Varda: A Framework for Compositional Distributed Programming

Laurent Prosperi[1(✉)], Ahmed Bouajjani[2], and Marc Shapiro[1]

[1] Sorbonne-Université, CNRS, Inria, LIP6, Paris, France
{laurent.prosperi,marc.shapiro}@lip6.fr
[2] IRIF, Université Paris Cité, Paris, France
abou@irif.fr

Abstract. A distributed system is made of interacting components. The current manual, *ad-hoc* approach to composing them cannot ensure that the composition is correct, and makes it difficult to control performance. The former issue requires reasoning over a high-level specification; the latter requires fine control over emergent run-time properties. To address this, we propose the Varda language (a work in progress) to formalize the *architecture* of a system, i.e., its components, their interface, and their orchestration logic. The Varda compiler checks the architecture description and emits *glue code*, which executes the orchestration logic and links to the components. The Varda system relies on a generic *interception mechanism* to act upon distribution-related system features in a transparent and uniform manner. Varda also takes into account important non-functional system properties, such as placement.

Keywords: Distributed programming · Language · Distributed system · Composition · Orchestration · Architecture

1 Introduction

The developer of a distributed system rarely implements it from scratch, as a monolithic program. Instead, a common approach is to compose independent components, either off-the-shelf or bespoke. For instance, a sharded key-value store might be composed of shard servers (`a` and `b` in Fig. 1a), with a router to direct client requests to the correct shard.

The composed system should both be safe and have good performance. This requires the developer to be able to: (1) formalize the individual components; (2) specify how they communicate [17,25,30]; (3) reason over both the static effects of the composed object [25], and its dynamic effects; and, (4) control and other non-functional and performance-related properties, such as co-location or inlining.

The current approach to compositional programming is *ad-hoc* and mostly manual. It consists of running components as processes that send messages to each others' API [30]. This satisfies in part Requirements 1 and 2 above, but does not express high-level safety [17,30], placement, or performance constraints.

M.-A. Koulali and M. Mezini (Eds.): NETYS 2022, LNCS 13464, pp. 16–30, 2022.
https://doi.org/10.1007/978-3-031-17436-0_2

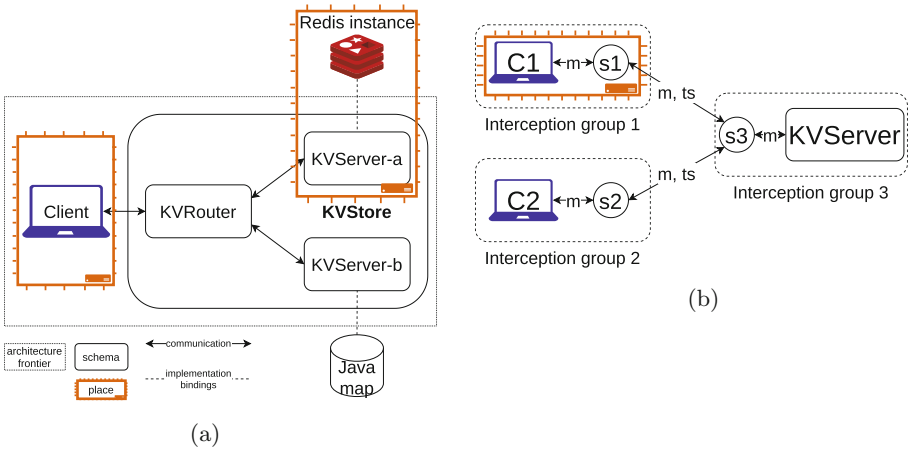

Fig. 1. (a) A key-value store (S-KV) composed of two shards and a router. (b) Adding causally consistent communication (cc) layer to our key-value store: with two client ($c1$, $c2$), one KVServer (kv) and three CC-sidecars ($s1$, $s2$, $s3$). m denotes a message and ts denotes a vector clock.

An improvement is to use an *orchestration engine*, such as Docker Compose, Kubernetes or OpenStack [3,4,15] to automate deployment and to control the topology. This addresses Requirement 4 only. Alternatively, a *protocol language* or an *orchestration language* can express some of the semantics. However, current languages do not satisfy Requirements 3 and 4, as we detail in Sect. 4.

We address these issues with Varda, our framework for compositional distributed programming. A system developer specifies the *architecture* of a distributed system in the Varda language. This enables to formally define the components of a system, their interface, their interconnection, and their placement. In particular, our *orchestration* sublanguage prescribes the run-time interactions between the components. Based on this specification, the Varda compiler performs static and run-time checks, and generates the interaction code between components, called the *glue*.

Note that an architecture description abstracts over issues not related to distribution. In particular, the individual components are imported and linked into the generated glue, but assumed implemented outside of our framework (written for instance in Java).

We claim the following contributions for this work:

- A language for expressing the component architecture and the orchestration of a distributed system (Sect. 2).
- A general *interception* mechanism for imposing orchestration logic and other transformations onto components (Sect. 3).
- As an example, we show how to impose transparently a common pattern: sharding (Sect. 3).

This paper does not yet provide an experimental evaluation, as the implementation is progress.

2 Programming Model

2.1 Concepts

Let us first explain the Varda concepts and terminology, based on the example in Fig. 1a. It represents the architecture of a key-value store. Its components are a client on the left, and the key-value store proper on the right, itself composed of a router in the centre and two servers on the middle right. The router forwards client requests to the appropriate server.

The two servers are distinct *activations* of the same *schema*; these concepts are somewhat to instantiations and classes in object-oriented languages respectively.

A schema, written in the Varda language, describes the component's interactions with other components. In our example, a server schema accepts *get* and *put* invocations, which it executes against its storage backend. The router schema accepts the same *get*/*put* signature as a server, but its behaviour is different: based on the arguments, it forwards the invocation to the appropriate server activation, awaits the response, and forwards the response to the client.

An activation can link to an *implementation*, a black-box executable component exported as a library. In the figure, the implementation of Activation `a` stores its data in a Redis implementation [36], whereas that of Activation `b` uses a custom storage logic.

Finally, the figure shows *places*, i.e., physical or logical locations in the distributed system. In this example, the client is in its own place, and Server `a` is in the same place as its implementation. Placement is an important consideration, for instance for performance, fault tolerance or physical security.

2.2 Components

Recall that a schema is the code for a class of components. The Varda schema code has several parts, each described in an appropriate sublanguage. Its *signature* declares the names and types of messages it can send and receive, using both classical (declarative) types and (imperative) safety assertions. Its *protocol* describes the sequencing of such messages, expressed in the language of *session types* [14]. *Ports* are communication entry and exit points; a port is described by its name, signature, and protocol.

The *orchestration* logic describes how the component behaves, in a Turing-complete imperative sublanguage. It can specify a callback method to be invoked when a given type of message is received. It also includes specific methods for creating and destroying an activation of the component (called `onStartup` and `onDestroy` respectively). Orchestration logic can maintain local state, can send messages, and can invoke the implementation.

The binding between the component, and its implementation written in some external programming language, is expressed using imperative templates that embed fragments of the external language.[1]

[1] Currently our compiler supports only Java bindings.

A component schema may contain sub-components. The scope of a sub-component is the enclosing component, i.e., a sub-component cannot be invoked from the outside.

An instance of a schema at run time is called an *activation*. The activation is the smallest grain of distribution and concurrency. Computation within an activation is sequential. Receiving a message, instantiating the activation or terminating it run the corresponding callback method. A method executes until it terminates, or until it waits for an asynchronous invocation.

2.3 Interaction Interface

This subsection details how two components interact. Activations communicates by sending messages to each other. Programmers group message into protocols. A protocol describes the type and the order of events. Session types [14] directly inspire protocols. Between activations, those messages are flowing through channels. A channel interconnects ports of multiple components. A programmer formalizes components interface by defining *ports*: `inport`, to communicate with the outside, and `outport`, to listen for incoming messages.

The interaction should both be safe and have good performance. This requires the developer to be able to: (a) constraints the communication topology to explicitly specify which component is talking to whom; (b) interacting component have to agree on the order and the type of messages they exchange to perform lightweight verification and to drive the code-generation of the networking interfaces; (c) represents the underlying network layer to do specialize the code-generation and to represents assumption on the underlying network in the architecture description; and, (d) (weakly) isolate component functionalities from each other.

Events. To communicate, activations exchange events. Each event is strongly typed and can carry a payload. Its payload should be serializable.

A programmer can manually define an event *key* carrying a `string` payload with `event key of string;`. Otherwise, a programmer can send classical serializable types without defining events. The Varda compiler auto-box those types into events and un-box them at reception. Event auto-boxing alleviate the programmer from the burden of defining events for base types (e.g., `int`).

Varda type system supports type evolution of event through subtyping. Subtyping define a relation of substitutability between data types. Substitutability is a property where code written to operate on the supertype can safely be substituted for any of the subtypes in the subtyping relationship [27].

A component can send a message with more information than expected. For instance, lets assume than Activation b expects messages of type record: `{value: string}`. An Activation a can send to b a message of type: `{checksum: string; value: string;}`. At reception, b considers only the field `value`.

Protocols. Protocols address Requirement b. It constraints communication between activations: programmer attaches a protocol to each channel and each port. A (binary) session represents one instance of a protocol. An Activation i can creates a session `s` with Activation j by calling `initiate_session_with (outport_of_i, j)`. Let's assume that the protocol of the port is `protocol p_get = !key?value. ;`. Where key and value are event types. The session type implicitly bound to `p_get` guarantees that a communication through s is as follows: i starts by sending a message of type `key` and ends by receiving a message of type `value`.

Varda exposes classical communication primitives managing session [14]: asynchronous message sending `fire(s, msg)`, asynchronous receiving using callback (ports) or `receive` primitive, non-deterministic branching `branch` and recursive protocol. Each of this operations returns a new session types with the protocol of the continuation and preserves the session identity.

Channels. Channels address Requirement c and their types solves the static part of Requirement a. A channel can interconnect multiple activations, of different component schemas. A channel can represents different communication guarantees, provided by the underlying network primitives. For instance, a channel can be protected by TLS encryption or can guarantee point to point FIFO communication, which is the default guarantee. A channel is compiled directly to network layer code to preserve performance.

A channel definition is asymmetric for communication establishment to statically constrain communication topology. A channel of type `channel<A, B, protocol>` guarantees that only activations of type `A` can initiate a request to activations of type `B`. Bidirectional channels can be constructed using union type: `channel<A|B, A|B, protocol>`.

Ports. The set of ports of a schema defines its interaction signature. Ports solves Requirement d. Each port define a functionality of a component: A port only accept communication that follow a given session type. Moreover, ports reduce the complexity of the component code: Ports abstract away the communication interconnection between the component inner logic (statically defined) and the activations interactions over channels (dynamic bindings). For instance, sessions primitives take ports as arguments and not channels.

Ports are static since they define the signature of a component schema: new ports can not be added nor removed at runtime. However, bindings between ports and channels depend of activation identity. Those bindings can evolve dynamically, and transparently for the inner activation logic. Operationally, a programmer binds a channel with port using the initial knowledge provided at activation creation (thanks to parameters) or by exchanging channel identity over existing sessions.

```
protocol p_kv = &{ (* non-deterministic choice *)
    "get": !key?value.; (* send key, receive value *)
    "put": !tuple<key,value>?bool.; (* or send tuple, receive ack *)
};

(* channel<active,passive,protocol> *)
channel<Client, KVServer, p_kv> chan = channel(p_kv);
(* start new component at a given location *)
activation_ref<KVServer> kv_a = spawn KVServer(chan) @ place_redis;
(* start and connect a client *)
activation_ref<Client> c = spawn Client(chan, kv_a);
```

Listing 1: The minimal key-value store in Varda.

2.4 Orchestration Logic

The objective of the orchestration is to write executable code doing dynamic interaction whereas the interaction interfaces describes what messages can be exchanged between components.

The main work of orchestration is to spawn activations and to interconnect ports using channels. Inside a component schema, the orchestration logic is in charge of doing the bindings between communication interfaces with procedural ones. For instance, this is the only work of the `callback` method of Listing 2. A programmer can also write the core behaviour of orchestration schemas (e.g., `KVRouter`) using the Varda orchestration logic in order to be completely agnostic to the underlying language. The Varda compiler generates the effective implementation.

In addition to component schema description language, Varda proposes a small imperative and Turing-complete language to write the orchestration logic. Varda language contains classical language constructs (e.g., binders, expression, function, control-flow statement and inductive type); communication primitives to exchange between activations using sessions; and, activation creation primitive: `spawn Schema(arguments) @ place;`.

2.5 Example: A Minimal Key-Value Store

Listing 1 presents the architecture of a warm-up case study: a key-value store composed of one server and one client. This warm-up example is a piece of Fig. 1a: a `KVServer` (`kv_a`), without sharding, that serves requests of one client such that the server use a Redis backend and is collocated with it. This example assumes that the Redis server is already running before spawning a `KVServer`. `KVServer` serves as a proxy to the Redis server.

In Listing 2, `KVServer` specifies the interface of a Redis server. Conversely, `Client` specifies of the interface of an application using the key-value store. `KVServer` exposes a *communication interface*, composed of its port `p_in` and the communication handling logic `callback` method; a *procedural interface* composed of two abstract methods `get` and `put` bound the black-box service (resp.

```
component KVServer {
    (* Method triggered at spawn, binds the channel *)
    onstartup (channel<Client, KVServer, p_kv> chan){
        this.chan = chan;
    }
    (* Communication interface *)
    channel<Client, KVServer, p_kv> chan;
    (* Liste on channel [this.chan] for session with
       (dual p_kv) type.  Upon reception, message is handled by
       [this.callback]. *)
    inport p_in on this.chan expecting (dual p_kv) = this.callback;
    (* Bindings between interaction interface and procedural interface *)
    void callback (blabel msg, p_kv s) {
        branch s on msg { (* non deterministic choice*)
            | "get" => s -> {
                tuple<key, ?value.> tmp = receive(s); (* wait for key *)
                (* return the value bound to the received key [tmp.0] *)
                fire(tmp.1, get(tmp.0));
            }
            | "put" => s -> { ... }
        }
    }
    (* procedural interface *)
    value get(key k);
    bool put(key k, value v);
}
```

Listing 2: KVServer component schema

```
    target akka;

    impl headers {=
        (* use the java-redis-client library *)
        import nl.melp.redis.protocol.Parser;
    =}

    (* binding for the get method *)
    impl method KVServer::get {=
        (* Open a socket to the local redis backend *)
        nl.melp.redis.Redis r = new nl.melp.redis.Redis(new
        ↪  Socket({{ip(current_place())}}, {{port(current_place())}});
        (* perform the GET request on key [k] *)
        return r.call("GET", {{string_of_ley(k)}});
    =}
```

Listing 3: `KVServer::get` bindings for the Akka target. The compiler interprets Varda `{{expression}}` strings inside an `impl` body.

implementation) and no orchestration logic. The compiler specializes the two abstract methods `get` and `put` during code generation according to implementation bindings (Listing 3). Moreover, the communication handling logic (here the `callback` method) is in charge of doing the binding between the communication interface and the procedural interface.

A channel `chan`, guaranteeing FIFO delivery for point-to-point communication, interconnects the client with the server (Listing 1). Both client and server discover `chan` as an argument. `chan` is asymmetric and constrains the communication topology: the left hand side of the channel type (e.g., `Client`) initiate the communication, the right hand side can not. Moreover, communication follows the protocol `p_kv` (technically, a session type [14]): a client can choose between two operations `put` or `get`. Once client chooses the `get` (resp. `put`) case, the communication must follow the pattern: client sends a `key` and expects to receive a `value` before the session is closed (resp. put).

3 Interception

At this point, one major remaining question is how to easily and safely enrich (or trim) system's functionalities. For instance, manually sharding the minimal key-value (Listing 1) would be time consuming: a programmer needs to manually (1) create the sharding logic (the router); (2) creates new channels to interconnect the shards (resp. the clients) with the sharding logic; (3) instantiate a router with correct channels interconnections; and (4) for each shards, bind correctly the new channels.

We propose that Steps (2), (3) and (4) should be automatically handled during compilation while followings this requirements: (a) impose arbitrary interception, orthogonal from placement and communication topology, and prevent intercepted activation to bypass the interception mechanism (b) be non invasive and transparent to avoid to the programmer to edit the whole architecture; (c) be generic and modular (d) should be executed efficiently to preserve performance; and, (e) be preserved by composition: multiple alterations could be nested to modularly build a major functionalities.

To address this problem, Varda leverages the interception mechanism as the core Varda primitive used to uniformly apply the orchestration logic. Developers write the interception logic at same abstraction level, and in the same language as the orchestration logic. Then, the Varda transparently and statically rewrite the architecture by adding proxies [38] in between groups of activations.

Varda interception is an architecture construction. This helps preserving the preexisting semantics and formalizing the new architecture. Other approaches work on the network layer and do dynamic interception, as we describe it in Sect. 4.4.

In the following, we review what a programmer can achieve using interception:

– *Message redirection* can be achieved by using the same interception instrumentation as sharding, with a custom routing policy (e.g., round-robin for *loadbalancing* and broadcasting for *replication*).

- *Dynamically constraining topology* (e.g., access control) can be done as long as dropping communication take place at session establishment since sessions can not be discard arbitrarily due to session type guarantees.
- *Encapsulating messages or piggy-packing metadata* between activations can also be done even if it is a bit more tricky: the programmer needs to introduce a new intermediate protocol without breaking transparency.
- *Changing the communication behaviour* can be performed by intercepting the communication and implementing the communication behaviour inside the interception logic. For instance, programmers can transparently replace a point to point communication by a broadcast.
- *Any combinations of those patterns* can be achieved using nested interception contexts.

3.1 What is Interception?

The interception concernes a group of channels in between an *internal* group of activations and the *external one* composed of all the remaining activations. In Varda, the programmer has only to enclose the creations of activations, she want to intercept, into an interception scope (using a `intercept` statement). The interception scope is part of the orchestration code. Therefore, applying interception is orthogonal to defining the logic of the both groups, their interactions and their placement. This solves Requirement a. Whereas, the interception behaviour can depend on those three elements.

Interception concerns both the session establishment and the messages exchanged inside the session. Interception give the ability to the programmer to alter arbitrarily the communications between two groups of activations: message value and session can be alter or delayed. However, the type of the protocol can not be altered arbitrarily, this point will be discussed when detailing transparency.

What is not Interception? Interception is not designed for ensuring security isolation. Interception can no prevent malicious activations to communicate with the external worlds. Indeed, interception works with Varda communication primitives whereas a malicious activation could *bypass it from below* by using arbitrary communication primitives provided by external code (e.g., sockets). Even if activations only communicate with Varda communication primitives, interception isolation could also be *breached by above* if an intercepted protocol allows channel exchange (recall that channels are first-class value) and if the intercepted activation dynamically binds this received channel to one of its ports. Varda compiler does not prevent this: breaching interception could be used to removed interception at some point to preserved performance, for instance once an activation has migrated. However, this kind of breaches can by either *forbidden*: by disallowing channel transmission in the protocol definition: or, *mitigated*: by checking the identity of forwarded channels inside the interception logic.

```
intercept<KVRouter, anonymous> interception_policy {
    activation_ref<KVServer> kv_a = spawn KVServer(chan);
    activation_ref<KVServer> kv_b = spawn KVServer(chan);
}

activation_ref<Client> c = spawn Client(chan, kv_a);
```

Listing 4: Intercepting KVServer from Listing 1 to support sharding

3.2 Example: A Sharded Key-Value Store

With Varda, transforming the simple key-value store example into a sharded version is a matter of transparently created an interception context containing two KVServer, Listing 4. Such that the interception logic, defined as a component called KVRouter, implements the sharding strategy. The `interception_policy` instantiates a singleton KVRouter activation for the whole interception context. The KVRouter postpones the establishment of a session between the interceptor and a KVServer until the client give enough knowledge (e.g., the key) to select the right KVServer. Delaying messages can be tricky, since arbitrary long delay between messages of the same session could be trigger a timeout depending of session implementation.

3.3 Expressing Interception

To setup interception context, programmers have three things to do: (1) define the *interception logic* by providing an interceptor component schema (e.g., KVRouter); (2) delimit the interception scope using a *intercept* block statement and (3) describe what interceptor activation is in charge of which intercepted activations thanks to an *interception policy*.

Interception Logic. The interception logic is in charge of processing (alteration, delaying and forwarding) session establishments and messages between internal and external activations. The interception logic is defined as annotated methods to remains generic and not to be specific to a given interception context. That way, programmers do not have to take care of creating the communication interface of the interceptor which depends on the interception context. The compiler is in charge of *specializing the interceptor component schema*, for each context, in order to create the needed ports according to the intercepted bridges. It binds the annotated methods with generated ports based on methods signature: intercepted session type (and message type for `@msginterceptor`) and the topology of the communication (defined by `from` and `to` schemas).

Varda provides three methods annotations: `@sessioninterceptor`, `@msginterceptor` and `@onboard`. `@onboard` methods are triggered at the creation of an intercepted activation. Interceptor needs onboarding to distinguish internal activations from externals one. To preserve transparency, onboarding

must be hidden to the intercepted activation (resp. external activations). Hence, it is up to the activation running the interception context to trigger the onboarding. Interception logic can access the set of in the set of `onboarded_activations`. `@sessioninterceptor` methods are triggered when a session is established, conversely `@msginterceptor` methods are triggered when a message cross the interception border.

Interception Context. Programmers define interception context inside the orchestration logic using a syntactic scope introduce by the `intercept` statement (Listing 4). Activations spawned inside this scope are intercepted, the others are not. Interception context does not behave like a classical syntactic scope. Indeed, to make the interception fully transparent in term of variable bindings, the interception scope exposes its binders. The parent scope, of the `intercept`, contains the variables bound inside the interception context. Activation and channels must process with special care not to break isolation, we only describe the activation case for brevity. Activations variables are exposed with the same type but, outside the interception context, they are references to their interceptor activation. This work transparently since the compiler specialize the interceptor schema into a subtype of any intercepted schema, i.e., communication interfaces are equivalent. Moreover, exposed activations may need to embed additional identity information. There are two different use cases: (1) for sharding, `KVServer` identity (`kv_a` and `kv_b`) are not exposed because a `Client` does not need to distinguish between intercepted activation; whereas (2) to achieve access control with interception, the intercepted activation identity must be exposed since sending a request to `kv_a` differs from sending one to `kv_b`. Identity exposition is managed by using the `anonymous` modifier of the `intercept` statement: `intercept<BaseInterceptor>` preserve identity whereas `intercept<BaseInterceptor, anonymous>` erase identity of intercepted activations.

User Defined Interception Policy. Neither the interception logic nor the interception context can expressed how and where interceptor activations are spawned and what is the relation between intercepted activation and interceptor activation (e.g., one to one or many to one). To achieve this, the `intercept` statement expect a user defined function called `interception policy`. Listing 5 defines a singleton interceptor activation in charge of all intercepted `KVServer`.

Programmers can use the interception policy to (1) define the relation between *intercepted activations* and *interceptor activation* by splitting intercepted activation in groups managed by an interceptor (according to their place, schema and identity); (2) to *reuse interceptor activation(s)* between interception context; (3) to *choose where to place interceptors*; and, (4) to *customize interceptor arguments in a per context basis.*

The policy is called at each spawn of an intercepted activation and it attributes an interceptor activation to each spawned activations. The arguments `intercepted_component_schema` denotes the schema of the intercepted activation and `p_of_intercepted` denotes its place. To make policy generic and

```
activation_ref<KVStore> policy(
    place -> activation_ref<KVStore> factory,
    string intercepted_component_schema,
    place p_of_intercepted
){
    if(this.singleton_interceptor == none()){
        this.singleton_interceptor = some(factory(current_place()));
    }

    return option_get(this.singleton_interceptor);
}
```

Listing 5: Interception policy for S-KV

strongly typed, the compiler does not pass arguments of the intercepted spawn to the policy.

To relieve programmers of binding the generated ports, of the specialized interceptor, with intercepted bridges (remember that both depends on the context and not only of the interceptor schema). The `factory` function spawns interceptor's activations to relieve programmers of binding the generated ports, of the specialized interceptor, with intercepted bridges (remember that both depends on the context and not only of the interceptor schema). The compiler provides and specializes a factory function for each context.

4 Related Work

4.1 Programming Languages

Classical programming models for distributed computing are actor model [6,8, 13], service oriented computing [26], dataflow [9] or reactive programming and tierless programming [10]. Recent evolutions tend to focus on easing specific distribution features by incorporating them into programming languages like consistency handling [18,29,32,33], placement aware-computation [37,40] and builtin fault-tolerance with [13,23,34] or without manual control [13]. However, they are not designed to compose black boxes easily and transparently while preserving programmer control on low-level details. This has a high cognitive cost for the programmer and a performance overhead.

4.2 Interface Description Languages

Interface description languages permits to formalize API to some extent and often to derive serialization mechanism and interfaces skeleton. Google's Protocol Buffer [21] and Apache's Thrift [19] provide basic typed specification of exchange messages. Hagar [1] extends the type system with polymorphism and generics. However, all of them tend to be limited on *what they can specify*: they can not reason on values; and, *they must be used manually in combination with other tools* to build a system which implies that they can not capture the orchestration nor the non-functional requirements.

4.3 Composition Framework

Currently composition mostly rely on interconnecting containerized application [5,12,16] or even serverless approach [7,20,22,31]. However, composition frameworks do not achieve safe composition [17,30]. They mostly work at the network layer which hamper reasoning on the semantics of the composition and of working on non-functional requirements. At a higher level of abstraction, CORBA [39] permits to transparently compose heterogeneous components with well-defined interfaces. They all deport the dynamic interconnections description and management into each component implementation without any general plan, except in English written documents. Regis [28] models communications and dynamic interconnection logic. However, this work do not address non-functional aspect of composition and they do not provide the ability to transform the architecture (like our interception mechanism) which means that every patterns must be established by hand.

4.4 Dynamic Interception

Other approaches providing interception mostly focus on dynamic interception. The use network based interception mechanisms: firewall-like features (e.g., iptables [2], mesh-services [11,24]) or service workers [35] embedded in browsers. They all lack the ability to describe the effects of the interceptions on the system's behaviour.

5 Conclusion

We present Varda, an architectural framework designed to build performant and safe distributed systems by composing heterogeneous components. Furthermore, it *discharges the programmer* from bridging the gap between implementation and design architecture; and that simplifies the writing of classical distribution patterns using a language-based interception mechanism. Varda model rests on three principles: **(1)** strict separation of concern between architecture and component implementation: one architecture can be used to generated multiple distributed systems; **(2)** interception is the core primitive to uniformly and transparently apply distribution patterns using static architecture rewriting; and, **(3)** preserve programmer control on distribution by incorporating dynamic aspects of the architecture (orchestration logic) and by embedding low-level details as first class value (e.g., place, bridges).

We are currently working on the evaluation of Varda: we are investigating the cognitive cost of the model and the performance overhead of the generated glue. Futures works can be divide in two branches: a) the first one targets performance, for instance optimizing the architecture using rewriting (e.g., merging components to avoid context switching); whereas, b) the second one explores how to improve the dependability of distributed system using Varda (e.g., enriching the type system or adding dynamic contracts).

References

1. Hagar. https://github.com/ReubenBond/Hagar
2. Iptables. https://www.netfilter.org/projects/iptables/index.html
3. Kubernetes. http://kubernetes.io
4. OpenStack. https://www.openstack.org/
5. Podman. https://podman.io/
6. Akka: Akka. https://akka.io/
7. Amazon: Aws lambda. https://aws.amazon.com/lambda/
8. Armstrong, J.: Erlang. Commun. ACM **53**(9), 68–75 (2010)
9. Bainomugisha, E., Carreton, A.L., Cutsem, T.V., Mostinckx, S., Meuter, W.D.: A survey on reactive programming. ACM Comput. Surv. (CSUR) **45**(4), 1–34 (2013)
10. Boudol, G., Luo, Z., Rezk, T., Serrano, M.: Reasoning about web applications: an operational semantics for HOP. ACM Trans. Program. Lang. Syst. (TOPLAS) **34**(2), 1–40 (2012)
11. Buoyant Inc.: Linkerd. https://linkerd.io/
12. Burns, B., Oppenheimer, D.: Design patterns for container-based distributed systems. In: Clements, A., Condie, T. (eds.) 8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, 20–21 June 2016. USENIX Association (2016). https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns
13. Bykov, S., Geller, A., Kliot, G., Larus, J.R., Pandya, R., Thelin, J.: Orleans: cloud computing for everyone. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, p. 16. ACM (2011)
14. Dezani-Ciancaglini, M., de'Liguoro, U.: Sessions and session types: an overview. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 1–28. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14458-5_1
15. Docker Inc.: Docker Compose. https://docs.docker.com/compose/
16. Docker Inc.: Docker Engine. https://www.docker.com/
17. Emerick, C.: Distributed systems and the end of the API. https://writings.quilt.org/2014/05/12/distributed-systems-and-the-end-of-the-api/
18. Eskandani, N., Köhler, M., Margara, A., Salvaneschi, G.: Distributed object-oriented programming with multiple consistency levels in ConSysT. In: Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, pp. 13–14. ACM (2019)
19. Apache Foundation: Thrift. https://thrift.apache.org/
20. Google: Cloud functions. https://cloud.google.com/functions/
21. Google: Protocol buffers. https://developers.google.com/protocol-buffers
22. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless computation with OpenLambda. In: Clements, A., Condie, T. (eds.) 8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, 20–21 June 2016. USENIX Association (2016). https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson
23. Hutchinson, N.C., Raj, R.K., Black, A.P., Levy, H.M., Jul, E.: The Emerald programming language (1991)
24. Istio: https://istio.io/

25. Kramer, J., Magee, J., Finkelstein, A.: A constructive approach to the design of distributed systems. In: 10th International Conference on Distributed Computing Systems (ICDCS 1990), 28 May–1 June 1990, Paris, France, pp. 580–587. IEEE Computer Society (1990). https://doi.org/10.1109/ICDCS.1990.89266
26. Lewis, J., Fowler, M.: Microservices. https://martinfowler.com/articles/microservices.html
27. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. (TOPLAS) **16**(6), 1811–1841 (1994)
28. Magee, J., Dulay, N., Kramer, J.: Regis: a constructive development environment for distributed programs. Distrib. Syst. Eng. **1**(5), 304–312 (1994). https://doi.org/10.1088/0967-1846/1/5/005
29. Meiklejohn, C., Van Roy, P.: Lasp: a language for distributed, coordination-free programming. In: Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming - PPDP 2015, pp. 184–195. ACM Press (2015)
30. Meiklejohn, C.S., Lakhani, Z., Alvaro, P., Miller, H.: Verifying interfaces between container-based components (2018)
31. Microsoft: Azure functions. https://functions.azure.com/
32. Milano, M., Myers, A.C.: MixT: a language for mixing consistency in geodistributed transactions. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2018, pp. 226–241. ACM Press. https://doi.org/10.1145/3192366.3192375. http://dl.acm.org/citation.cfm?doid=3192366.3192375
33. Milano, M., Recto, R., Magrino, T., Myers, A.: A tour of gallifrey, a language for geodistributed programming. In: Lerner, B.S., Bodík, R., Krishnamurthi, S. (eds.) 3rd Summit on Advances in Programming Languages (SNAPL 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 136, pp. 11:1–11:19. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019). https://doi.org/10.4230/LIPIcs.SNAPL.2019.11. http://drops.dagstuhl.de/opus/volltexte/2019/10554
34. Mogk, R., Baumgärtner, L., Salvaneschi, G., Freisleben, B., Mezini, M.: Fault-tolerant distributed reactive programming. In: 32nd European Conference on Object-Oriented Programming, ECOOP 2018, 16–21 July 2018, Amsterdam, The Netherlands, pp. 1:1–1:26 (2018)
35. Mozilla: Service Worker. https://developer.mozilla.org/fr/docs/Web/API/Service_Worker_API
36. Redis: Redis. https://redis.io/
37. Sang, B., Roman, P.L., Eugster, P., Lu, H., Ravi, S., Petri, G.: Plasma: programmable elasticity for stateful cloud computing applications. In: Proceedings of the Fifteenth European Conference on Computer Systems, pp. 1–15 (2020)
38. Shapiro, M.: Structure and encapsulation in distributed systems: the proxy principle. In: International Conference on Distributed Computing Systems (ICDCS), Camchannel, MA, USA, pp. 198–204. IEEE (1986). https://hal.inria.fr/inria-00444651
39. Vinoski, S.: CORBA: integrating diverse applications within distributed heterogeneous environments. IEEE Commun. Mag. **35**(2), 46–55 (1997). https://doi.org/10.1109/35.565655
40. Weisenburger, P., Köhler, M., Salvaneschi, G.: Distributed system development with ScalaLoci. Proc. ACM Program. Lang. **2**(OOPSLA), 129 (2018)