Project no.: H2020-MSCA-RISE-2017-778233
Project full title: Behavioural Application Program Interfaces
Project Acronym: BEHAPI
Deliverable no.: D.2.2 (M12)
Title of Deliverable: Languages and models for b-API description
Work package: WP2 (tasks 2.2 and 2.7)
Type: R
Lead Beneficiary: **UNIVERSITY OF TORINO (BEN 9, UNITO)**
Dissemination Level: PU
Number of pages: 19
Contract Delivery due date: 28/02/2019 (M12)

Acknowledgement: This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 778233

Abstract:

The goals of work package (WP) 2 "API provision" are the development and enhancement of languages tailored to the rigorous and human friendly description of b-APIs. In particular:

- to define techniques to (semi-)automatically generate b-API descriptions from source code.
- to develop validation and verification techniques to assure that the documentation of a b-API is aligned with its implementation, in both its functional and non-functional requirements.
- to provide both static and dynamic techniques, such as typing, testing and monitoring, to ensure that b-APIs meet their requirements.

This report summarizes the initial activities related to WP2, and particularly to Tasks T.2.2 "Models for behavioural APIs" and T.2.7 "Inferring behavioural descriptions for b-APIs". These activities have been carried out by the following beneficiaries according to the list of secondments below:

- University of Malta (BEN 1, UOM)

- University of Leicester (BEN 2, ULEIC)
- IT-Universitet I Kobenhavn (BEN 7, ITU)
- Alma Mater Studiorum - Università di Bologna (BEN 8, UNIBO)
- Università degli Studi di Torino (BEN 9, UNITO)
- Universidad de Buenos Aires (BEN 20, UBA)

Fellow Id	Category	Declaration Number	Person/Month
1	ER	1	0.7
2	ER	2	0.83
3	ER	4 and 8 and 12	0.5 and 0.2 and 0.93
4	ER	5	0.73
5	ESR	6	1.3
8	ER	9	1.07
9	ER	10 and 11	0.5 and 2.1
10	ESR	13	2.07
11	ER	14	1.07
12	ER	15	0.27
13	ER	16	1.07

Table of contents

1. Introduction

This deliverable is structured in two parts. In Section 2 we provide a survey of the most relevant models and languages that can be used to provide behavioural descriptions of APIs. The survey covers both models actively studied by academic researchers and languages developed and adopted by the industry. For each model or language, we give an informal description of its main features and provide references to the related literature. In Section 3, we describe the activities of BehAPI participants related to WP2 and show their connections with the models and languages listed in Section 2. This allows us to identify the cornerstone formalisms and technologies at the basis of the ongoing and future research activities of the BehAPI project.

In general, we observe that most of the currently used languages for describing provided APIs are quite limited compared to the models that are being studied in academia. In particular, most of the present languages only allow for very simple behavioural specifications of interacting processes, often limited to monotyped input/output ports, thereby limiting the availability and effectiveness of both static and dynamic analysis techniques.

2. Models and languages relevant to BehAPI

In this section we review the most relevant models and languages for the behavioral description of communication channels, processes, objects. We make a distinction between models - formally defined behavioral descriptions - and languages - actual technologies that are currently in use for this purpose. This distinction is sometimes blurred: many models are supported by concrete implementations that make them usable for the development of real-world applications. Dually, some languages have taken inspiration from formal models. We give a short, informal description of each model/language, highlighting its defining features and some of the most notable refinements and extensions. We also provide references to the literature where these models/languages have been defined and studied. Recent state-of-the-art reports provide even more detailed overview of most of these models and their applications [Hüttel et al. 2016, Bartoletti et al. 2015, Ancona et al. 2016].

2.1 Models

2.1.1 Web Service Contracts

Contracts are behavioural descriptions of Web Services that specify the kind, direction and order of messages exchanged between a client and a service [Castagna et al. 2009]. Typical models for Web service contracts are regular expressions extended with non-deterministic choice [Bravetti and Zavattaro 2008] and sequential CCS-like processes [Bravetti and Zavattaro 2009, Castagna et al. 2009]. By describing the behavioural API of a web service, a contracts allows forms of static conformance verification - so that a client is checked to be conformant to a service, leading to deadlock-free and possibly fairly terminating interactions - as well as dynamic discovery of Web services [Padovani 2008, Castagna et al. 2009]. In the latter case, the contract of a service is used as search key for querying Web service repositories. Web service contracts can be extracted from BPEL specifications of Web services [Laneve and Padovani, 2015]. Several theories of subcontract relations have been studied to provide conformance-preserving compatibility relations between contracts, possibly enabled by the use of mediator processes or orchestrators [Padovani 2008].

2.1.2 Session types

A *binary session* is a private communication established between two peer processes (like a client and a service), each process using one of the *session endpoints*. A session type describes the kind, direction and order of messages exchanged through an endpoint [Honda 1993] and, as such, provide a behavioral description of a process restricted to those I/O actions that pertain to one particular endpoint. Suitably designed type systems are capable of checking whether a process (modeled in a process algebra or in a core programming language) is well typed with respect to the session types of the session endpoints it uses. The type system makes sure that peer endpoints of a session have dual session types, so that each input operation on one endpoint is matched by a corresponding output operation on the other endpoint and vice versa. Duality ensures communication safety and progress. In order to guarantee protocol fidelity, namely the property that a well-typed process adheres to the communication protocol described by a session type, it is also necessary to ensure that session endpoints are used *linearly*, namely that they are not duplicated nor discarded.

There are analogies between contracts and session types [Laneve and Padovani 2008, Bernardi and Hennessy 2016]. Subtyping relations have been studied that preserve communication safety [Gay and Hole 2005], liveness properties [Padovani 2013, 2016] and that allow for permutations of actions not related by causal dependencies [Mostrous and Yoshida 2015, Bravetti, Carbone and Zavattaro 2017]. Recently, session types have been extended with a general sequential composition operator to allow the description and inference of context-free communication protocols [Thiemann and Vasconcelos 2016, Padovani 2017].

Hybrid techniques based on a mixture of static and dynamic checks can be used for enforcing more fine-grained properties related to the content of messages [Melgratti and Padovani 2017, Gommerstadt, Jia and Pfenning 2018] and for the gradual adoption of session types in programs mixing weakly- and strongly-typed modules [Igarashi, Thiemann, Vasconcelos and Wadler 2017].

2.1.3 Multiparty session types

Multiparty session types [Honda, Yoshida and Carbone 2016] are a generalization of binary session types allowing the description of communication protocols between an arbitrary number (but often fixed) of processes. In these cases a session is more akin to a "room" in which the conversation between processes takes place. In this setting, there are two behavioral descriptions involved called *global* and *local* types. A global type describes the conversation from a neutral point of view, whereas a local type describes the conversation from the viewpoint of a particular process. Local types are akin to session types, except that I/O actions are annotated with the identifier of the multiparty session participant to which a message is sent or from which a message is expected. Global and local types are related by a notion of *projection*, that allows to compute the local type (hence the behaviour) of a particular process from the global type of the overall conversation. Like session types, local types are suitable to be used as behavioral APIs, whereas global types are more useful during the design phase of a system. Session type systems similar to those used for binary sessions have been defined for checking well-typedness of processes. Global and local types have also been extended to provide refined behavioral specifications that include assertions on the content of messages [Bocchi, Honda, Tuosto and Yoshida 2010], timing information on the synchronizations [Neykova, Bocchi and Yoshida 2017], exceptions [Carbone, Yoshida and Honda 2009, Capecchi, Giachino and Yoshida 2016]. An important advantage of multiparty sessions with respect to binary sessions is that they guarantee progress for an arbitrary number of processes, provided that they do not interleave communications actions pertaining different sessions. Refinements of

multiparty session types have been studied to enforce progress also in presence of session interleaving [Coppo, Dezani-Ciancaglini, Padovani, Yoshida 2013, Coppo, Dezani-Ciancaglini, Yoshida, Padovani 2016].

2.1.4 Choreographies

Choreographies are syntactic descriptions of the overall coordination of a system in terms of interactions between autonomous participants. A choreography captures how two or more participants exchange messages during their execution from a global viewpoint, instead of a collection of programs that define individually the behavior of each endpoint. A notion of projection, similar to that for global types, can be used to obtain the code of each participant from the choreography. Unlike global types, choreographies provide a description of the actual computation taking place in an interaction. As such, they cannot be considered behavioural types in a strict sense and cannot be used as behavioural specifications of APIs. However, choreographies can be integrated with multiparty sessions [Carbone and Montesi 2013] and can be used as backbones for the design of systems making use of behavioural APIs.

2.1.5 Mailbox types

In contrast with processes that communicate through channels, *actors* and *concurrent objects* communicate through mailboxes. A *mailbox* is a communication medium associated with an actor or a concurrent object, with the property that only the associated actor or concurrent object is allowed to retrieve messages from it. Other actors and concurrent objects can only write messages in a mailbox that is not their own. A *mailbox type* [de' Liguoro and Padovani 2018] describes the legal configurations of messages that can be stored into a mailbox. Unlike contracts and session types, which describe the order of communications, mailbox types describe the combinations of messages can be present in a mailbox at any moment during its lifetime. Mailbox types are suitable for describing the protocol of concurrent objects and of actors because they do not impose a linear use of endpoints as is the case for session types.

2.1.6 Typestate

The notion of typestate has been introduced for describing in abstract terms the state of an object whose API changes during its lifetime. The term "object" is used here in its most general sense. The first work on typestate [Strom and Yemini 1986] considered variables, which can be in states uninitialized, assigned, or

deallocated. Later, typestate has been extended to object-oriented programming [DeLine and Fähndrich 2004] and has been used for the behavioural description and analysis of objects with non-uniform APIs [Vasconcelos and Ravara 2017]. Typical examples of objects whose API can be described by typestate are files (which can be read or written only when they are open and can be opened only when the are closed) and iterators (which can be advanced only if they are not finished).

The significant number of objects with non-uniform interfaces in typical object-oriented applications [Beckman, Kim and Aldrich 2011] has led researchers to propose a methodology called TypeState-Oriented Programming (TSOP) [Aldrich, Sunshine, Saini, Sparks 2009, Sunshine, Naden, Stork, Aldrich and Tanter 2011, Garcia, Tanter, Wolff and Aldrich 2014] that uses dedicated constructs and type annotations for implementing and using such objects. TSOP has also been extended to concurrent objects [Crafa and Padovani 2017] and subsequently refined for deadlock analysis [Padovani 2018]. Algorithms for checking protocol conformance of concurrent objects implemented using concurrent TSOP are available [Padovani 2018 bis]. The object types introduced by Crafa and Padovani [2017] for concurrent TSOP have inspired - and have been generalized to - mailbox types [de'Liguoro and Padovani 2018].

Somehow related to typestate are Enabledness Preserving Abstractions (EPA) of an object protocol, which are finite transition systems grouping concrete states of the potentially infinite state and non-deterministic object protocol into sets such that two concrete states are grouped if they enable the same method calls [de Caso et al. 2011]. EPAs can be built automatically from source code or from a formal specification of the concrete intended protocol of an implementation. A technique for the automatic construction of EPAs using static analysis of code has been proposed by de Caso et al. [2013].

2.2 Languages

2.2.1 Web Service Description Languages

The Web Service Description Language (WSDL) [Christensen et al. 2001, Chinnici et al. 2006] provides a standardized technology for describing the API exposed by a Web service. Such description includes the service location, the format (or schema) of the exchanged messages, the transfer mechanism to be used (e.g. remote procedure calls), and the service contract. In WSDL, contracts are basically limited to one-way (asynchronous) and request/response (synchronous) interactions. This limitation is partly overcome in the Web Service Conversation

Language (WSCL) [Banerji et al. 2002], which extends WSDL contracts by allowing the description of arbitrary, possibly cyclic sequences of exchanged messages between communicating parties.

2.2.2 Languages for REST(ful) APIs

REST (REpresentational State Transfer) is a client-server architectural style [Fielding and Taylor 2000] that relies on stateless communication, i.e., each request is treated independently from the previous interactions with the same client and, consequently, should provide all required information to complete it. The HATEOAS (Hypertext As The Engine Of Application State) principle states that clients are guided by the responses provided by the API, i.e., within a response the server indicates the actions that a client can perform in the current state of the conversation. Resources are used to represent both state and functionalities, and they are the building blocks of REST architecture. Consequently, the description of REST services/APIs mainly consists of the definition of resource representation and resource operations.

REST architecture is customary implemented on top of HTTP (HyperText Transfer Protocol), and HTTP commands are used for retrieving (i.e., GET) and modifying (i.e., POST, PUT and DELETE) resources.

Below we summarise the basics of approaches currently adopted by the industry and tailored to the description of Restful APIs.

2.2.2.1 RAML (Restful API Modeling Language)

RAML [RAML Spec 2019] is a language for the specification of RESTful APIs whose syntax builds on YAML (YAML Ain't Markup Language) [Ben-Kiki et al. 2009]. A RAML description is conceptually a type definition; consequently, RAML provides a set of primitive built-in types (such as **number** and **string**) and type constructors. A user-defined type is an object type, i.e., a labelled record that associates properties (also member names) to types. User-defined types are analogous to JSON (JavaScript Object Notation) and XML (eXtensible Markup Language) schemas. Object properties can be either standard properties or **facets**, i.e., properties aimed at defining variant configurations for an object (e.g., to define the default value of a type, the minimum size of a collection, etc).

Each resource is defined by a type declaration that provides the operations that can be applied to it. A resource definition may include the following operations: **get**, **patch**, **put**, **post**, **delete**, **head** and **options**; which are in one-to-one correspondence with the homonymous HTTP commands. As standard in object models, operations are just properties whose types describe their intended

meaning. In RAML, an operation (called **method**) is described also as an object type that contains (among others) the following properties:

- **queryParameters**: the parameters allowed/required by the operation.
- **body**: the payload of a method invocation (i.e., the media type for body of the associated HTTP request).
- **responses**: a (partial) mapping that associates HTTP response codes with payload type.
- **protocols**: the protocols allowed for method invocation.
- **securedBy**: mechanisms used for securing data access and identification.

In addition, types can be organised into inheritance lattices through type derivation from **resource types** and **traits**. RAML supports multiple inheritance.

Then, a RAML specification defines a collection of resources, each of them identified by a relative URI (Uniform Resource Identifier) and associated with an object type. Resources are organised hierarchically as a forest; roots are called **top—level** resources while the remaining ones are called **nested** resources. The nesting structure is reflected by the prefix relation on the URIs assigned to the resources.

RAML can be used to automatically generate client and server code as well as documentation. There are several tools based on RAML that give support to different activities in the lifecycle of an API; among others, edition (API Designer), evaluation (API console), testing (API Notebook), and documentation (RAML for JAX-RS).

2.2.2.2 Open API

An OpenAPI specification is either a YAML or a JSON document that defines the following 8 object types:

- **openapi**: it is a textual reference to the version of the OpenAPI Specification used in the document; it is a meta-information that establishes the intended semantics of the remaining objects.
- **info**: it contains basic information about the API that is being defined, such as its title, description, and version, and may provide links to licences and terms of services.
- **servers**: an array of objects, each of which provides a basepath (i.e., the prefix for the URLs) to access the operations provided by the API. Different servers may provide access to operations in different environments (e.g., test, beta, production).

- **paths**: a collection of objects, each of them providing a relative path (for the URL) to access an operation of the API and an object describing the operation itself. As in RAML, each operation is defined as a property named as an HTTP request (e.g., **get**, **post**, etc). The type of each of such properties is operation object, which contains (among others) the members **parameters**, **requestBody**, **responses** and **security** analogous to the corresponding RAML properties. Differently from RAML, operations may stipulate **callbacks**, i.e., describe the requests that may be initiated by the API provider together with the expected responses.
- **components**: auxiliary type definitions that enable reuse and modularisation. They are the counterpart of type objects and traits in RAML. Components are instrumental for the definition of the remaining object in the document, they are used by referencing them.
- **security**: describes the default security mechanisms that apply to all operations provided by the API. However, each operation may override this default definition by establishing particular security requirements with the property **security** of the object **path**.
- **tags**: they provide meta-data for paths; such information can used, e.g., to relate different operations and exploited by tools for browsing API specifications.
- **externalDocs**: it provides links to external (extended) documentation.

Swagger gives tool support to different activities related the provision of APIs based on OpenAPI specification. Among such activities are editing specifications, enforcing style guidelines, versioning, inferring specifications. OpenAPI specification is supported by several integration platforms such as GoogleCloud.

2.2.2.3 Blueprint

Blueprint is description language for RESTful APIs [API Blueprint]. A Blueprint specification is a Markdown document that conforms the GitHub Flavored Markdown syntax [Flavored Markdown syntax]. A Blueprint specification is structured around resources, which are defined in the Resource section of the document. As in other approaches, a resource is specified in terms of its URI and the set of actions that can be performed over it. Actions are defined by providing:

- a name (optional)
- an URI (or template)
- the associated HTTP method (e.g., GET, POST, etc.).
- the request description that establishes the types of the required and optional parameters as well as the payload specification, i.e., a specification

of the information transferred as the payload of the corresponding HTTP request.
- the response specification, which maps HTTP status code with the payload specification.

A payload definition in Blueprint allows for the specification of both the attributes of a message-body and a validation schema. Attributes are specified as data structures using the MSON (Markdown Syntax for Object Notation) [Markdown Syntax for Object Notation] while a validation schema is a JSON Schema that establishes the valid shapes of a payload.

2.2.2.4 OData

A REST API is described in ODATA [OData Version 4.01] in terms of a model of the data exposed by a service, called Entity Data Model (EDM). The building blocks of a EDM are entities, which are instances of entity types. Entity types are object types that can be organised into a inheritance hierarchy. Each entity type has an associated key, which is defined in terms of a subset of its properties. Entities can be related to each-other by using navigation properties. Operations that can be applied to entities. There two kind of operations: Functions, which do not have side-effects, and actions.

Services expose each of their entities (or properties of their entities) by providing a read URL (for read-only access) and edit URL (for updating and elimination).

As in previous languages, ODATA allows for the specification of request parameters (headers and payload) and interpretation of server responses (i.e., HTTP status code).

Analogously to the previous approaches, operations are executed by sending HTTP commands GET, POST, etc.

2.2.3 Query Languages for APIs

GraphQL [GraphQL Draft] is a query language that allow clients to retrieve data from services; basically clients submit GraphQL queries containing the shape of the requested data and the server responds back a JSON document containing the requested data shaped as requested by the client. Differently from a REST service, a GraphQL API provide a single endpoint that gives access to all capabilities of the service. GraphQL APIs are organised around types and fields, not single resources/endpoints. In this way a single request over a GraphQL service may obtain all the needed data and avoid accessing several resources (as typically needed in a REST API).

In terms of API provision, a GraphQL service publishes the capabilities that a client is allowed to query. Conceptually, a server defines a type system used for validating client's requests and providing guarantees about the structure of the responses, i.e., a server defines a set of types (usually object types) that characterises the data that can be queried on that service. The type system associated to a service is defined through a GraphQL document containing schema definition. A schema associates operations to the object types that the server responses will contain. A schema may contain three kind of operations:

- **query**: for read-only data retrieval.
- **mutation**: for an operation that is expected to have side-effects on the underlying data.
- **subscription**: the client make a request to keep receiving data associated with future events on the server.

A schema must provide at least a query operation but mutation and subscription operations are optional.

In addition, GraphQL features introspection capabilities, i.e., a service can be asked about the queries it supports. This is achieved by querying the schema field, which is always available on a GraphQL service.

2.2.4 Remote Procedure Call (RPC) APIs

The main idea behind RPC (Remote Procedure Call) systems is the use of an interface that defines the methods that must be implemented by a server and can be invoked remotely by clients. Such interface defines the parameter types and the return types for each method. A server implements such interface and clients invoke operations locally through a stub. An underlying protocol handles communications. Recent approach for handling RPC systems is gRPC [gRPC guides], which relies on Protocol Buffers [Protocol Buffers] for serializing structured data. Then, an API description in gRPC consists on providing the signature of provided operations, where data types are defined by using protocol buffer syntax (alternatively, data can be described as JSON documents).

gRPC allows for the definition of four different kind of methods:

- Unary calls: standard call-return request, in which the client invokes an operation and awaits for the response.
- Server streaming: the client sends a request and awaits for a stream to read a sequence of messages.
- Client streaming: the client sends a sequence of messages over a provided stream and the server returns a response.

- Bidirectional streaming: the server and the client write and read sequences of messages over a read-write stream.

Analogous systems are Avro [Apache Avro™ Documentation] and Thrift [Apache Thrift Documentation] by Apache Software Foundation.

3. Progress report on activities of WP2

Work has been done on understanding how the use of behavioural types for shared objects and concurrency (Section 2.1.6) can help in the context of virtual machines for sensor networks developed by GreenByWeb. With this aim in mind, a behavioural type system is being developed for a realistic object-oriented language ruling out null-pointer dereferencing and memory leaks as a by-product of a safety property (protocol fidelity) and of a weak liveness property (object protocol completion for terminated programs). This type system will be at the basis of a behavioural type inference algorithm (T.2.7).

A preliminary study has been conducted on the use of session types (Section 2.1.2) to instruct monitors that check whether endpoints communicating via First-In-First-Out (FIFO) buffers deviate from some expected behavior. This is particularly useful in situations in which static analysis techniques cannot be applied (e.g. there is no a-priori knowledge of the behavior of the endpoints). The idea is to investigate a notion of refinement (T.2.4) that reflects, at the level of monitors, the typical notion of session subtyping. This is a key notion in case one would like not only to synthesize monitors, but also to type check a monitor against a given session type. Several notions of refinement appear interesting to be investigated, depending on the ability of the monitor to interact with the monitored system: simply observe the exchanged messages, or drop some unexpected messages, or even inject messages in the communication queues.

Quality of Service (QoS) and, in general, non-functional properties of distributed and communicating processes have received fewer attention by the community interested in behavioural types. Yet, these aspects are fundamental for enterprises. Work has been initiated to extend choreographies (Section 2.1.4) so as to to characterize QoS of processes interacting in a distributed system. This is made possible by introducing non-functional attributes, called *quantitative attributes*, to classify functionally equivalent services by the QoS they provide. A method for automatically analyzing this specifications is being investigated. These are necessary steps to equip behavioural APIs with non-functional information on the behaviour of processes and to check whether processes adhere to these APIs.

Work has been done on adding dynamic checks (Section 2.1.2) to Jolie [Montesi et al. 2014], which is a programming language tailored to the definition of base services and their composition into a service architecture, and the programming of orchestrators that monitor the correct interaction among the services. Concretely, the work addressed the problems of (i) defining a suitable DSL (Domain Specific Language) for expressing contracts in the form of FSM (Finite State Machines) with *guards*, (ii) evaluate alternative for integrating contracts to the main abstractions of Jolie, i.e., interfaces, ports (either input or output) or services, (iii) integrating contracts to the mechanisms for monitoring provided by Jolie; two strategies were identified: intrusive (monitor intermediates all exchanged messages and allows only right messages), non-intrusive (monitoring goes in parallel with the execution and eventually notifies deviations but without altering the interaction). UNIBO and UBA will develop further this model in the context of task T.2.5 (Dynamic Analysis).

A model for data-driven choreographies has been proposed by Bruni et al. [2019], where interaction over tuple-spaces replaces the standard mechanism of interaction based on message passing (as described in Section 2.1.4). Instead of primitives for sending and receiving messages, the primitives now are the ones for inserting a tuple on a tuple space, for reading (without consuming) a tuple from a tuple space or for retrieving a tuple from a tuple space. Unlike behavioural types such as session types (Section 2.1.2), these specifications express the data flow across distributed tuple spaces rather than detailing the communication pattern of processes. In the context of T.2.2, we plan to study the suitability of data-driven approaches as formal models for resource/graph-based languages and technologies (Sections 2.2.2 and 2.2.3). Also, mailbox types (Section 2.1.5) could provide a suitable model on which building an API expressing valid tuple configurations.

Work has been done on the definition of choreographic models (Section 2.1.4) accounting for refinement, i.e., a partially defined choreography can be refined by substituting partially defined interactions by protocols. The work is aimed at providing a formal notion of refinement in choreography models and a criterion for deciding when a particular refinement of a choreography is admissible, i.e., the obtained model is still realisable. Future work is aimed at establishing the correspondence between notions of refinement at choreographic level and refinement at service level. The outcome of this activity is expected to lay the basis for the development of engineering tasks for behavioural APIs, such as testing refined APIs (Task 2.7).

Work has been done on type inference algorithms for the synthesis of EPAs (Section 2.1.6) suitable for test case generation techniques (Task T.2.6). A new coverage criteria over EPAs has been defined and search-based test case

generation techniques aimed at achieving high EPA coverage has been developed. The initial results suggest that the proposed coverage criteria and fitness functions can provide better fault-detection capabilities (for general, but particularly for protocol failures) when compared to random testing and search-based test generation for standard structural coverage.

Bibliographic references

- Jonathan Aldrich, Joshua Sunshine, Darpan Saini, Zachary Sparks: Typestate-oriented programming. OOPSLA Companion 2009: 1015-1022
- Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, Nobuko Yoshida: Behavioral Types in Programming Languages. Foundations and Trends in Programming Languages 3(2-3): 95-230 (2016)
- Banerji, A., Bartolini, C., Beringer, D., Chopella, V., et al.: Web Services Conversation Language (WSCL) 1.0 (March 2002), http://www.w3.org/TR/2002/NOTE-wscl10-20020314
- Massimo Bartoletti, Ilaria Castellani, Pierre-Malo Deniélou, Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Jovanka Pantovic, Jorge A. Pérez, Peter Thiemann, Bernardo Toninho, Hugo Torres Vieira: Combining behavioural types with security analysis. J. Log. Algebr. Meth. Program. 84(6): 763-780 (2015)
- Nels E. Beckman, Duri Kim, Jonathan Aldrich: An Empirical Study of Object Protocols in the Wild. ECOOP 2011: 2-26
- Giovanni Bernardi, Matthew Hennessy: Using higher-order contracts to model session types. Logical Methods in Computer Science 12(2) (2016)
- Laura Bocchi, Kohei Honda, Emilio Tuosto, Nobuko Yoshida: A Theory of Design-by-Contract for Distributed Multiparty Interactions. CONCUR 2010: 162-176
- Mario Bravetti, Marco Carbone, Gianluigi Zavattaro: Undecidability of asynchronous session subtyping. Inf. Comput. 256: 300-320 (2017)
- Mario Bravetti, Gianluigi Zavattaro: A Foundational Theory of Contracts for Multi-party Service Composition. Fundam. Inform. 89(4): 451-478 (2008)
- Mario Bravetti, Gianluigi Zavattaro: A theory of contracts for strong service compliance. Mathematical Structures in Computer Science 19(3): 601-638 (2009)

- Sara Capecchi, Elena Giachino, Nobuko Yoshida: Global escape in multiparty sessions. Mathematical Structures in Computer Science 26(2): 156-205 (2016)
- Marco Carbone, Fabrizio Montesi: Deadlock-freedom-by-design: multiparty asynchronous global programming. POPL 2013: 263-274
- Marco Carbone, Nobuko Yoshida, Kohei Honda: Asynchronous Session Types: Exceptions and Multiparty Interactions. SFM 2009: 187-212
- Giuseppe Castagna, Nils Gesbert, Luca Padovani: A theory of contracts for Web services. ACM Trans. Program. Lang. Syst. 31(5): 19:1-19:61 (2009)
- Chinnici, R., Moreau, J.-J., Ryman, A., Weerawarana, S.: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language (March 2006), http://www.w3.org/TR/2006/CR-wsdl20-20060327
- Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1 (2001), http://www.w3.org/TR/2001/NOTE-wsdl-20010315
- Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, Nobuko Yoshida: Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions. COORDINATION 2013: 45-59
- Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Luca Padovani: Global progress for dynamically interleaved multiparty sessions. Mathematical Structures in Computer Science 26(2): 238-302 (2016)
- Silvia Crafa, Luca Padovani: The Chemical Approach to Typestate-Oriented Programming. ACM Trans. Program. Lang. Syst. 39(3): 13:1-13:45 (2017)
- Ugo de' Liguoro, Luca Padovani: Mailbox Types for Unordered Interactions. ECOOP 2018: 15:1-15:28
- Robert DeLine, Manuel Fähndrich: Typestates for Objects. ECOOP 2004: 465-490
- Simon J. Gay, Malcolm Hole: Subtyping for session types in the pi calculus. Acta Inf. 42(2-3): 191-225 (2005)
- Ronald Garcia, Éric Tanter, Roger Wolff, Jonathan Aldrich: Foundations of Typestate-Oriented Programming. ACM Trans. Program. Lang. Syst. 36(4): 12:1-12:44 (2014)
- Hannah Gommerstadt, Limin Jia, Frank Pfenning: Session-Typed Concurrent Contracts. ESOP 2018: 771-798
- Kohei Honda, Nobuko Yoshida, Marco Carbone: Multiparty Asynchronous Session Types. J. ACM 63(1): 9:1-9:67 (2016)
- Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara,

Emilio Tuosto, Hugo Torres Vieira, Gianluigi Zavattaro: Foundations of Session Types and Behavioural Contracts. ACM Comput. Surv. 49(1): 3:1-3:36 (2016)
- Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, Philip Wadler: Gradual session types. PACMPL 1(ICFP): 38:1-38:28 (2017)
- Cosimo Laneve, Luca Padovani: An algebraic theory for web service contracts. Formal Asp. Comput. 27(4): 613-640 (2015)
- Cosimo Laneve, Luca Padovani: The Pairing of Contracts and Session Types. Concurrency, Graphs and Models 2008: 681-700
- Hernán C. Melgratti, Luca Padovani: Chaperone contracts for higher-order sessions. PACMPL 1(ICFP): 35:1-35:29 (2017)
- Dimitris Mostrous, Nobuko Yoshida: Session typing and asynchronous subtyping for the higher-order π-calculus. Inf. Comput. 241: 227-263 (2015)
- Rumyana Neykova, Laura Bocchi, Nobuko Yoshida: Timed runtime monitoring for multiparty conversations. Formal Asp. Comput. 29(5): 877-910 (2017)
- Luca Padovani: Contract-Directed Synthesis of Simple Orchestrators. CONCUR 2008: 131-146
- Luca Padovani: Fair Subtyping for Open Session Types. ICALP (2) 2013: 373-384
- Luca Padovani: Fair subtyping for multi-party session types. Mathematical Structures in Computer Science 26(3): 424-464 (2016)
- Luca Padovani: Context-Free Session Type Inference. ESOP 2017: 804-830
- Luca Padovani: Deadlock-Free Typestate-Oriented Programming. Programming Journal 2(3): 15 (2018)
- Luca Padovani: A type checking algorithm for concurrent object protocols. J. Log. Algebr. Meth. Program.100: 16-35 (2018)
- Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, Éric Tanter: First-class state change in plaid. OOPSLA 2011: 713-732
- Peter Thiemann, Vasco T. Vasconcelos: Context-free session types. ICFP 2016: 462-475
- Cláudio Vasconcelos, António Ravara: From object-oriented code with assertions to behavioural types. SAC 2017: 1492-1497
- Roy T. Fielding, Richard N. Taylor: Principled design of the modern Web architecture. ICSE 2000: 407-416
- RAML Version 1.0: RESTful API Modeling Language (Spec). https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/. Accessed Feb, 2019.

- Oren Ben-Kiki, Clark Evans, C., Ingy döt Net. YAML Ain't Markup Language (YAML™) version 1.2. https://yaml.org/spec/1.2/spec.html (2009)
- API Blueprint. Format 1A revision 9. https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md#api-blueprint (Latest access February 2019).
- GitHub Flavored Markdown syntax. https://help.github.com/articles/basic-writing-and-formatting-syntax/ (Latest access February 2019)
- Markdown Syntax for Object Notation. https://github.com/apiaryio/mson (Latest access February 2019)
- GraphQL. Current Working Draft. https://facebook.github.io/graphql/draft/ (Latest access February 2019)
- gRPC Guides. https://grpc.io/docs/guides/ (Latest access February 2019)
- Protocol Buffers Guides. https://developers.google.com/protocol-buffers/docs/overview (Latest access February 2019)
- OData Version 4.01. Part 1: Protocol Committee Specification 01-30 January 2018
- Apache Avro™ 1.8.2 Documentation. http://avro.apache.org/docs/current/.
- Apache Thrift™ Documentation. https://thrift.apache.org/docs/ (Latest access February 2019)
- Fabrizio Montesi, Claudio Guidi, Gianluigi Zavattaro: Service-Oriented Programming with Jolie. Web Services Foundations 2014: 81-107
- Roberto Bruni, Andrea Corradini, Fabio Gadducci, Hernán Melgratti, Ugo Montanari, and Emilio Tuosto. Data-driven choreographies à la Klaim. Submitted 2019.
- Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, Sebastián Uchitel: Program abstractions for behaviour validation. ICSE 2011: 381-390
- Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, Sebastián Uchitel: Enabledness-based program abstractions for behavior validation. ACM Trans. Softw. Eng. Methodol. 22(3): 25:1-25:46 (2013)

Note: The results reported in this deliverable have not been published elsewhere. We expect new developments of languages and models to appear in conference proceedings and/or journal articles. This deliverable will be made available on the project Web site https://www.um.edu.mt/projects/behapi/ in due course.