Resource interfaces

Samuel Chassot *School of Computer Science*
EPFL
Lausanne, Switzerland
samuel.chassot@epfl.ch

***Abstract*—Resource usage, in particular memory, is a critical aspect of a system when analysing its performance and correctness. In this work, we explore the concept of a resource interface for the memory resource. This concept is analog to the semantic interface concept which is widely used in computer systems conception and analysis as it is crucial for the concept of abstraction. We propose a way to represent these interfaces and how to extract them statically. We then explore how to apply them to the serverless computing paradigm. Finally, we present the prototype we built that is capable of extracting some basic interfaces for a Python function. This prototype is meant to form a basis for future development.**

***Index Terms*—resource interface, static analysis, formal methods, serverless computing, Function-as-a-Service, memory usage**

I. Introduction

Memory usage of programs is a critical metric to predict the performance and correctness of the systems with which these programs interact. Running out of memory leads often to a crash of the program and the consequences of that crash can be critical. Therefore, resource usage, in particular memory, plays a crucial role in the correctness of some applications.

Semantic interfaces are the basis of abstraction in modern programming. All library documentations contain the semantic interfaces of the offered functions or objects. In addition, the object-oriented paradigm is based on this concept. We argue that this concept can be extended to resource interfaces.

These resource interfaces would be conceptually close to the semantic interfaces but would express the resource usage with respect to the input parameters and other variables if needed. These interfaces are meant to be extracted statically (just as the semantic interfaces are). In this project, we focus on memory as the resource represented by these interfaces. The resource interfaces format and challenges are discussed in the section III.

In this report, we propose a use case for these interfaces: the serverless computing paradigm. Serverless computing is another name for Function-as-a-Service and is a cloud computing paradigm in which the users do not manage the infrastructure at all but only upload functions, the provider taking care of the rest. A more detailed description of serverless computing and the current state of the available solutions are exposed in section II.

We think that serverless computing services would benefit from the resource interfaces in several ways. This would allow the users and the providers to know the memory usage of a function (or at least an accurate estimation) without having to execute it. This would help the scheduling of the functions as well as the billing process and cost estimation. We develop these points in detail in the section III-C.

II. Background: serverless computing

A. Definitions

There is no formal definition for the concept of serverless computing and the services it offers [1]. We however provide the definitions proposed by Shafiei et al. in their review [1].

Function-as-a-Service (FaaS) is a paradigm in cloud computing and is often what serverless computing refers to. FaaS is a paradigm in which users run and manage functions without managing the infrastructure [1].

Serverless computing can also refer to the Backend-as-a-Service paradigm. In this paradigm, the unit the users access in the cloud is an entire service, handling a specific task like authentification or notification [1].

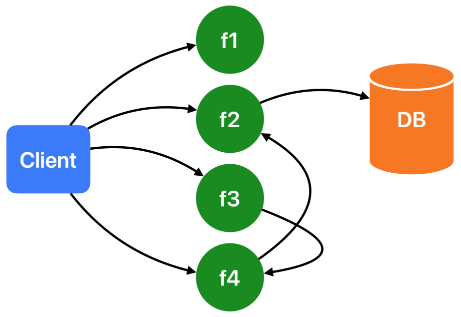
Neither FaaS nor BaaS requires any resource management from the users. In the case of FaaS, the users only manage the functions and the cloud provider takes care of provisioning the machines, creating and removing instances of the functions, and invoking them. In the case of BaaS, the users do not even manage functions and have direct access to the complete service.

In this report, when we talk about serverless computing, we refer to FaaS. Here are the important characteristics of a serverless service:

- The execution environment (e.g., Virtual Machines (VMs), containers, Operating Systems (OSes)) and its management is hidden from the users.
- The cloud provider provides auto-scaling service, i.e., the resources are made available on demand.
- The billing mechanism only reflects what has been consumed (i.e., pay-as-you-go).
- The provider handles the requests the best it can (i.e., best-effort policy)
- The basic elements are the functions. They are not hidden from the provider, which knows their dependencies, runtime environment, and state during execution. The provider also sees the inputs and outputs of those functions.

Let us now define what is a serverless application. A serverless application is generally composed of two parts: the client and the registered functions. The client sits in between the end users and the FaaS provider: it invokes the functions and translates their results into views. The registered functions are the functions uploaded to the FaaS provider: instances of those functions are created or deleted by the provider to handle requests sent by the client. This architecture is shown in Figure 1.

Fig. 1. Serverless application architecture example.



B. Market

The main providers for serverless computing services are Amazon with AWS Lambda, Microsoft with Azure Functions, and Google with Google Cloud Functions. Apache offers an open-source platform called OpenWhisk. According to [2], Auth0 was also offering a serverless computing service called Webtask, but it was discontinued [3].

AWS Lambda is by far the most used, with more than 70% of the surveyed companies using it in this survey by Datadog [4]. It is followed by Azure Functions and Google Cloud Functions with both around 55% of the surveyed companies as customers [4]. According to [2], which is based on a previous version of the Datadog report [4], in 2019, OpenWhisk and Webtask were both used by 2% of the surveyed companies. It also mentions that, in 2019, AWS Lambda was used by 96% of the companies, Azure Functions by 6%, and Google Cloud Functions by 4%. We can then see that even though AWS Lambda is still the leader of the market, Microsoft and Google are gaining in popularity. It is also interesting to point out that AWS Lambda was the first of these services, introduced in 2014 [5]. Azure Functions followed in 2016, first in preview since March with an official release in November [6]. Google Cloud Functions was introduced in beta in 2016 and became officially available in 2018 [7].

C. Opportunities

We now discuss why serverless computing offers an interesting paradigm and in which cases it is preferable to other computing paradigms according to Shafiei et al. [1].

The first and most obvious advantage of serverless computing is the absence of infrastructure management. Infrastructure as a Service (IaaS) removed the hassle of physical management of computing infrastructure. With this solution, however, the management of virtual resources is still a time-consuming and non-trivial operation. FaaS removes the management of resources entirely, allowing the users to simply upload functions to the platform, without having to think in terms of VMs or containers.

Then, another benefit of this paradigm is affordable scalability. As the user does not provide more than a function to run, the service appears as a black box to the user and thus the provider is free to optimise the resource utilisation of its infrastructure. For the user, this means that the provider provides scalability automatically by running the number of instances needed for each function. For the provider, this enables multiplexing of the resources as they are shared among the users and managed entirely by the provider. Also, the provider is free to use a more heterogeneous infrastructure, by using older machines that would not be competitive as VMs on the market. This more efficient utilisation of the resources and the heterogeneity of the infrastructure are the main sources of serverless computing affordability. It is however important to note that serverless computing is not the most affordable option for all applications: for some, renting some VMs is cheaper.

Finally, this architecture enables functions to be shared in a marketplace. Developers can indeed share their functions on a marketplace like *AWS Serverless Applications Repository* [8]. This creates competition and with it an incentive to write better code, with better documentation which is of course beneficial for the users. These marketplaces also offer an interesting application for our resource interfaces: having the resource consumption of the function directly available to the user on the marketplace would indeed help them make an informed decision, especially concerning the price per request of the function (as explained in section II-E and III-C).

D. Languages

Serverless computing services offer support for various languages for developing functions. The complete list of the languages supported by the main providers (Google Cloud Functions, AWS Lambda, and Microsoft Azure functions) can be found in table I.

Providers	**Languages**
AWS Lambda	Java, Go, PowerShell, Node.js, C#, Python, Ruby
Google Cloud Functions	Node.js, Python, Go, Java, Ruby, PHP, .NET Core
Microsoft Azure Functions	C#, Javascript (Node.js), F#, Java, Powershell, Python, Typescript (through JS)
Common	Java, Javascript, Python, C#

TABLE I
SUPPORTED LANGUAGES IN SERVERLESS COMPUTING ENVIRONMENTS [9]–[11].

The list of supported languages includes some without a runtime environment but most of them have one. In practice, the most used languages are Python and Javascript, as shown by the survey by Datadog: over 90% of the surveyed companies use Python and Node.JS for their serverless functions [4]. In proportion of functions, 47% of the functions on AWS Lambda are written in Python, and 39% in Node.JS [12]. So Python and Node.JS represent together 86% of all the functions.

It is then clear that most of the functions running on FaaS service rely on a language runtime environment, and therefore our resource interfaces must take that into account.

E. Pricing

An important aspect of serverless computing is its pricing. We will not discuss the absolute price of the service as it is dependent on the provider and changes over time; we will focus on how the total price is computed.

First of all, the FaaS paradigm is built upon the "pay-as-you-go" concept. It is indeed one of the main advantages for the users: they pay for executed functions, which means there is no cost induced by an idle infrastructure. To explain the billing method, we take the example of AWS Lambda but the principles are the same for the other providers.

The basic cost formula is the following:

$$Cost = \sum_{i=1} M \times t_i \times P$$

where N is the number of requests, M is the memory allocated to the function's instance, t_i for $i = 1, ..., N$ is the duration of execution of the ith request and P is the price for 1 GB-second.

We can observe that the price depends on M, which is the amount of memory allocated to the function. In practice, the providers require the user to fix M a priori. The providers indeed do not perform any analysis or prediction for the memory usage of a function: the user defines the amount of memory that will be allocated to each instance of that function and the provider allocates it. If the function exceeds the allocated amount of memory, it is killed. Some providers emit a warning if the amount by which it exceeded the allocated memory is small enough.

Another interesting fact about this is the fact that the computing resource depends on the amount of memory. The performance of the CPU increases with the allocated amount of memory. So, in the case of a function demanding on CPU but not memory, the users must allocate more memory than needed to get better computing performance.

With these two aspects in mind, picking a value for M can become a challenge for the users. It is indeed non-trivial to decide a priori how much memory a function will need, especially given the nature of those functions. They are most likely written in a language with a runtime environment (see section II-D) and calling some external libraries.

To get an idea of the scale, as we are writing this report, the current value of P on AWS Lambda is of $\$1.83334 \times 10^{-5}$.

F. Scheduling & allocation

Providers are discrete about their architectures and do not reveal many details. According to Singhvi et al., [13], current providers do not take the execution time of the functions into account when scheduling them. It is also unclear whether providers prioritise functions and, if they do, how.

Tariq et al. [14] performed measurements on the latency of function requests and concluded that the providers were using the FIFO policy. As the serverless platforms are black boxes, we think these results are only indicative but they show that the scheduling of the functions does not seem to obey complex rules.

G. Applications

In this section, we give some examples of applications of serverless computing.

The benchmarking suite SeBS proposed by Copik et al. in [15] contains examples of serverless applications based on use cases found in the literature. They classify their benchmarking applications into different categories each containing a few applications:

- Web applications
  - **Dynamic HTML**: generates dynamic HTML from a template.
  - **Uploader**: uploads a file from a provided URL to the cloud storage.
- Multimedia
  - **Thumbnailer**: generates a thumbnail for a given image. The input image is stored on the cloud storage and the key is passed in the request.
  - **Video processing**: adds a watermark onto a video and generates a gif file of the result. The input video is stored on the cloud storage and the key is passed in the request.
- Utilities
  - **Compression**: compresses a directory containing files into a .zip file and uploads it to the cloud storage returning the key for the user to download. The input files are on the cloud storage too, the key is passed in the request.
- Inference
  - **Image recognition**: classifies the input image (on the cloud storage with the key passed in the request) using ResNet and Pytorch.
- Scientific
  - **Graph-Pagerank**: generates a graph using the Barabasi method with the size given in the request and computes its PageRank.
  - **Graph-MST**: generates a graph using the Barabasi method with the size given in the request and computes its Minimum Spanning Tree.
  - **Graph-BFS**: generates a graph using the Barabasi method with the size given in the request and performs a Breadth-First-Search on it.
  - **DNA visualisation**: gets a file from the cloud storage corresponding to the given key and computes a visualisation of the DNA sequence it contains. The visualisation is stored back in the cloud storage.

The "cloud storage" mentioned above depends on the chosen service. In the benchmarking suite, it depends on the provider on which the suite is run. For example, it is Amazon S3 when running on AWS Lambda.

We can observe interesting properties of the typical serverless functions from the source code of those. First, they are small: in the order of 10 lines of code per function. Second, they are mostly preparing the inputs (i.e., extracting parameters from the request body, downloading files from the cloud storage, ...) and calling external libraries to process them. Some are using external executables called from Python like *ffmpeg* for the video processing function. The executable is downloaded from the web at the beginning of the function.

Netflix is an example from the industry that uses AWS Lambda in their infrastructure (at least in 2014), to remove polling and use declarative rules instead so that events trigger computations [16]. They use it for the encoding of media files for example. When a studio sends a media file to Netflix, the processing pipeline first chops it up in chunks of 5 minutes. All these chunks are then encoded in parallel. Once it is done, the chunks are merged back together and the media file is deployed for CDN (Content Delivery Network) use. This process is well suited for serverless computing: each of the functions from the chopping to the encoding can be stateless, the instances can be spawned whenever a new file is received, and the number of instances depends on the number of files entering and their length. Also, all stages of this pipeline are triggered by events ("a new file enters", "all chunks are ready", etc...) which is a good fit for this paradigm.

One last interesting point we observe in the functions we analysed is the fact that these functions contain no loop. We, therefore, argue that a significant portion of the serverless functions does not contain loops.

H. Practical experience

We experimented with AWS Lambda and Google Cloud Functions to get a sense of the workflow and do some experiments on memory usage.

To test the memory consumption of the functions and see how the service reacts when it exceeds the limit, we are using a basic function computing the sum of integers from 1 to a given n by first filling an array with all integers from 1 to n and then computing the sum by iterating through that array. The core of the function is the following:

```
int_array = range(n + 1)
sum = 0
for i in int_array:
    sum += i
return sum
```

The memory usage of this piece of code is largely dominated by the array itself and thus depends directly on the value of n. We can see the evolution of memory usage compared to the evolution of the value of n, and this includes all the environment. We can also observe what happens when the value of n becomes too large and the memory usage exceeds the allocated amount.

We are not interested in the absolute value of the amount of memory, but in the trends and what happens at the limit.

Figure 2 shows the memory usage and the execution duration of that function for different values of n. n varies from 0 to 500'000'000 with increments of 200'000, with 40 requests for each. First of all, we can observe that for $n = 0$, the amount of memory used is around 30MB which is because of the Python environment and the container itself. Then, we can see that the used amount of memory exceeds the allocated amount (120MB) for some values of s. We can see in the duration graph that after that point is reached (the used memory exceeds the allocated amount), some requests are dropped, but this is not systematic. This is probably due to the variability of the memory used between runs, and also possibly some AWS Lambda settings allowing some exceeding under some conditions we do not know.

Another interesting fact we observed during our experimentation is that the "out-of-memory error" can be caught in the Python code. The `try-except` construct can then be used to gracefully fail or fall over to another computation path that is less memory intensive.

III. Memory interfaces

In this section, we expose the memory interfaces as we propose them in detail.

A. Representation

We propose to represent the memory interfaces as programs, more specifically Python programs. The interface for a given function would have the same signature as the original except for the return type, which is now the amount of memory that the function would demand if run with the given arguments. The memory demand is defined as the memory the function would request while running (for its heap, stack, global and local state, etc...) but this amount could be different from the actual amount of memory allocated by the OS (due to different mechanism, e.g., page swap). This memory demand amount does not take the Garbage Collection mechanism into account and so it is the sum of all memory the function will allocate, without recollection.

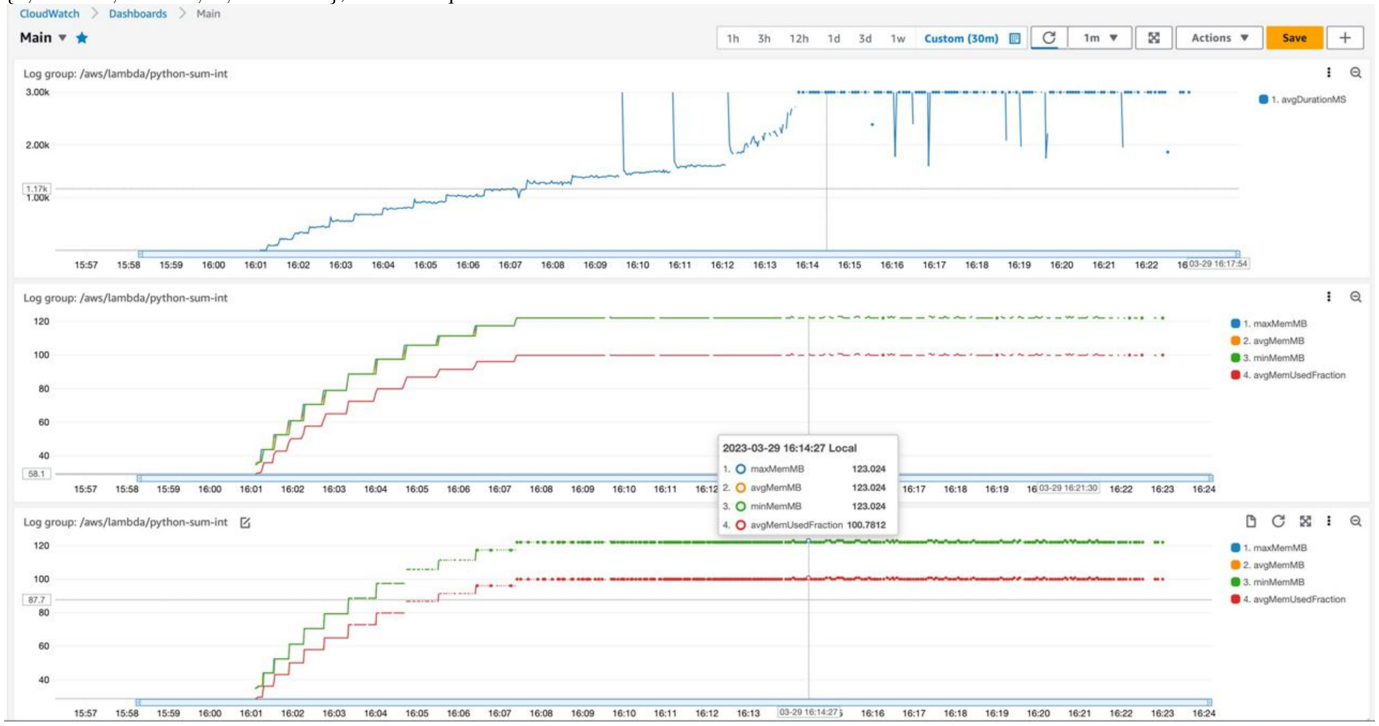
The advantages of the code format for the interfaces are the following. First of all, the interface is readable by both humans and machines. The code can indeed be run easily and developers are used to reading code so if the output code is runnable and small enough, it would fit both cases. Second, the code format is modifiable by a developer with reasonable effort. Third, as the structure of the input function and the interface format are close, it helps the automatic extraction.

B. Extraction

As it is one of the most used programming languages in general and the most used in particular for serverless computing, we develop our memory interfaces for Python. Python is an interpreted language with an intermediate bytecode to which the interpreter dynamically compiles the code before executing it. The standard implementation of Python, *CPython*, is implemented in C and Python. Python typing system is dynamic, which means that variables do not have a fixed type, but it is also strong, which means that an operation called on an object of a wrong type would fail (as opposed to the interpreter trying to make sense of the operation on the given object anyway). A lot of Python libraries are also written in other languages like C or C++ for performance reasons.

The complexity of the environment, the heterogeneity of the libraries' source languages, and the typing system are presenting some challenges to analyse statically the memory consumption of a Python program.

Fig. 2. Memory usage and average duration of calls to the sum of integer function on AWS Lambda. The function was called with $n \in \{0, 200'000, 400'000, ..., 5'000'000\}$, with 40 requests for each n.


CloudWatch > Dashboards > Main
Main
1h 3h 12h 1d 3d 1w Custom (30m) 1m Actions Save
Log group: /aws/lambda/python-sum-int
1. avgDurationMS
Log group: /aws/lambda/python-sum-int
1. maxMemMB
2. avgMemMB
3. minMemMB
4. avgMemUsedFraction
2023-03-29 16:14:27 Local
1. maxMemMB 123.024
2. avgMemMB 123.024
3. minMemMB 123.024
4. avgMemUsedFraction 100.7812
Log group: /aws/lambda/python-sum-int
1. maxMemMB
2. avgMemMB
3. minMemMB
4. avgMemUsedFraction

To tackle these challenges, we propose a hybrid approach to the analysis. We would use static analysis for the frequently modified code (i.e., user-written functions) and we would rely on learning (i.e., based on running traces) for the less frequently modified part (i.e., the environment, the libraries, ...). With this setup, we can leverage static analysis to compute the interface without running the function and tackle challenges posed by the presence of the environment.

Concretely, we would extract the call graph of the analysed function (lazily meaning we do not necessarily have the entire graph computed). Then, we would statically analyse the calls performed by the function and create a symbolic formula of the memory demand conditioned by the input arguments of the function and other variables this amount depends on. At each level of the tree, the analysis continues by calling the interfaces of the called functions recursively. We would stop this analysis at a given depth, returning a symbolic formula for the memory demand.

The execution traces would be used to compute memory usage distribution for function calls. Once these distributions are available, they can be used as concrete values for the symbolic variables in the formula.

By using learning only at a given depth of the graph, we would need traces only for the common code among different applications like the standard library or popular libraries like *numpy* or *scypy*. For these functions, the traces can be obtained by serverless computing providers. The amount of traces would therefore be big and therefore the distributions have a better chance to be accurate.

For this reason, we argue that learning from data does not make sense at the top-level function (i.e., the user's function) as it would require executing the function to get traces and getting enough of them would take a non-negligible amount of time.

Resource and performance interfaces have a major difference from semantic interfaces: the broken abstraction. The resource usage, just like the performance, depends on more parameters than the function arguments. Iyer et al. call them *PCVs* for Performance Critical Variables [17] so, in our work, we will use the name *MCVs* for Memory Critical Variables. With semantic interfaces, the implementation details are abstracted away behind the interface, which makes reasoning about a component easier. With resource usage interfaces, this is not possible anymore, as resource usage heavily depends on the implementation details. We thus need to include these as variables in the symbolic memory demand formula.

The formula can be used as is, without replacing the MCVs with concrete values. This can be sufficient for the developers, as it can give insight into what affects the memory demand and how it affects it (e.g., linearly, exponentially, ...). In the cases where this is not sufficient, we need to take the workload of the program into account. From this workload, probability distributions for these variables can be extracted and we can use these distributions and the formula to compute a distribution for the memory usage. For some MCVs, we might need to collect actual execution traces, for example, if they depend on the underlying systems. For others, the workload known a priori might be sufficient, for example, the amount of data a query to a database will return can be known approximately if the structure of that database is known in advance.

C. Application: serverless computing

We propose different aspects of serverless computing that our memory interfaces would improve.

The first aspect is the configuration and the billing process. As explained in the section II, currently the user has to decide a priori the amount of memory that will be allocated to the function. This amount, in addition to being critical for the correctness of the system, is conditioning the cost of the function. For the user, determining the minimum required amount is a really difficult task, and therefore the temptation of over-provisioning is high, leading to overpriced function executions.

If our memory interface is integrated directly into the upload process of the function by the provider, it would release the user from the burden of deciding the memory amount. The memory demand of the function would automatically be stored as the memory to allocate to an instance and to provide an accurate cost estimate to the user. The user could then accept or rewrite the function to either use less memory to make the cost go down or improve performance if the cost is within her budget. Even if the interface is not integrated by the providers to upload process directly, it can still be used by the users locally (or as a stage of a Continuous Integration (CI) pipeline) to guide the development and to enter an accurate memory provision when uploading to their serverless computing platform.

The second aspect is indicating the memory demand of functions directly in the marketplaces. As explained in the section II, most serverless computing providers offer a marketplace for users to share and download functions. We propose to add the memory interface to the upload pipeline so that a potential user exploring available functions directly gets their memory demand and therefore the cost of these functions' execution. This would enable the developers to offer different versions of the same functionality with different performance-cost trade-off choices.

IV. Related work

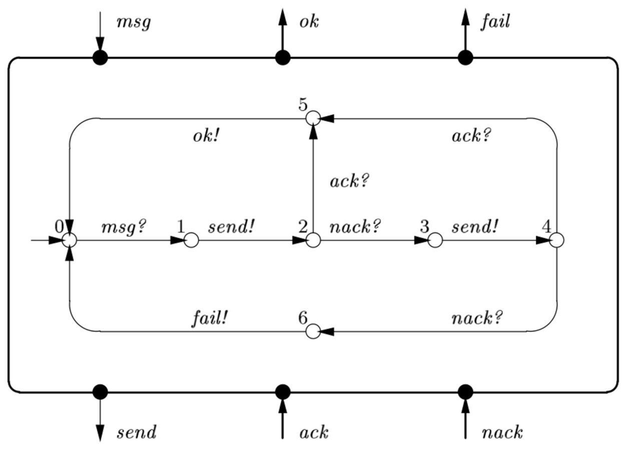
Interfaces for systems, and resource interfaces in particular, have been studied notably by T. Henzinger and L. de Alfaro, later joined by A. Chakrabarti.

In Interface Automata [18], they propose a *modeling language* to describe system components. They propose to use these models to validate, verify, and document system components. They also propose a way of combining these interfaces and a definition of compatibility for components, based on the combination of their interfaces.

The proposed modeling describes the temporal externally observable behaviours of a component (i.e., what other components it calls and when it either gets or sends information). These models are represented as automata, with the externally observable behaviours as transition labels. There are two types of transitions: either the component receives a new input and thus transitions to a new state, or the transition is taken directly and this symbolises the component sending an event to the outside environment.

Let us take an example. A component has a method `msg` which sends a message and returns either `ok` or `fail`. This component internally uses a communication channel that offers a method `send` which can return either `ack` or `nack`. The component tries to send the message and, if it receives a `nack`, retries once and then returns the result. The interface automaton for that component is represented in figure 3. If no transition labeled with a particular input goes out of a state, this input is *illegal* if the automaton is in that state. Illegal inputs represent the assumption about the environment, i.e., the component assumes that the environment would not send an illegal input.

Fig. 3. Interface automaton for the component sending a message with retry [18].



The composition of two components' interfaces is defined as the product of their automata $C_1 \otimes C_2$ for two given automata C_1 and C_2.

The definition of compatibility of 2 components that Henzinger and de Alfaro propose is an *optimistic compatibility*. This means that 2 components are considered compatible if there exists an environment that makes them work together, i.e., which does not send illegal inputs. This is opposed to a *pessimistic compatibility* which would state that 2 components are considered compatible only if all possible environments would send only legal inputs. In the automaton representation, this means that the combined automaton can contain illegal states. The environment would make these states unreachable by not sending illegal commands. In this report, we omit the formal definition as well as the algorithm used to compute and prove this compatibility.

The justification for such an optimistic definition is that the developers can use some components together in an environment known to be helpful. There is therefore no reason to declare these components as incompatible.

A. Chakrabarti et al. propose to apply these interface automata to resource usage in Resource Interfaces [19]. The formalism and the modeling language are similar but are now used to model the resource requirements of a given component. In this case, resources they consider include energy, memory, network bandwidth, and more.

They propose some algebra to work on those automata. Examples of questions that these interfaces and algebra can answer are: "Can two given components work together without exceeding the energy stored in the battery?" or "Can two given components work together without exceeding the peak power available on the device?".

To answer those questions, the interfaces are seen as a *game* between two players: *input* and *output*. Intuitively, at each state the player *input* chooses a combination of input variables that respects the input assumptions. Then, the player *ouput* chooses a combination of the output variables that respects the output conditions (i.e., the interface). This produces a *run*, i.e., a sequence $q_0, (v_0^i, v_0^o), q_1, (v_1^i, v_1^o)$ where q_k is a state of the automaton and v_k^i and v_k^o are the input and output variables combination. Properties and invariants can then be computed on this *run*. In this report, we omit the formalism and the algorithms.

An important point to note about these interfaces is that they are extracted by hand. This means that such an interface is a model of a component, written by hand by the developer of the system. Interesting and useful properties can be computed with them but we argue that for our use case and for verification, it is better to extract these interfaces automatically and statically. This improves the ease of use and the trust we can have in these interfaces.

Other pieces of work explore memory usage static analysis.

Kim et al. [20] propose a technique to statically analyse a C program to determine its dynamic memory usage to find vulnerabilities. The first step of their analysis is to extract the flow of memory allocation at the source code level. Each variable receives a size (depending on its type) and a life duration (i.e., the scope for local variables and between `malloc` and `free` calls for pointers). Then they propose that existing techniques like Call Flow analysis can be used. The second step is the static prediction of the dynamic memory usage by analysing this flow extracted in the first step.

Braberman et al. [21] estimate an upper bound of the dynamic memory usage of Java programs. They build a tool to compute a formula representing the quantity of memory allocated by a method as a function of its parameters. Their experimental results show that their tool estimates precisely the amount of memory in most of the cases and is a close approximation in the other cases, between 0% and 5% error with 4 methods for which the error is 10%, 21.1%, 22.22% and 36.2% respectively. This does not take the Garbage Collector (GC) into account and so the memory amount is the sum of all allocated objects size, without taking into account the fact that the GC (Garbage Collector) is recovering the memory of inaccessible ones.

Albert et al. [22] propose a way to give better estimations, taking the GC activity into account. They propose to parameterise the GC to give an estimation closer to reality, without assuming a particular GC model. The parameterisation is done on the lifetime of the objects (i.e. when they are eligible for collection). The produced estimation can then be used in different scenarios and is sound at least for the following two: when the GC collects objects as soon as they become available, and when the GC collects eligible objects when the heap size reaches a certain limit (which is then the upper bound of the memory usage that the tool would output). The first scenario is interesting from a theoretical point of view, while the second is realistic.

V. Current prototype

In this section, we expose the prototype we developed and how it works.

A. Scalpel framework

To implement our prototype, we use the Scalpel framework [23][1]. Scalpel is a framework built to serve as a basis for static analysis tools for Python. It offers different modules to manipulate or analyse Python code. The list of modules includes a type inference module, an alias analysis module, a module to perform constant propagation, and a call graph generation module for example.

In this project, we use the control-flow graph (CFG) construction module which can extract the control flow of a Python program. For example, the CFG of the function `handler` shown in figure 4 is shown in figure 5. The produced CFG is a Directed Graph in which each vertex contains a block of the code with its statements, link(s) to the next block(s), and the condition of this(these) branch(es) is any.

Fig. 4. A example Python function.

```python
def handler(event: dict[str, Any]):
    x = f()
    y = g()
    if x > 0:
        z = h()
    else:
        z = i()
    if y > 0:
        w = j()
    else:
        w = k()
    return w + z
```

While developing our prototype, we discovered a bug in the library: when called on a piece of code containing a `try-except-finally` construct, the code in the `finally` block and after was ignored. For example, the piece of code in figure 6 was generating the CFG shown in figure 7.

As we can see, the `finally` block does not appear in the CFG, nor does the `return` statement.

We opened a PR[2] to correct the bug and the patch will be added to the next release.

B. Sympy

To represent the memory demand as a symbolic formula, we use the Sympy library[3]. This library is a symbolic mathematics library. In our case, we only need to create free variables and

[1]https://github.com/SMAT-Lab/Scalpel

[2]https://github.com/SMAT-Lab/Scalpel/issues/98

[3]https://www.sympy.org/en/index.html

Fig. 5. CFG of the `handler` function in figure 4.

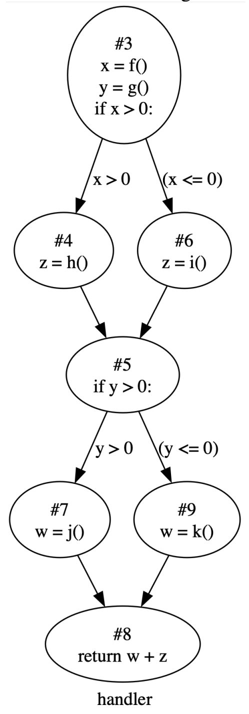
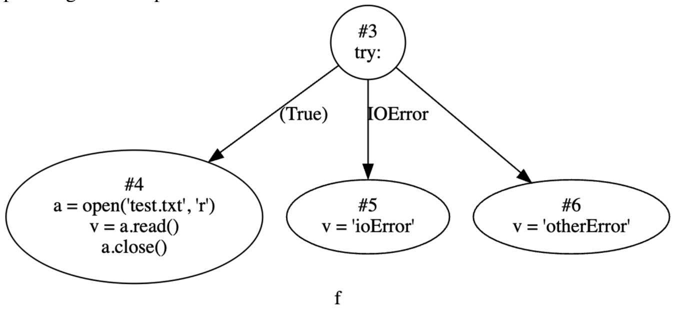


Fig. 6. A example Python function with a `try-except-finally` construct.

```
def f():
    try:
        a = open("test.txt", "r")
        v = a.read()
        a.close()
    except IOError:
        v = "ioError"
    except:
        v = "otherError"
    finally:
        a = 123
    return v + str(a)
```

Fig. 7. CFG of a function with `try-except-finally` construct before patching the Scalpel module.



create a formula with these variables and basic operations like additions.

The resulting formula can be later analysed by a processing tool. The library indeed offers various operations to be performed on formulas like getting the list of free variables, computing derivatives, and more. For example, computing the derivative with respect to a free variable (i.e., a function called in the analysed function) can be used to determine how calling this function affects the final memory amount (i.e., linearly, quadratically, ...).

C. Prototype

We develop a prototype that can extract a memory interface for a given function, exploring the call graph with a maximum depth of 1 (as explained in the section III-B). Given a Python function, our tool constructs another Python function which is the memory interface itself. This function returns a `sympy` formula which is a sum of free variables, representing the functions called in the original function. The interface function has the same control-flow structure as the original program so it returns a different formula for each path. In some cases, the function has to take more arguments than the original function because of the MCVs, as explained in the section III-B. Indeed, the branch conditions of the original functions can depend on variables different than the arguments of the function (e.g., local computations, values returned by some function calls, ...). So for the interface to be runnable, we need to take these variables as arguments. The interface function then takes as arguments the same arguments as the original function along with all variables that appear in a branch condition (removing duplicates).

We now go through the extraction step by step.

First of all, the tool extracts the CFG of the function using Scalpel. Then it extracts from the CFG a recursive structure of the blocks. We indeed need to recover the program structure in this form to be able to correctly recreate the indentation in the new program. For example, the CFG in the figure 5 would have the structure `[3, ([4], [6]), 5, ([7], [9]), 8]`. The structure is a list of block ids with a tuple for each branch. The tuple contains the list corresponding to the branch(es), recursively defined. For now, we do not support CFG in which a block has more than 2 exits or in which there are loops.

Then the tool constructs a new Python AST object for the interface function. This construction is done recursively. For each block, the tool extracts the functions called in the block. If there are multiple levels of calls (i.e., calls of the form `f()()` with two or more levels), the tool adds one variable per level (e.g., for `f()()` two variables are added, `'f(...)'` and `'f(...)(...)'`). The tool then adds a statement for each of these variables, adding them to an accumulator variable.

Once the AST for the body of the interface is ready, the tool extracts the list of variables appearing in branch conditions and constructs a function definition AST node.

The new function is then written to a file.

As an example, the interface for the function `handler` in figure 4 can be seen in figure 8.

Fig. 8. The memory interface of the function `handler` in figure 4.

```
def handler_interface(event, x, y):
    mem_demand = 0
    mem_demand += sp.symbols('f(...)')
    mem_demand += sp.symbols('g(...)')
    if x > 0:
        mem_demand += sp.symbols('h(...)')
    else:
        mem_demand += sp.symbols('i(...)')
    if y > 0:
        mem_demand += sp.symbols('j(...)')
    else:
        mem_demand += sp.symbols('k(...)')
    return mem_demand
```

VI. Limitations

The prototype tool that we propose is not useful in itself at that stage. It represents a starting point to develop such a tool. It poses the basis such as the structure, the methodology, and the libraries to use.

It, therefore, has several limitations. First of all, and most importantly, it explores only one depth of the call graph, which reduces its practical usage. Then, it does not perform analysis on the MCVs and simply adds them as arguments. However, in some cases, some MCVs can be computed from the input arguments. For example, they could correspond to values extracted from a dictionary received as arguments. Also, it ignores completely the arguments passed to the functions called. Finally, it does not add any learning capabilities, therefore, only does the first part of the analysis, i.e., constructing the symbolic formula containing the function called.

VII. Conclusion

In this work, we explored the idea of an interface to represent the memory usage of a function. We explored the use cases of such an interface, how it should be represented, and how to extract it. To achieve that, we review the literature on this topic, and the available commercial serverless computing services.

We then developed a proof of concept that is meant to be used as a starting point to develop a tool for extracting such interfaces.

References

[1] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless Computing: A Survey of Opportunities, Challenges, and Applications," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 239:1–239:32, Nov. 2022.

[2] S. Watts, "State of Serverless Today," https://www.bmc.com/blogs/state-of-serverless/.

[3] "Future of rules without webtask.io," https://community.auth0.com/t/future-of-rules-without-webtask-io/48554, Aug. 2020.

[4] Datadog, "The State of Serverless," https://www.datadoghq.com/state-of-serverless/.

[5] "AWS Lambda – Run Code in the Cloud — AWS News Blog," https://aws.amazon.com/blogs/aws/run-code-cloud/, Nov. 2014.

[6] M. Azure, "Announcing general availability of Azure Functions — Azure Blog — Microsoft Azure," https://azure.microsoft.com/en-us/blog/announcing-general-availability-of-azure-functions/, Nov. 2016.

[7] F. Lardinois, "Google's Cloud Functions serverless platform is now generally available," Jul. 2018.

[8] "AWS Serverless Application Repository - Amazon Web Services," https://aws.amazon.com/serverless/serverlessrepo/.

[9] "AWS Lambda – FAQs," https://aws.amazon.com/lambda/faqs/.

[10] "Cloud Functions execution environment — Cloud Functions Documentation," https://cloud.google.com/functions/docs/concepts/execution-environment.

[11] ggailey777, "Supported languages in Azure Functions," https://learn.microsoft.com/en-us/azure/azure-functions/supported-languages, Oct. 2021.

[12] B. G. Rama and 02/04/2020, "Report: AWS Lambda Popular Among Enterprises, Container Users -," https://awsinsider.net/articles/2020/02/04/aws-lambda-usage-profile.aspx.

[13] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A Scalable Low-Latency Serverless Platform," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 138–152.

[14] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling quality-of-service in serverless computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 311–327.

[15] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "SeBS: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 64–78.

[16] "Netflix & AWS Lambda Case Study," https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/.

[17] R. Iyer, K. Argyraki, and G. Candea, "Performance Interfaces for Network Functions," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 567–584.

[18] L. de Alfaro and T. A. Henzinger, "Interface automata," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 109–120, Sep. 2001.

[19] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga, "Resource Interfaces," in *Embedded Software*, ser. Lecture Notes in Computer Science, R. Alur and I. Lee, Eds. Berlin, Heidelberg: Springer, 2003, pp. 117–133.

[20] S. Kim and J. Ryou, "Source Code Analysis for Static Prediction of Dynamic Memory Usage," in *2019 International Conference on Platform Technology and Service (PlatCon)*, Jan. 2019, pp. 1–4.

[21] V. Braberman, D. Garbervetsky, and S. Yovine, "A Static Analysis for Synthesizing Paramet- ric Specifications of Dynamic Memory Consumption," *Journal of Object Technology*, vol. 5, pp. 31–58, Jan. 2006.

[22] E. Albert, S. Genaim, and M. Gómez-Zamalloa, "Parametric inference of memory requirements for garbage collected languages," *ACM SIGPLAN Notices*, vol. 45, no. 8, pp. 121–130, Jun. 2010.

[23] L. Li, J. Wang, and H. Quan, "Scalpel: The Python Static Analysis Framework," Feb. 2022.