{\downarrow}\}, \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b$)

1: **if** $v_2^{\downarrow}$ is present **then**
2: $s \leftarrow$ SGROUND$(v_2^{\downarrow}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b)$
3: **else**
4: $s \leftarrow \langle(\mathsf{true})\colon \mathsf{nil}\rangle$
5: **end if**
6: s is a switch term of the form $\langle(\tilde{g}_1)\colon \tilde{t}_1, \ldots, (\tilde{g}_p)\colon \tilde{t}_p\rangle$
7: **for** $1 \leq i \leq p$ **do**
8: **if** $\tilde{t}_i$ is a record **then**
9: $(\mathsf{B}, \tilde{s}) \leftarrow$ COMBINERECORDS$(\{\mathsf{l}_2^1(g_2^1)\colon t_2^1, \ldots, \mathsf{l}_2^m(g_2^m)\colon t_2^m\}, \tilde{t}_i, \mathsf{B}, b \wedge \tilde{g}_i)$
10: $\tilde{s}$ is a switch term of the form $\langle(\hat{g}_1)\colon \{\hat{\mathsf{l}}_{11}(\hat{g}_{11})\colon \hat{t}_{11}, \ldots, \hat{\mathsf{l}}_{1r_1}(\hat{g}_{1r_1})\colon \hat{t}_{1r_1}\}, \ldots, (\hat{g}_q)\colon \{\hat{\mathsf{l}}_{q1}(\hat{g}_{q1})\colon \hat{t}_{q1}, \ldots, \hat{\mathsf{l}}_{qr_q}(\hat{g}_{qr_q})\colon \hat{t}_{qr_q}\}\rangle$
11: **for** $1 \leq x \leq q$ **do**
12: **for** $1 \leq y \leq r_q$ **do**
13: **if** exists $w : 1 \leq w \leq n$ and $l_1^w = \hat{l}_{xy}$ **then**
14: $(\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}) \leftarrow$ SOLVE$(t_1^w, \hat{t}_{xy}, \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b \wedge \hat{g}_x \wedge \hat{g}_{xy} \wedge g_w)$
15: **if** $v_1^{\downarrow}$ is present **then**
16: $(\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}) \leftarrow$ SET$(v_1^{\downarrow}, \{\hat{\mathsf{l}}_{xy}(\hat{g}_{xy})\colon \hat{t}_{xy}\}, \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b \wedge \hat{g}_x \wedge \neg g_w)$
17: **end if**
18: **else**
19: **if** $v_1^{\downarrow}$ is present **then**
20: $(\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}) \leftarrow$ SET$(v_1^{\downarrow}, \{\hat{\mathsf{l}}_{xy}(\hat{g}_{xy})\colon \hat{t}_{xy}\}, \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b \wedge \hat{g}_x)$
21: **else**
22: $\mathsf{B} \leftarrow \mathsf{B} \cup \{\neg(b \wedge \hat{g}_u \wedge \hat{g}_{xy})\}$
23: **end if**
24: **end if**
25: **end for**
26: **end for**
27: **else**
28: $\mathsf{B} \leftarrow \mathsf{B} \cup \{\neg(b \wedge \tilde{g}_i)\}$
29: **end if**
30: **end for**
31: **return** $(\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow})$

Example 7.5.2. Assume that $\vec{a}^{\downarrow} = (\langle(x)\colon \texttt{int}, (\neg x)\colon \texttt{double}\rangle, \langle(y)\colon \texttt{int}, (\neg y)\colon \texttt{double}\rangle)$ is

Algorithm 6 SOLVECC$((:\mathsf{l}_1^1(g_1^1)\colon t_1^1, \ldots, \mathsf{l}_1^{\mathsf{n}}(g_1^n)\colon t_1^n \mid v_1^{\uparrow}:), (:\mathsf{l}_1^1(g_1^1)\colon t_1^1, \ldots, \mathsf{l}_1^{\mathsf{m}}(g_1^m)\colon t_1^m \mid v_2^{\uparrow}:)\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b)$

1: **if** $v_1^{\uparrow}$ is present **then**
2: $s \leftarrow$ SGROUND$(v_1^{\uparrow}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b)$
3: **else**
4: $s \leftarrow \langle(\mathsf{true})\colon \mathsf{none}\rangle$
5: **end if**
6: s is a switch term of the form $\langle(\tilde{g}_1)\colon \tilde{t}_1, \ldots, (\tilde{g}_p)\colon \tilde{t}_p\rangle$
7: **for** $1 \leq i \leq p$ **do**
8: **if** $\tilde{t}_i$ is a choice **then**
9: $(\mathsf{B}, s) \leftarrow$ COMBINECHOICES$((:\mathsf{l}_1^1(g_1^1)\colon t_1^1, \ldots, \mathsf{l}_1^{\mathsf{n}}(g_1^n)\colon t_1^n:), \tilde{t}_i, \mathsf{B}, b \wedge \tilde{g}_i)$
10: s is a switch term of the form $\langle(\hat{g}_1)\colon (:\hat{\mathsf{l}}_{11}(\hat{g}_{11})\colon \hat{t}_{11}, \ldots, \hat{\mathsf{l}}_{1\mathsf{r}_1}(\hat{g}_{1r_1})\colon \hat{t}_{1r_1}:),$ $\ldots,$ $(\hat{g}_q)\colon (:\hat{\mathsf{l}}_{\mathsf{q}1}(\hat{g}_{q1})\colon \hat{t}_{q1}, \ldots, \hat{\mathsf{l}}_{\mathsf{qr_q}}(\hat{g}_{qr_q})\colon \hat{t}_{qr_q}:)\rangle$
11: **for** $1 \leq x \leq q$ **do**
12: **for** $1 \leq y \leq r_q$ **do**
13: **if** exists $w : 1 \leq w \leq m$ and $l_2^w = \hat{l}_{xy}$ **then**
14: $(\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}) \leftarrow$ SOLVE$(\hat{t}_{xy}, t_2^w, \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b \wedge \hat{g}_x \wedge \hat{g}_{xy} \wedge g_2^w)$
15: **if** $v_2^{\uparrow}$ is present **then**
16: $(v_2^{\uparrow}, (:\hat{\mathsf{l}}_{\mathsf{xy}}(\hat{g}_{xy})\colon \hat{t}_{xy}:), \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}) \leftarrow$ SET$(\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b \wedge \hat{g}_x \wedge \neg g_2^w)$
17: **end if**
18: **else**
19: **if** $v_2^{\uparrow}$ is present **then**
20: $(v_2^{\uparrow}, (:\hat{\mathsf{l}}_{\mathsf{xy}}(\hat{g}_{xy})\colon \hat{t}_{xy}:), \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}) \leftarrow$ SET$(\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b \wedge \hat{g}_x)$
21: **else**
22: $\mathsf{B} \leftarrow \mathsf{B} \cup \{\neg(b \wedge \hat{g}_x \wedge \hat{g}_{xy})\}$
23: **end if**
24: **end if**
25: **end for**
26: **end for**
27: **else**
28: $\mathsf{B} \leftarrow \mathsf{B} \cup \{\neg(b \wedge \tilde{g}_i)\}$
29: **end if**
30: **end for**
31: **return** $(\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow})$

a vector containing a conditional approximation for variables $p^{\downarrow}$ and $q^{\downarrow}$. Then

$$\begin{aligned}\text{SGROUND}(\{\mathsf{a}(z)\colon p^{\downarrow}, \mathsf{b}(u)\colon q^{\downarrow}\}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, \mathsf{true}) = \langle &(x \wedge y)\colon \{\mathsf{a}(z)\colon \texttt{int}, \mathsf{b}(u)\colon \texttt{int}\},\\ &(x \wedge \neg y)\colon \{\mathsf{a}(z)\colon \texttt{int}, \mathsf{b}(u)\colon \texttt{double}\},\\ &(\neg x \wedge y)\colon \{\mathsf{a}(z)\colon \texttt{double}, \mathsf{b}(u)\colon \texttt{int}\},\\ &(\neg x \wedge \neg y)\colon \{\mathsf{a}(z)\colon \texttt{double}, \mathsf{b}(u)\colon \texttt{double}\}\rangle\end{aligned}$$

Algorithm 7 SOLVETERMSWITCH($\langle (g_1^1)\colon t_1^1, \ldots, (g_1^n)\colon t_1^n \rangle, t_2, \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b$)

1: $(\mathsf{B}_0, \vec{a}_0^{\uparrow}, \vec{a}_0^{\downarrow}) \leftarrow (\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow})$
2: **for** $1 \leq i \leq n$ **do**
3: $\quad (\mathsf{B}_i, \vec{a}_i^{\uparrow}, \vec{a}_i^{\downarrow}) \leftarrow \text{SOLVE}(t_1^i, t_2, \mathsf{B}_{i-1}, \vec{a}_{i-1}^{\uparrow}, \vec{a}_{i-1}^{\downarrow}, b \wedge g_1^i)$
4: **end for**
5: **return** $(\mathsf{B}_n, \vec{a}_n^{\uparrow}, \vec{a}_n^{\downarrow})$

Algorithm 8 SOLVESWITCHTERM($t_1, \langle (g_2^1)\colon t_2^1, \ldots, (g_2^n)\colon t_2^n \rangle, \mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow}, b$)

1: $(\mathsf{B}_0, \vec{a}_0^{\uparrow}, \vec{a}_0^{\downarrow}) \leftarrow (\mathsf{B}, \vec{a}^{\uparrow}, \vec{a}^{\downarrow})$
2: **for** $1 \leq i \leq n$ **do**
3: $\quad (\mathsf{B}_i, \vec{a}_i^{\uparrow}, \vec{a}_i^{\downarrow}) \leftarrow \text{SOLVE}(t_1, t_2^i, \mathsf{B}_{i-1}, \vec{a}_{i-1}^{\uparrow}, \vec{a}_{i-1}^{\downarrow}, b \wedge g_2^i)$
4: **end for**
5: **return** $(\mathsf{B}_n, \vec{a}_n^{\uparrow}, \vec{a}_n^{\downarrow})$

△

Algorithm 11 demonstrates the function SET(...). The function updates the approximation for some variable v with a new value t_2. Essentially, the function produces the new approximation as the meet term (join term) of the existing approximation and the new value if t_2 is a down-coerced term (an up-coerced term). If the meet term (join term) does not exist, the function adds a Boolean constraint that forbids such approximation.

Algorithm 9 COMBINERECORDS($\mathsf{B}, b, \{\mathsf{l}_1^1(g_1^1)\colon t_1^1, \ldots, \mathsf{l}_1^{\mathsf{n}}(g_1^n)\colon t_1^n\}$, $\{\mathsf{l}_2^1(g_2^1)\colon t_2^1, \ldots, \mathsf{l}_2^{\mathsf{m}}(g^m)\colon t_2^m\}$)

1: **if** $n = 0$ **then**
2: $\quad$ **return** $(\mathsf{B}, \langle (\mathsf{true})\colon \{\mathsf{l}_1^1(g_1^1)\colon t_1^1, \ldots, \mathsf{l}_2^{\mathsf{m}}(g_2^m)\colon t_2^m\}\rangle$
3: **end if**
4: **if** exists $j : 1 \leq j \leq m$ and $l_1^1 = l_2^j$ **then**
5: $\quad (\mathsf{B}, s) \leftarrow$ COMBINERECORDS($\mathsf{B}, b, \{\mathsf{l}_1^2(g_1^2)\colon t_1^2, \ldots, \mathsf{l}_1^{\mathsf{n}}(g_1^n)\colon t_1^n\}, \{\mathsf{l}_2^1(g_2^1)\colon t_2^1, \ldots,$ $\mathsf{l}_2^{\mathsf{j}-1}(g_2^{j-1})\colon t_2^{j-1}, \mathsf{l}_2^{\mathsf{j}+1}(g_2^{j+1})\colon t_2^{j+1}, \ldots, \mathsf{l}_2^{\mathsf{m}}(g_2^m)\colon t_2^m\}$)
6: $\quad s_1$ is a switch term obtained by transforming every element $\hat{g} : \{\tilde{\mathsf{l}}_1(\tilde{g}_1)\colon \tilde{t}_1, \ldots, \tilde{\mathsf{l}}_{\mathsf{p}}(\tilde{g}_p)\colon \tilde{l}_p\}$ of s to $\hat{g} \wedge \neg g_j' : \{\tilde{\mathsf{l}}_1(\tilde{g}_1)\colon \tilde{t}_1, \ldots, \tilde{\mathsf{l}}_{\mathsf{p}}(\tilde{g}_p)\colon \tilde{l}_p, \mathsf{l}_1^1(g_1^1)\colon t_1^1\}$
7: $\quad s_2$ is a switch term obtained by transforming every element $\hat{g} : \{\tilde{\mathsf{l}}_1(\tilde{g}_1)\colon \tilde{t}_1, \ldots, \tilde{\mathsf{l}}_{\mathsf{p}}(\tilde{g}_p)\colon \tilde{l}_p\}$ of s to $\hat{g} \wedge \neg g_1 : \{\tilde{\mathsf{l}}_1(\tilde{g}_1)\colon \tilde{t}_1, \ldots, \tilde{\mathsf{l}}_{\mathsf{p}}(\tilde{g}_p)\colon \tilde{l}_p, \mathsf{l}_2^{\mathsf{j}}(g_2^j)\colon t_2^j\}$
8: $\quad$ **return** $(\mathsf{B} \cup \{\neg(b \wedge g_1 \wedge g_j')\}$, a switch term that is a concatenation of s_1 and $s_2)$
9: **else**
10: $\quad (\mathsf{B}, s) \leftarrow$ COMBINERECORDS($\mathsf{B}, b, \{\mathsf{l}_2(g_2)\colon t_2, \ldots, \mathsf{l}_{\mathsf{n}}(g_n)\colon t_n\}, \{\mathsf{l}_1'(g_1')\colon t_1', \ldots,$ $\mathsf{l}_{\mathsf{m}}'(g_m')\colon t_m'\}$)
11: $\quad s'$ is a switch term obtained by adding an element $\mathsf{l}_1(g_1)\colon t_1$ to every record in s
12: $\quad$ **return** (B, s')
13: **end if**

Algorithm 10 SGROUND$(t, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b)$

match t **with**
  case symbol s
    return $\langle (b)\colon t \rangle$
  case down-coerced or up-coerced variable v
    Let $\langle (b_1)\colon t_1, \ldots, (b_n)\colon t_n \rangle$ be an approximation of v in $\vec{a}$
    return $\langle (b \wedge b_1)\colon t_1, \ldots, (b \wedge b_n)\colon t_n \rangle$
  case $(t_1 \ \ldots \ t_n)$
    for $1 \leq i \leq n$ **do**
      $s_i \leftarrow$ SGROUND$(t_i, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b)$
      Let s_i be of the form $\langle (b_i^1)\colon t_i^1, \ldots, (b_i^{m_i})\colon t_i^{m_i} \rangle$
    end for
    return $\langle (b \wedge b_1^1 \cdots \wedge b_{n-1}^1 \wedge b_n^1)\colon (t_1^1 \ \ldots \ t_{n-1}^1 \ t_n^1),$
      $(b \wedge b_1^1 \cdots \wedge b_{n-1}^1 \wedge b_n^2)\colon (t_1^1 \ \ldots \ t_{n-1}^2 \ t_n^2),$
      $\ldots,$
      $(b_1^{m_1} \wedge \cdots \wedge b_n^{m_n})\colon (t_1^{m_1} \ \ldots \ t_n^{m_n}) \rangle$
  case $\{\mathsf{l}_1(g_1)\colon t_1, \ldots, \mathsf{l}_n(g_n)\colon t_n \mid v^{\downarrow}\}$
    for $1 \leq i \leq n$ **do**
      $s_i \leftarrow$ SGROUND$(t_i, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b)$
      Let s_i be of the form $\langle (b_i^1)\colon t_i^1, \ldots, (b_i^{m_i})\colon t_i^{m_i} \rangle$
    end for
    $r \leftarrow \langle (b_1^1 \cdots \wedge b_{n-1}^1 \wedge b_n^1)\colon \{\mathsf{l}_1(g_1)\colon t_1^1, \ldots, \mathsf{l}_{n-1}(g_{n-1})\colon t_{n-1}^1, \mathsf{l}_n(g_n)\colon t_n^1\},$
      $(b_1^1 \cdots \wedge b_{n-1}^1 \wedge b_n^2)\colon \{\mathsf{l}_1(g_1)\colon t_1^1, \ldots, \mathsf{l}_{n-1}(g_{n-1})\colon t_{n-1}^1, \mathsf{l}_n(g_n)\colon t_n^1\},$
      $\ldots,$
      $(b_1^{m_1} \cdots \wedge b_{n-1}^{m_{n-1}} \wedge b_n^{m_n})\colon \{\mathsf{l}_1(g_1)\colon t_1^{m_1}, \ldots, \mathsf{l}_{n-1}(g_{n-1})\colon t_{n-1}^{m_{n-1}}, \mathsf{l}_n(g_n)\colon t_n^{m_n}\} \rangle$
    if $v^{\downarrow}$ is present **then**
      $\tilde{s}' \leftarrow$ SGROUND$(v^{\downarrow}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b)$
      return r combined with s'
    else
      return r
    end if
  case $(:\mathsf{l}_1(g_1)\colon t_1, \ldots, \mathsf{l}_n(g_n)\colon t_n \mid v^{\uparrow}:)$
    A semi-ground term for a choice is computed in the same way as for a record
  case $\langle (b_1)\colon t_1, \ldots, (b_n)\colon t_n \rangle$
    for $1 \leq i \leq n$ **do**
      $s_i \leftarrow$ SGROUND$(t_i, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b \wedge b_i)$
    end for
    return a switch term that is a composition of $s_1, \ldots, s_n$

7.6 Support for Multiple Flow Inheritance in the Algorithm

The algorithm in Section 7.5 solves the CSP-WS for service interfaces that do not contain a UNION term. In Chapter 8 and Chapter 9 we argue that it is impossible to support

Algorithm 11 SET$(v, t_2, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b)$

t_2 is a semi-ground switch of the form $\langle (b_2^1)\colon t_2^1, \ldots, (b_2^n)\colon t_2^n \rangle$
$t_1 \leftarrow$ SGROUND$(v, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b)$
t_1 is a semi-ground switch of the form $\langle (b_1^1)\colon t_1^1, \ldots, (b_1^m)\colon t_1^m \rangle$
s is the empty switch term
for $1 \leq i \leq m$ **do**
    for $1 \leq j \leq n$ **do**
        add the element $(b \wedge g_1^i \wedge \neg g_2^j)\colon t_1^i$ to s
        add the element $(b \wedge \neg g_1^i \wedge g_2^j)\colon t_2^j$ to s
        if t_1^i and t_2^j are down-coerced terms and the join term $t_1^i \sqcup t_2^j$ exists **then**
            add the element $(b \wedge g_1^i \wedge g_2^j)\colon t_1^i \sqcup t_2^j$ to s
        else if t_1^i and t_2^j are up-coerced terms and the meet term $t_1^i \sqcap t_2^j$ exists **then**
            add the element $(b \wedge g_1^i \wedge g_2^j)\colon t_1^i \sqcap t_2^j$ to s
        else
            $\mathsf{B} \leftarrow \mathsf{B}\{\neg(b \wedge g_1^i \wedge g_2^j)\}$
        end if
    end for
end for
if v is a down-coerced term **then**
    update the value of v in $a^{\downarrow}$ to s
else
    update the value of v in $a^{\uparrow}$ to s
end if
return $(\mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow})$

stateful services in the interface configuration protocol without having support for UNION in the CSP-WS.

Support for UNION can be added on top of the algorithm presented in Section 7.5. We introduce an additional case to Algorithm 3, which is triggered if t_1 is a UNION tuple in the constraint $t_1 \sqsubseteq t_2$ (t_2 can be any term). In this case, the algorithm performs steps as follows:

1. First, the algorithm replaces the tuple with a free term variable (for each UNION tuple, the variable is generated only once). By replacing all UNION tuples with variables, the set of constraints becomes tuple-free.
2. Then, the algorithm calls a TRANSFORMUNION function (see Algorithm 12), which sets an approximation for the variable.
3. Finally, the algorithm recursively calls SOLVE for the constraint $t_1 \sqsubseteq t_2$, where the union t_1 is replaced with the variable.

We implement the three steps as follows. Assume that $t_1 = (\mathsf{union}\ p^{\downarrow}\ q^{\downarrow})$ and $r^{\downarrow}$

Algorithm 12 TRANSFORMUNION$(t, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b)$

if t is a down-coerced variable $r^{\downarrow}$, $\mathrm{U}(r^{\downarrow})$ exists and $\mathrm{U}(r^{\downarrow}) = (p^{\downarrow}, q^{\downarrow})$ **then**
  $t' \leftarrow \textsc{SGround}(r, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b)$
  t' is a semi-ground switch of the form $\langle (b_1^1)\colon t_1^1, \ldots, (b_1^m)\colon t_1^m \rangle$
  for $1 \leq i \leq m$ **do**
    if t_1^i is not a record **then**
      add $\neg(b \wedge b_1^i)$ to B
    else
      if $\mathrm{C}(\mathsf{B}(b \wedge b_1^i), r^{\downarrow})$ does not exist **then**
        Assume $t_1^i = \{\tilde{\mathsf{l}}_1(\tilde{b}_1^i)\colon \tilde{t}_1^i \ldots \tilde{\mathsf{l}}_k(\tilde{b}_k^i)\colon \tilde{t}_k^i\}$
        $r_1 \leftarrow \{\tilde{\mathsf{l}}_1(\tilde{b}_1^i)\colon \tilde{t}_1^i \wedge \bar{b}_1 \ldots \tilde{\mathsf{l}}_k(\tilde{b}_k^i \wedge \bar{b}_k)\colon \tilde{t}_k^i\}$, where $\bar{b}_1, \ldots \bar{b}_k$ are new b-variables
        $r_2 \leftarrow \{\tilde{\mathsf{l}}_1(\tilde{b}_1^i \wedge \neg\bar{b}_1)\colon \tilde{t}_1^i \ldots \tilde{\mathsf{l}}_k(\tilde{b}_k^i \wedge \neg\bar{b}_k)\colon \tilde{t}_k^i\}$
        the value of $p^{\downarrow}$ in $\vec{a}^{\downarrow}$ must be r_1 given b
        the value of $q^{\downarrow}$ in $\vec{a}^{\downarrow}$ must be r_2 given b
        $(\mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) \leftarrow \textsc{Set}(p, r_1, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b)$
        $(\mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}) \leftarrow \textsc{Set}(q, r_2, \mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow}, b)$
      end if
    end if
  end for
end if
return $(\mathsf{B}, \vec{a}^{\downarrow}, \vec{a}^{\uparrow})$

is a variable that replaces the union tuple. We introduce two dictionaries $\mathrm{U}(r^{\downarrow})$ and $\mathrm{C}(\mathsf{SAT}(b), r^{\downarrow})$. The former maps a down-coerced variable that corresponds to the union tuple to a pair of down-coerced variables that are subterms in the tuple. The latter maps a set of solutions to a Boolean constraint and the down-coerced variable to a set of new Boolean variables.

C stores newly generated b-variables that corresponds to some logical expression and a down-coerced variable for a union ($r^{\downarrow}$, in our example). While solving the CSP-WS the algorithm generates a new b-variable that is inserted to C as illustrated in Algorithm 12.

TRANSFORMUNION$(\ldots)$ refines a conditional approximation for the down-coerced variable $r^{\downarrow}$ providing that the variable is present in $\mathrm{U}(r^{\downarrow})$. Next, for every value of $r^{\downarrow}$, two records r_1 and r_2 are generated. Then, an auxiliary b-variable is added to every guard in r_1. As a result, r_2 is a record that contains the same labels and terms as in r_1, yet the auxiliary b-variable is negated. r_1 and r_2 are records that $p^{\downarrow}$ and $q^{\downarrow}$ must be coerced to. This guarantees that the records $p^{\downarrow}$ and $q^{\downarrow}$ contain all elements that are in $r^{\downarrow} = (\mathsf{union}\ p^{\downarrow}\ q^{\downarrow})$, but not in the both records.

Example 7.6.1. Consider a simple example in Figure 7.4 that consists of a merger and a service that consumes an output from the merger. The merger is generic: it receives two

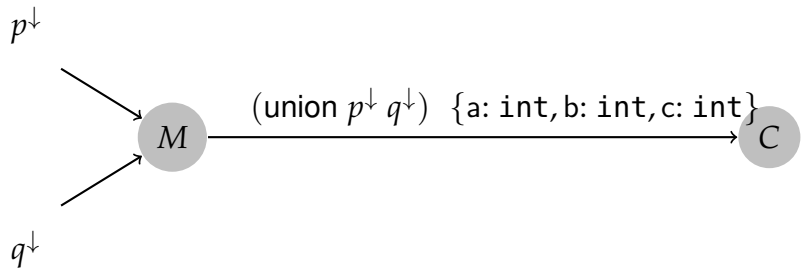


Figure 7.4: A merger and its consumer

messages as records and combines them in a single output message. A constraint follows from the interconnection:

$$(\mathsf{union}\ p^{\downarrow}\ q^{\downarrow}) \sqsubseteq \{\mathtt{a}\colon \mathtt{int}, \mathtt{b}\colon \mathtt{int}, \mathtt{c}\colon \mathtt{int}\}.$$

To resolve the union, the algorithm performs the following steps:

1. Generates a new t-variable $r^{\downarrow}$ such that $r^{\downarrow} = (\mathsf{union}\ p^{\downarrow}\ q^{\downarrow})$ and adds a new entry to U: $\mathrm{U}(r^{\downarrow}) = (p^{\downarrow}, q^{\downarrow})$.
2. Replaces the union in the constraint with $r^{\downarrow}$:

$$r^{\downarrow} \sqsubseteq \{\mathtt{a}\colon \mathtt{int}, \mathtt{b}\colon \mathtt{int}, \mathtt{c}\colon \mathtt{int}\}.$$

3. Calls TRANSFORMUNION$(r^{\downarrow}, \dots)$ for $r^{\downarrow}$ and
   (a) Generates new b-variables x, y and z;
   (b) Sets approximations for $p^{\downarrow}$ and $q^{\downarrow}$:

$$p^{\downarrow} = \{\mathtt{a}(x)\colon \mathtt{int}, \mathtt{b}(y)\colon \mathtt{int}, \mathtt{c}(z)\colon \mathtt{int}\}$$

   and

$$q^{\downarrow} = \{\mathtt{a}(\neg x)\colon \mathtt{int}, \mathtt{b}(\neg y)\colon \mathtt{int}, \mathtt{c}(\neg z)\colon \mathtt{int}\};$$

   (c) Sets $\mathrm{C}(\mathsf{true}, r^{\downarrow}) = \{x, y, z\}$.

As a result, the algorithm produces solutions (the number of ways to arrange three elements in two sets) for $p^{\downarrow}$ and $q^{\downarrow}$. Additional constraints may reduce the set of potential solutions to $p^{\downarrow}$, $q^{\downarrow}$ and $r^{\downarrow}$. △

Chapter 8

Interface Configuration Protocol for Service-Based Applications

A service-oriented approach (SOA) facilitates software development and maintenance. As a relatively new technology, SOA relies heavily on existing technologies, such as OOP. Object-oriented programming (OOP) is widely used in the implementation of web services. It provides encapsulation, polymorphism, and inheritance — concepts that simplify modern software development and maintenance. Many platforms for implementing service-oriented applications are designed as object-oriented frameworks. For example, Java Business Integration (JBI) [Bin08], Windows Communication Foundation (WCF) [MMW06], and the data distribution service (DDS) [PC03] support the development of service-oriented applications on top of OOP.

However, moving from an object-oriented approach to a service-oriented approach does not come without its challenges. One reason is that OOP was not originally designed to represent services. Object-oriented methods assume that users of an object are application developers. Typically, classes and class hierarchies are developed by one team of developers within one application. Therefore, class interfaces are tightly coupled: any modification to a method definition requires modification to the caller's code.

Service loose coupling is the key principle of SOA. In contrast to an objects-based application, 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 interface must be generic and should be adaptable to a good many contexts. The service interface must meet the requirements of all clients that use the service, because it is impossible to manually adjust the service to meet the needs of all clients.

Service interoperability is typically achieved by tuning an interface at the client end so it matches an interface provided at the server end. Therefore, web services

are tightly coupled, which contradicts the principle of Service Oriented Architecture (SOA) [AGR13a]. The problem is caused by the fact that the service is represented as an object that contains a set of methods. A message is received by calling a service's method; a service sends a message by returning a value from the method. Interface flexibility can be achieved by using parametric polymorphism, but this is not enough to achieve service loose coupling.

Services are often organised in long computational pipelines, which process an input message step-by-step. A service modifies some elements of the input message and sends the transformed message further down in the pipeline. In fact, the elements can be contained in any message. Therefore, a reusable service must contain a specific interface for elements that are being modified. The rest of the service must remain generic. Moreover, any part of the input message that remains unmodified must automatically be inherited from the input to the output, so that it can be processed by other services in the pipeline. Such behaviour is called *flow inheritance* [GSS08, GSS10].

Unfortunately, flow inheritance cannot be achieved in a web service that is implemented as a class that exposes a set of methods to other services. A method contains a set of predefined arguments that cannot be extended if a method caller provides more arguments than the method requires. Neither method arguments can automatically be attached to the return value. The method can explicitly be overloaded with additional arguments, but this delivers tight coupling between all services in the network.

We solve this problem and provide support for flow inheritance using an interface configuration mechanism as presented in this chapter. Instead of representing a service as an object, we model it as a black box, which exposes input and output ports to the environment. A port has a name and an interface. To interact with the service, application designers must declare communication channels between the service and clients. Clients must provide and demand messages of formats that are compatible with those ones specified in the port interface.

Instead of representing a service interface as a set of class methods, we encode it as an MDL term. The interface is associated with each port and it specifies the format of a message that can be sent to or received from the port. In our approach, a topology of the application is statically defined. An application designer must connect the ports of interacting services by means of communication channels. As a result, each communication channel raises a seniority constraint on the MDL terms.

The MDL term associated with a generic service (i.e. a service that does not depend upon a particular context) may contain variables. Variables are essentially a form of parametric polymorphism. The variable is matched with data of any format which is provided by a communicating service. Furthermore, the variables provide a mechanism

for wiring interfaces in a service. Putting the same variable into several interfaces for the service allows data formats to be propagated across channels. The same format is expected in all places in which the variable is present.

After creating a service application by connecting the services via communication channels, it must be ensured that the communication is safe.[1] In order to achieve this, the CSP-WS must be solved (see Chapter 5). A solution for the CSP-WS instantiates the variables with values that satisfy the constraints. In this way, it can be guaranteed that the services in the application are compatible. Instantiating variables with the values contextualises the interfaces. The variable values can also specialse service code, similarly to the specialisation of C++ templates.

We provide support for flow inheritance using this approach. Firstly, we define a fixed format for service interfaces. We represent an interface as a choice term at the top level. Flow inheritance can be supported using tail variables in the choice. If choices in input and output interfaces contain the same tail variable, then we guarantee that data that is matched by the tail variable is automatically propagated to the output (see Figure 2.2 on page 27).

Inheritance in OOP is a local mechanism for structuring stateful objects. A programmer always explicitly declares the relation between classes, which is something that does not depend upon the context. In contrast to OOP, flow inheritance is a non-local mechanism for structuring transformations (a transformation is a pair of input and output interfaces) without taking states into consideration. The interfaces are extended automatically provided that the extended interfaces satisfy client requirements. Furthermore, our mechanism guarantees that the inherited data is always propagated from input to output.

In this chapter we present an interface configuration protocol, which supports flow inheritance in services that are coded in C++. In a nutshell, the protocol specifies the following steps in order:

1. deriving the MDL interfaces from the code;
2. constructing the CSP-WS constraints;
3. providing a solution for the CSP-WS;
4. propagating the CSP-WS solution back to the service code.

Furthermore, we developed a toolchain that performs these steps and, as a result, specialises generic services within the context.

[1] By safe communication we mean only interface compliance. There are other approaches, such as session types [CHY07], that provide stronger guarantees about communication safety given a communication protocol of each service. In contrast to these approaches, our method relies solely on the interfaces and is not aware of service behaviour.

In order to address the names matching for messages and processing functions, we introduce service separation into a core and a shell. The core is the code that implements the functionality of the service. In our protocol, the core is essentially a set of C++ functions, each of them being responsible for processing a message without taking environment details into account.

The shell is a layer that connects the core with the environment. Using the shell, the application designer can match element names in output messages with those which are expected by consumers. Furthermore, the designer can reroute messages to various channels from the shell.

Our protocol supports not only records as messages, but also abstract data types, i.e. collections that, in addition to the data, store methods which can operate upon the data.

An advantage of SOA is the concurrent processing of input messages. Incoming messages can be processed in multiple threads, where each thread is defined by its internal state. Thread-safety can be enabled by externalising the state in a message; that is to say that if a thread crashes, its state can be recovered from the message. In this case, a generic service has at least two input ports: one for receiving an input message and another one that represents a state.

We discuss the generic services in Chapter 9. In this chapter we consider only those services that do not have a port for injecting a message with a state in a service. This allows us to focus on an interface configuration protocol for a simple case. Such 'stateless' services are widely used in the industry. These, for example, are query services: those services that provide information to a requester without modifying an internal state of the service.

In Chapter 9 we show that it is possible to reduce the interface configuration protocol for 'stateful' services (those which have a port for injecting a message which carries a state in the service) to the one for 'stateless' services. We achieve this by introducing a special form of services, called *synchronisers*, which can merge multiple input messages into a single output message. To reflect this message relation in the interfaces, we introduce a special MDL term called a *union* (see Section 3.5). The union is a macro for a message format which is constructed by merging two other messages together.

8.1 Overview

We present an interface configuration protocol in Figure 8.1 for services which are coded in C++. Supporting the C++ services allows us to consider not only web-oriented applications but also performance-critical applications. In general, the interface configuration protocol is rather generic and can easily be extended for services written in other

languages.

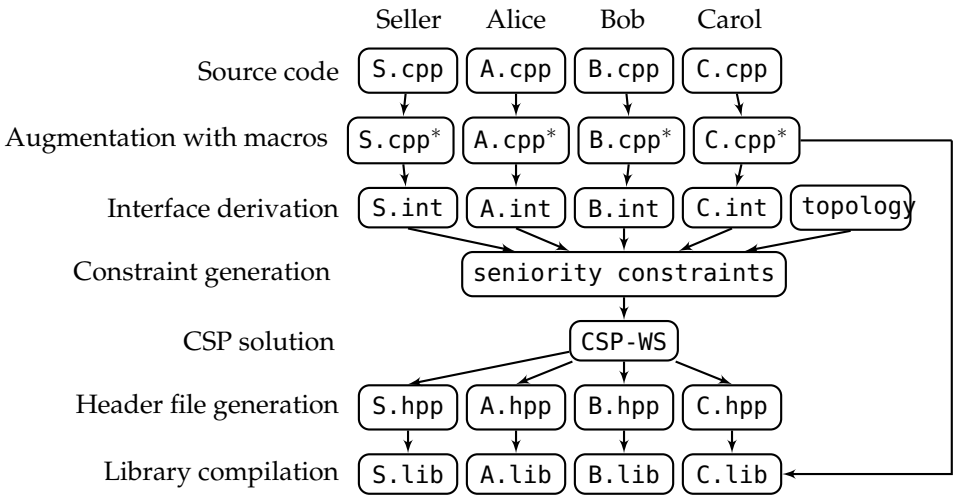


Figure 8.1: An interface reconciliation workflow performed by the toolchain

In our protocol we assume that each service provider has a source code for a service that they own. The protocol specifies a sequence of steps that needs to be performed in order to check the consistency of service interfaces and to configure the interfaces according to the context in which the services are used. The steps, which are automatically performed by our toolchain, are defined as follows.

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-WS 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-WS.
5. If a solution to the CSP-WS 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 Figure 8.1, `.cpp` are the service source files. `.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 that are generated from the CSP-WS solution. `.lib` are generated libraries for the services.

Currently, the protocol supports only stateless services. We discuss a general approach for supporting stateful services in Chapter 9. Integrating support for the stateful services in the protocol is a problem for further work.

We illustrate the configuration protocol using the example from Section 3.6. Figure 8.2 is implementation of the Seller service. `request_1` and `payment_1` are processing functions that are triggered when an input message arrives; the input data is passed to the function with arguments. The processing functions are distinguished from 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 the output as a message. Using this mechanism, each service produces zero or more output messages as a response to a single input message.

```
1  salvo response_1(string title, int money);
2  salvo invoice_1(int id);
3  salvo error_2(string msg);
4  service request_1(string title) {
5      try {
6          int price = ...
7          response_1(title, price);
8      } catch (exception e) {
9          error_2(e.what());
10     }
11 }
12 service payment_1(string title, int money) {
13     try {
14         int invoice_id = ...
15         invoice_1(invoice_id);
16     } catch (exception e) {
17         error_2(e.what());
18     }
19 }
```

Figure 8.2: The code of the Seller service

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 that allows to redefine a mapping between function/salvo names and ports.

The protocol provides a facility for renaming ports (see Chapter 4 for the port definition) and message routing. This is performed from a service component called a *shell*. The shell is described in Section 8.1.2. In general, the shell is a service wrapper that provides facilities for port rerouting and function name renaming.

In the example from Figure 8.2, error messages are sent to the second output port. 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.

Figure 8.3 illustrates MDL interfaces derived by our configuration toolchain. In the protocol we define a fixed interface format for the services. 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. As a result, a message is structured as a labelled data record, where the label specifies the function that processes the message. The name of a salvo corresponds to a label in an output choice term. Compatibility of two communicating services is defined by the seniority relation. The interfaces can automatically be derived using the tool that we developed. The tool is described in Section 8.4.

IN #1: $(:\mathsf{request}(x)\colon \{\mathsf{title}\colon \texttt{string} \mid a^{\downarrow}\}, \mathsf{payment}(y)\colon \{\mathsf{title}\colon \texttt{string}, \mathsf{money}\colon \texttt{int} \mid b^{\downarrow}\} \mid c^{\uparrow}:)$
OUT #1: $(:\mathsf{response}(x)\colon \{\mathsf{title}\colon \texttt{string}, \mathsf{money}\colon \texttt{int} \mid d^{\downarrow}\}, \mathsf{invoice}(y)\colon \{\mathsf{id}\colon \texttt{int} \mid e^{\downarrow}\} \mid c^{\uparrow}:)$
OUT #2: $(:\mathsf{error}(x \vee y)\colon \{\mathsf{msg}\colon \texttt{string} \mid f^{\downarrow}\}:)$

Figure 8.3: 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

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.

In our example, response is present in the output interface #1 only if request is present

in the input one; similarly, error is present in the output interface #2 only if request or payment is present in the input interface.

The effect achieved by Boolean variables is similar to intersection types [Pie97]. Intersection types increase expressiveness of function signatures. For example, the signature $(a \to c) \wedge (b \to d)$ is more expressive comparing to $(a \wedge b) \to (c \wedge d)$. Likewise, for configuration of a generic service it is essential to know the relation between input and output data. Otherwise, it is impossible to track flow of data in the application network.

Now we discuss the structure of constraints and implementation of flow inheritance in Figure 8.3. $c^{\uparrow}$ is an up-coerced variable that stores variants. $c^{\uparrow}$ specifies unused functionality of services inherited from Seller's producers. The variants are automatically propagated to output port #1, because $c^{\uparrow}$ is also present in the output interface OUT #1. We designed inheritance to be configurable and, therefore, it can be either enabled or disabled. In the example, inheritance in the output port #2 is disabled, and, therefore, OUT #2 lacks the tail variable $c^{\uparrow}$.

Besides inheritance for variants, the protocol supports flow inheritance for record elements. In the example, down-coerced variables $d^{\downarrow}$, $e^{\downarrow}$ and $f^{\downarrow}$ specify data that is demanded by Seller's consumers. This requirement is propagated back to the input using variables $a^{\downarrow}$ and $b^{\downarrow}$. Relation between variables in the input and output interfaces is maintained using auxiliary constraints $a^{\downarrow} \sqsubseteq d^{\downarrow}$, $a^{\downarrow} \sqsubseteq f^{\downarrow}$, $b^{\downarrow} \sqsubseteq e^{\downarrow}$ 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 element 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.

Now we illustrate back-propagation of the CSP-WS 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 Figure 8.4. `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. For example, 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:[2]

```
#define COMMA ,
#define TV_a COMMA int id
#define TV_d_decl COMMA int id
#define TV_d_use COMMA id
```

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

```
#define TV_f_decl COMMA float rank
#define TV_f_use COMMA rank
```

Providing that the header file is included in the augmented source files, the above definitions replace their corresponding macro names (see Figure 8.4). As a result, the data that 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.

```
#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
```

Figure 8.4: The code for the Seller service augmented with preprocessor's directives. `BV_...` and `TV_...` are macro names

We not discuss a structure of individual services, in particular, the core and the shell.

8.1.1 The Core

The core is the code of the service structured in a predefined format as illustrated in Figures 8.2 and 8.5. The core is context-independent and can be seen as a reusable basic module. The code of the core is accessible only by a designer of the service. In overall, the core is observed as a black box. The services interact with each other only using a predefined API. As a result, our mechanism supports proprietary services, which cannot publicly expose their source code.

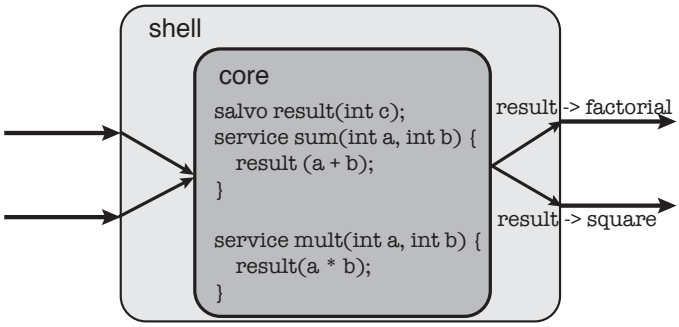


Figure 8.5: The core and the shell of the service

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 a name, the format of its input message and the 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 a record are ordered by inclusion, although C++ function arguments are identified by a position. The application layer, which triggers processing functions, must reorder record elements in a way they are specified in a processing function. On the other hand, the problem disappears in languages with support for labelled arguments, such as Python [Glo] or OCaml [OCa].

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

In our mechanism the input and output interfaces of the interacting services are not required to be exactly the same. 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.[3]

Furthermore, the MDL interfaces support polymorphism, which allows to integrate the configuration mechanism with C++ templates.[4] 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 (`vector` $t^{\downarrow}$), where $t^{\downarrow}$ is a free t-variable. After solving the CSP-WS, $t^{\downarrow}$ is instantiated with a specific value, which can be propagated back to the service as specialisation of T.

Finally, the main contribution of this work is support for flow inheritance. The service can not only accept 'more' data that required, but also propagate excessive data (the one that is provided, but not required by a consumer) to other services. A solution to CSP-WS 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 Figure 8.1, which illustrates configuration process in the context).

8.1.2 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.

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 Figure 8.6.

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.

[3] Generation of such preprocessing function has not been done as part of this work though.[4] The existing implementation of the configuration mechanism lacks such integration.

There are two ways to specify port routing. First, the routing can be 'hard-coded' in services as illustrated in Figure 8.2. 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 specifiy 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. Figure 8.6 is the shell file that corresponds to name transformation and salvo routing from Figure 8.5.

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

Figure 8.6: 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`

8.2 Qualifiers

Messages in the protocol may contain elements of various types. Using MDL symbols, we can represent basic types, such as `int`, `string`, `double`, etc. In Section 8.1.1 we also explained how C++ templates can be supported. For example, (`map` $k^{\downarrow}$ $v^{\downarrow}$) is a `map` template and (`map int string`) is a template's specialisation with `int` as a key and `string` as a value.

Furthermore, we introduce support for `const` and `volatile` C++ qualifiers [CPP] using tuples and records.[5] We wrap all qualified types in a tuple. The first element of the tuple is a record that contains qualifiers as labels. The second element of the tuple is an unqualified type itself. For example, ({} `int`) represents `int`, ({const: nil} `int`) represents `const int`, ({volatile: nil} `int`) represents `volatile int` and ({const: nil, volatile: nil} `int`) represents `const volatile int`, respectively. Such representation of qualified types allows us to achieve the partial ordering as illustrated in Figure 8.7.

8.3 Interface Classes and Objects

In addition to basic types, polymorphic types (i.e. templates) and qualified types, we also added support for objects in the form of MDL terms to the configuration protocol.

[5] Although there is no practical reason for having `volatile` qualifier in web services, we use it for demonstrating support for multiple qualifiers.

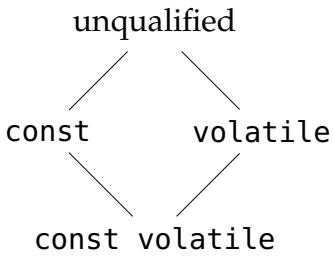


Figure 8.7: Partial ordering of C++ `const` and `volatile` qualifiers

Supporting object orientation in web services is challenging. Object representation must be preserved when the object is sent from a producer to a consumer. On the other hand, the producer and the consumer can be written in different languages. In this case, it is difficult to guarantee that object representation is identical before and after serialisation. Object binary properties, such as memory offset for fields, is difficult to encode in structural form in XML or JSON [LM07].

Currently, interoperability of objects is achieved by manual tuning of client's interfaces. Furthermore, objects that are used in configuration between external web services do not support subtyping [AGR13a]. As a result, the services are tightly coupled, which contradicts the principles of SOA. In this section we demonstrate that expressiveness of the MDL is sufficient to solve the interoperability problem between objects in web services.

We encode structure of objects as C++ classes. We divide classes into two categories: *global classes* and *interface classes*.

Global classes are available to all services. The purpose of global classes is to provide basic primitives and most commonly used operations to all services. Since global classes are available to all services, objects that belong to global classes can be safely transferred between the services. On the other hand, a disadvantage is that all services are required to share class definitions, which violates decontextualisation principles.

As a solution to the problem, we introduce another category called interface classes. An interface class is a class that is defined inside a service. For an object that belong to an interfaces class, a class representation must also be transferred along with the object. Support for the interface classes requires the following properties:

Serialisability. For each interface class, there must exist its representation as a self-contained MDL term. The term must contain all details about the class and must unambiguously map to the class representation in binary format.

Flexibility and Compatibility. Typically, only objects of identical classes can be safely transferred in web services. This breaks flexibility, because in this case use of

service in various contexts is limited. Our goal is to introduce class subtyping in web services. With support for subtyping, a consumer can accept an object of any compatible class. To achieve this, we must ensure that the seniority relation for terms representing interface objects holds only if the objects are compatible.

We designed configurable interfaces classes that have the following features:

1. In C++ class fields are identified by the memory offset. We provide support for field inheritance in classes (i.e. fields from other interface classes can be added) and address the problem of offset misalignment.

   Assume a producer declares the interface class `struct A { int x; int y; };` and a consumer expects to receive `struct B { int y; };`. [6] Therefore, the memory layout of `B` must be changed to `struct B { int id; int y;};`, where `id` is an arbitrary identifier (the consumer does not use this field anyway).
2. We allow forward-declaration in interface classes. Specifically, a consumer may specify only declaration of a method in the interface object and a producer can provide implementation.

   Consider the following example. Assume the producer sends the interface class `struct A { int foo() { ... } };` to the consumer and the consumer expects to receive `struct B { int foo(); }`. The latter contains only declaration of `foo()`. By solving the CSP-WS the definition is propagated from `A` to `B`. In our approach, definitions of various methods can be provided various services.
3. As part of our configuration mechanism, we resolve a 'conflict' between method definitions in a producer's and a consumer's class. The conflict arises if the producer provides a definition of the method in the interface object that is already present in the consumer's interface.
4. We introduce support for C++ class private members. Private members are members of the class that can only accessible within a service where the class is defined. On the other hand, the private members cannot be removed from the class, because it would change field offsets in the class.

   To solve the problem, we propose to rename private members to random names before they are sent to other services. In this way, we keep the private members in the object and make sure that they cannot be accessed outside the service.

An overview of the existing approaches in [AGR13a, AGR13b] shows that structural support of objects in web service is limited. Comparing to the existing approaches, our solution is more functional and expressive.

[6] In C++, **struct** is a class with exclusively public fields.

8.3.1 Structure of the Interface Class

For simplicity, we consider only basic features of the C++ class. In this model, we see class as a collection of public fields and methods. In the following subsections, we expand the class with other features, such as private members, templates, etc.

In the MDL we encode a class as a record term, where members of the class map to the record elements. A record is an extendable term. While configuring the interfaces, we store additional elements, which encode inherited class members, in the tail variable of the record.

Class fields and methods are encoded in the MDL in the following way. If a class member is a field, then the record contains an element with the label equal to the field name; similarly, if a class member is a method, then the record contains an element with the label equal to the method's name and argument types provided as a string.

For example, the class `struct A { int a; string foo(int x) { ... } };` is described by the following term in the MDL: $\{\mathsf{a}\colon \texttt{int}, \mathsf{foo}(\mathsf{int})\colon (\texttt{string}\ c^{\uparrow})\ |t^{\downarrow}\}$.

If an element of the record represents a class field, then the record subterm encodes the field type as a symbol (or a tuple, if qualifiers from Section 8.2 are supported). Similarly, if the element represents a method, then the record subterm encodes the method return type and a reference to the method implementation, both stored in a tuple. In the example above, the implementation is stored in the variable $c^{\uparrow}$. The structure of $c^{\uparrow}$ is discussed in Section 8.3.3.

Such representation follows the overloading rules for class members in C++ [Str13]. The variables cannot be overloaded, therefore, the class record may contain only one element for each field in the class. Using method overloading, one can define methods with the same name, but different arity or argument type. Therefore, an element of the record for a method is uniquely identified by the name and a list of argument types. For example, there are two elements for a method overloaded with `string foo(int x)` and `string foo(string x)`: an element with the label `foo(int)` and an element with the label `foo(string)`.

8.3.2 Field Inheritance

Inheritence of fields in the interface class is similar to the inheritance of message elements as described in Section 8.1. The interface class can be extended with additional fields.

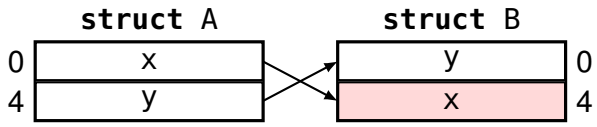
We illustrate field inheritance using the following example. Assume that the producer declares the class `struct A { int x; int y; };` as an interface class sent to the output. Then, the consumer may contain any subset of the fields in `A`. For example, `A` can be matched by `struct B { int y; };`. On the other hand, lack of `x` in `B` changes object

memory layout. Therefore, all fields that are present in A, must be inherited in B. This can be achieved using the constraint

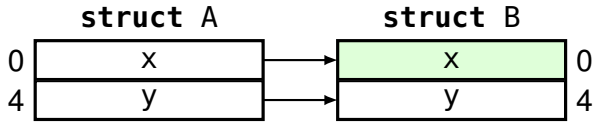
$$\underbrace{\{\texttt{x: int}, \texttt{y: int}\}}_{\texttt{struct A}} = \underbrace{\{\texttt{y: int} \mid a^{\downarrow}\}}_{\texttt{struct B}},$$

where the left term represents the interface of A and the right term represents the interface of B (the equality relation on terms is defined in Eq. (3.1)). $a^{\downarrow}$ inherits fields that are present in A, but not present in B. Furthermore, if B contains fields that are not present A, then the constraint is unsatisfied and a communication error occurs.

If the constraint is satisfied and B is provided with all fields that it lacks from A, then the memory layout problem needs to be fixed. Indeed, in the current example, inserting `int x` to the end of B leads class incompatible with A (see illustration in Figure 8.8a). After configuration one must ensure that the offsets for all fields in A and B are exactly the same. For these purposes, our mechanism reorders class members in lexicographical order on label names.



(a) The field x added to the end of the class breaks compatibility between A and B



(b) The field x added to the correct position, which does not break class compatibility

Figure 8.8: Inherited data placement in the class affect correctness and compatibility of class memory layouts

8.3.3 Method Inheritance

Configuration of interface class methods is more difficult comparing to configuration of the fields. We consider several cases.

Consumer's methods that contain declaration only

In the first case, a consumer may contain an interface object's method with only declaration. We illustrate it using the following example.

A consumer expects an instance of a class that represents a geographical map. Furthermore, the class declares a method `findLocation` that retrieves coordinates of the

location given an address or a name (for example, Google Places API provides such kind of functionality [Goo]):

```
struct Map {
    GeoMap data;
    // 'findLocation' returns longitude and latitude as a pair
    pair<double, double> findLocation(string place);
};
```

The producer must provide an implementation of `findLocation` encoded as an MDL term.

Implementation of the method is encoded in the MDL in a following way. We specify a term that stores the code as a singleton choice: $c^{\uparrow} = (\text{:code: } \texttt{hash\_12345}\text{:})$. Here the code is a keyword and `hash_12345` is a hash code of the string representing the method's code. A lookup table that maps a hash to the source code (encoded as a string) is shared between all services in the network.

Defining the term for a code as a choice allows the producer to propagate the method definition on the corresponding class in the consumer's interface. In the example above, assume that the producer intends to provide the consumer with **`struct`** `Map` that contains the field `data` and the definition of `findLocation` method:

```
struct Map {
    GeoMap data;
    // 'findLocation' returns longitude and latitude as a pair
    pair<double, double> findLocation(string place) { ... }
};
```

The relation is specified by the following constraint:[7]

$$\{\text{`findLocation(string)'}\text{: (`}\texttt{pair<double, double>}\text{' (:code: }\texttt{hash\_12345}\text{:))}\,|p^{\downarrow}\} = \{\text{`findLocation(string)'}\text{: (`}\texttt{pair<double, double>}\text{' } c^{\uparrow})\,|q^{\downarrow}\}, \quad (8.1)$$

where $c^{\uparrow}$ is a variable, which corresponds to the definition of `findLocation` method. The variable is set to none initially (that is, no definition is present, only declaration). $c^{\uparrow} = (\text{:code: } \texttt{hash\_12345}\text{:})$ is a solution to the constraint (the code can be obtained by getting a value for the hash `12345` in the code lookup table. Notice that the $c^{\uparrow}$ would remain equal to the initial approximation nil if a record term would be used instead of a choice to represent the method implementation.

[7] Quotation marks in labels and names are introduced to prevent term ambiguity caused by presence of spaces and parenthesis in the symbols.

Availability of method's source code across the services does not raise privacy issues. The services exchange the interface objects anyway, so behaviour of the interface objects is explicitly public.

The constraint for the methods (8.1) is defined in a form of an equality on records with tail variables. It allows the producer to contain methods that are not required by the consumer and the consumer to contain methods that are not provided by the producer. The latter is perfectly acceptable if the consumer's method contains the definition (that is, the consumer does not require any information from the producer). Otherwise, the consumer lacks the method's definition and the error must be reported. In terms of the MDL interfaces, $c^{\uparrow}$ remains none in the latter case. Since $c^{\uparrow} = \text{none}$ is a satisfiable solution, detection of the error while solving the CSP-WS is impossible, but can be done during the next configuration step (*Header file generation* step in Figure 8.1) by simply checking the value of *code variables* that must not be equal to none.

Consumer's methods containing definitions

In general case, consumer's interface objects may contain method definitions, similarly to producer's interface objects too. On the other hand, if both the producer and the consumer contain the same method's definition, a conflict arises. In this case, there are three scenarios:

1. In the most trivial scenario, the code provided by the producer is the same as the one defined by the consumer. To recall, in the MDL we represent the code as a symbol that is a hash of the original code string. As a result, with the current implementation, the code of the producer and the consumer can match only if the strings representing the code are literally identical. For example, the code `{ return f(n-1) + f(n-2); }` matches only itself, and not `{ return f(n-1)+f(n-2); }` due to different formatting. Clearly, in real-world scenarios with services developed independently, the code is almost never matched given this approach. More intelligent 'code matching' algorithm, specifically the one that compares behaviour specification, can be used without breaking consistency of the overall configuration mechanism.

2. In the second scenario, the consumer may want to ignore declaration or definition of the method provided by the producer and use its own definition instead. In this case, the definition encoded in the MDL represents a switch like in the following

example:

$$\{\text{'findLocation(string)'}\text{: (}\texttt{'pair<double, double>'}\ \text{(:code: }\texttt{hash\_12345}\text{:))}\ |p^{\downarrow}\} = \{\text{'findLocation(string)'}\text{: (}\texttt{'pair<double, double>'}\ \langle(r)\text{: }\texttt{hash\_67890}, (\neg r)\text{: } c^{\uparrow}\rangle)\ |q^{\downarrow}\}. \tag{8.2}$$

The term $\langle(r)$: `hash_67890`, $(\neg r)$: $c^{\uparrow}\rangle$ in the consumer's interface contains the b-variable r that is true if no definition is provided by the producer or the provided one matches the consumer's method (in this case, refer to Item 1); alternatively, r equals false if the definition provided by the producer does not match the consumer's definition. In this case, a hash code of the string representing the method definition is stored as a value of $c^{\uparrow}$. If the consumer wants to use it's own implementation, then it ignores the value in $c^{\uparrow}$ and uses the value that corresponds to the hash `hash_67890`.

3. Finally, in the third scenario a consumer contains the method's definition but prefers to replace it with the producer's method definition if provided. The definition encoded in the MDL as a switch term (like in the case 1) handles this case too. If the overrided method is provided, then r equals false and the configuration algorithm must select the hash value stored in $c^{\uparrow}$. Otherwise, r is false and no overrided version is provided.

The cases 2 and 3 are alternative scenarios. The configuration mechanism selects one of them depending on service settings. Specifically, for each method it must be specified whether the method overriding is supported or not. We achieve it by reusing the C++ keyword **`volatile`** (the original meaning is meaningless in the context of web services). The methods that support overloading must have the **`volatile`** specifier, which is used during interface derivation step and is removed before the code derivation.

8.4 Implementation

The toolchain is developed in C++ and OCaml, also using Clang for the C++ code analysis and the PicoSAT solver as part of the CSP-WS solver [Zai17]. The toolchain configures the service interfaces in five steps as illustrated in Figure 8.1:

1. For each file that contains a service source code in C++, augment them with macros acting as placeholders for the code that enables flow inheritance (as illustrated in Figure 8.4);
2. Derive the interfaces from the code (see Figure 8.2);

3. Given the interfaces and the application topology, construct the constraints to be passed on to the CSP-WS algorithm;
4. Run the CSP-WS solver;
5. Based on the solution, generate header files for every service with macro definitions. In addition, the tool generates the API functions to be called when a service sends or receives a message;
6. Given the modified service source files and the generated header files, create a binary library for every service.

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

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

- Interfaces and the code behind them can be generic as long as they are sufficiently configurable. No communication between services designers is necessary to ensure consistency in the design.
- Configuration and compilation of every service is separated from the rest of the application. This prevents source code leaks in proprietary software running in the Cloud.[8]

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

Chapter 9

Message Synchronisation in Stateful Services

The interface configuration protocol which was presented in Chapter 8 is applicable only for stateless services. Such services are typically used for querying information without modifying the service state itself. The interface configuration protocol for such services is simpler than is the protocol for generic stateful services. This means that interaction between service input messages is not allowed. In other words, the interface configuration protocol from the previous chapter can only be applied to services in which an output message is a function of a single input message.

However, a generic service may indeed have a state. In order to be able to argue about the correctness of stateful services we need a service behaviour specification to be expressed in some sort of protocol. There exist approaches, such as multiparty session types [HYC08], which can be used to prove communication correctness in generic services. In our approach, we do not prove communication correctness in stateful services. However, our mechanism can be useful for showing the correctness of message composition in services, where an output message is produced as a combination of multiple input messages.

We support stateful services in the configuration protocol by externalising a service state in the form of a message (see Figure 9.1). This technique is useful for service reliability and fault-tolerance, because the state of the service can be recovered from a 'state' message if the service has crashed. As a result, each service with an externalised state has at least two input ports, one of them for receiving messages that represent the externalised state of the service.

We apply the interface configuration protocol for stateful services (in addition to stateless services) using *synchronisers*. Synchronisers are special services that combine several input messages into one message and forward the constructed message to the

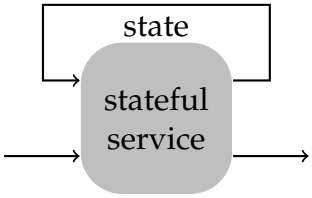


Figure 9.1: A stateful service that stores its state as a message

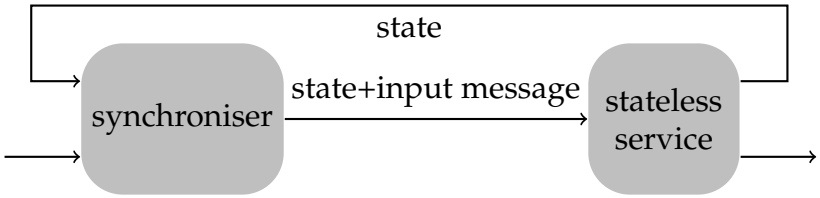


Figure 9.2: A stateful service is modelled as a synchroniser, which merges an input message and a state, and a stateless service, which receives a message from the synchroniser

output. Each stateful service can be represented as a synchroniser prepended to a stateless service (see Figure 9.2).

A synchroniser is defined as a state transition system in a domain-specific language. The synchroniser has a set of predefined states and transitions between them. A transition is associated with one of the input ports and is triggered upon receipt of a message from the port. The synchroniser may also have a storage for integer values, which is called *state variables*, and a storage for input messages, which is called *store variables*. In each state, the synchroniser can do the following:

- set or update the value for state or store variable;
- send a message, which is formed using an input message, or a state or store variable, to output ports (one or more);
- switch a state.

We extend the interface configuration protocol with a mechanism that analyses the synchroniser and automatically derives interfaces from the synchroniser. This allows us to reuse the protocol from Chapter 8 for stateful services as well. The interface configuration protocol for stateful services is reduced to the protocol for stateless services. We replace each stateful service with a pair consisting of a synchroniser followed by a stateless service (see Figure 9.2). The synchroniser combines input messages with a state message. As a result, the service receives its previous state and an input in a single message.

In this chapter we present the syntax for the synchronisers. The interface configuration protocol does not support the synchronisers as it is; however, below we discuss how we can represent synchroniser interfaces and internal state transitions in the form of the seniority constraints. We specialise the synchroniser interfaces by resolving the

constraints in the CSP-WS. We use the MDL term *union* (see Section 3.5) to describe a message that is constructed as a composition of two messages.

The synchronisers described in this chapter are supported in a coordination language AstraKahn [Sha13], which currently exists in the form of a prototype. The syntax and the semantics of the language for synchronisers are provided in [Sha13, Kuz16, Tik15].

9.1 Syntax

We define the language of synchronisers as follows.

9.1.1 Header

⟨*synch_decl*⟩ ::= ‘`synch`’ ⟨*synch_name*⟩ ‘`(`’ ⟨*ports_decl*⟩ ‘`)`’
‘`{`’ ⟨*variable_decl*⟩* ⟨*state_decl*⟩$^+$ ‘`}`’

⟨*synch_name*⟩ ::= ⟨*identifier*⟩

The header of the synchroniser contains the name of the synchroniser and a set of ports that specify a set of input and output ports. The body of the synchroniser is specified in braces: the body contains declarations of *state* and *store* variables that are followed by a description of a finite-state machine, which specifies the synchroniser behaviour.

⟨*ports_decl*⟩ ::= ⟨*input_port_name*⟩ [‘`,`’ ⟨*input_port_name*⟩]* ‘`|`’
⟨*output_port_name*⟩ [‘`,`’ ⟨*output_port_name*⟩]*

⟨*input_port_name*⟩ ::= ⟨*integer*⟩

⟨*output_port_name*⟩ ::= ⟨*integer*⟩

The ports in the header declare a set of input and output ports that are used for wiring services. Following a convention from the configuration protocol, which is presented in Chapter 8, the port names are integers.

9.1.2 Body

Variable declaration

⟨*variable_decl*⟩ ::= ‘`store`’ ⟨*identifier_list*⟩ ‘`;`’
| ‘`state`’ ⟨*type*⟩ ⟨*init_decl*⟩ [‘`,`’ ⟨*init_decl*⟩]* ‘`;`’

⟨*type*⟩ ::= ‘`int`’

⟨*init_decl*⟩ ::= ⟨*identifier*⟩ [‘`=`’ ⟨*integer*⟩]

⟨*identifier_list*⟩ ::= ⟨*identifier*⟩ [‘`,`’ ⟨*identifier*⟩]*

Store variables are used for storing messages. The variables are typed. A type of a store variable is specified as an MDL term, which can be derived automatically (see Section 9.2). If a variable store a message from an input, then the type equals to the MDL term associated with an input port where the message comes from. Otherwise, if a stored message is constructed within the synchroniser using a union operation `||`, which is presented in Section 9.1.3, then a union term (`union` ...) is used to represent the type. Similarly to service interface, the types of store variables may be generic (i.e. they may contain t-variables and b-variables) outside the environment, and contextualised later using the CSP-WS, when the synchroniser is inserted in the context.

A sychroniser is a coordination primitive. Its purpose is to combine multiple messages into one before the messages are sent to a stateless service. The synchroniser should be expressive enough to be able to specify various kinds of synchronisation primitives. For this purpose, we introduce support for integer arithmetic to synchronisers (see *int_expr* rule in Section 9.1.3). The integers can be used for storing array indices, collection sizes, message depths, etc. We introduce state variables to store a computation result for integer expressions.

9.1.3 States

⟨*state_decl*⟩ ::= ⟨*state_name*⟩ ‘`{`’ ‘`on:`’ ⟨*transition*⟩$^+$ [‘`elseon:`’ ⟨*transition*⟩$^+$]* ‘`}`’

⟨*state_name*⟩ ::= ⟨*identifier*⟩

⟨*transition*⟩ ::= ⟨*port_name*⟩ [‘`.`’ ⟨*msg_pattern*⟩] [‘`&`’ ⟨*predicate*⟩]
‘`{`’ ⟨*statements*⟩ ‘`}`’

⟨*port_name*⟩ ::= ⟨*input_port_name*⟩
| ⟨*output_port_name*⟩

⟨*msg_pattern*⟩ ::= ⟨*record_pattern*⟩
| ‘`else`’

⟨*record_pattern*⟩ ::= ‘`(`’ ⟨*identifier_list*⟩ [‘`||`’ ⟨*tail*⟩]‘`)`’

⟨*tail*⟩ ::= ⟨*identifier*⟩

⟨*predicate*⟩ ::= ⟨*bool_expr*⟩

Each state has a name and a set of transitions associated with the state. A synchroniser must have an initial state, which is called `start` by convention.

Transition declaration starts with `on:` or `elseon:` keyword followed by the pattern expression. When a new message arrives to one of the input ports, it gets matched

against one of the predefined patterns in the current state. The pattern consists of a port name, a record pattern and a Boolean expression that involves local (visible within the current transition only) and state variables. If a single input message is match by multiple patterns, by convention the priority is given to the top-most transition that matches the pattern.

Example 9.1.1.

```
1.(x, y || z) & p = 0
```

is a predicate on the port #1, which is matched if a record in the input message contains elements with labels `x` and `y` and the state variable `p` is zero. For example, a message `{x: 1, y:2, z: 3, w: 4}` is matched by the predicate. Furthermore, in this case the synchroniser will create and initialise local variables `x = 1`, `y = 1`, `z = 1`, and `w = {z: 3, w: 4}`. △

Transition

A transition contains a set of statements that are executed when the transition pattern is matched. The statements are optional, but the statements are executed in the order they are specified.

⟨*statements*⟩ ::= [⟨*set_stmt*⟩] [⟨*send_stmt*⟩] [⟨*goto_stmt*⟩]

⟨*set_stmt*⟩ ::= '`set`' ⟨*assign*⟩ ['`,`' ⟨*assign*⟩]* '`;`'

⟨*assign*⟩ ::= ⟨*identifier*⟩ '`=`' (⟨*int_exp*⟩ | ⟨*data_exp*⟩)

⟨*send_stmt*⟩ ::= '`send`' ⟨*dispatch*⟩ ['`,`' ⟨*dispatch*⟩]* '`;`'

⟨*dispatch*⟩ ::= ⟨*data_exp*⟩ '`=>`' ⟨*output_port_name*⟩

⟨*goto_stmt*⟩ ::= '`goto`' ⟨*state_name*⟩ ['`,`' ⟨*state_name*⟩]* '`;`'

There are three statement categories. The first statement type starts from the `set` keyword. Essentially, the statement assigns new values to store or state variables. *int_expr* is an expression that evaluates to an integer, and *data_expr* is an expression that produces a new message (its syntax is provided below).

The statement that starts with `send` keyword sends a message (input message, the one that is stored in a store variable, or a message constructed from other messages) to the specified output port.

Finally, the last type of the statement starts with `goto` keyword. It specifies the next state of the synchroniser. If various states are provided, then the state is selected non-deterministically.

The syntax for message constructing expressions is the following:

⟨*data_exp*⟩ ::= ⟨*atom_list*⟩ | '(' ⟨*atom_list*⟩ ')'

⟨*atom_list*⟩ ::= ⟨*atom*⟩ ['||' ⟨*atom*⟩]*

⟨*atom*⟩ ::= 'this' | ⟨*identifier*⟩ ':' ⟨*rhs*⟩

⟨*rhs*⟩ ::= ⟨*int_exp*⟩ | ⟨*identifier*⟩

A message can be produced from 1) a store variable 2) a key-value pair that is transformed to a singleton record 3) the input message accessed by a keyword `this` and 4) a union operation `||` that combines two message expressions into one.

Example 9.1.2. The synchroniser that implements a simple merger (Figure 3.4) can be defined as follows:

```
synch merger (1, 2 | 3) {
    store a, b;
    start {
        on: 1 { set a = this; goto wait2; }
        elseon: 2 { set b = this; goto wait1; }
    };
    wait2 {
        on: 2 { send (this || a) => 3; goto start; }
    };
    wait1 {
        on: 1 { send(this || b) => 3; goto start; }
    };
}
```

In the `start` state, the merger expects a message from either a port `1` or `2`. If the message from `1` is received, the merger stores the input message in the variable `a` and changes its state to `wait2`; likewise, if the message from `2` is received, the merger stores the input message in the variable `b` and changes the state to `wait1`. In the states `wait1` and `wait2` the merger waits for a message from another port, sends it to the output port and transits to the `start` state. △

Below we informally describe a mechanism for deriving interfaces from the synchroniser. This allows to integrate synchronisers into the interface configuration protocol and ensure that store variables declared in the synchronisers and their interface formats are compatible with interfaces of other services in the environment.

9.2 Constraint Derivation

In this section we describe an idea of an algorithm for deriving generic interface from a synchroniser. After deriving the interfaces, the interfaces can be configured while solving the CSP-WS. As a result, the interface configuration protocol for stateless services can be extended with support for the synchronisers.

The algorithm that derives the MDL interfaces from a synchroniser consists of two parts. First, with each input and output port in the synchroniser we associate a free t-variable. A free t-variable encodes the most generic interface. Next, the synchroniser is analysed and auxiliary seniority constraints which restrict variable values are derived from the synchroniser. Particular values that satisfy the constraints are found by solving the CSP-WS.

In order to produce a set of the auxiliary constraints, all transitions in the synchroniser must be traversed. A transition is triggered when an input message received from a given port matches a record pattern and a Boolean predicate (if the latter is specified). The record pattern specifies record labels that must be present in the message. For example, given a record pattern `( a, b, c || t)` and t-variable $p^{\downarrow}$, which is associated with an input port, the constraint

$$p^{\downarrow} \sqsubseteq \{\mathsf{a}\colon v_1^{\downarrow}, \mathsf{b}\colon v_2^{\downarrow}, \mathsf{c}\colon v_3^{\downarrow} \,|\, t\},$$

where $v_1^{\downarrow}$, $v_2^{\downarrow}$ and $v_3^{\downarrow}$ are free variables, specifies that the input message defined by $p^{\downarrow}$ matches the pattern. In this example, elements identified by the labels `a`, `b`, and `c` can be accessed as local variables. For example, if a predicate `a < 0` is used in a condition, then `a` must be an integer and the constraint

$$v_1^{\downarrow} \sqsubseteq \mathsf{int}.$$

must be produced.

In a single state only one transition is triggered when the synchroniser receives a message. As a result, constraints that correspond to different transitions in one state are alternative. This behaviour can be specified by using a switch term.

In addition to the constraints that correspond to transition patterns, we must add constraints for statements in a synchroniser. For this purpose with each store variable we associate a free t-variable. The t-variables represent the format of a message stored in a given variable. The t-variables can be refined

For an assignment statement `set` that involves store variables, we can generate constraints of the form

$$t \sqsubseteq a^{\downarrow},$$

where $a^{\downarrow}$ is an interface variable associated with the store variable being assigned, and t represents a term constructed for an expression *data_exp* (a procedure for constructing terms for *data_expr* is explained below). For the `send` operator, we generate a constraint of the same form, where t corresponds to the message being dispatched and $a^{\downarrow}$ is the interface variable of the output port. For `goto` statement we don't add any additional constraints. Instead, we change the state and repeat the constraint derivation algorithm. If multiple states are specified in the `goto` statement, then the algorithm must include all constraints derived from all of the states to a single set, because at runtime the synchroniser can switch to any of the states.

The *data_exp* expression is constructed from the input message, messages from the store variables, a singleton record and `||` union operation. An MDL term t that corresponds to the expression can be constructed in a straightforward manner. The MDL term for the input message is the variable associated with the input port; store variables have a corresponding term as well; a singleton record maps to an MDL record. These three kinds of the MDL terms can be united with a union tuple (see Sections 3.5 and 7.6).

Example 9.2.1. For the synchroniser in Example 9.1.2, the interfaces and the constraints are derived as follows.

Let $a^{\downarrow}$ and $b^{\downarrow}$ be t-variables that are associated with store variables `a` and `b`, respectively; and let $c^{\downarrow}$, $d^{\downarrow}$ and $e^{\downarrow}$ be t-variables that are associated with input ports #1, #2 and output port #3, respectively.

The constraint aggregation starts from the state `start`. There are two transitions that correspond to two input ports. In the first transition, `this` corresponds to a message from the input port #1; in the second transition, `this` corresponds to the input port #2. From the assignment statements in the `start` state we derive two constraints:

$$c^{\downarrow} \sqsubseteq a^{\downarrow}$$
$$d^{\downarrow} \sqsubseteq b^{\downarrow}.$$

From the `start` state the synchroniser can move to states `wait2` and `wait1`. Therefore, the constraints that follow from these states must be generated. In both states, the union operator `||` is used. As a result, from `wait2` the constraint

$$(\mathsf{union}\ d^{\downarrow}\ a^{\downarrow}) \sqsubseteq e^{\downarrow}$$

is generated. Similarly, from `wait1` the constraint

$$(\mathsf{union}\ c^{\downarrow}\ b^{\downarrow}) \sqsubseteq e^{\downarrow}$$

is generated.

The derivation result consists of the interfaces $c^{\downarrow}$, $d^{\downarrow}$ and $e^{\downarrow}$ associated with the ports and a set of the constraints that specify relations between the interfaces. △

Chapter 10

Image Processing Use Case for Interface Configuration Protocol

In this chapter we illustrate the interface configuration protocol, which we presented in Chapter 8, using a practical use case.

The use case is an image segmentation algorithm that is based on k-means clustering [Mac67]. The structure of the application is shown in Figure 10.1. We selected the application that contains only stateless services, because synchronisers (see Chapter 9) are not yet properly supported in the protocol.

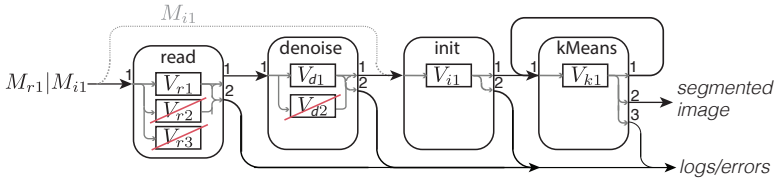


Figure 10.1: An image segmentation algorithm based on k-means clustering that is implemented as a service-oriented application

The application network represents a pipeline composed of four services:

- The service `read` opens an image file using an input message M_{r1} with the file name, and sends it to the first output port. The service contains 3 functions that overload behaviour of the service. In terms of the MDL, the input interface of the component is defined by 3 choice variants: 1) V_{r1} loads the colour image in RGB format; 2) V_{r2} loads the grayscale image as an intensity one; and 3) V_{r3} loads the image as it is stored in the file. The second output port of the service is intended for sending auxiliary and debugging information, such as logs and error messages.
- The service `denoise` preprocesses the input image by removing the noise from the images. The service contains two functions that remove the noise from colour and

grayscale images, respectively. Both functions produce images of the same format, which are directed to the first output port. Similarly, to the `read` service, `denoise` has the second output port for producing error messages.

- The service `init` sets initial parameters for the k-means algorithm. The service contains one processing function V_{i1}. The input message can come either from the service `denoise` or from the environment with an input message M_{i1} if it has been opened and preprocessed before. The input message must contain the number of clusters `K` and the image itself. Similarly to the `read` and `denoise` service, `init` has a second output port for sending auxiliary output.
- The `kMeans` service represents an iterative implementation (defined as a function V_{k1}) of the k-means algorithm. The result of each iteration is sent to the first output port, which is circuited back to the input port of the component itself. This kind of design gives an opportunity to manage system load in the run-time and execute the next iteration only when sufficient resources are available. Once the cluster centres have converged, the algorithm yields the result to the second output port. Furthermore, the service has a third output port for sending logs.

Using flow inheritance, a variant M_{i1} is routed directly to the `init` service bypassing the services `read` and `denoise`. Similarly, again using flow inheritance, a parameter `K` that is contained in M_{r1} is implicitly bypassed through `read` and `denoise` to `init`.

The interface reconciliation algorithm is capable of finding out that V_{r2} and V_{r3} are not used with the provided input, and functions containing the implementations will be excluded from the generated code.

In Appendices A.1.1 to A.1.4 we provide the code for these four services. All services include the configuration header `mdl.h` that contains protocol-specific declarations, such as `salvo`, `service`, as well as API for service communication with the environment. The core of a service (see Section 8.1.1) contains implementation of the service. The implementation is structured as a set of processing functions that process input messages of various formats (these functions are distinguished from other functions by the return type `service`). Each processing function calls functions with the return value `salvo`, which triggers message dispatch to one or more communication ports depending on a channel routing specified in the shell (see Section 8.1.2).

The shell is a file that is associated with each service. It allows to

1. rename labels in a choice term that represents output message variants;
2. rename labels in a record term that represents an output message;
3. customise output message port routing.

The shell provides a mechanism to solve the problem of service interface mismatch that is caused by different labels or ports. The problem is addressed during service composition, because service code should remain unaffected. Indeed, a service designer can choose any label names and ports without relying on other services and the context.

The network topology for the image processing example is provided in Appendix A.2. The shells below are created in a way that the names exported from the producing service match the names in the consuming service.

The shell for the `read` service is specified as follows:

```
send_color -> 1
send_grayscale -> 1
1[send_color/denoise_color]
1[send_grayscale/denoise_grayscale]
```

The shell redirects `send_color` and `send_grayscale` salvos to the first output port and renames the salvo names from `send_color` and `send_grayscale` to `denoise_color` and `denoise_grayscale`, respectively.

The shell for the `denoise` service is specified as follows:

```
1[send_img/init]
```

The shell renames the salvo `send_img` to `init` in the first output port.

We do not provide the shell for the service `init`. Therefore, the default port routing, which is specified in the source file, is used.

The shell for the `kMeans` service is specified as follows:

```
result -> 2
error -> 3
1[loop/kMeans]
```

The shell redirects the salvos `result` and `error` renames the salvo that is sent to the first output port.

The shell is analysed during the interface derivation (see Section 10.2) in the interface configuration protocol.

An application interacts with the environment. The environment provides an input for the application and consumes output messages produced by the application. We model environment as a separate service. In contrast to other services, the environment does not have an implementation. Instead, it explicitly exposes its interfaces as the terms. The interfaces expose the data format that is expected or demanded in the environment. The terms that specify the environment interfaces do not contain variables, because the format of the provided data is always known.

10.1 Service Code Transformation

In the first step of the interface configuration protocol, we modify the code for each service and introduce a configuration macros. The macros are placeholders for the code that implements flow inheritance (Section 10.4 contains an example). Furthermore, we annotate the code with macros that allows to exclude redundant (in the given context) functions from the library.

Below we demonstrate a transformed code for the `read` service. A complete code snippet for transformed services is provided in Appendices A.3.1 to A.3.4.

While transforming the code we wrap salvos and function declarations/definitions in `#if ... #endif` macro:

```
#if defined(f_read_color_1) || defined(f_read_unchanged_1)
salvo send_color(std::vector<std::vector<double>> img
                 read_read_color_1_read_unchanged_1_send_color_decl);
#endif
```

In this example `f_read_color_1` and `f_read_unchanged_1` represent Boolean variables. Values for Boolean variables are resolved while the CSP-WS is solved. `f_read_color_1` is true only if the function `read_color_1` is required in the context; similarly, `f_read_unchanged_1` is true only if the function `read_unchanged_1` is used in the given application. `read_color_1` and `read_unchanged_1` are functions that can produce `send_color` salvo. If none of these functions is needed, the salvo definition can safely be removed from the library.

Furthermore, we add a macro `read_read_color_1_read_unchanged_1_send_color_decl` to the declaration of the salvo `send_color`. The macro corresponds to a tail variable in an MDL term and implements flow-inheritance: the declaration and the definition of the function is expanded with additional arguments after the CSP-WS is solved.

We transform a function definition in a similar way:

```
#ifdef f_read_color_1
service read_color_1(std::string fname read_read_color_1_decl) {
    cv::Mat image = cv::imread(fname, CV_LOAD_IMAGE_COLOR);
    if(!image.data)
        error_2("Could not open or find the image"
                read_read_color_1_read_grayscale_1_read_unchanged_1_error_use);
    else {
        send_color(image
                   read_read_color_1_read_unchanged_1_send_color_use);
    }
}
#endif
```

In this snippet, the function `read_color_1` must be present in a library only if the Boolean variable `f_read_color_1` is true. Otherwise, the function is not used and can be removed from the library.

Within the service, flow inheritance is propagated using variables `read_read_color_1_decl`, `read_read_color_1_read_grayscale_1_read_unchanged_1_error_use` and `read_read_color_1_read_unchanged_1_send_color_use`. `read_read_color_1_decl` provides a list of arguments to propagate to output; `read_read_color_1_read_grayscale_1_read_unchanged_1_error_use` and `read_read_color_1_read_unchanged_1_send_color_use` must contain subsets of arguments from `read_read_color_1_decl`. Depending on the context, CSP-WS finds subsets with elements demanded by other services in the pipeline. The relation between the variables is specified by generating auxiliary constraints in the constraint generation phase:

$$\text{read\_read\_color\_1\_decl} \sqsubseteq \text{read\_read\_color\_1\_read\_grayscale\_1\_read\_unchanged\_1\_error\_use}$$
$$\text{read\_read\_color\_1\_decl} \sqsubseteq \text{read\_read\_color\_1\_read\_unchanged\_1\_send\_color\_use}.$$

In addition to annotating services and salvos, in this step we generate an API. Specifically, we generate 1) functions that are called by the run-time environment when a message arrives to the service, and 2) functions that are called within a library to notify the run-time environment when the service sends a message.

```
void input_1(salvo&& _msg) {
#ifdef f_read_color_1
    if (_msg.getType() == "read_color") {
        cereal::BinaryInputArchive iarchive(_msg.ss);
        std::tuple<std::string read_read_color_1_types > _data;
        iarchive(std::get<0>(_data) read_read_color_1_tuple_get );
        read_color_1(std::get<0>(_data) read_read_color_1_tuple_get );
        return;
    }
#endif
#ifdef f_read_grayscale_1
...
#endif
#ifdef f_read_unchanged_1
...
#endif
read_read
}
```

`input_1` is a function called by a run-time system when the message `_msg` is received in the input port #1. Every message has a 'type': a choice variant that the message belongs to.

The function unpacks the message, deserialises[1] it and calls the service function `read_color_1` with the unpacked message provided in arguments. The function also contains the code for unwrapping the messages for `read_grayscale_1` and `read_unchanged_1` functions. `read_read` is a placeholder macro for message variants that are inherited and bypassed further in the pipeline.

Implementation of the salvo function `send_color` is provided below.

```
#if defined(f_read_color_1) || defined(f_read_unchanged_1)
salvo send_color(std::vector<std::vector<double> > img
                 read_read_color_1_read_unchanged_1_send_color_decl) {
    for (const std::pair<int, std::string>& _p : read_DOWN_send_color_ochannels)) {
        Message _msg;
        cereal::BinaryOutputArchive oarchive(_msg.ss);
        oarchive(img read_read_color_1_read_unchanged_1_send_color_use);
        _msg.setType(_p.second);
        output(_p.first, std::move(_msg));
    }
}
#endif
```

It receives the element of the output message `img` and the inherited arguments stored in the macro `read_read_color_1_read_unchanged_1_send_color_decl`. The function serialises the message and sends the output message to the runtime environment.

After the CSP-WS is solved, the code can be compiled into a library that contains all functions required for interaction with the run-time.

10.2 Interface Derivation

In the next step, the algorithm derives MDL terms from the annotated service code. Since the format of the service code is fixed, the algorithm uses code analysis to derive the structure of `salvo` and `service` functions. For analysis we use tools and libraries from LLVM project, such as `LibTooling` and `ASTMatchers`.

As a result, for the `read` service we obtain the following term as the interface for the input port #1:

$$(:\ \mathsf{read\_color}(f\_read\_read\_1\_color)\colon \{\mathsf{fname}\colon \texttt{string}\,|read\_color\_1^{\downarrow}\},$$
$$\mathsf{read\_grayscale}(f\_read\_read\_grayscale\_1)\colon \{\mathsf{fname}\colon \texttt{string}\,|read\_grayscale\_1^{\downarrow}\},$$
$$\mathsf{read\_unchanged}(f\_read\_read\_1\_unchanged)\colon \{\mathsf{fname}\colon \texttt{string}\,|read\_unchanged^{\downarrow}\}|read^{\uparrow}\ :)$$

[1] In our implementation with use `cereal` serialisation library (http://uscilab.github.io/cereal/).

At the top level, the term is a choice term that contains three variants that represent messages of various formats. Each variant contains a record. In this example, the record contains a name of an input file. Furthermore, every record contains a tail variable for supporting flow-inheritance for records; similarly, the choice contains a tail variable $read^{\uparrow}$ for supporting flow-inheritance for variants. The tail variables uniquely map to macros in the modified source files. Later, values for tail variables are transformed into macros definitions that are propagated back to the service code.

The rest of the interfaces are structured in a similar way. The interface for the input port #2 is specified as follows:

$$(:\ \mathsf{denoise\_color}(f\_read\_read\_color\_1 \vee f\_read\_read\_unchanged\_1) :$$
$$\{\mathsf{img}:\ \texttt{vec<vec<double> >} \,|\, read\_color\_1\_read\_unchanged\_1\_send\_color^{\downarrow}\},$$
$$\mathsf{denoise\_grayscale}(f\_read\_read\_grayscale\_1) :$$
$$\{\mathsf{img}:\ \texttt{vec<vec<double> >} \,|\, read\_grayscale\_1\_send\_grayscale^{\downarrow}\} | read^{\uparrow} :),$$

`vec<vec<double> >` is a shorthand for `vector <vector <double> >` — a 2-dimensional array of doubles, which represents an image matrix.

Similarly,

$$(:\ \mathsf{error}(f\_read\_read\_color\_1 \vee f\_read\_read\_grayscale\_1 \vee f\_read\_read\_unchanged\_1) :$$
$$\{\mathsf{msg}:\ \texttt{string} \,|\, read\_color\_1\_read\_grayscale\_1\_read\_unchanged\_1\_error^{\downarrow}\} :)$$

is the interface for the output port #1.

In addition to term derivation, we derive auxiliary constraints that specify a relation between t-variables in input and output interfaces:

$$read\_color\_1^{\downarrow} \sqsubseteq read\_color\_1\_read\_unchanged\_1\_send\_color^{\downarrow} \tag{10.1}$$
$$read\_unchanged\_1^{\downarrow} \sqsubseteq read\_color\_1\_read\_unchanged\_1\_send\_color^{\downarrow} \tag{10.2}$$
$$read\_color\_1^{\downarrow} \sqsubseteq read\_color\_1\_read\_grayscale\_1\_read\_unchanged\_error^{\downarrow} \tag{10.3}$$
$$read\_grayscale\_1^{\downarrow} \sqsubseteq read\_color\_1\_read\_grayscale\_1\_read\_unchanged\_error^{\downarrow} \tag{10.4}$$
$$read\_unchanged\_1^{\downarrow} \sqsubseteq read\_color\_1\_read\_grayscale\_1\_read\_unchanged\_error^{\downarrow} \tag{10.5}$$
$$read\_grayscale\_1^{\downarrow} \sqsubseteq read\_grayscale\_1\_send\_grayscale^{\downarrow}. \tag{10.6}$$

$read\_color\_1\_read\_unchanged\_1\_send\_color^{\downarrow}$ is a variable that specifies the data inherited in `send_color` salvo. Since the salvo is produced by `read_color` and `read_unchanged` services (where $read\_color\_1^{\downarrow}$ and $read\_unchanged\_1^{\downarrow}$ are inheriting variables), the variable $read\_color\_1\_read\_unchanged\_1\_send\_color^{\downarrow}$ must contain a subset of elements from

*read_color*_1$^{\downarrow}$ and *read_unchanged*_1$^{\downarrow}$ (not all elements need to be inherited). Similarly, elements in *read_color*_1_*read_grayscale*_1_*read_unchanged_error*$^{\downarrow}$ must be a subset of elements in *read_color*_1$^{\downarrow}$, *read_grayscale*_1$^{\downarrow}$ and *read_unchanged*_1$^{\downarrow}$, and elements in the record *read_grayscale*_1_*send_grayscale*$^{\downarrow}$ must be a subset of *read_grayscale*_1$^{\downarrow}$.

Services interact with the environment. Terms that are associated with the environment ports are provided explicitly. They specify the format of messages that is received from the external context.

10.3 Constraint Satisfaction

After generating terms and auxiliary constraints, the interface configuration protocol constructs a set of constraints $\mathcal{C}$ as the input for the CSP-WS. The auxiliary constraints are simply added to $\mathcal{C}$. Also, new constraints are constructed from the terms: for every communication channel in the topology graph, we take a pair of terms and construct a new constraint.

The constraints are solved using the algorithm from Section 7.5. The algorithm produces a set of possible solutions for the CSP-WS that depend on instantiations of Boolean variables. One can use different heuristics to select a solution that fits their needs. For example, a solution that contains more false variables is a solution that contains less data (because elements from collections are excluded by false guards). This allows us to drop all record/choice elements (i.e. to inherit less data) if they are not required by consumers. Other strategies for selecting a solution can be applied as well.

10.4 Library Generation

After finding a solution to the CSP-WS, we need to transform the solution to macro definitions and propagate them back to the service code. The transformation is straightforward. MDL symbols, such as `string`, `int` `vector<vector<double>`, are mapped to their corresponding C++ types. Only trueBoolean values are translated to macros; macros that corresponds to falsevalues remains undefined (as a result, condition `#ifdef` evaluates to false).

In the current version of the protocol, choices and records can appear on the top two levels of interface terms, and therefore, they cannot encode C++ types. The top level choices are transformed as illustrated in the following example. The algorithm finds a value for *read*$^{\uparrow}$ variable:

$$read^{\uparrow} = (:\ \mathsf{init}:\ \ldots :),$$

It means that `read` service must redirect messages with the tag `init` to the output. To achieve this, we generate a definition of the macro `read`:

```
#define read_read do {\
    if (_msg.getType() == "init") {\
        output(1, std::move(_msg));\
        return;\
    }\
} while (0);
```

This is an implementation of flow inheritance for elements of a top-level choice term. The code is triggered when an input message is received in port #1. If the message has tag `init`, it directly gets forwarded to the output port.

Flow inheritance for records is implemented as illustrated in the following example. Consider variables *read_color*_1$^{\downarrow}$ and *read_color*_1_*read_unchanged*_1_*send_color*$^{\downarrow}$, which are evaluated to the records:

$$\begin{aligned} \textit{read\_color\_1}^{\downarrow} &\sqsubseteq \{\texttt{K: int}\} \\ \textit{read\_color\_1\_read\_unchanged\_1\_send\_color}^{\downarrow} &\sqsubseteq \{\texttt{K: int}\}. \end{aligned}$$

*read_color*_1$^{\downarrow}$ is the tail variable in the input interface of the `read_color` function. Similarly, *read_color*_1_*read_unchanged*_1_*send_color*$^{\downarrow}$ is the tail variable in the output interface of the `send_color` salvo, which is called from `read_color`. Therefore, according to our configuration protocol, the argument `int K` must be propagated through the processing function to the output. To achieve this, we construct the following macros based on the CSP-WS solution:

```
#define read_read_color_1_decl COMMA int K
#define read_read_color_1_use COMMA K
#define read_read_color_1_read_unchanged_1_send_color_decl COMMA int K
#define read_read_color_1_read_unchanged_1_send_color_use COMMA K
```

If we include a header with the macros in the transformed service `read` (Appendix A.3.1, a configured service that propagates `K` argument from input to output will be obtained.

In such a way, we specialise all services in the network. After this, the service source files can be separately compiled into contextualised libraries. Such compilation does not raise security or privacy, because the code of individual services is not exposed to each other. Instead, the services only exchange their interface terms. As a result, our mechanism prevents source code leaks in proprietary software running in the Cloud.

Chapter 11

Conclusions and Outlook

Web services are often developed independently and without aiming for a specific context. As a result, the error-free composition of such services into a service-based application requires a great deal of effort. This often requires the modification of the service code, which is impossible for proprietary services (the ones for which the code is not available publicly).

In this thesis we solved the composition problem for web services. We developed a mechanism for configuring interfaces for generic services in the context. In our approach, services designers need to expose only service interfaces for application composition, and are not required to expose the code behind the services. We designed a language for describing service interfaces in the form of a term algebra with support for subtyping, polymorphism, flow inheritance [GSS08, GSS10], and configuration parameters. The configuration mechanism analyses the application topology and derives communication constraints for pairs of the interfaces. In order to accomplish this, we designed a simple type system, which collects relations between services in the application.

The constraints give rise to a constraint satisfaction problem (CSP). The problem cannot be solved in a straightforward manner using the existing SMT solvers. Furthermore, the presence of Boolean variables makes the problem NP-complete.

On theoretical grounds, our contribution is twofold:

- We designed a fixed-point algorithm that solves the CSP for constraints that do not contain Boolean variables.
- We designed a fixed-point algorithm that solves the CSP and improves the brute-force approach. The latter applies the fixed-point algorithm for each of the 2^n instantiations of Boolean variables. In contrast, the improved approach applies the fixed-point algorithm once for all Boolean variables.

On the practical side, we developed an interface configuration protocol for configuring

interfaces for web services coded in C++ [Zai17]. The protocol performs the following steps:

1. it automatically derives the interfaces from the code and constructs the communication constraints;
2. it solves the CSP using the original algorithm;
3. it propagates the solution back to the services in a form of configuration parameters.

The protocol is compatible with the proprietary service: the services do not need to expose their code to configure the interfaces. Furthermore, the protocol does not rely on a run-time model or a particular framework for implementing web services.

Currently, the protocol supports only stateless services, but it can also be extended to support stateful services. The stateful services can be described in the form of synchronisers. The synchronisers are state-transition systems for the specification of arbitrary message synchronisation patterns. In this thesis we described how interfaces can be derived from the synchroniser and how the interface configuration protocol can be reused for applications that contain synchronisers.

Unfortunately, the configuration mechanism currently lacks error reporting, something which is highly 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.

We designed the Message Definition language (MDL) to be flexible enough to be able to specify messages of any format. It provides support not only for basic formats, such as symbols, integers and labelled collections, but also for objects. The objects are data types that contain methods in addition to data fields, which can be inherited in pipelines using flow inheritance. In the future, it will be useful to add support for other data formats, such as generic collections, subtyped arrays [Sch03], and functions with support for behavioural subtyping (subtyping on object methods is not currently supported).

The efficiency of the proposed algorithm will be improved in future work. We can use the advantage of having an incremental SAT solver (MiniSAT, for example) [ES03]. It efficiently solves a sequence of incrementally-generated SAT instances, which is relevant to the algorithm. We can also provide an ordering of constraints in the problem, which minimises the number of constraint traversals.

The next major milestone in this direction of the research will be the integration of the interface configuration mechanism with a full fledged service management system (such as Google Borg [VPK+15]) or a stream-based coordination language (such as AstraKahn [Sha13]). These systems can provide an execution environment for managing data and computations, but they lack a mechanism for checking the 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.

Appendix A

Additional Details of Image Processing Use Case

A.1 Source Code

A.1.1 Read Service

```
#include "cal.h"

salvo send_color(vector<vector<double>> img);
salvo send_grayscale(vector<vector<double>> img);
salvo error_2(string msg);

service read_color_1(string fname) {
  cv::Mat image = cv::imread(fname, CV_LOAD_IMAGE_COLOR);
  if(!image.data)
    error_2("Could not open or find the image");
  else {
    send_color(image);
  }
}

service read_grayscale_1(string fname) {
  cv::Mat image = cv::imread(fname, CV_LOAD_IMAGE_GRAYSCALE);
  if(!image.data)
    error_2("Could not open or find the image");
  else {
```

```
    send_grayscale(image);
  }
}

service read_unchanged_1(string fname) {
  cv::Mat image = cv::imread(fname, CV_LOAD_IMAGE_UNCHANGED);
  if(!image.data)
    error_2("Could not open or find the image");
  else {
    send_color(image);
  }
}
```

A.1.2 Denoise Service

```
#include "cal.h"

salvo send_img(vector<vector<double>> img);
salvo error_2(string msg);

sevice denoise_color_1(vector<vector<double>> img) {
  cv::Mat result;
  cv::fastNlMeansDenoisingColored(img, result);
  if(result.empty())
    error_2("error");
  else
    send_img(result);
}

service _1_denoise_grayscale(vector<vector<double>> img) {
  cv::Mat result;
  cv::fastNlMeansDenoising(img, result);
  if(result.empty())
    error_2("error");
  else
    send_img(result);
}
```

A.1.3 Init Service

```
#include "cal.h"
#include <time.h>

salvo kMeans_1(vector<vector<double>> img,
               vector<vector<double>> old_centers_v,
               int K, double epsilon);
salvo error_2(string msg);

static void generateRandomCenter(const vector<cv::Vec2f>& box,
                                 float* center) {
  size_t j, dims = box.size();
  float margin = 1.f / dims;
  for (j = 0; j < dims; ++j)
    center[j] = ((float) rand() * (1.f + margin * 2.f) - margin) *
                (box[j][1] - box[j][0]) + box[j][0];
}

service init_1(vector<vector<double>> img, int K) {
  cv::Mat data0(img);
  bool isrow = data0.rows == 1 && data0.channels() > 1;
  int N = !isrow ? data0.rows : data0.cols;
  int dims = (!isrow ? data0.cols : 1) * data0.channels();
  int type = data0.depth();

  if (!(data0.dims <= 2 && type == CV_32F && K > 0 && N >= K)) {
    error_2("Cannot perform K-means algorithm for this configuration");
    return;
  }

  cv::Mat data(N, dims, CV_32F, data0.ptr(),
               isrow ? dims * sizeof(float) :
                       static_cast<size_t>(data0.step));

  cv::Mat centers(K, dims, type), old_centers(K, dims, type),
          temp(1, dims, type);
```

```
36    vector<int> counters(K);
37    vector<cv::Vec2f> _box(dims);
38    cv::Vec2f* box = &_box[0];
39    double best_compactness = DBL_MAX, compactness = 0;
40    int a, iter, i, j, k;
41    double epsilon = 0.;
42
43    const float* sample = data.ptr<float>(0);
44    for (j = 0; j < dims; ++j)
45      box[j] = cv::Vec2f(sample[j], sample[j]);
46
47    for (i = 1; i < N; ++i) {
48      sample = data.ptr<float>(i);
49      for (j = 0; j < dims; ++j) {
50        float v = sample[j];
51        box[j][0] = min(box[j][0], v);
52        box[j][1] = max(box[j][1], v);
53      }
54    }
55
56    generateRandomCenter(_box, centers.ptr<float>(k));
57
58    vector<vector<double>> _centers;
59    centers.copyTo(_centers);
60    kMeans_1(img, _centers, K, epsilon);
61  }
```

A.1.4 KMeans Service

```
1  #include "cal.h"
2
3  salvo loop_1(vector<vector<double>> img,
4               vector<vector<double>> old_centers_v,
5               int K, double epsilon);
6  salvo result(vector<vector<double>> centers);
7  salvo error(string msg);
8
```

```
service kMeans_1(vector<vector<double>> img,
                 vector<vector<double>> old_centers_v,
                 int K, double epsilon) {
  cv::Mat centers(cv::Scalar(0)), old_centers(old_centers_v);
  cv::Mat data0(img);
  bool isrow = data0.rows == 1 && data0.channels() > 1;
  int N = !isrow ? data0.rows : data0.cols;
  int dims = (!isrow ? data0.cols : 1) * data0.channels();
  int type = data0.depth();

  if (!(data0.dims <= 2 && type == CV_32F && K > 0 && N >= K)) {
    error("Cannot perform K-means algorithm for this configuration");
    return;
  }

  cv::Mat data(N, dims, CV_32F, data0.ptr(),
               isrow ? dims * sizeof(float) :
                       static_cast<size_t>(data0.step));
  cv::Mat temp(1, dims, type);

  vector<int> counters(K, 0);
  const float* sample = data.ptr<float>(0);

  double max_center_shift = 0;

  for (int k = 0; k < K; ++k) {
    if (counters[k] != 0)
      continue;

    int max_k = 0;
    for (int k1 = 1; k1 < K; ++k1) {
      if (counters[max_k] < counters[k1])
        max_k = k1;
    }

    double max_dist = 0;
    int farthest_i = -1;
```

```
46      float* new_center = centers.ptr<float>(k);
47      float* old_center = centers.ptr<float>(max_k);
48      float* _old_center = temp.ptr<float>();
49      float scale = 1.f/counters[max_k];
50      for (int j = 0; j < dims; ++j)
51        _old_center[j] = old_center[j]*scale;
52
53      for (int i = 0; i < N; ++i) {
54        sample = data.ptr<float>(i);
55        double dist = cv::normL2Sqr_(sample, _old_center, dims);
56
57        if (max_dist <= dist) {
58          max_dist = dist;
59          farthest_i = i;
60        }
61      }
62
63      counters[max_k]--;
64      counters[k]++;
65      sample = data.ptr<float>(farthest_i);
66
67      for (int j = 0; j < dims; ++j) {
68        old_center[j] -= sample[j];
69        new_center[j] += sample[j];
70      }
71    }
72
73    for (int k = 0; k < K; ++k) {
74      float* center = centers.ptr<float>(k);
75      if (counters[k] == 0) {
76        error("One of the clusters is empty");
77        return;
78      }
79      float scale = 1.f/counters[k];
80      for (int j = 0; j < dims; ++j)
81        center[j] *= scale;
82
```

```
83      double dist = 0;
84      const float* old_center = old_centers.ptr<float>(k);
85      for (int j = 0; j < dims; ++j) {
86        double t = center[j] - old_center[j];
87        dist += t * t;
88      }
89      max_center_shift = max(max_center_shift, dist);
90    }
91
92    vector<vector<double>> _centers;
93    centers.copyTo(_centers);
94    if (max_center_shift <= epsilon) {
95      result(_centers);
96    } else {
97      loop_1(img, _centers, K, epsilon);
98    }
99  }
```

A.2 Topology Description

```
environment@1 1@read
read@1 1@denoise
read@2 2@environment
denoise@1 1@init
denoise@2 2@environment
init@1 1@kMeans
init@2 2@environment
kMeans@1 1@kMeans
kMeans@2 1@environment
kMeans@3 2@environment
```

A.3 Source Code Augmented with Macros

A.3.1 Transformed Read Service

```
#include "cal.h"

```

```
#if defined(f_read_color_1) || defined(f_read_unchanged_1)
salvo send_color(vector<vector<double>> img
                 read_read_color_1_read_unchanged_1_send_color_decl);
#endif

#ifdef f_read_grayscale_1
salvo send_grayscale(vector<vector<double>> img
                     read_read_grayscale_1_send_grayscale_decl);
#endif

#if defined(f_read_color_1) || defined(f_read_grayscale_1) || \
    defined(f_read_unchanged_1)
salvo error_2(string msg
              read_read_color_1_read_grayscale_1_read_unchanged_1_error_decl);
#endif

#ifdef f_read_color_1
service read_color_1(string fname read_read_color_1_decl) {
  cv::Mat image = cv::imread(fname, CV_LOAD_IMAGE_COLOR);
  if(!image.data)
    error_2("Could not open or find the image"
            read_read_color_1_read_grayscale_1_read_unchanged_1_error_use);
  else {
    send_color(image
               read_read_color_1_read_unchanged_1_send_color_use);
  }
}
#endif

#ifdef f_read_grayscale_1
service read_grayscale_1(string fname read_read_grayscale_1_decl) {
  cv::Mat image = cv::imread(fname, CV_LOAD_IMAGE_GRAYSCALE);
  if(!image.data)
     error_2("Could not open or find the image"
             read_read_color_1_read_grayscale_1_read_unchanged_1_error_use);
  else {
    send_grayscale(image
```

```
40                    read_read_grayscale_1_send_grayscale_use);
41    }
42  }
43  #endif
44
45  #ifdef f_read_unchanged_1
46  service read_unchanged_1(string fname read_read_unchanged_1_decl) {
47    cv::Mat image = cv::imread(fname, CV_LOAD_IMAGE_UNCHANGED);
48    if(!image.data)
49      error_2("Could not open or find the image"
50              read_read_color_1_read_grayscale_1_read_unchanged_1_error_use);
51    else {
52      send_color(image
53                 read_read_color_1_read_unchanged_1_send_color_use);
54    }
55  }
56  #endif
57
58  void input_1(salvo&& _msg) {
59  #ifdef f_read_color_1
60    if (_msg.getType() == "read_color") {
61        cereal::BinaryInputArchive iarchive(_msg.ss);
62        tuple<string read_read_color_1_types > _data;
63        iarchive(get<0>(_data) read_read_color_1_tuple_get );
64        read_color_1(get<0>(_data) read_read_color_1_tuple_get );
65        return;
66    }
67  #endif
68  #ifdef f_read_grayscale_1
69    if (_msg.getType() == "read_grayscale") {
70      cereal::BinaryInputArchive iarchive(_msg.ss);
71      tuple<string read_read_grayscale_1_types > _data;
72      iarchive(get<0>(_data) read_read_grayscale_1_tuple_get );
73      read_grayscale_1(get<0>(_data) read_read_grayscale_1_tuple_get );
74      return;
75    }
76  #endif
```

```
77 #ifdef f_read_unchanged_1
78   if (_msg.getType() == "read_unchanged") {
79     cereal::BinaryInputArchive iarchive(_msg.ss);
80     tuple<string read_read_unchanged_1_types > _data;
81     iarchive(get<0>(_data) read_read_unchanged_1_tuple_get );
82     read_unchanged_1(get<0>(_data) read_read_unchanged_1_tuple_get );
83     return;
84   }
85 #endif
86   read_read
87 }
88
89 #if defined(f_read_color_1) || defined(f_read_grayscale_1) || \
90     defined(f_read_unchanged_1)
91 salvo error_2(string msg
92               read_read_color_1_read_grayscale_1_read_unchanged_1_error_decl) {
93   for (const pair<int, string>& _p : read_error_2_ochannels) {
94     Message _msg;
95     cereal::BinaryOutputArchive oarchive(_msg.ss);
96     oarchive(msg read_read_color_1_read_grayscale_1_read_unchanged_1_error_use);
97     _msg.setType(_p.second);
98     output(_p.first, move(_msg));
99   }
100 }
101 #endif
102
103 #if defined(f_read_color_1) || defined(f_read_unchanged_1)
104 salvo send_color(vector<vector<double> > img
105                  read_read_color_1_read_unchanged_1_send_color_decl) {
106   for (const pair<int, string>& _p : read_send_color_ochannels) {
107     Message _msg;
108     cereal::BinaryOutputArchive oarchive(_msg.ss);
109     oarchive(img
110              read_read_color_1_read_unchanged_1_send_color_use);
111     _msg.setType(_p.second);
112     output(_p.first, move(_msg));
113   }
```

```
}
#endif

#ifdef f_read_grayscale_1
salvo send_grayscale(vector<vector<double> > img
                     read_read_grayscale_1_send_grayscale_decl) {
  for (const pair<int, string>& _p : read_DOWN_send_grayscale_ochannels) {
    Message _msg;
    cereal::BinaryOutputArchive oarchive(_msg.ss);
    oarchive(img read_read_grayscale_1_send_grayscale_use);
    _msg.setType(_p.second);
    output(_p.first, move(_msg));
  }
}
#endif
```

A.3.2 Transformed `Denoise` Service

```
#include "cal.h"

#if defined(f_denoise_color_1) || defined(f_denoise_grayscale_1)
salvo send_img(vector<vector<double>> img
               denoise_denoise_color_1_denoise_grayscale_1_send_img_decl);
#endif

#if defined(f_denoise_color_1) || defined(f_denoise_grayscale_1)
salvo error_2(string msg
              denoise_denoise_color_1_denoise_grayscale_1_error_decl);
#endif

#ifdef f_denoise_color_1
service denoise_color_1(vector<vector<double>> img denoise_denoise_color_1_decl)
{
  cv::Mat result;
  cv::fastNlMeansDenoisingColored(img, result);
  if(result.empty())
    error_2("error" denoise_denoise_color_1_denoise_grayscale_1_error_use);
```

```
  else
    send_img(result denoise_denoise_color_1_denoise_grayscale_1_send_img_use);
}
#endif

#ifdef f_denoise_grayscale_1
service denoise_grayscale_1(vector<vector<double>> img
                            denoise_denoise_grayscale_1_decl) {
  cv::Mat result;
  cv::fastNlMeansDenoising(img, result);
  if(result.empty())
    error_2("error" denoise_denoise_color_1_denoise_grayscale_1_error_use);
  else
    send_img(result denoise_denoise_color_1_denoise_grayscale_1_send_img_use);
}
#endif

void input_1(Message&& _msg) {
#ifdef f_denoise_color_1
  if (_msg.getType() == "denoise_color") {
    cereal::BinaryInputArchive iarchive(_msg.ss);
    tuple<vector<vector<double>> denoise_denoise_color_1_types > _data;
    iarchive(get<0>(_data) denoise_denoise_color_1_tuple_get );
    denoise_color_1(get<0>(_data) denoise_denoise_color_1_tuple_get );
    return;
  }
#endif
#ifdef f_denoise_grayscale_1
  if (_msg.getType() == "denoise_grayscale") {
    cereal::BinaryInputArchive iarchive(_msg.ss);
    tuple<vector<vector<double> > denoise_denoise_grayscale_1_types > _data;
    iarchive(get<0>(_data) denoise_denoise_grayscale_1_tuple_get );
    denoise_grayscale_1(get<0>(_data) denoise_denoise_grayscale_1_tuple_get );
    return;
  }
#endif
  denoise_UP_denoise
```

```
57 }
58
59 #if defined(f_denoise_color_1) || defined(f_denoise_grayscale_1)
60 salvo error_2(string msg denoise_denoise_color_1_denoise_grayscale_1_error_decl)
61 {
62   for (const pair<int, string>& _p :
63       vector<pair<int, string>>({ denoise_error_2_ochannels })) {
64     Message _msg;
65     cereal::BinaryOutputArchive oarchive(_msg.ss);
66     oarchive(msg denoise_denoise_color_1_denoise_grayscale_1_error_use);
67     _msg.setType(_p.second);
68     output(_p.first, move(_msg));
69   }
70 }
71 #endif
72
73 #if defined(f_denoise_color_1) || defined(f_denoise_grayscale_1)
74 salvo send_img(vector<vector<double> > img
75                denoise_denoise_color_1_denoise_grayscale_1_send_img_decl) {
76   for (const pair<int, string>& _p :
77       vector<pair<int, string>>({ denoise_send_img_ochannels })) {
78     Message _msg;
79     cereal::BinaryOutputArchive oarchive(_msg.ss);
80     oarchive(img denoise_denoise_color_1_denoise_grayscale_1_send_img_use);
81     _msg.setType(_p.second);
82     output(_p.first, move(_msg));
83   }
84 }
85 #endif
```

A.3.3 Transformed `Init` Service

```
1 #include "cal.h"
2 #include <time.h>
3
4 #ifdef f_init_1
5 salvo kMeans_1(vector<vector<double>> img,
```

```
6                 vector<vector<double>> old_centers_v,
7                 int K, double epsilon init_init_1_kMeans_decl);
8   #endif
9
10  #ifdef f_init_1
11  salvo error_2(string msg init_init_1_error_decl);
12  #endif
13
14  static void generateRandomCenter(const vector<cv::Vec2f>& box,
15                                   float* center) {
16    size_t j, dims = box.size();
17    float margin = 1.f / dims;
18    for (j = 0; j < dims; ++j)
19      center[j] = ((float) rand() * (1.f + margin * 2.f) - margin) *
20                  (box[j][1] - box[j][0]) + box[j][0];
21  }
22
23  #ifdef f_init_1
24  service init_1(vector<vector<double>> img, int K init_init_1_decl) {
25    cv::Mat data0(img);
26    bool isrow = data0.rows == 1 && data0.channels() > 1;
27    int N = !isrow ? data0.rows : data0.cols;
28    int dims = (!isrow ? data0.cols : 1) * data0.channels();
29    int type = data0.depth();
30
31    if (!(data0.dims <= 2 && type == CV_32F && K > 0 && N >= K)) {
32      error_2("Cannot perform K-means algorithm for this configuration"
33              init_init_1_error_use);
34      return;
35    }
36
37    cv::Mat data(N, dims, CV_32F, data0.ptr(),
38                 isrow ? dims * sizeof(float) :
39                         static_cast<size_t>(data0.step));
40
41    cv::Mat centers(K, dims, type), old_centers(K, dims, type),
42            temp(1, dims, type);
```

```
43    vector<int> counters(K);
44    vector<cv::Vec2f> _box(dims);
45    cv::Vec2f* box = &_box[0];
46    double best_compactness = DBL_MAX, compactness = 0;
47    int a, iter, i, j, k;
48    double epsilon = 0.;
49
50    const float* sample = data.ptr<float>(0);
51    for (j = 0; j < dims; ++j)
52      box[j] = cv::Vec2f(sample[j], sample[j]);
53
54    for (i = 1; i < N; ++i) {
55      sample = data.ptr<float>(i);
56      for (j = 0; j < dims; ++j) {
57        float v = sample[j];
58        box[j][0] = min(box[j][0], v);
59        box[j][1] = max(box[j][1], v);
60      }
61    }
62
63    generateRandomCenter(_box, centers.ptr<float>(k));
64
65    vector<vector<double>> _centers;
66    centers.copyTo(_centers);
67    kMeans_1(img, _centers, K, epsilon init_init_1_kMeans_use);
68  }
69  #endif
70
71  void input_1(Message&& _msg) {
72  #ifdef f_init_1
73    if (_msg.getType() == "init") {
74      cereal::BinaryInputArchive iarchive(_msg.ss);
75      tuple<vector<vector<double>>, int init_init_1_types> _data;
76      iarchive(get<0>(_data), get<1>(_data) init_init_1_tuple_get);
77      init_1(get<0>(_data), get<1>(_data) init_init_1_tuple_get);
78      return;
79    }
```

```
#endif
  init_init
}

#ifdef f_init_1
salvo kMeans_1(vector<vector<double>> img,
               vector<vector<double>> old_centers_v,
               int K, double epsilon init_init_1_kMeans_decl) {
  for (const pair<int, string>& _p : init_kMeans_1_ochannels) {
    Message _msg;
    cereal::BinaryOutputArchive oarchive(_msg.ss);
    oarchive(img, old_centers_v, K, epsilon init_init_1_kMeans_use);
    _msg.setType(_p.second);
    output(_p.first, move(_msg));
  }
}
#endif

#ifdef f_init_1
salvo error_2(string msg init_init_1_error_decl) {
  for (const pair<int, string>& _p : init_error_2_ochannels) {
    Message _msg;
    cereal::BinaryOutputArchive oarchive(_msg.ss);
    oarchive(msg init_init_1_error_use);
    _msg.setType(_p.second);
    output(_p.first, move(_msg));
  }
}
#endif
```

A.3.4 Transformed KMeans Service

```
#include "cal.h"
#include "kMeans_CAL_FI_variables.h"

#ifdef f_kMeans_1
salvo loop_1(vector<vector<double>> img,
```

```
6               vector<vector<double>> old_centers_v,
7               int K, double epsilon kMeans_kMeans_1_loop_1_decl);
8   #endif
9
10  #ifdef f_kMeans_1
11  salvo result(vector<vector<double>> centers
12              kMeans_kMeans_1_result_decl);
13  #endif
14
15  #ifdef f_kMeans_1
16  salvo error(string msg kMeans_kMeans_1_error_decl);
17  #endif
18
19  #ifdef f_kMeans_1
20  service kMeans_1(vector<vector<double>> img,
21                   vector<vector<double>> old_centers_v,
22                   int K, double epsilon kMeans_kMeans_1_decl) {
23    cv::Mat centers(cv::Scalar(0)), old_centers(old_centers_v);
24    cv::Mat data0(img);
25    bool isrow = data0.rows == 1 && data0.channels() > 1;
26    int N = !isrow ? data0.rows : data0.cols;
27    int dims = (!isrow ? data0.cols : 1) * data0.channels();
28    int type = data0.depth();
29
30    if (!(data0.dims <= 2 && type == CV_32F && K > 0 && N >= K)) {
31      error("Cannot perform K-means algorithm for this configuration"
32            kMeans_kMeans_1_error_use);
33      return;
34    }
35
36    cv::Mat data(N, dims, CV_32F, data0.ptr(),
37                                  isrow ? dims * sizeof(float) :
38                                          static_cast<size_t>(data0.step));
39    cv::Mat temp(1, dims, type);
40
41    vector<int> counters(K, 0);
42    const float* sample = data.ptr<float>(0);
```

```
43
44    double max_center_shift = 0;
45
46    for (int k = 0; k < K; ++k) {
47      if (counters[k] != 0)
48        continue;
49
50      int max_k = 0;
51      for (int k1 = 1; k1 < K; ++k1) {
52        if (counters[max_k] < counters[k1])
53          max_k = k1;
54      }
55
56      double max_dist = 0;
57      int farthest_i = -1;
58      float* new_center = centers.ptr<float>(k);
59      float* old_center = centers.ptr<float>(max_k);
60      float* _old_center = temp.ptr<float>();
61      float scale = 1.f/counters[max_k];
62      for (int j = 0; j < dims; ++j)
63        _old_center[j] = old_center[j]*scale;
64
65      for (int i = 0; i < N; ++i) {
66        sample = data.ptr<float>(i);
67        double dist = cv::normL2Sqr_(sample, _old_center, dims);
68
69        if (max_dist <= dist) {
70          max_dist = dist;
71          farthest_i = i;
72        }
73      }
74
75      counters[max_k]--;
76      counters[k]++;
77      sample = data.ptr<float>(farthest_i);
78
79      for (int j = 0; j < dims; ++j) {
```

```
80        old_center[j] -= sample[j];
81        new_center[j] += sample[j];
82      }
83    }
84
85    for (int k = 0; k < K; ++k) {
86      float* center = centers.ptr<float>(k);
87      if (counters[k] == 0) {
88        error("For some reason one of the clusters is empty"
89              kMeans_kMeans_1_error_use);
90        return;
91      }
92      float scale = 1.f/counters[k];
93      for (int j = 0; j < dims; ++j)
94        center[j] *= scale;
95
96      double dist = 0;
97      const float* old_center = old_centers.ptr<float>(k);
98      for (int j = 0; j < dims; ++j) {
99        double t = center[j] - old_center[j];
100       dist += t * t;
101     }
102     max_center_shift = max(max_center_shift, dist);
103   }
104
105   vector<vector<double>> _centers;
106   centers.copyTo(_centers);
107   if (max_center_shift <= epsilon) {
108     result(_centers kMeans_kMeans_1_result_use);
109   } else {
110     loop_1(img, _centers, K, epsilon kMeans_kMeans_1_loop_use);
111   }
112 }
113 #endif
114
115 void input_1(Message&& _msg) {
116 #ifdef f_kMeans_1
```

```
117   if (_msg.getType() == "kMeans") {
118     cereal::BinaryInputArchive iarchive(_msg.ss);
119     tuple<vector<vector<double>>,
120           vector<vector<double>>,
121           int, double kMeans_kMeans_1_types> _data;
122     iarchive(get<0>(_data),
123              get<1>(_data),
124              get<2>(_data),
125              get<3>(_data) kMeans_kMeans_1_tuple_get);
126     kMeans_1(get<0>(_data),
127              get<1>(_data),
128              get<2>(_data),
129              get<3>(_data) kMeans_kMeans_1_tuple_get);
130     return;
131   }
132 #endif
133   kMeans_UP_kMeans
134 }
135
136 #ifdef f_kMeans_1
137 salvo loop_1(vector<vector<double>> img,
138              vector<vector<double>> old_centers_v,
139              int K, double epsilon kMeans_kMeans_loop_1_decl) {
140   for (const pair<int, string>& _p : kMeans_loop_1_ochannels) {
141     Message _msg;
142     cereal::BinaryOutputArchive oarchive(_msg.ss);
143     oarchive(img, old_centers_v, K, epsilon kMeans_kMeans_1_loop_use);
144     _msg.setType(_p.second);
145     output(_p.first, move(_msg));
146   }
147 }
148 #endif
149
150 #ifdef f_kMeans_1
151 salvo error(string msg kMeans_kMeans_1_error_decl) {
152   for (const pair<int, string>& _p : kMeans_DOWN_error_ochannels) {
153     Message _msg;
```

```
154     cereal::BinaryOutputArchive oarchive(_msg.ss);
155     oarchive(msg kMeans_kMeans_1_error_use);
156     _msg.setType(_p.second);
157     output(_p.first, move(_msg));
158   }
159 }
160 #endif
161
162 #ifdef f_kMeans_1
163 salvo result(vector<vector<double>> centers kMeans_kMeans_1_result_decl) {
164   for (const pair<int, string>& _p : kMeans_result_ochannels) {
165     Message _msg;
166     cereal::BinaryOutputArchive oarchive(_msg.ss);
167     oarchive(centers kMeans_kMeans_1_result_use);
168     _msg.setType(_p.second);
169     output(_p.first, move(_msg));
170   }
171 }
172 #endif
```

References

[AAF+02] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, et al. Web service choreography interface (WSCI) 1.0. *Standards proposal by BEA Systems, Intalio, SAP, and Sun Microsystems*, 2002. (Cited on page 21.)

[ABB+13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013. (Cited on page 38.)

[ABB+16] Davide Ancona, Viviana Bono, Mario Bravetti, G Castagna, J Campos, Pierre-Malo Deniélou, S Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. *Behavioral types in programming languages*. Now Publishers Incorporated, 2016. (Cited on page 3.)

[ABP12] Vasilios Andrikopoulos, Salima Benbernou, and Michael P Papazoglou. On the evolution of services. *IEEE Transactions on Software Engineering*, 38(3):609–628, 2012. (Cited on page 2.)

[ACD+03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, et al. Business process execution language for web services, 2003. (Cited on page 19.)

[ACG86] Sudhir Ahuja, N Curriero, and David Gelernter. Linda and friends. *Computer*, 19(8), 1986. (Cited on page 22.)

[AGR13a] Diana Allam, Hervé Grall, and Jean-Claude 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. (Cited on pages 17, 25, 106, 117, and 118.)

[AGR13b] Diana Allam, Hervé Grall, and Jean-Claude Royer. The substitution principle in an object-oriented framework for web services: From failure to success. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, page 250. ACM, 2013. (Cited on pages 4, 19, and 118.)

[AHS93] Farhad Arbab, Ivan Herman, and Pål Spilling. An overview of manifold and its implementation. *Concurrency: practice and experience*, 5(1):23–70, 1993. (Cited on page 23.)

[Apa11] Apache. Apache Thrift. `https://thrift.apache.org/`, 2011. (Cited on page 25.)

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science*, 14(03):329–366, 2004. (Cited on page 23.)

[B+81] Barry W Boehm et al. *Software engineering economics*, volume 197. Prentice-hall Englewood Cliffs (NJ), 1981. (Cited on page 16.)

[BBB+02] Arindam Banerji, Claudio Bartolini, Dorothea Beringer, Venkatesh Chopella, Kannan Govindarajan, Alan Karp, Harumi Kuno, Mike Lemon, Gregory Pogossiants, Shamik Sharma, et al. Web services conversation language (wscl) 1.0. *W3C Note*, 14, 2002. (Cited on page 2.)

[BBC05] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005. (Cited on page 17.)

[BBC+10] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010. (Cited on page 22.)

[BBDCL97] Viviana Bono, Michele Bugliesi, Mariangiola Dezani-Ciancaglini, and Luigi Liquori. Subtyping constraints for incomplete objects. In *TAPSOFT'97: Theory and Practice of Software Development*, pages 465–477. Springer, 1997. (Cited on page 33.)

[BCDGM05] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, and Massimo Mecella. Automatic composition of process-based web services: a challenge. In *Proc. of the WWW*, volume 5, 2005. (Cited on page 20.)

[BCK+09] Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari. Multi-core implementations of the concurrent collections programming model. In *CPC'09: 14th International Workshop on Compilers for Parallel Computers*, 2009. (Cited on page 22.)

[BCP04] Antonio Brogi, Carlos Canal, and Ernesto Pimentel. On the specification of software adaptation. *Electronic Notes in Theoretical Computer Science*, 97:47–65, 2004. (Cited on page 17.)

[BCP06] Antonio Brogi, Carlos Canal, and Ernesto Pimentel. Component adaptation through flexible subservicing. *Science of Computer Programming*, 63(1):39–56, 2006. (Cited on page 17.)

[BCS+09] Robert D Bjornson, Nicholas J Carriero, Martin H Schultz, Patrick M Shields, and Stephen B Weston. Networkspace: a coordination system for high-productivity environments. *International journal of parallel programming*, 37(1):106–125, 2009. (Cited on page 22.)

[BCZ15] Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. Compliance in behavioural contracts: a brief survey. In *Programming Languages with Applications to Biology and Security*, pages 103–121. Springer, 2015. (Cited on pages 11 and 15.)

[BDO05] Alistair Barros, Marlon Dumas, and Phillipa Oaks. A critical overview of the web services choreography description language. *BPTrends Newsletter*, 3:1–24, 2005. (Cited on page 21.)

[BEK+00] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (SOAP) 1.1, 2000. (Cited on page 18.)

[BFT10] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB standard — version 2.5, 2010. (Cited on page 57.)

[BHM05] Tomás Barros, Ludovic Henrio, and Eric Madelaine. Behavioural models for hierarchical components. In *Model Checking Software*, pages 154–168. Springer, 2005. (Cited on pages 11 and 15.)

[Bin08] CA Binildas. Service oriented java business integration. *Birmingham-Mumbai: Packt Publishing*, 2008. (Cited on page 105.)

[BK98] Nat Brown and Charlie Kindel. Distributed component object model protocol – DCOM/1.0. *Online, November*, 1998. (Cited on page 18.)

[BLHL+01] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001. (Cited on page 20.)

[BPR02] Antonio Brogi, Ernesto Pimentel, and Ana M Roldan. Compatibility of Linda-based component interfaces. *Electronic Notes in Theoretical Computer Science*, 66(4):82–96, 2002. (Cited on page 17.)

[BS90] Bonnie Berger and Peter W Shor. Approximation alogorithms for the maximum acyclic subgraph problem. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 236–243. Society for Industrial and Applied Mathematics, 1990. (Cited on page 51.)

[BSD13] Athman Bouguettaya, Quan Z Sheng, and Florian Daniel. *Advanced web services*. Springer, 2013. (Cited on pages 19 and 41.)

[BSS12] Scott Bourne, Claudia Szabo, and Quan Z Sheng. Ensuring well-formed conversations between control and operational behaviors of web services. In *Service-Oriented Computing*, pages 507–515. Springer, 2012. (Cited on page 21.)

[BW98] Alan W Brown and Kurt C Wallnau. The current state of CBSE. *IEEE software*, (5):37–46, 1998. (Cited on page 17.)

[CCKT86] David Callahan, Keith D Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *ACM SIGPLAN Notices*, volume 21, pages 152–161. ACM, 1986. (Cited on page 61.)

[CCM+01] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (WSDL) 1.1, 2001. (Cited on pages 21 and 31.)

[CDM09] Anis Charfi, Tom Dinkelaker, and Mira Mezini. A plug-in architecture for self-adaptive web service compositions. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 35–42. IEEE, 2009. (Cited on page 21.)

[cF16] Standard C++ Foundation. Why can't i separate the definition of my templates class from its declaration adn put it inside a .cpp file. `https://isocpp.org/wiki/faq/templates#templates-defn-vs-decl`, 2016. (Cited on page 5.)

[CGP09] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(5):19, 2009. (Cited on page 2.)

[CHY07] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *Programming Languages and Systems*, pages 2–17. Springer, 2007. (Cited on pages 1, 20, and 107.)

[CJ99] Grady H Campbell Jr. Adaptable components. In *Proceedings of the 21st international conference on Software engineering*, pages 685–686. ACM, 1999. (Cited on page 17.)

[CL06] Samuele Carpineti and Cosimo Laneve. A basic contract language for web services. In *Programming Languages and Systems*, pages 197–213. Springer, 2006. (Cited on page 25.)

[CPP] cv (const and volatile) type qualifiers. `http://en.cppreference.com/w/cpp/language/cv`. (Cited on page 116.)

[CRR98] Lobel Crnogorac, Anand S Rao, and Kotagiri Ramamohanarao. Classifying inheritance mechanisms in concurrent object-oriented programming. In *ECOOP'98 Object-Oriented Programming*, pages 571–600. Springer, 1998. (Cited on pages 13 and 14.)

[Cus91] Elspeth Cusack. Inheritance in object oriented Z. In *ECOOP'91 European Conference on Object-Oriented Programming*, pages 167–179. Springer, 1991. (Cited on page 13.)

[DAH01] Luca De Alfaro and Thomas A Henzinger. Interface automata. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 109–120. ACM, 2001. (Cited on pages 11 and 15.)

[Dij82] Edsger W Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982. (Cited on pages 11 and 13.)

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. (Cited on page 57.)

[DS05] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International journal of web and grid services*, 1(1):1–30, 2005. (Cited on pages 19 and 20.)

[DSWdS09] Rodrigo Pereira Dos Santos, Cláudia Maria Lima Werner, and Marcio Antelio da Silva. Incorporating information of value in a component repository to support a component marketplace infrastructure. In *Information Reuse & Integration, 2009. IRI'09. IEEE International Conference on*, pages 266–271. IEEE, 2009. (Cited on page 17.)

[DYV12] Qiang Duan, Yuhong Yan, and Athanasios V Vasilakos. A survey on service-oriented network virtualization toward convergence of networking and cloud computing. *Network and Service Management, IEEE Transactions on*, 9(4):373–392, 2012. (Cited on pages 19 and 41.)

[Erl05] Thomas Erl. *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 2005. (Cited on page 9.)

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003. (Cited on pages 83 and 146.)

[Fer04] Andrea Ferrara. Web services: a process algebra approach. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251. ACM, 2004. (Cited on page 45.)

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM, 1998. (Cited on page 15.)

[FSW+06] Peter Feiler, Kevin Sullivan, Kurt Wallnau, Richard Gabriel, John Goodenough, Richard Linger, Thomas Longstaff, Rick Kazman, Mark Klein, Linda Northrop, and Douglas Schmidt. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon University, 2006. (Cited on page 11.)

[G+02] Web Services Choreography Working Group et al. Web services choreography description language, 2002. (Cited on pages 1, 2, and 21.)

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, 1992. (Cited on page 22.)

[GGRV15] Simon J Gay, Nils Gesbert, António Ravara, and Vasco Thudichum Vasconcelos. Modular session types for objects. *Logical Methods in Computer Science*, 11(4), 2015. (Cited on page 14.)

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994. (Cited on page 15.)

[GJ96] Benedict R Gaster and Mark P Jones. A polymorphic type system for extensible records and variants. 1996. (Cited on page 31.)

[Glo] Python Glossary. Term argument. `https://docs.python.org/3/glossary.html#term-argument`. (Cited on page 114.)

[Goo] Google. Places API. `https://developers.google.com/places/`. (Cited on page 121.)

[Goo08] Google. Protocol buffers. `https://developers.google.com/protocol-buffers`, 2008. (Cited on pages 3 and 25.)

[GS14] Vincent Gramoli and Andrew E Santosa. Why inheritance anomaly is not worth solving. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*, page 6. ACM, 2014. (Cited on page 14.)

[GSS08] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components. *Parallel Processing Letters*, 18(02):221–237, 2008. (Cited on pages 5, 22, 26, 106, and 145.)

[GSS10] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. Asynchronous stream processing with S-Net. *International Journal of Parallel Programming*, 38(1):38–67, 2010. (Cited on pages 5, 22, 26, 38, 53, 106, and 145.)

[Had06] Marc J Hadley. Web application description language (WADL). 2006. (Cited on page 21.)

[HC01] George T Heineman and William T Councill. Component-based software engineering. *Putting the Pieces Together, Addison-Westley*, 2001. (Cited on page 16.)

[HM05] Seyyed Vahid Hashemian and Farhad Mavaddat. A graph-based approach to web services composition. In *Applications and the Internet, 2005. Proceedings. The 2005 Symposium on*, pages 183–189. IEEE, 2005. (Cited on pages 2 and 20.)

[HNT08] Ramy Ragab Hassen, Lhouari Nourine, and Farouk Toumani. Protocol-based web service composition. In *Service-Oriented Computing–ICSOC 2008*, pages 38–53. Springer, 2008. (Cited on page 20.)

[HO98] George T Heineman and Helgo Ohlenbusch. An evaluation of component adaptation techniques. 1998. (Cited on page 17.)

[Hon93] Kohei Honda. Types for dyadic interaction. In *CONCUR'93*, pages 509–523. Springer, 1993. (Cited on pages 11 and 15.)

[HS11] Torsten Hoefler and Marc Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the international conference on Supercomputing*, pages 75–84. ACM, 2011. (Cited on page 53.)

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices*, 43(1):273–284, 2008. (Cited on pages 1, 2, 41, and 125.)

[Inc03] Sun Microsystems Inc. Specification, enterprise JavaBeans. Technical report, 2003. (Cited on page 18.)

[Ing81] Daniel HH Ingalls. Design principles behind Smalltalk. *BYTE magazine*, 6(8):286–298, 1981. (Cited on page 13.)

[JEA+07] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, et al. Web services business process execution language version 2.0. *OASIS standard*, 11(120):5, 2007. (Cited on page 2.)

[JF88] Ralph E Johnson and Brian Foote. Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35, 1988. (Cited on page 13.)

[Jon97] Capers Jones. *Applied software measurement*, volume 8. McGraw Hill New York, 1997. (Cited on page 16.)

[JRSS14] Andrea Janes, Tadas Remencius, Alberto Sillitti, and Giancarlo Succi. Towards understanding of structural attributes of web apis using metrics based on api call responses. In *Open Source Software: Mobile Open Source Technologies*, pages 83–92. Springer, 2014. (Cited on page 1.)

[Kah62] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962. (Cited on page 51.)

[KDPG16] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J Gay. Typechecking protocols with mungo and stmungo. 2016. (Cited on page 3.)

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of AspectJ. In *ECOOP 2001 Object-Oriented Programming*, pages 327–354. Springer, 2001. (Cited on page 15.)

[Kil73] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973. (Cited on pages 58, 59, 61, 62, and 64.)

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997. (Cited on pages 15 and 16.)

[KMNB+09] Woralak Kongdenfha, Hamid Reza Motahari-Nezhad, Boualem Benatallah, Fabio Casati, and Regis Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. *Services Computing, IEEE Transactions on*, 2(2):94–107, 2009. (Cited on page 21.)

[Kno02] Kirk Knoernschild. *Java design: objects, UML, and process*. Addison-Wesley Professional, 2002. (Cited on page 15.)

[Kos03] Jussi Koskinen. Software maintenance costs. *Information Technology Research Institute, ELTIS-Project University of Jyväskylä*, 2003. (Cited on page 16.)

[Kum92] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, 13(1):32, 1992. (Cited on page 57.)

[Kuz16] Maksim Kuznetcov. Data-driven self-tuning in a coordination programming language. Master's thesis, University of Hertfordshire, 2016. (Cited on page 127.)

[LC10] Thomas Y Lee and David W Cheung. Formal models and algorithms for XML data interoperability. *Journal of Computing Science and Engineering*, 4(4):313–349, 2010. (Cited on page 25.)

[Lei05] Daan Leijen. Extensible records with scoped labels. *Trends in Functional Programming*, 5:297–312, 2005. (Cited on page 31.)

[Len73] Hendrik Willem Lenstra. The acyclic subgraph problem. 1973. (Cited on page 51.)

[LFMS10] Xitong Li, Yushun Fan, Stuart Madnick, and Quan Z Sheng. A pattern-based approach to protocol mediation for web services composition. *Information and Software Technology*, 52(3):304–323, 2010. (Cited on page 21.)

[LM07] Ralf Lämmel and Erik Meijer. Revealing the X/O impedance mismatch. In *Datatype-Generic Programming*, pages 285–367. Springer, 2007. (Cited on pages 25 and 117.)

[LNW07] Kim G Larsen, Ulrik Nyman, and Andrzej Wąsowski. Modal i/o automata for interface and product line theories. In *European Symposium on Programming*, pages 64–79. Springer, 2007. (Cited on page 2.)

[LP07] Cosimo Laneve and Luca Padovani. The must preorder revisited. In *International Conference on Concurrency Theory*, pages 212–225. Springer, 2007. (Cited on page 2.)

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008. (Cited on page 2.)

[MA03] Brandon Morel and Perry Alexander. Automating component adaptation for reuse. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 142–151. IEEE, 2003. (Cited on page 17.)

[Mac67] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297. University of California Press, 1967. (Cited on page 135.)

[MASM13] Sujith Samuel Mathew, Yacine Atif, Quan Z Sheng, and Zakaria Maamar. The web of things-challenges and enabling technologies. In *Internet of things and inter-cooperative computational technologies for collective intelligence*, pages 1–23. Springer, 2013. (Cited on page 19.)

[MBE03] Brahim Medjahed, Athman Bouguettaya, and Ahmed K Elmagarmid. Composing web services on the semantic web. *The VLDB Journal — The International Journal on Very Large Data Bases*, 12(4):333–351, 2003. (Cited on page 20.)

[MBH+04] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, et al. Owl-s: Semantic markup for web services. *W3C member submission*, 22:2007–04, 2004. (Cited on page 20.)

[Men09] Gregoris Mentzas. *Semantic Enterprise Application Integration for Business Processes: Service-Oriented Frameworks: Service-Oriented Frameworks*. IGI Global, 2009. (Cited on page 25.)

[MG06] A Bucciaroneand Maurice and S Gnesi. A survey on service composition approaches: From industrial standards to formal methods. In *Proceedings of the 2nd Inter Conf on Internet and Web Applications and Services (ICIW'07)*, pages 10–129, 2006. (Cited on page 19.)

[MKD93] Jeff Magee, Jeff Kramer, and Naranker Dulay. Darwin/mp: An environment for parallel and distributed programming. In *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, volume 2, pages 337–346. IEEE, 1993. (Cited on page 23.)

[MM04] Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Computing*, (6):51–59, 2004. (Cited on page 19.)

[MMW06] Craig McMurtry, Marc Mercuri, and Nigel Watling. *Microsoft Windows communication foundation: hands-on!* Sams Publishing, 2006. (Cited on page 105.)

[MNXB10] Hamid Reza Motahari Nezhad, Guang Yuan Xu, and Boualem Benatallah. Protocol-aware matching of web service interfaces for adapter development. In *Proceedings of the 19th international conference on World wide web*, pages 731–740. ACM, 2010. (Cited on page 21.)

[MS98] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *ECOOP'98 Object-Oriented Programming*, pages 355–382. Springer, 1998. (Cited on page 13.)

[MS04] Giuseppe Milicia and Vladimiro Sassone. The inheritance anomaly: ten years after. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1267–1274. ACM, 2004. (Cited on page 14.)

[MY93] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. *Research directions in concurrent object-oriented programming*, 3:107–150, 1993. (Cited on page 13.)

[OCa] Variant types and labeled arguments. `http://caml.inria.fr/pub/docs/u3-ocaml/ocaml051.html#toc23`. (Cited on page 114.)

[Par71] David Lorge Parnas. Information distribution aspects of design methodology. Technical report, 1971. (Cited on page 13.)

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972. (Cited on pages 13 and 15.)

[PC03] Gerardo Pardo-Castellote. Omg data-distribution service: Architectural overview. In *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pages 200–206. IEEE, 2003. (Cited on page 105.)

[Pel03] Chris Peltz. Web services orchestration and choreography. *Computer*, (10):46–52, 2003. (Cited on page 20.)

[Pet06] Helmut Petritsch. Service-oriented architecture (SOA) vs. component based architecture. *Vienna University of Technology, Vienna*, 2006. (Cited on page 18.)

[Pie97] Benjamin C Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(02):129–193, 1997. (Cited on pages 29, 32, and 112.)

[PL03] Randall Perrey and Mark Lycett. Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pages 116–119. IEEE, 2003. (Cited on pages 1 and 18.)

[PMR99] Gian Pietro Picco, Amy L Murphy, and Gruia-Catalin Roman. Lime: Linda meets mobility. In *Proceedings of the 21st international conference on Software engineering*, pages 368–377. ACM, 1999. (Cited on page 22.)

[Pro13] ProgrammableWeb. ProgrammableWeb research center. `http://www.programmableweb.com/api-research`, 2013. (Cited on pages 1 and 19.)

[RDL+09] Vinay Kumar Reddy, Alpana Dubey, Sala Lakshmanan, Srihari Sukumaran, and Rajendra Sisodia. Evaluating legacy assets in the context of migration to soa. *Software Quality Journal*, 17(1):51–63, 2009. (Cited on page 9.)

[RGB+11] Mohammad Javad Rashti, Jonathan Green, Pavan Balaji, Ahmad Afsahi, and William Gropp. Multi-core and network aware mpi topology functions. In *Recent Advances in the Message Passing Interface*, pages 50–60. Springer, 2011. (Cited on page 53.)

[RS05] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In *Semantic Web Services and Web Process Composition*, pages 43–54. Springer, 2005. (Cited on page 19.)

[SAK07] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007. (Cited on pages 4 and 25.)

[SBMN09] Quan Z Sheng, Boualem Benatallah, Zakaria Maamar, and Anne HH Ngu. Configurable composition and adaptive provisioning of web services. *Services Computing, IEEE Transactions on*, 2(1):34–49, 2009. (Cited on page 21.)

[SBS06] Gwen Salaun, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. *International Journal of Business Process Integration and Management*, 1(2):116–128, 2006. (Cited on pages 17 and 45.)

[SBW99] Clemens Szyperski, Jan Bosch, and Wolfgang Weck. Component-oriented programming. In *Object-oriented technology ECOOP'99 workshop reader*, pages 184–192. Springer, 1999. (Cited on page 16.)

[Sch03] Sven-Bodo Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *Journal of functional programming*, 13(06):1005–1059, 2003. (Cited on page 146.)

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. Traits: Composable units of behaviour. In *ECOOP 2003–Object-Oriented Programming*, pages 248–274. Springer, 2003. (Cited on page 15.)

[Sha13] Alex Shafarenko. Astrakahn: A coordination language for streaming networks. *arXiv preprint arXiv:1306.6029*, 2013. (Cited on pages 127 and 146.)

[Sie00] Jon Siegel. *CORBA 3 fundamentals and programming*, volume 2. John Wiley & Sons New York, NY, USA:, 2000. (Cited on page 18.)

[SK03] Biplav Srivastava and Jana Koehler. Web service composition-current solutions and open problems. In *ICAPS 2003 workshop on Planning for Web Services*, volume 35, pages 28–35, 2003. (Cited on page 19.)

[SMY+14] Quan Z Sheng, Zakaria Maamar, Lina Yao, Claudia Szabo, and Scott Bourne. Behavior modeling and automated verification of web services. *Information Sciences*, 258:416–433, 2014. (Cited on pages 20 and 21.)

[SPK+12] Hari Subramoni, Sreeram Potluri, Krishna Kandalla, B Barth, Jérôme Vienne, Jeff Keasler, Karen Tomko, K Schulz, Adam Moody, and Dhabaleswar K Panda. Design of a scalable infiniband topology service to enable network-topology-aware placement of processes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 70. IEEE Computer Society Press, 2012. (Cited on page 53.)

[SQV+14] Quan Z Sheng, Xiaoqiang Qiao, Athanasios V Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. Web services composition: A decade's overview. *Information Sciences*, 280:218–238, 2014. (Cited on pages 19 and 20.)

[Sta09] Stack Overflow. Why can templates only be implemented in the header file? `http://stackoverflow.com/questions/495021/`, 2009. (Cited on page 5.)

[Ste06] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *ACM Sigplan Notices*, volume 41, pages 481–497. ACM, 2006. (Cited on page 16.)

[Str13] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013. (Cited on page 119.)

[T+55] Alfred Tarski et al. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955. (Cited on page 76.)

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE'94 Parallel Architectures and Languages Europe*, pages 398–413. Springer, 1994. (Cited on pages 11 and 15.)

[Tik15] Anna Tikhonova. A synchronisation facility for a stream processing coordination language. Master's thesis, University of Hertfordshire, 2015. (Cited on page 127.)

[TYN13] Ewan Tempero, Hong Yul Yang, and James Noble. What programmers do with inheritance in Java. In *ECOOP 2013–Object-Oriented Programming*, pages 577–601. Springer, 2013. (Cited on page 15.)

[VPK+15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015. (Cited on page 146.)

[VVR06] Antonio Vallecillo, Vasco T Vasconcelos, and António Ravara. Typing the behavior of software components using session types. *Fundamenta Informaticæ*, 73(4):583–598, 2006. (Cited on page 17.)

[WDW07] Matthias Weidlich, Gero Decker, and Mathias Weske. Efficient analysis of BPEL 2.0 processes using p-calculus. In *Asia-Pacific Service Computing Conference, The 2nd IEEE*, pages 266–274. IEEE, 2007. (Cited on page 20.)

[WHW+11] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011. (Cited on page 22.)

[Wis06] Ryan J Wisnesky. The inheritance anomaly revisited, 2006. (Cited on page 14.)

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. *Computing Systems*, 9:265–290, 1996. (Cited on page 19.)

[XFZ10] PengCheng Xiong, YuShun Fan, and MengChu Zhou. A Petri net approach to analysis and composition of web services. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 40(2):376–387, 2010. (Cited on page 21.)

[YT86] Akinori Yonezawa and Mario Tokoro. *Object-oriented concurrent programming*. The MIT Press, Cambridge, MA, 1986. (Cited on page 14.)

[Zai17] Pavel Zaichenkov. Interface configuration protocol for web services. `https://github.com/zayac/joule`, 2017. (Cited on pages 6, 123, and 146.)

[ZGG+14] Pavel Zaichenkov, Bert Gijsbers, Clemens Grelck, Olga Tveretina, and Alex Shafarenko. A case study in coordination programming: Performance evaluation of

s-net vs intel's concurrent collections. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1059–1067. IEEE, 2014. (Cited on pages 22 and 23.)

[ZL13] Zibin Zheng and Michael R Lyu. Personalized reliability prediction of web services. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(2):12, 2013. (Cited on page 21.)