An Internal DSL for Long-Running Processes in Python

Emil Platz, 201206048

Master's Thesis, Computer Science

January 2018

Advisor: Henrik Bærbak Christensen

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

I'm not interested in having an orchestra sound like itself. I want it to sound like the composer.
- Leonard Bernstein

Abstract

uOrchestrate is an integral system used by uDeploy (Uber's internal interface to their shared compute platform) responsible for Web Service Orchestration according to user-defined strategies (Workflows) inside Uber's infrastructure. Workflows are described using Workflow Language v1.0 (WL1), an internally developed Domain-Specific Language (DSL). With the increasing complexity of orchestration and Workflows within Uber, uOrchestrate suffers from lack of usability due to design choices based on an idea of less complexity. Therefore, I propose Workflow Language v2.0 (WL2), a WL1-interoperable DSL that tries to solve WL1's issues by replacing the existing syntax with an internal DSL written in Python with the goal of better usability. An evaluation of WL2 shows that it indeed does improve on the established issues of WL1.

Acknowledgements

The work presented in this thesis was done in Uber Technologies Inc., particularly at the Aarhus office, a remote engineering office focused on scaling Uber's core infrastructure[18]. This thesis focuses on a Domain-Specific Language (DSL) used by uOrchestrate, an integral system in Uber's infrastructure. uOrchestrate was originally designed and implemented by Claus Thrane, Mathias Schwarz and Kristian Lassen.

I would like to thank my thesis advisor Henrik Bærbak Christensen, both for keeping me on the right track and providing great advice, but also for being one of the factors that made this collaboration a reality. I would also like to thank Claus Thrane for his great mentorship, not only in regards to my work at Uber Technologies Inc., but also in regards to this thesis. Another thanks goes out to Mathias Schwarz and Sebastian Kallestrup Bogner for proofreading the thesis and providing great feedback, and to Mathias Bak Bertelsen for helping out in the 11th hour.

This thesis wouldn't have been possible without the help of Steffen Grarup and Gustav Wibling, who, along with Claus, put faith in the work and paved the way for this collaboration, so a huge thanks to you and to the rest of the Uber Aarhus office.

Finally, I would like to thank my girlfriend, Nina, for her endless love and support the last 5 months.

Emil Platz,
Aarhus, January 15, 2018.

Contents

Chapter 1

Introduction

In this thesis, we'll be studying a DSL used by several of Uber's infrastructure services to coordinate web services, with the intention of proposing a substantial redesign to address a number of short-comings. Before we dive into the details, let's try to understand the context of Uber's architecture and one of the main systems consuming this DSL, the internal deployment system uDeploy.

Uber is one of the original Unicorns (a startup company valued at over $1 billion). It is a technology company, that provides transportation services, through state-of-the-art technology. Uber has grown at incredible rates, and while this is generally a good thing, it can be extremely hard to keep up with explosive growth - especially since Uber's core products are all about being stable and available at all times. If the infrastructure can't handle the growth, it affects stability and availability, which will in turn affects growth. Limit growth to relieve infrastructure is obviously not ideal. The following will take a look at the background of Uber's size, the problems that arises with it and the systems Uber uses to solve these problems.

1.1 Growing a Technological Infrastructure

Originally, Uber had a Monolithic Architecture (a relatively centralized system[25, p. 2]) with a single PostgreSQL instance handling all their data[23]. It kept up with business, but as Uber grew and the traffic with it, this monolith became a limitation. The main application, although centralized code-wise, could be distributed out onto many application servers, but it was a single point of failure code-wise, since introducing an error would affect all instances, hurting availability[25, p. 322]. The PostgreSQL instance, on the other hand, ran on a single machine[23], making it a big concern hardware-wise, since it physically was running out of space to put in more storage. It stored all the trip data, and since number of daily trips doubled often, it was going to run out of disk space at some point. It was also having a hard time keeping up with the number of requests from an IO perspective (reads/writes). Another issue came from introducing new functionality to the lone repository. Adding new features, fixing bugs or resolving technical debt started requiring tribal knowledge, since the codebase had become too complex. Also, in order to make changes take effect (no matter the size or importance), a complete redeployment of the codebase was required.

Maintaining and operating a company's technological infrastracture under rapid growth is a notoriously hard situation to manage, only seen previously at a few companies (such as Google and Facebook), that have since become industry leaders. It does however also open up for the possibility of Uber joining the ranks of these

industry leaders, should they play their cards right. In order to get there, Uber needs a stable platform. Scalability, reliability, availability and extensibility are four important concepts in Distributed Systems theory[25], that are all needed in order to have a stable distributed system. Here's a quick walkthrough of the requirements to give an overview and set the stage for why they are hard to get right (especially with a Monolithic Architecture).

Scalability represents the flexibility in the size and capacity of a system[25, p. 9]. This is both in matters of storage and processing power. If the system is handling 100,000 requests per second during the weekdays, but it peaks at 500,000 during the weekend, how is this done without wasting capacity in the weekdays? How is growth handled (i.e. add hosts or capacity) when business is expanding to a new city every week? For reference, Uber currently has tens of thousands of hosts, and over 10x of what it had 3 years ago. With the Monolithic Architecture, the business logic could be distributed across multiple application services, but the PostgreSQL instance could not (atleast not without a larger distribution effort).

Reliability represents the probability that a system behaves as intended[25, p. 322]. This generally becomes harder the larger the system is, since program flow becomes harder to keep track of. The Monolithic Architecture had an enourmous code-base with a lot of different functionality. The Monolithic Architecture also meant a single point of failure, which isn't very reliable.

Availability represents the system uptime[25, p. 322]. If a system has a availability requirement of 99%, it is allowed to be unreachable less than 15 minutes per day. For reference, most of Uber's systems have an availability requirement of 4 9's (99.99%), which is less than 9 seconds per day (or slightly below an hour per year), and many critical systems have an even higher requirement. What unreachable implies depends on the business impact of the system. To be more precise, any system will have a Service-Level Agreement (SLA) that describes exactly what requirements are set around the system. It can be any kind of unreachable, i.e. not only during failures, but also during upgrades. As mentioned before, the Monolithic Architecture means a single point of failure, which results in bad availability.

Extensibility represents the possibility for future growth[25, p. 8]. A Monolithic Architecture, as mentioned, innately has bad extensibility. Extending it requires an update of a very large system, which can cause downtime.

Table 1.1 describes where the requirements conflict:

{row} is hard\When having {column}	Scalability	Reliability	Availability	Extensibility
Scalability		Scaling up and down in size means that service instances needs to be started or shut down which can be hard due to distributed state	Adding or removing hosts needs precise coordination and can result in downtime due to synchronization	Knowing what is critical and what isn't is crucial to knowing how to scale up or down and a very extensible architecture can complicate this
Reliability	When the infrastructure is scalable, it is very hard to mitigate issues, since every instance of a service needs to be updated, rolled back or paused		Availability and reliability in general don't conflict since high amount of one results in high amount of the other	When systems are highly extensible there usually ends up being a lot of dependencies. If a service relies on dependencies that are unreliable, it also becomes unreliable
Availability	Synchronization of systems during scaling can easily end up causing downtime	Availability and reliability in general don't conflict since high amount of one results in high amount of the other		Since reliability is hard to achieve with high extensibility, availability is also hard to achieve
Extensibility	It is hard to make an extensible system with dependencies since scaling a service could suffer from bottleneck dependencies. Scalability requires alignment and homogeneity, which conflicts with extensibility	It is hard to make an extensible system when fault tolerance is a requirement	It is hard to extend systems with high availability since extending often means bringing down the existing system to plug the extension into	

Table 1.1: Conflicts between different requirements

1.1.1 Web Services

In order to handle the previously mentioned requirements, the Monolithic Architecture had to go. A completely distributed and decentralized system, such as a Service-Oriented Architecture[20] (SOA), would require a big migration and introduce other challenges and complexity, such as obviousness[16] (making interaction between distributed resources obvious), but given the nature of Uber's growth, it was the only way forward. This way new functionality could be added modularly as a web service and be updated independently instead of being part of the primary code base.

Uber now has an SOA[16] (specifically a microservice architecture[14]) with thousands of different microservices, each with many instances, running across Uber's private datacenters. In terms of the previous requirements, a microservice architecture opens up to other challenges.

Scalability isn't a question of having enough application servers anymore. Instead the responsibility has shifted towards services and their owners instead, which can be hard to manage.

Reliability can be hard to get right. States can get messed up. Debugging suddenly happens across different services, processes or even machines, instead of being contained within a single codebase.

Availability is now on a per service basis as well.

In regards to extensibility, a microservice architecture can also be extremely hard to manage when it grows out of control. Microservices and their dependencies can be hard to keep track of and therefore hard to decommission. Uber has around thousands of different microservices, but only an estimated 50% of them are actively used. Initiatives to shut them down are hard, since nobody knows the full dependency graph.

Even though managing an SOA can be hard, it is still a huge improvement from the Monolithic Architecture. Uber is still continuously expanding its business with great velocity, which would have been slowed down greatly, if the Monolithic Architecture hadn't been replaced.

1.2 Reliable Microservice Deployment

With a SOA and the different requirements it introduces, letting a developer or team manage their own service from end-to-end (meaning server provisioning, building an image, installing the image, monitoring) quickly becomes infeasible. First of all, time is wasted in the process of doing all of these steps manually. Secondly, manual and cumbersome work is more prone to errors, due to the human element. Thirdly, the systems will quickly become heterogeneous (unless concrete policies exist, but even with them, they will be hard to enforce), which complicates troubleshooting, error handling and outage mitigation. Consistency is key. Lastly, it is insecure, since service owners have to have direct production access. To get rid of the human element and all of these issues, a Shared Compute Platform (SCP, Grid Computing[25, p. 17-20]) is introduced in order to automate and align the processes. A SCP lets a user use resources without having any direct access to them. It also provides efficiency, since multiple services with different owners can live on the same resource independently, allowing for CPU scavenging. It requires alignment between service owners and the platform, since a SCP innately has a high degree of heterogeneity (it's what makes it different from traditional clusters) - although not as high as on an unmanaged SOA. Service management is basically made scalable, but at the price of extensibility, since alignment usually means restrictions. Scalability here is, however, extremely important, so it gets priority, and at the end of the day,

extensibility is still preserved with certain restrictions. Automation also provides reliability and availability, since it is less error-prone, however this is only under the assumption that the automation works, since errors can also happen here.

1.2.1 uDeploy

uDeploy[24] is Uber's interface to their SCP. It will be used the primary example of the use-case of the DSL throughout this thesis. It was created to relieve the service owners from worrying about single hosts and cluster management, by instead providing an interface to the infrastructure as a resource pool, exposing generic resources in certain locations. uDeploy itself consists of many microservices[24] (uBuild, uDeploy Aggregator etc.), that has to do work in a specific order (build, upgrade etc.). This makes uDeploy a complex service[25, p. 553]. uDeploy basically makes it possible to manage a service's life cycle, i.e. deployment, rollback, upgrade etc. Depending on the policy, an operation (e.g. a rollout or a rollback) can finish within minutes, without any manual work from the user except for pushing a button. Which policy depends on a lot of things, but in general, Uber categorizes services by tiers depending on their business impact (from 0 to 5, 0 being mission critical services) with different policies applying to different tiers.

Deployment Policies

Beyer et al. argue that running reliable services require reliable release processes[8]. To give an idea of how management of a service in uDeploy can vary, here are a few examples of different deployment policies that uDeploy offers, in order to make releasing reliable.

Incremental Deployment is the most common deployment policy at Uber, where a service will be deployed incrementally in zones of increasing size. Say deployment happens in 4 increments, then the first bump could represent a Canary release[22] of 5% of the desired instances. Once faith in the iteration has been established (through e.g. monitoring or smoke tests), the release will proceed to the next bump which could be 15% (resulting in deployment to a total of 20%), then 30% (to reach 50% in total) and at last the remaining 50%. This means an error in the service being upgraded will only affect a small fraction of the service's instances. Even in situations where a zone can affect a whole failure domain (isolated area that will be compromised), incremental deployment will only deploy to one failure domain at a time. It's mandatory for tier 0 and 1 services (with very strict SLA's) to upgrade using Incremental Deployment, since their business impact is critical to Uber. Figure 1.1 shows this policy as a flow diagram (or state machine).

Blue Green Deployment is another well-known deployment policy, where instances are divided into two deployments, a blue and a green one (figuratively speaking). At all times, only one of the deployment is serving live traffic. Meanwhile, the other one is either upgrading or acting as a staging environment to give confidence in the release. This one isn't used actively at Uber, but would be very easy to implement using uDeploy. A similar situation with the two deployments also sort of happens with Incremental Deployment, since it also only deploys to one datacenter at a time and then monitors that for a while. This means that Uber can always fall back to the other datacenter during the monitoring if the release blows up the entire stack. Uber does, however, serve live traffic out of all datacenters by default.

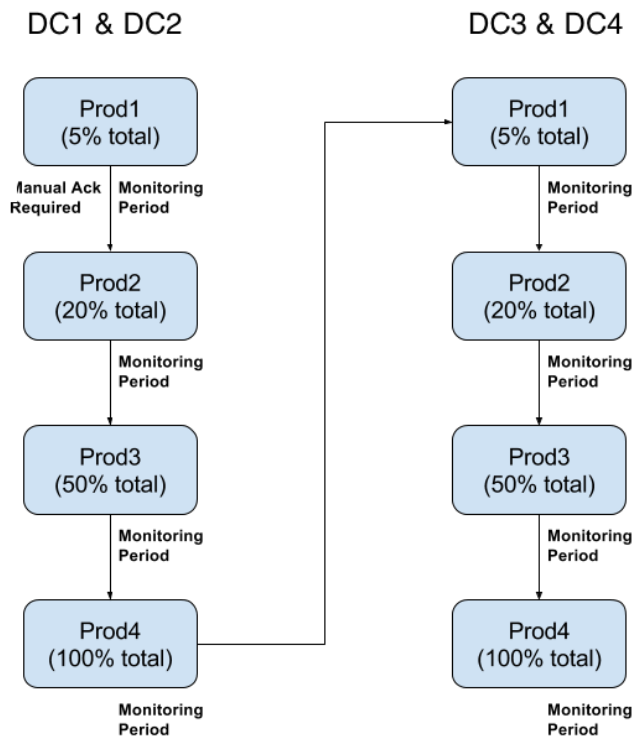

Figure 1.1: Incremental Deployment on two sets of Data Centers

Other types of deployment policies are available for services, depending on their requirements. E.g. Uber has emergency policies, where the different safe-guards are deprioritized in order to speed up the process.

1.3 Web Service Orchestration

As mentioned in section 1.2.1, uDeploy is a complex service. It is responsible for transactional 'operations' implemented by invoking multiple web services (implementing the upgrade mechanisms, build system etc.) in the right order. Web Service Orchestration (Tanenbaum calls it Web Services Coordination[25, p. 552-554]) is a way to handle the coordination needed, when calling microservices in a specific order according to some policy (Tanenbaum calls it a Coordination Protocol, although it will henceforth be referred to as a policy). Deployment Policies could be examples of such policies. A mental model of a policy could for example be a Petri net[21] (a graph structure), where places will be states and transitions will be steps. Figure 1.2 shows a simplified Petri net (with transitions being rectangular and places being oval) modelling a specific policy. The policy first builds an image of the service and then iteratively deploys it. For uDeploy to work, it needs an orchestrator to coordinate the steps needed to execute the policy.

1.3.1 uOrchestrate

uOrchestrate is an orchestrator used by uDeploy. Basically, it coordinates what needs to happen according to a Workflow (a policy, Uber's own Coordination Pro-

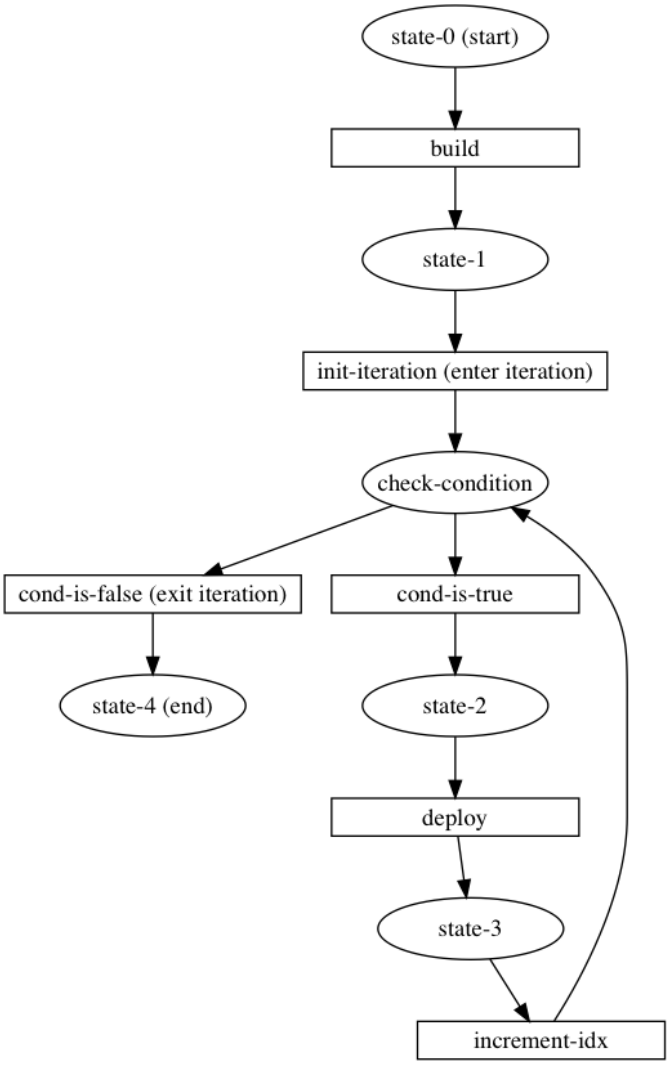

Figure 1.2: A mental model of a policy modelled as a Petri net

tocol) provided by a user. Figure 1.3 describes the guarantees uOrchestrate provide. A Workflow consists of a number of steps (or Actions), governed by a control-flow. An Action represents an RPC to an arbitrary web service. The result of the call is read back into the state of the Workflow to be available during the rest of the execution. When a Workflow is activated (either manually or invoked by another service), uOrchestrate will make sure it gets executed, no matter how long it might take (which can be a while, considering lengthy monitoring periods). Distributed workers continuously and repeatedly poll a shared queue for the next step (along with its state), that is ready to be executed. Once a worker retrieves a step, it executes the step and puts the next step (which depends on the result of the previous step) back in the queue, meaning Workflows can be run concurrently. In figure 1.2, transitions are steps and places are states. uOrchestrate can be scaled according to the load, due to its design. uOrchestrate is general-purpose, in the sense that it has lot of different use cases and is used in many different internal systems, but it was originally developed for uDeploy, where it handles the coordination of steps needed for specific task. Technically, a Workflow is modelled by a Petri net (making the mental and the actual model converge) and executed by a Petri net simulator.

- Reliability: Guarantee that the Workflow runs to the end
- Persistence: Do not lose the Workflow state, even if a machine crashes
- Observability: Report progression and current state
- Reusability: Primitives for a service should be usable in multiple workflows

Figure 1.3: Guarantees provided by uOrchestrate

There are different tasks when managing a service's life cycle and different types of services and deployment policies and neither remain static as Uber's situation changes. The Cartesian product results in a large number of different policies needed. It simply isn't scalable to have a set amount of policies, meaning it should be possible to create, import and execute user-defined Workflows decoupled from deployment of uOrchestrate (meaning all of uOrchestrate shouldn't have to be upgraded everytime a Workflow needs to be added).

Currently, there exists more than 300 different policies, divided into bundles with specific areas of responsibility, all of them hand-tailored to the specific task they are meant to fulfill. Some of them are simple and short, others consist of complex graphs with a lot of data. uOrchestrate has many use cases and it isn't only used by uDeploy anymore. Other potential users, however, hesitate to embrace it, primarily because it's complex to use.

1.4 Workflow Language

In order to let users of uOrchestrate define their own Workflows (such as the uDeploy developers defining Deployment Policies), a Domain-Specific Language[13] (DSL) was introduced. A DSL is a special-purpose programming language focused completely on its domain. This allows the creator of the language to model it precisely after the problem it needs to solve, in order to be able to express it more clearly. This can be done by following particular programming language paradigms, that fits well with the model. The user can then use the DSL to solve problems in the domain. WL1's documentation can be found in appendix B. Figure 1.4 shows a sample Workflow that implements the simplified Petri net in figure 1.2. It has name and description as meta-data (lines 1-2), takes two typed arguments with description meta-data (lines 3-9) and returns a map with the key build_size pointing at the data stored in `build_ref['size']` (lines 10-11). The *workflow* (line 12) key describes the program flow, which in this case consists of a single *build* invocation (line 13) and an iteration (starts at line 17) over the list *clusters* that in each loop invokes *deploy* (line 21).

1.4.1 Philosophy

When the original DSL (referred to as WL1) was defined, it was inspired by Petri nets (as mentioned in 1.3.1) and a number of different programming language paradigms. The language has inspirations from both data-driven programming, declarative programming and imperative programming. It is data-driven in the sense, that the data defines the flow of the program. When running, it exposes its entire state through endpoints, giving the outside world a way to define an interface to the Workflow. It is declarative in the sense that the goal is to declare the problem instead of declaring how to solve it. This was especially true in early development where Workflows were thought to be mostly sequential pipelines without

```
name: deploy_service_to_cluster
description: Deploys service to cluster
input:
  - name: service
    type: service_id
    description: service_id of service
  - name: clusters
    type: list
    description: clusters to deploy service to
output:
  - build_size: build_ref['size']
workflow:
  - build:
      name: build_ref
      args:
        service: service
  - iterate:
      for: cluster
      in: clusters
      sequential:
        - deploy:
            args:
              build_ref: build_ref
              cluster: cluster
```

Figure 1.4: Workflow sample

any control flow. It has imperative and procedural aspects, when looking at more complex Workflows consisting of subroutines or control flow statements. This is the philosophy in regards to defining program flow. In regards to actually modelling the flow, we also use Petri nets. Petri nets provide us with an exact mathematical definition and theory surrounding process analysis, which we can derive guarantees from. Hamadi et al.[17] defined an algebra with corresponding formal semantics for Web Service Orchestration, that directly translates into Petri nets. Petri nets also support iteration and concurrency, which is a big part of WL1's power. It provides the user with data transformation through Python expressions. Python was primarily chosen due to the expressiveness of single expressions. It is very powerful when it comes to transforming data objects, but it also restricts the user to one-line expressions, enforcing conciseness and avoiding a polluted scope. When discussing DSLs, there is a clear distinction between internal and external DSLs[13]. Fowler describes an Internal DSL as a DSL hosted by another programming language, meaning the DSL is written using the other programming languages syntax (or atleast some subset of it) and an External DSL as a DSL with its own syntax and parser. Depending on perspective, WL1, in a sense, is a either a hybrid (meaning it's hosted within YAML, but in a way where it has its own syntax and can be evaluated at runtime) or neither of them (meaning it doesn't have its own parser and its host language isn't a programming language - just the syntax).

1.4.2 Workflow Language Compiler

Figure 1.5 shows a simplified version of the current compilation pipeline for WL1. YAML is loaded into a Python dictionary that serves as an intermediate data struc-

ture, before further compilation. It is then parsed into an AST that in turn is used to generate a Petri net. The Petri net is executed in a runtime environment hosted by uOrchestrate (basically making it a Petri net interpreter).

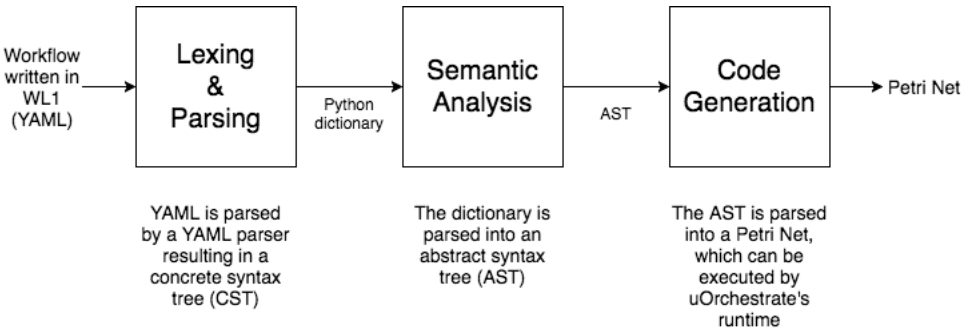

Figure 1.5: uOrchestrate's Workflow compiler

1.4.3 Syntax and Semantics

WL1's syntax is a subset of YAML with embedded Python expressions. YAML was chosen since it fits well together with the data-driven and declarative philosophy. It is great for structuring data and it has a declarative feel to it. WL1 makes it possible to use different strategies when managing a service in uDeploy by executing different Workflows according to the strategy. Figure 1.6 shows the syntactic components of WL1 (except for phases) as a sort of pseudo-Backus-Naur form (BNF).

WL1 consists primarily of Workflows and Actions (statements). Actions, as mentioned, are RPCs and Workflows are small programs that define program flow. Program flow is constructed using different mechanisms of the language, such as invocations (calls) and control structures. Workflows and Actions can be invoked by other Workflows. In terms of control structure WL1 has sequential and parallel execution of statements, iterative or parallel loops and conditional branching. In order to handle data, Python expressions can be used in certain situations to do data transformation.

Workflows and Actions are described using the DSL, imported into uOrchestrate, which compiles it into a Petri net and stores it. The syntax generates a Petri net, which will be finite state (meaning the DSL isn't Turing complete). Control structures are translated into Petri nets (meaning there is a well-defined set of Petri nets corresponding to the control structures) similarly to how Hamadi et al.[17] does it, e.g. a Parallel Loop will be translated into a Petri net consisting of many places and transitions, where the transitions manipulate the state, such that the exit place of the Petri net will have the state resulting from a Parallel Loop done on the entry place's state. This proves by construction, that the Petri net will behave correctly when generated from the syntax. On request of an execution (or run) of a Workflow, the first step is added to the queue for the workers to pick up.

1.4.4 Runtime

As mentioned in section 1.3.1, uOrchestrate has workers repeatedly polling a shared queue. Once a Workflow is activated, it will be put in the queue where the state will be the entry place. Everytime a worker pulls it from the queue, the Workflow and its state will either be looked up in cache or compiled into a Petri net (and stored in memory). uOrchestrate then works as a Petri net interpreter, where it can execute transitions on a Petri net given its state. As mentioned in the previous

section, control structures will be compiled into Petri nets with many places and transitions, meaning the Petri net will have more steps than what appers from the syntax. One step, though, can cover multiple transitions, if none of them has any external side effects. The resulting state and the Workflow will be put back in the queue. Python expressions are executed with the state given as the scope as part of transitions.

1.4.5 Tooling

Tooling refers to the language toolchain, i.e. a set of programming tools that can in some way improves development in the particular language. This can be everything from performing complex tasks such as debugging to improving developer experience through editor integration. WL1, being an internally developed DSL, doesn't have an impressive toolchain, but there is some. Using YAML for its syntax provides it with some of YAMLs tooling. That is, WL1 benefits from the tooling that isn't subject to what Fowler calls Symbolic Integration and the Symbolic Barrier[13]. Symbolic Integration focuses on the integration between the base language and the DSL, and the Symbolic Barrier is what limits our ability to manipulate the overall program. More specifically, YAML has editors that provide syntax highlighting and linters that highlights syntax errors, but the lack of Symbolic Integration between WL1 and YAML creates a Symbolic Barrier, that makes the tooling less powerful, since syntax highlighting in YAML isn't syntax highlighting in WL1 and linting in YAML isn't linting in WL1.

Dedicated Tooling

Dedicated tooling has been developed for WL1. One example is orchestratorman which is a pytest[1] plugin that makes it possible to integration-test Workflows on an actual uOrchestrate instance. Actions and Workflows can be stubbed or mocked and state can be inspected in order to ensure correct behaviour. Another example is different runtime handles that provide various debug functionality.

1.4.6 Namespaces and Bundles

When uploading Workflows to uOrchestrate, they are uploaded as part of a bundle. The use of bundles is how uOrchestrate handles dependencies - any Workflow in a bundle can call all the other Workflows or Actions in that same bundle. Furthermore, a bundle can import other bundles as dependencies, which allows the bundle's Workflows to call the imported bundle's Workflows and Actions as well. As mentioned earlier, there exists more than 300 different Workflows, all of them written in WL1, split 30 bundles.

1.4.7 Example: Incremental Deployment

This example takes a look at Incremental Deployment (described in section 1.2.1) as it would be implemented using uOrchestrate with both WL1 and Workflow Language v2.0 (WL2). To reiterate, deployment is basically done incrementally, meaning an upgrade is deployed to a subset of service instances at a time to allow the service to run on the subset until confidence in the revision (the upgrade) has been established. The release process will then proceed with the next subset (until all service instances have been upgraded). Figure 1.7 shows Incremental Deployment as a stack diagram (or a nested state machine), where each frame represents a Workflow. A simplified version of each Workflow (as both YAML and Python) can be found in figure 1.8. First, an image of the service is built from a service name and

a git reference (line 8-12 in figure 1.8g and line 3 in figure 1.8h). It is then deployed to two pairs of data centers sequentially. It is then deployed to the two data centers in each pair in parallel with the rollout Workflow (figure 1.8e and 1.8f). Each data center is partitioned into four zones of increasing size. The service is then deployed sequentially to each zone with the rollout_cluster Workflow (figure 1.8c and 1.8d). After each deployment, the health of the service is monitored (automatically and manually), in order to ensure that everything is working as intended in the particular zone. The Workflow will continue to the next zone, when certain requirements have been satisfied, e.g. a period of time with automated monitoring or manual acknowledgement. In each zone, an upgrade_deployment Workflow is started, that transfers the build to the zone, does a rolling upgrade (a few at a time) of the current service instances to the new one and monitors the newly updated service instances (figure 1.8a and 1.8b). After having deployed the first pair of data centers, the Workflows moves on to the next pair.

1.4.8 Conclusion

The number of Workflows and Bundles indicates that WL1 has definitely been useful. Usability has always been one of the goals of uOrchestrate and WL1 was designed accordingly with simplicity in mind, but due to the rapid growth of Uber and the increasing complexity of the technological infrastructure, complexity of Workflows increased. The intended simplicity made it overly hard to implement complex solutions, and due to compounded technical debt, it has never been resolved. Users have noticed this, and this is the reason this thesis sets out to solve this problem.

```
# Workflow
name: <string>
description: <string>
input:
  - <variable>
output:
  - <variable>: <expression>
workflow:
  - <statement>

# Action
name: <string>
extends: <string>
action:
    condition:
        pre: <expression>
        post: <expression>
        post_feedback: <expression>
    request:
        url: <string>
        parameters:
          - <variable>: <expression>
        method: <string>
        body:
          - <variable>: <expression>
        headers:
          - <variable>: <expression>
        returns: <integer>
        retry:
            wait: <integer>
            max-retries: <integer>
    output:
      - <variable>: <expression>
    input:
      - <variable>
```

Figure 1.6: WL1 syntactic components (Workflow and Action)

```
# Conditional
do:
  if: <expression>
  sequential:
    - <statement>

# Iterate
iterate:
  for: <variable>
  in: <expression>
  sequential:
    - <statement>

# Parallel Loop
run_concurrently:
  for: <variable>
  in: <expression>
  output:
    - <variable>: <expression>
  sequential:
    - <statement>

# Parallel Block
parallel:
  - <statement>

# Sequential Block
sequential:
  - <statement>

# Invocation
<reference>:
  name: <variable>
  args:
    - <variable>: <expression>
```

Figure 1.6: WL1 syntactic components - statements (and Control Flow)

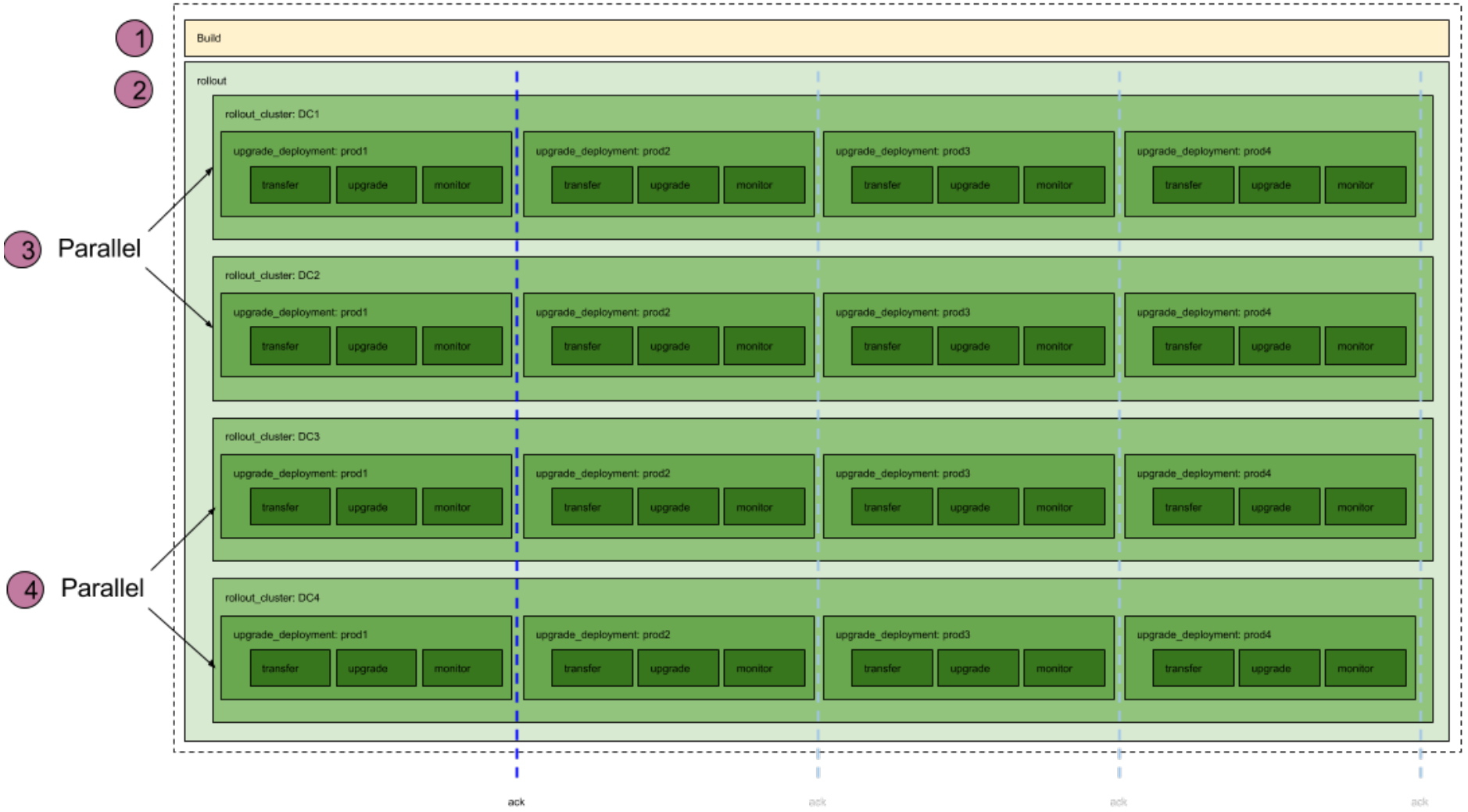

Figure 1.7: Stack diagram of Incremental Deployment

1.5 Hypothesis

The usability of WL1 in uOrchestrate suffers under issues described in section 3 specific to the DSL. In practice, this means that WL1 struggles to gain traction and adaptation inside the Uber organization.

With these considerations in mind, a new and improved DSL can improve both the usability and testability of writing Workflows for uOrchestrate. Specifically, by using WL2, described in section 3, any user will be able to describe a complex Workflow (consisting of 3 loops and 4 subroutines from 2 imported bundles as described in section 1.4.7) twice as fast with half as many errors and half as many lookups in the documentation. The specific improvements can be seen in table 1.2, described using Quality Attribute Scenarios[6] (QAS). Note that Source, Artifact and Environment are fixed in all 4 QASes[1].

Stimulus	**Response - system provides:**	**Response Measure**
(a) Learn to use system	- suggestions & information (through tooling) - more 'programming-like'-environment	Knowledge gain from tooling, syntax and Python up by 100%
(b) Use system efficiently	- look-ahead and auto-completion - easy navigation (e.g jump to implementation) - static analysis (correct syntax) - integrated Python expressions - local validation	Task time down by 50% Number of errors down by 50%
(c) Feel comfortable	- more 'programming-like'-environment - integrated Python expressions	User satisfaction rate up by 50%
(d) Minimize impact of errors	- local validation, resulting in: - correct arguments for calls - required fields being surfaced more quickly - format is correct - quicker testing	Amount of time lost on error down by 75%

Table 1.2: QAS for improvements

1.6 Method

In order to get an understanding of Uber's distributed platform, we have used theory from Distributed Systems: Principles and Paradigms by Tanenbaum and van Steen[25]. We have analysed the usability of their current solution for defining Workflows, WL1, using a software architectural approach as described in Software Architecture in Practice by Bass et al.[6]. We have tried to connect the usability issues with DSL theory from Language Workbenches: The Killer-App for Domain Specific Languages? by Fowler[13]. Using the results from the analysis, we have established requirements and specified a design for a more usable solution using theory from Bass et al. and Fowler. A protoype of the proposed solution was built using the overall design and by following good programming practice by amongst other things using various patterns specified in Design Patterns: Elements of Reusable Object-oriented Software by Gamma et al.[15].

In order to measure the success of the proposed solution, we have specified 4 Quality Attribute Scenarios on Usability with specific goals. We have also pointed

[1]Source: Workflow designer (user), Artifact: Workflow Design, Environment: Runtime & Configuration

out 3 particular weaknesses of WL1, that can be evaluated in isolation. Evaluation was done in two parts: One through a workshop with a variety of participants doing an experiment in order to evaluate the 4 QAS. The other through a questionnaire sent to a mix of workshop participants and actual users of the new language.

```
name: upgrade_deployment
description: upgrade deployment
input:
  - service
  - bld_ref
  - path

workflow:
  - udeploy.transfer:
    args:
      - service: service
      - build_ref: bld_ref
      - uns_path: path
  - udeploy.upgrade:
    args:
      - service: service
      - build_ref: bld_ref
      - uns_path: path
  - udeploy.monitor:
    args:
      - service: service
      - build_ref: bld_ref
      - uns_path: path
```

(a) upgrade_deployment in WL1

```
@workflow('upgrade deployment')
def upgrade_deployment(service, bld_ref, path):
    udeploy.transfer(service, bld_ref, path)
    udeploy.upgrade(service, bld_ref, path)
    udeploy.monitor(service, bld_ref, path)
```

(b) upgrade_deployment in WL2

```
name: rollout_cluster
description: rollout to cluster
input:
  - service
  - build_ref
  - datacenter

workflow:
  - iterate:
    for: zone
    in: >
          "['prod1',
            'prod2',
            'prod3',
            'prod4']"
    sequential:
      - upgrade_deployment:
        args:
          - service: service
          - bld_ref: build_ref
          - path: "'uns://{dc}:{zone}'.
            ↪  format(dc=datacenter, zone=zone)"
```

(c) rollout_cluster in WL1

```
@workflow('rollout to cluster')
def rollout_cluster(service,
                    build_ref,
                    datacenter):
    with Loop(['prod1', 'prod2', 'prod3',
               'prod4']) as zone:
        uns_path = Format('uns://{dc}:{z}',
                          dc=datacenter,
                          z=zone)
        upgrade_deployment(service,
                           build_ref,
                           uns_path)
```

(d) rollout_cluster in WL2

Figure 1.8: Incremental Deployment example

```
name: rollout
description: rollout build
input:
  - service
  - build_ref

workflow:
  - iterate:
    for: dcs
    in: >
          "[['DCA1', 'PVG1'],
            ['SJC1', 'PEK1']]"
    sequential:
      - run_concurrently:
        for: dc
        in: dcs
        sequential:
          - rollout_cluster:
            args:
              - service: service
              - build_ref: build_ref
              - datacenter: datacenter
```

(e) rollout in WL1

```
@workflow('rollout build')
def rollout(service, build_ref):
    with Loop([['DCA1', 'PVG1'],
               ['SJC1', 'PEK1']]) as dcs:
        with LoopAsync(dcs) as dc:
            rollout_cluster(service,
                            build_ref,
                            datacenter)
```

(f) rollout in WL2

```
name: incremental_deployment
description: deploy service
input:
  - service
  - gitref

workflow:
  - ubuild.build:
    name: build_ref
    args:
      - service: service
      - gitref: gitref
  - rollout:
    args:
      - service: service
      - build_ref: build_ref
```

(g) incremental_deployment in WL1

```
@workflow('deploy service', main=True)
def incremental_deployment(service, gitref):
    build_ref = ubuild.build(service, gitref)
    rollout(service, build_ref)
```

(h) incremental_deployment in WL2

Figure 1.8: Incremental Deployment example (cont.)

Chapter 2

Related work

Many big companies working with large distributed systems have a way to execute simple jobs under specific guarantees. These guarantees vary on the company and the requirements of the jobs they want executed. Usually, reliability and scalability are a given, due to the nature of the companies, but other guarantees might only be relevant to certain use cases. The following is a discussion of some of these companies approaches to their language design for job definitions.

2.1 Amazon

Ever since their expansion into on-demand cloud with AWS launching in 2006, Amazon has been a pioneer in large distributed systems. They have been in the same situation as Uber (albeit many years ago) and they emerged victorious. They operate at many times the size of Uber (host-wise) with millions of servers. From an infrastructure perspective, they are definitely one of the best in the world, so it's an obvious direction to look for inspiration. And it just so happens, that one of their products are very similar to uOrchestrate: Their Simple Workflow Service[5] (SWF). SWF also does Web Service Orchestration (as described in section 1.3) and provides users with an easier way to combine the power of different services from their product line.

2.1.1 Simple Workflow Service

SWF makes it possible for AWS users to create Complex Services (as described in 1.2.1), by letting them define transactional 'operations' (policies) consisting of invocations of multiple services in a specific order. A policy is also called a Workflow in SWF and they are similar in concept to the Workflows of uOrchestrate. They both represent a series of steps (that can be run either synchronous or asynchronous), where each step can potentially change the state of the Workflow. Although SWF is very general and can be used in many ways (which also goes for uOrchestrate), the original intent was to have steps be interactions between their different services, which is also the case for uOrchestrate. A comparison of the two can be seen in table 2.1.

When comparing the two, it is crucial to keep the end-users and the development time in mind. SWF has a much larger user base, and they have to persuade them to use their product. However, it has also had far longer to mature to the state it is in now, which probably means a smaller amount of technical debt than uOrchestrate. SWF and uOrchestrate both aim to be reliable and consistent solutions to the same problem. In addition to being reliable and consistent, SWF has been able

	Amazon Simple Workflow Service	Uber uOrchestrate
User Base	External (and internal due to dogfooding).	Internal only (for now)
System Maturity	Amazon uses dogfooding a lot to get the best possible product. SWF was released to the public in 2012, but was probably used internally since way before that. It probably has more dedicated developers than uOrchestrate as well. In general, SWF is a lot more mature, so we have to keep that in mind, when we compare the two.	uOrchestrate has only existed for 2 years, but it has been used a lot in those two years (especially by the developers - dogfooding). Not nearly as mature as SWF, so still contains a fair amount of technical debt. For this reason, we have to be more selective and restrictive when it comes to the goals and guarantees we want to achieve. We can only prioritize a few.
Design Goals	Reliability, scalability, ease of use and flexibility	Primary: Reliability, persistence, observability and reusability Secondary: Usability, testability
Language Choice	SWF is an external product, so they want it to be language-agnostic. Due to that Amazon has developed libraries for any big language.	uOrchestrate is only used internally. The YAML language didn't take advantage of this since everybody has to learn a new language to be able to use it. Using one of Ubers primary languages (Go, Python or Java) would be preferred.

Table 2.1: Comparison of uOrchestrate and Amazon SWF

to achieve ease of use (a lot of time shouldn't be spent on acquiring the skills to design and implement Workflows) and flexibility by implementing SDKs for many different, popular languages such as Python and Java, that makes it possible for any developer to write in their preferred language in a procedural way. 'Ease of use'- and flexibility-wise, uOrchestrate doesn't offer much. When writing Workflows in WL1, it feels like writing a document, not coding a program. It is procedural, but tainted by YAMLs syntax. Language-wise, there are no alternatives to WL1.

uOrchestrate only has internal users and it's original Return on Investment (ROI) isn't based on convincing users to use it. First and foremost, it was made for uDeploy and was designed with that in mind. It has only gotten more users because it actually is very useful to other parts of the organization. With the technical debt, it is rather impressive that the user group grew without any major evangelism. The benefit from convincing other internal teams to use it is to save developer time from Uber's perspective, which of course is important, but as mentioned, priorities were elsewhere at the time. That is why the neglected goals (such as usability and testability) haven't been achieved. The sacrifices were necessary at the time, but now we try to redeem it and hopefully we can learn from SWF.

Usability-wise in general, uOrchestrate might have a slight edge compared to SWF if the low-hanging fruit are prioritized. SWF is more intuitive than uOrchestrate at the moment, but SWF has to integrate with all of Amazons product line, fit into as many language models as possible and in general has to be more general-use than uOrchestrate, so there are a lot of constraints as well. If uOrchestrate's usability got improved, it could feel as intuitive or more. Since all of the communication happens through user-defined requests (RPCs), it doesn't have to integrate directly with any other systems - that is on that particular system to expose the right end-point. It also doesn't have to accept any language, since our customer base is internal. Thus, uOrchestrate is more simple structural and functionality-wise, even though it in a way stays as powerful through facilitating coordination of powerful services.

This means that we, contrary to Amazon, can pick whatever language we prefer and tune it to fit into our desired model. The whole ecosystem around uOrchestrate is written in Python and parts of the Workflows are even written in Python, so where Amazon has to be completely language-agnostic, we can choose Python which gives us a big advantage.

By looking at what currently makes Amazon SWF more usable than uOrchestrate, but also identifying uOrchestrate's strengths and the benefits from Ubers particular situation, we can take what Amazon has done well and make it even

better, by sticking to a single language and focusing entirely on exploiting that. uOrchestrate might not be as flexible, but it becomes very easy to use.

2.2 Netflix

Netflix is another large company that has a very large infrastructure with a microservice architecture. They use AWS as their cloud provider, but have built their own continuous delivery platform called Spinnaker[19]. Netflix has open-sourced Spinnaker, which means that anybody can set it up and use it. Companies the size of Netflix open-source products (Uber does it aswell, but haven't open-sourced uOrchestrate) for a number of reasons, where some of them could be: ideological (belief in the open-source movement), to create a community around the product (and hopefully get them to contribute) and to attract talent (by showing the cool software the company makes). There can definitely be other benefits as well, but it is also hard work, since it puts a lot of other requirements on the product. The two most important being: the code is available to anybody, so every release has to be carefully reviewed, in order to avoid backlash and there suddenly is an external user group as well, increasing the number of feedback channels. The first requirement is something that should be prioritized anyway. Basically, like any email should be written as if it could end up on the front page of The Guardian, any code should be written as if it could be reviewed by anybody in the world (although this can be hard, when technical debt is actively being accumulated). The second one is more interesting, since this relates directly to the usability of the product. Comparing Spinnaker and uOrchestrate is like comparing apples to oranges - Spinnaker is the complete continuous delivery platform, where uOrchestrate is only a specific part. We need to compare Spinnaker to uDeploy to see if we can find any takeaways.

Netflix Spinnaker is definitely interesting to analyse, since it solves many of the same problems uOrchestrate does, but because it is open-sourced and available to the general public, it also **has** to be usable.

2.2.1 Spinnaker

As mentioned, Spinnaker should be compared to uDeploy and not uOrchestrate. If we want to compare uOrchestrate (or more precisely Workflows) to anything, it should be Pipelines in Spinnaker. However, comparing uDeploy and Spinnaker could also be useful. Table 2.2 shows a comparison of the two.

Spinnaker primarily lets users define Pipelines through their own UI, although templating tools have been created by the community (reward from open-sourcing). Netflix has also started development on their own codifying tool[27] (due to popular demand). We could try to improve usability on uOrchestrate by providing a UI for the users, but considering Netflix's experience with community driven templating, their effort towards creating their own codifying tool and an internal observation in Uber on preference of CLIs and language-based solution over GUIs, this might not be the best approach. We can however try and take the best from both world. The UI provides a natural flow to create Pipelines and their stages. It exposes the necessary fields at any given time through forms made for that particular purpose. Expressions in Spinnaker are written in the UI using SpEL, Spring's Expression Language[4], presumably because Orca was originally written using Spring Batch[12]. In uOrchestrate, it neither feels natural nor is any help given regarding what to write or fill out. One advantage with WL1 is that expressions are written in Python, which is a more popular language than SpEL. If we try to make the language-based solution more natural (e.g. by making it feel like actual programming) and provide the user with suggestions and auto-completion (e.g. through

IDEs or tooling), we can maybe achieve this easy way of writing Workflows.

It is clear that we don't want to completely copy the process of defining Spinnaker's Pipelines by providing an UI, but by identifying what works and what doesn't, maybe we can incorporate some of the features in our own language-based solution. If we change our language to a properly tooled one and fix the syntax to something closer to actual programming, then we might get the UI we'd like and the benefits from Spinnakers way to do things.

	Netflix Spinnaker	**Uber uDeploy**
User Base	External and internal (open-source)	Internal only (for now)
System Maturity	Spinnaker was open sourced in November 2015, but used internally before that. It has been under development at least since 2014. Before Spinnaker, Netflix used Asgard, so they have probably brought a lot of experience from Asgard on to Spinnaker. Furthermore, Spinnaker was open sourced in partnership with Google, so that's one of the benefits of open-source - a wider community.	uDeploy was released in 2014. It has not been open sourced so the entire development has happened internally in Uber. It is developed for Ubers particular stack, so it wouldn't be easy to open-source and it would probably not get many users either. Where Spinnaker and Asgard was developed specifically for cloud solutions (AWS at first, but currently many others aswell), uDeploy integrates with the different internal solutions, which at the moment covers both on-premise and cloud (AWS only).
State Machine Solution	Spinnaker has Pipelines as the abstraction for strategy. A Pipeline defines the particular strategy regarding a deployment from when it's built (baked) to the final monitoring before the deployment is completed. Each step is a stage declaring what should happen. Pipelines are defined in a web UI or through Spinnakers API and executed by Orca (a Spinnaker component). Expressions are written using SpEL.	uDeploy has Workflows as the abstraction for strategy. Workflows also handle deployment from start to finish. They are defined in a DSL (YAML based) created specifically for uDeploy and executed by uOrchestrate. The DSL compiles to a Petri net, so there can be many more states than it might seem at first. A Workflow can, however, be separated into phases that can represent particular steps of the strategy. Expressions are written in Python. Pipelines and Workflows have the same functionality, but in a different package.
Usability	In order to achieve usability in Spinnaker, Netflix has not defined any default language to write Pipelines in. They provide the user with a web UI (consisting primarily of forms to fill out representing stages and metadata) and an API that takes the same data. The Spinnaker community have however made efforts towards templating and codifying Pipelines and as a response to that, Spinnaker is in the process of developing their own templating solution that is also based on codifying Pipelines.	uDeploy doesn't provide any UI to define Workflows, so the usability comes from the structure and the tooling surrounding the language. Unfortunately, uDeploy doesn't have much of that at the moment. It could be possible to build a UI similar to Spinnakers, but since our users are internal and in general seem to prefer programming and CLIs to UIs, that would probably not help us. However, the idea of prompting the user for the necessary information is good and definitely lacking in the current language. Could come through suggestions and auto-completion from the tooling used for developing.

Table 2.2: Comparison of uDeploy and Netflix Spinnaker

Chapter 3

Workflow Language: from v1.0 to v2.0

This chapter will present an analysis of the particular problem we're trying to solve. First, by taking a look at the current solution and its issues, and then by proposing a second solution, which tries to solve these issues.

3.1 Workflow Language v1.0

The current Workflow Language (described in section 1.4) was designed with a different idea of Workflows than how they turned out to be. As mentioned in section 1.4.1, they were originally thought of as primarily being sequential pipelines (which is fairly simple to imagine as a sequence of declaration statements). Eventually, Workflows became more complex than expected, which the language wasn't designed to handle very well. WL1, based on YAML, worked well as a declarative DSL, but once writing Workflows became more imperative, it started to suffer from usability issues, relating to YAML originally being markup language/data serialization language.

3.1.1 Issues with Workflow Language v1.0

As described in section 1.4.1, WL1 is either a hybrid of an internal and an external DSL or neither of them. What this really means is that WL1 gets none of the advantages of being either, but suffers from disadvantages from both categories.

Syntax

WL1 is limited to its host languages syntax (internal DSL disadvantage), but it also suffers from the Language Cacophony problem[13] (external DSL disadvantage), meaning a new language has to be learned in order to be able to use it. YAML isn't very intuitive to use for imperative programming/scripting. Programming entirely in lists and maps (which is what a YAML document consists of) isn't what users are used to, and especially when it comes to control flow statements, it isn't obvious how to construct them without referring to the documentation. In a sense, it becomes too declarative (which is part of what it aims to be) which results in verbosity instead of concise problem formulation. YAML is indentation based, which has the advantage of being readable, but can be bothersome to construct. The same could be said about Python, but in Python, it only affects statements, where in YAML it affects every single detail (combining maps and lists can be particularly tricky).

Tooling

As described in section 1.4.5, WL1 has fairly limited tooling. Fowler describes the decoupling between a DSL and the base language as Symbolic Integration[13] (external DSL disadvantage). Even though WL1 is hosted in YAML, we still suffer from this. YAML isn't a programming language, which means it has a very limited toolchain. YAML editors are limited to basic linting (since they can't resolve what level an object should actually be in) and YAML pointers (which WL1 doesn't use). The Symbolic Barrier[13] between WL1 and YAML means no syntax checking in regards to WL1. Referencing other objects, Workflows or Actions isn't a thing, since YAML has its own pointer system that editors use. To reference a variable, the developer has to keep track of the scope himself, since it will be represented as a string (which editors won't suggest). To call another Workflow or an Action, the developer has to remember the name, the parameters, where it's located etc. Support for auto-completion or suggestions is limited to previous occurences (so basically non-existent). Templating doesn't exist. This is also the case for more general structures and reserved words, such as control flow statements. See figure 3.1 for an example of a conditional, that checks whether a build_size is larger than 20,000 and makes an alert if so. There is no help from the tooling in regards to enforcing this structure, which means not knowing exactly how the map should look costs time, since an error wouldn't surface before uploading the Workflow. In general, the language lacks System Initiative[6, p. 122], i.e. it doesn't provide the user with any help based on related models (task, user or system).

Python Integration

Python expressions can be used in particular parts of Workflows and Actions, such as conditionals (branching), when outputting results etc. WL1 integrates badly with these Python expressions. They have to be passed as strings, so validation is needed in order to ensure that they behave as intended under the particular scope in an external environment. It also divides development into two parts or contexts. Python expressions have to written and tested in a Python environment, and then copied into a Workflow environment. Testing also becomes troublesome, since these expressions only exists as strings, which means they have to be manually extracted to unit-test them. There exists a testing framework (orchestratorman, described in section 1.4.5), but it takes time to learn and it's primarily meant for testing entire Workflows and Actions, not for unit-testing expressions. This is no different from SpEL, the language Netflix Spinnaker (described in section 2.2.1) uses for expressions, so it might be a hard problem to solve.

The lack of usability in WL1 isn't just a nuisance to the users. Using uDeploy as an example, the issues can also indirectly lead to outages[1]. Users are prone to making errors, when they find the tool hard to use or the results hard to test - just as with any other programming language. The only difference from a normal programming language is that uOrchestrate and WL1 is used to orchestrate uDeploy, which is an integral system with a large business impact within Uber. A uDeploy outage can be very costly to Uber, since we are dealing with an integral part of Uber's deployment pipeline. An example could be an error-ridden rollback feature. If rollbacks suddenly stop working, the risk of developers deploying important pieces of code with critical bugs is suddenly a lot bigger, since it will be harder to mitigate the damage done.

[1] Unavailability of integral parts of Uber

Conclusion

In essence, we have a language with a steep learning curve, used to orchestrate integral services within Uber, meaning errors can be very costly (e.g. when affecting uDeploy). It also presents a big reward in terms of productiveness and automation (looking at the existing amount of work done by uOrchestrate). It has a small, albeit very active, user group, despite being a very powerful tool, although it might be useful to other teams within Uber, given its broad use-case.

```
1 do:
2   if: int(result['build_size']) > int(20000)
3   sequential:
4     - alert_big_build
```

Figure 3.1: Conditional Workflow step as YAML

3.2 Workflow Language v2.0

We would like to improve on our current solution by making a new and better Workflow language, based on the knowledge and experience derived from the existing language. We want to make deliberate decisions and make the right choices in advance in order to achieve this. The following is a discussion of aspects of the new language to decide.

3.2.1 Fundamental Changes

First of all, we need to look at whether we're satisfied with WL1's philosophy. This is crucial to be able to determine which parts of the current language to replace. If we can already now decide on keeping parts, that would alleviate our workload, so instead of starting from scratch designing a new language, let's take a look at what needs to be changed. The following will be based on the discussion of YAML as syntax of WL1 in section 3.1.1 and the compilation pipeline described in section 1.4.2.

The current language aims to be primarily data-driven and declarative, but also allows the user to write procedural and imperative Workflows (which is necessary for more complex Workflows). It suffers from verbosity and annoying syntax, but the actual features of the language and Petri nets, along with the runtime, are useful and work well for defining the program flow. Data transformation provided through the Python expressions is a crucial feature and nearly all Workflows use them, so they are clearly important. The original idea behind using Python for transformation is preserved. Passing them as strings doesn't work too well though. We wish to make expressions:

- easier to use
- fit more seamlessly into the new language
- more testable

The last issue section 3.1.1 identified was limited tooling, which isn't really part of the philosophy, but definitely an important take-away when designing a new language. In general the issues with WL1 and what should be prioritized in WL2 can be summed up in three parts:

- YAML syntax is verbose and annoying -> design better syntax
- String Python expressions don't integrate very well -> provide similar functionality that is testable
- The toolchain surrounding YAML is limited to markup writing -> pick a language with a toolchain better suited for programming

In order to make WL2 more succesful than WL1, these problems need to get solved. A quick analysis of the problem can be found in table 3.1, where the problem and the most suited approach to a solution in particular areas are described.

Problem	Internal vs External DSL	Language bias	Action item	Contributes to (QAS)
YAML syntax is verbose and annoying	Both, but external gives the most control over the syntax	Extensible and dynamic languages such as LISP, Ruby and Python	Design better syntax	Row a, b and c in table 1.2
String Python expressions don't integrate very well	Internal, since an external DSL requires a lot more work to be able to handle expressions like that	1. Python 2. Any language with concise and powerful expressions	Provide similar functionality that is testable	Row c and d in table 1.2
The toolchain surrounding YAML is limited to markup	Internal, since a toolchain doesn't come for free with an external	Any language with a big community. Python internally in Uber	Pick a language with a toolchain better suited for programming	Row a, b and c in table 1.2

Table 3.1: Analysis and action item for the 3 main WL1 issues

In conclusion, from the fact, that none of the three action items request fundamental changes of the Workflow Language, but rather of the syntax and the tooling, it seems clear, that our focus is on the frontend of the Workflow Language and not the underlying semantics of the language. What we need to achieve is a frontend that can produce the same type of structured dictionary that our YAML subset is parsed into, but without the issues that we're seeing at the moment. The earlier in the compilation pipeline (described in section 1.4.2) the two languages converge, the easier complete compatiblity between WL1 and WL2 can be achieved, which is an important feature. With compatibility between the two languages, no major migration effort of porting existing Workflows will be needed.

3.2.2 Internal vs External DSL

A big decision to make is whether to try and solve this through an internal or an external DSL. Looking at table 3.1, we see that an internal DSL is preferred in 2 of the identified issues and an external DSL is largely preferred in 1 (where internal also has some advantages).

External DSL

Fowler[13] argues that the advantage of an external DSL is the fact that the syntax and the form is completely up to the creator. This is particularly nice for declarative programming, which is why external DSLs are often used for query languages. It's also easy to evaluate at runtime. The corresponding disadvantage is that a parser needs to be implemented. Another substantial disadvantage is what Fowler refers to as the Symbolic Integration (also mentioned in section 3.1.1 as part of the current issues), which basically means that existing tooling doesn't integrate with the DSL. Since one of the current issues is directly linked to tooling, this is unfortunate. A more specific issue in our case has to do with Python integration, which will also be hard to do well with an external language (same issues as with WL1). We do, however, get to pick our own syntax.

Internal DSL

Some of the advantages of using an internal DSL are:

We don't need to reinvent the wheel: The host language provides us with a parser, data structures, expression functionality etc.

The host language brings its community with it: Depending on the language (which can be chosen deliberately), users might be experienced in the host language already. There might also be documentation and it could be battle tested.

The host language brings its ecosystem with it: Again, depending on the language (which can be chosen deliberately), the host language might have a very useful toolchain.

The biggest disadvantage is:

The DSLs syntax is limited to the host languages syntax: The degree of how limiting this is depends on the extensibility and how dynamic the language and its syntax is.

These all depend on the host language, so if we go with an internal DSL, the host language is key to making it good. Let's look into some potential host languages.

3.2.3 Language Choice

Usually, when wanting to write a DSL with no specific user base or system in mind, people will suggest Lisp or Ruby[13] (compared to say C, C++, Java or C#). They both have large communities around writing DSLs in them and great power to customize its syntax through macros and abstractions. They are famously dynamic and extensible. This, however, changes when wanting to write a DSL for an existing system with actual users. The same requirements apply, but they can be bent if there are low hanging fruits to pick by using another language. This is the case for the Workflow Language. The following paragraphs describe the decision to proceed with Python as the host language.

In terms of uOrchestrate, we have a system written in Python and a language that uses inline Python expressions. We have already established, that we will compile our language to an intermediate representation, so the Python-based system might not be that important, but it is however a small advantage, since it will be easier to ship a combined product in the same language instead of two different ones. We also have a CLI written in Python, which will be the first point of contact with the new language, making integration easier. The testing framework also revolves around Python and pytest. In regards to the inline Python expressions, there is a fairly big advantage to gain here. It can be solved using other languages, but that also means more for developers to get used to.

In terms of user base, Uber has traditionally been a Python house, so a lot of developers know Python and have an understanding of how to write good Python. In the same way that WL1's YAML hasn't been succesful in the long run, giving them another DSL in a language they don't know or feel familiar with might push them away again. Picking Python would contribute towards row c infigure 1.2. Other languages used in Uber are Java, Go and JavaScript (for UI). Neither are as dynamic or provides the same extensibility as Python (except for maybe JavaScript),

and they don't fit as well into the ecosystem as Python does either. Go is famously easy to pick up, but so is Python (which is widely regarded as one of the easiest[11] languages to learn and very approachable). Uber developers in general use Uber managed machines, that ships with supported versions and environments for all of these languages. Using other languages will require setup.

In terms of toolchain, which is one of the core benefits of using an internal DSL, we have to figure out what we need. A language debugger or a testing framework might not have very high priority, since integration with the DSL needs to be implemented in order to be able to apply to the Workflow Language (since DSL execution is very different from host language execution), but an editor or an IDE, package manager, integrations and community will all help the developer. As mentioned, testing uOrchestrate and Workflows already happen in Python, although it required pytest integration to be implemented. Since Uber historically has used Python a lot, their own Python package repository works well, which makes it easy to distribute and install packages as requirements in projects. In addition, Python is very popular and therefore has a lot of support in all these areas.

In terms of syntax, if we compare how extensible and dynamic Python is to Ruby and Lisp, we see why Ruby and Lisp are the preferred choice. Python doesn't have macros like Ruby or Lisp, which is an incredibly powerful tool syntax-wise. Instead, one has to rely on Python's other language constructs such as decorators etc.

3.2.4 Prototype Study

Now that we have settled on Python if we were to go with internal, lets look at two prototypes, one internal and one external, and a projection of one of the prototypes, and compare them with the original language to get a better idea of what solution to go with.

In figure 3.4a we see a Workflow (upgrade) written in WL1. It's quite long, spanning over 45 lines, but it is fairly easy to read and understand for an experienced Workflow author (in the sense that almost all the data is named and because indentation gives an initial idea of underlying structure). Looking at it, we see that it takes 3 input parameters and then does up to 3 different Actions in a flow depending on a conditional and an asynchronuous loop. YAML as a language is designed to be readable[7], which we reap the benefits from now. The Workflow designer, however, might have suffered from the issues touched upon in section 3.1.1.

In figure 3.4b we see upgrade written in an external DSL prototype. The external DSL is parsed into the intermediate representation by a parser written in Python. To write the parser, a grammar needs to be designed first. Design goals of the external DSL was to create a syntax that was both more expressive[2] (by inferring metadata from the program) and forgiving (spaces and tabs don't matter - we use { and } as scope delimiters). Python integration is improved a small amount by distinguishing it from normal strings with ‘ as a delimiter.

The language is shorter and more concise than YAML, but harder to read due to lack of syntax highlighting and familiarity with the language. Writing in this external DSL is potentially easier than writing in WL1, since the syntax is more forgiving, and more efficient, since more can be expressed in fewer lines and words,

[2]Expressivity meaning practical expressivity - express concisely and readily, i.e. easily express more in fewer words

but no further studies supporting this has been made, since the rewards from the effort were too small compared with the other prototypes. There were no obvious low-hanging fruits to improve the experience significantly.

In figure 3.4c we see upgrade written in an internal DSL prototype. The objects, we construct, build the intermediate representation on a .build() call. This is a very early prototype of the internal DSL. It is very verbose and doesn't offer much in terms of expressiveness or Python integration. The tooling advantage so far is very limited (most editors would autosuggest when it comes to filling out the constructor parameters). There are, however, a lot of low-hanging fruits that could easily improve the DSL. To give an idea of some of these low-hanging fruits, we have: language constructs (decorators, context managers etc.), better Python integration (needs to be serializable though) and taking advantage of the fact that the language exists inside the Python runtime with all the power it could potentially provide.

In figure 3.4d we see upgrade written in a much improved internal DSL written in Python. This is a projection of the final product. We have a callable Workflow with Python integration, static analysis (both from Python tooling, but also in regards to models) and a lot of different tools from Pythons toolchain (pytest, IDEs and editors with autocompletion and suggestions, Pythons module ecosystem etc.).

Overall, graph 3.2 shows a quick analysis of expressiveness on the vertical axis and efficiency on the horizontal axis based on the 4 examples.

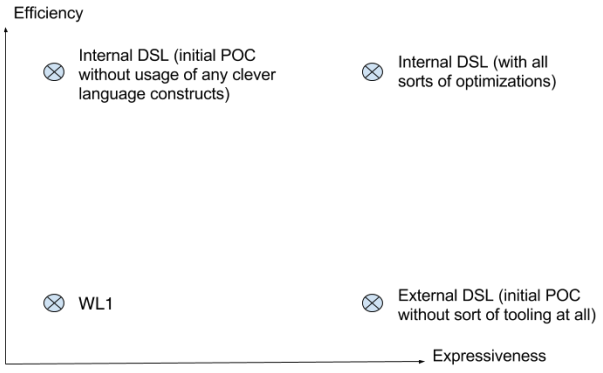

Figure 3.2: Different approaches primarily considering two aspects

3.2.5 Conclusion

WL2 is an internal DSL (although it still compiles to an intermediate format that can be executed at runtime), implemented as a Python library, that provides decorators/annotations, context managers and utility functions to write Workflows and Actions. WL2's documentation can be found in appendix C. See figure 3.5 for the Python version of figure 1.4. Decorators exist for Actions and Workflows, context managers exist for the different control flow statements (see figure 3.6) and the utility functions exist for everything that didn't work well as either a decorator or a context manager. Getting to figure 3.4d (which only has model classes) from 3.4c requires solving a lot of problems. Figure 3.3 shows a list of the problems and the necessary solutions. The technical challenges of the language will be presented and analysed in section 4.

- Reduce explicit object instantiation
  - Make Workflow and Action objects callable
  - Represent Workflow and Action objects as Decorated functions
    - Infer name and input from function object metadata
- Eliminate explicit program flow creation represented by lists
  - Create artifical scope
    - Decorated functions already does this for Workflows and Actions
    - Represent Control Structures as Context Managed objects
  - Create program flow capturing stack
- Eliminate *string expressions*
  - Create lazily evaluated serializable expressions
    - Create class capable of capturing operations
    - Inject newly introduced variables into scope as capturing objects
  - *(Runtime)* Deserialize (execute) serialized expressions

Figure 3.3: Hurdles for WL2

```yaml
name: upgrade
input:
    - name: service
      type: service_id
      description: The service to upgrade
    - name: deployment
      type: deployment_id
      description: The deployment to upgrade
    - upgrade_config

workflow:
    - chat.chat_room:
          name: chat_room
          args:
              service_id: service

          phase-start:
              id:
                  key: upgrade
                  context: [service, deployment]
                  display-data:
                      service: service
                      deployment: deployment

    - do:
          if: "not upgrade_config.get('next_rev_id')"
          sequential:
              - udeploy.chat_message:
                    name: msg_start_build
                    args:
                        room: "chat_room['room']"
                        color: "'yellow'"
                        message: "u'Requesting build of
                         ↪  {service}'.format(service=service)"
    - run_concurrently:
          for: failure_domain
          in: 'list(set(clusters["cluster_names"]) & set(["pek1", "dca1"]))'
          output:
              - failed: original_prod1_deployment
          sequential:
              - udeploy.get_active_revision:
                    name: original_prod1_deployment
                    args:
                        service_id: '"test-uber-service"'
                        deployment: '"prod1"'
                        dep_uid: '"{svc}: {dep}".format(svc="test-uber-service",
                         ↪  dep="prod1")'
```

(a) upgrade in YAML

```
upgrade(service: service_id - "The service to upgrade",
        deployment: deployment_id - "The deployment to upgrade",
        upgrade_config)

workflow: {
  chat_room = chat.chat_room(service_id=service)
  @phase-start(
    key=upgrade,
    context=[service, deployment],
    display-data=(
      service=service,
      deployment=deployment))

  if `not upgrade_config.get('next_rev_id')` {
    msg_start_build = udeploy.chat_message(room=`chat_room['room']`,
     ↪  color=`'yellow'`, message=`u'Requesting build of
     ↪  {service}'.format(service=service)`)
  }

  for failure_domain in `list(set(clusters["cluster_names"]) & set(["pek1",
   ↪  "dca1"]))` parallel {
    original_prod1_deployment =
     ↪  udeploy.get-active-revision(service_id=`"test-uber-service`,
     ↪  deployment=`"prod1"`, dep_uid=`"{svc}:
     ↪  {dep}".format(svc="test-uber-service", dep="prod1")`)
  } return (failed: original_prod1_deployment)
}
```

(b) upgrade in external DSL

```
def upgrade():
    service = {'name': 'service', 'type': 'service_id',
               'description': 'The service to upgrade'}
    deployment = {'name': 'deployment', 'type': 'deployment_id',
                  'description': 'The deployment to upgrade'}
    chat_room = Call('chat.chat_room', service='service').phase_start('upgrade',
     ↪  ['service', 'deployment'], display_data={'service': 'service',
     ↪  'deployment': 'deployment'}).returns('chat_room')
    msg_start_build = Call('udeploy.chat_message', room="chat_room['room']",
     ↪  color="'yellow'", message="u'Requesting build of
     ↪  {service}'.format(service=service)").returns( 'message_start_build')
    original_prod1_deployment = Call('udeploy.get_active_revision',
     ↪  service_id="'test_uber_service'", deployment="'prod1'", dep_uid='"{svc}:
     ↪  {dep}".format(svc="test-uber-service", dep="prod1")')

    return Workflow(service, deployment,
     ↪  'upgrade_config').with_name('upgrade').sequential( chat_room,
     ↪  Conditional("not upgrade_config.get('next_rev_id')").sequential(
     ↪  msg_start_build), Iteration('failure_domain',
     ↪  'list(set(clusters["cluster_names"]) & set(["pek1", "dca1"]))',
     ↪  Sequential(original_prod1_deployment), parallel=True).with_output(
     ↪  failed='original_prod1_deployment'))
```

(c) upgrade in initial internal Python DSL

```
@workflow()
def upgrade(service, deployment, upgrade_config):
    """
    :param service_id service: The service to upgrade
    :param deployment_id deployment: The deployment to upgrade
    """
    chat_room = chat.chat_room(service)
    phase_start('upgrade', [service, deployment], display_data={'service': service,
     ↪  'deployment': deployment})
    with If(not upgrade_config.get('next_rev_id')):
        msg_start_build = udeploy.chat_message(chat_room['room'], 'yellow',
         ↪  Format(u'Requesting build of {service}', service=service))
    with LoopAsync(list(set(clusters["cluster_names"]) & set(["pek1", "dca1"]))) as
     ↪  failure_domain:
        original_prod1_deployment =
         ↪  udeploy.get_active_revision('test_uber_service', 'prod1',
         ↪  Format('{svc}:{dep}', svc='test-uber-service', dep='prod1'))
        failed = LoopAsync.outputs(original_prod1_deployment)
```

(d) upgrade in improved internal Python DSL

Figure 3.4: upgrade Workflow in 2 different prototypes, a projection of the improved language and the original language

```
@workflow('Deploys service to cluster')
def deploy_service_to_cluster(service, clusters):
    """
    :param service_id service: service_id of service
    :param list clusters: clusters to deploy service to
    """
    build_ref = build(service)
    with Loop(clusters) as cluster:
        deploy(build_ref, cluster)
    return {'build_size': build_ref['size']}
```

Figure 3.5: Workflow sample as Python

```
with If(int(result['build_size']) > int(20000)):
  alert_big_build()
```

Figure 3.6: Conditional Workflow step as Python

Chapter 4

Implementation

Section 3 proposed an internal DSL written in Python as a solution to the problems described in section 3.1.1. Proceeding with the proposal, this section will take a look at how the DSL was implemented, starting at a simple library using Builder pattern (see example 3.4c) reaching a version, where nothing is created explicitly using a constructor, but instead created implicitly using all sorts of language constructs Python provides.

4.1 Model Classes

In order to create a new representation of an existing languages syntax, understanding the current representation is key. What that really means is that we have to look at the structure (e.g. through the BNF of a context-free grammar) of WL1, since we want to represent each structural component. Figure 1.6 shows a pseudo-BNF describing the syntactical components of WL1 (except for phases).

It splits up the Workflow language into these predefined structures: {workflow, action, call, parallel, sequential, do if, iterate, run_concurrently}. Phases aren't in the BNF, since they are a bit different than the others. Each of these structures have their own reserved words and they always translate into a similarly structured dictionary. In the original language, each of these structures have been proven to work (see section 1.4.3), meaning that Workflows constructed by using them will also work. If we make a one-to-one representation of each structure, we know that the guarantees from the original representation will be in effect here as well.

As a result, we know we can create a new representation by representing these particular structures. Two interfaces, Callable and Statement, have also been created in order to group the two different types of models. These groups will become useful in section 4.2.1 and section 4.3. A simplified UML diagram using the Module Viewpoint[9] of the Model class architecture can be seen in figure 4.1. See example 4.2 for the resulting classes.

We can use these classes as part of the Builder pattern[15, p. 97]. Once the builder knows what to build, we can build the intermediate representation from the given information. So far, WL1 and WL2 are not comparable, but WL2 have started to make progress in terms of the action items specified in table 3.1, particularly in terms of better tooling, since Python editors are now able to provide suggestions and auto-completion, instead of the user having to lookup in the documentation.

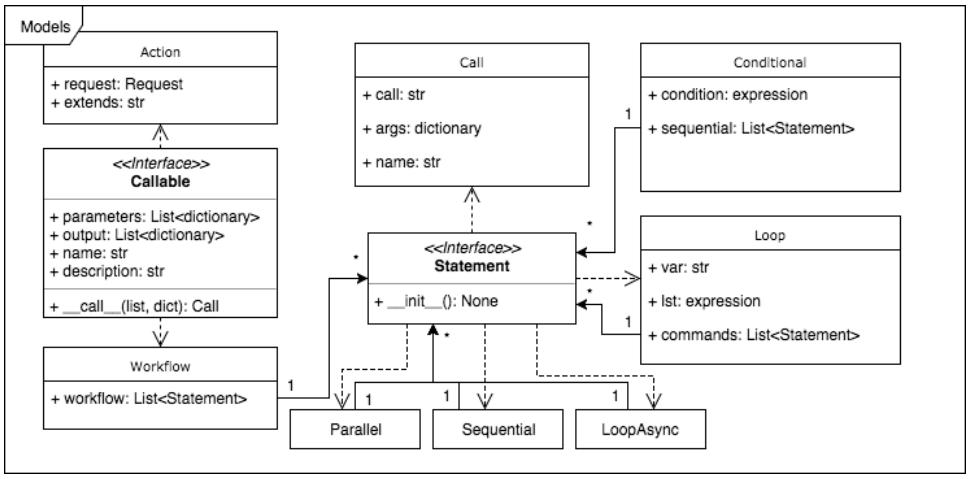

Figure 4.1: UML diagram showing the Model architecture from a Module Viewpoint

4.1.1 Schematics

Depending on intended functionality, implementing the logic in the model classes can take a lot of work. Python isn't statically typed, so in order to minimize impact of errors by doing local validation (table 1.2d), the model classes need to implement type checking, conversion and validation. Serialization is also a requirement. Instead of doing this from scratch, it is possible to use libraries that can generate the model classes from only the necessary information, such as the fields. WL2 uses Schematics[3], a library that provides typed models classes capable of the required functionality. It is already the primary model library in uOrchestrate, which makes it the obvious choice, since it fits into the rest of the data model pipeline (which is very handy once we start looking at bundles), ensuring consistency. Model classes with typed fields are made by subclassing Model and defining the fields as class attributes. Schematics provides recursive validation, conversion between popular data structures and conversion to serializable dictionaries.

From a Workflow Language v2 perspective it gives us the following advantages:

Validation that provides us with certainty and confidence in our objects, since they are validated after creation, effectively minimizing impact of errors (improving on figure 1.2d).

Serialization to dictionaries which we had to implement anyways, since that's our intermediate representation.

Alignment with uOrchestrate since we can plug our models into the existing models (very useful in regards to bundles, which will be elaborated on in section 4.5).

Less code to maintain.

The disadvantage is the fact that we don't have all the code under control. Changes to how Schematics do things are done through overrides, which isn't always very transparent.

```python
class Callable(object):
    def __call__(self):
        pass

class Workflow(Callable):
    def __init__(self, name, description, parameters, output, workflow):
        super(Callable, self).__init__()

class Action(Callable):
    def __init__(self, name, extends, description, parameters, output, request,
    ↪  conditions):
        super(Callable, self).__init__()

class Statement(object):
    def __init__(self):
        pass

class Call(Statement):
    def __init__(self, call, name, args):
        super(Statement, self).__init__()

class Parallel(Statement):
    def __init__(self, parallel):
        super(Statement, self).__init__()

class Sequential(Statement):
    def __init__(self, sequential):
        super(Statement, self).__init__()

class Conditional(Statement):
    def __init__(self, condition, sequential):
        super(Statement, self).__init__()

class Loop(Statement):
    def __init__(self, var, lst, commands):
        super(Statement, self).__init__()

class LoopAsync(Statement):
    def __init__(self, var, lst, output, commands):
        super(Statement, self).__init__()
```

Figure 4.2: Model Classes for Workflow Language structures

The advantages of using Schematics are substantial, and the disadvantage is limited to a few parts of Schematics (meaning only a few overrides is needed to overcome the challenge), so let's continue with Schematics. A simplified example of a model class written in Schematics can be seen in example 4.3.

4.2 Language Constructs in Python

As seen in example 3.4c, defining Workflows continues to be very verbose. All the information has to be passed into a constructor and there is no real flow to defining them. As mentioned in figure 3.3, reducing the amount of explicit object instantiations would help, i.e. create a leaner syntax. Python has a lot of functionality in order to automate explicit or repeated functionality. Decorators can dynamically alter functionality in regards to function definitions (basically wrapping a function to modify its behaviour). Context managers can handle an object's context, automating setup and teardown (which is useful in regards to the artificial scope mentioned in figure 3.3).

```
class Workflow(Callable):  # Callable now inherits from Model
    workflow = ListType(SimpleModelType(Statement), serialize_when_none=True)
    name = StringType(required=True)
    description = StringType()
    output = ListExpressionMapType()
    parameters = ListType(SimpleModelType(Input), default=[],
     ↪  serialize_when_none=True)
```

Figure 4.3: Schematics class for Workflow structure

Python also has a very dynamic object model, meaning everything is basically an object (including classes, functions etc.). This means we can have objects can act like functions by defining the `__call__`[2] method of its class. This makes it possible to make decorators, that will appear to decorate functions (i.e. wrap them with functionality), but instead of returning the decorated function, they will return an arbitrary object implementing `__call__` with no noticeable difference to the user.

It might be possible to take advantage of (or even exploit) this in order to improve the syntax.

Let's look at the different approaches to automation and inferring data and see whether they can help.

4.2.1 Callable Objects

In WL1, a non-reserved key represents a function call of an Action or a Workflow (WL1's de facto functions and the reason that the language is somewhat procedural). As seen in example 4.2, a function call in WL2 is represented by explicitly instantiating a `Call` object with a string describing what Action or Workflow to call, a name describing where to store the result and a list of arguments.

`__call__` is an overridable Instance Method, that makes objects callable, essentially making them act like functions. Using this, Actions and Workflows can be made callable, making them return a `Call` object. What we can do is make `__call__` take any amount of arguments, store them in `args`, infer the name from the object that is being called and return a `Call` based in this data. We still have to manually set the name field (representing the variable it gets stored in) though.

What we get is a more familiar and less verbose way to call Workflows and functions. It turns out particularly useful, when we start taking advantage of decorators as well. See example 4.4.

```
class Callable(Model):
    def __call__(self, **kwargs):
        return Call(call=self.name, args=kwargs)

# Example call with empty workflow
deploy = Workflow(name='deploy', input=['service', 'cluster'])

deploy_call = deploy(service='svc1', cluster='cluster1')
```

Figure 4.4: Callable Workflow class

4.2.2 Decorators

Proceeding with the idea of treating Workflows and Actions as functions, decorators are a great way to hide the fact that they (objects with an implemented `__call__`) aren't traditional functions. A decorated function isn't a traditional function either, but the general idea is that it should behave relatively close to the original function, meaning it becomes intuitive (for both the system in regards to tooling and the user) to call the function with the specified arguments like a normal function would typically be called. If we define a Workflow or Action by decorating a function, it will be obvious that it (the Workflow or Action represented by a decorated function) is callable. Decorators takes the function object as input and can return anything (usually this would be a wrapped function or a callable object). See example 4.5 for examples of decorated functions.

```
1  @action()
2  def message(message):
3      action.request.url = 'http://localhost:1337/message?msg={message}'  # typing
       ↪  action.request. will suggest url as a field
4      return {'response': response}
5
6  @workflow()
7  def deploy(service, cluster):
8      call = message(message='deploying now')
9      return [call], None
```

Figure 4.5: Sample decorated Workflow and Action

Function objects have richer metadata than normal objects, making it possible to infer data otherwise not available (such as reference name and parameters). Three of the fields on a Workflow or Action correspond to that of a function (name from the `__name__` field, input from parameter names and output from return values). We can use a couple of different techniques to gather data from the function body - by injecting some sort of aggregator, either as a parameter or directly into the scope, where data can be set or appended, or by collecting it from the return values. For Workflows, we'll just return what we need from the body. This isn't ideal, but we'll see a way to do this in a cleaner way in section 4.3. For Actions, we'll inject an Action object. The way this is done will be described in section 4.2.2. After executing the decorated function and having the necessary data at hand, we can construct a Workflow or Action object and return it with no noticeable difference to the user (since it is also callable). Modern IDEs or editors will even suggest the decorated functions parameters (from static analysis) instead of those of `__call__`, which is very useful, since they are the actual parameters needed to call the Workflow or Action. The docstring is also available in the decorator, so if we use that to annotate params with types or descriptions, we can extract them from the decorator (which a modern IDE will also do in order to do type hinting). See example 4.6 for an example of the two decorators.

Exploiting Predictions from Modern Editors

Figure 1.6 shows the structure of an Action. The structure is significantly more predictable than that of a Workflow. It has a bunch of fields describing what it needs to do. There is no control flow involved. What we really want is to have the object available with suggestions on the fields (since it can be hard to remember exactly what it stores). Modern editors or IDEs will do static analysis and infer the type of the object from its source. We can either inject the object into the function

as a parameter or directly into the scope, but neither of these solutions will work, since the editor or IDE won't know the source. We can explicitly denote the type either in the docstring or in a comment, which some editors or IDEs will pick up on, but this needs to be done for every action in order to get suggestions.

What we can do, however, is inject it directly into the scope under the same name as something already in scope and even make the object already in scope (which will be overwritten on runtime) have the fields, that we want to be suggested. We are already importing the `action` decorator, which is not used for anything inside an `@action` decorated function, and it even has the semantically correct name. In addition to this, decorators can be implemented as a class and classes can have fields, making it perfectly suited to act as a proxy for the injected object. See example 4.6b for a class implementing these tricks.

```
def workflow(description=None):
    def decorator(f)::
        parameters = f.__code__.co_varnames[:f.__code__.co_argcount]
        sequential, output = f(*parameters)
        w = Workflow(name=f.__name__, description=description,
         ↪  parameters=parameters, workflow=sequential, output=output)
        w.validate()
        return w
    return decorator
```

(a) Decorator for Workflow

```
class action(object):
    """
    :type request: Request
    :type condition: Condition
    """
    request = None
    condition = None

    def __init__(self, description=None, extends='base'):
        self._extends = extends
        self._description = description

    def __call___(f):
        parameters = f.__code__.co_varnames[:f.__code__.co_argcount]
        a = Action(name=f.__name__, extends=self._extends,
         ↪  description=self._description, parameters=parameters)

        temp_action = f.__globals__.get('action')
        f.__globals__['action'] = a  # inject action
        a.output = f(*parameters)
        f.__globals__['action'] = temp_action

        a.validate()
        return a
```

(b) Decorator for Action

Figure 4.6: Decorators for Action and Workflow

4.2.3 Context Managers

We also want to represent control flow statements. WL1 provides us with multiple ways to define control flow, such parallel blocks and asynchronuous loops, sequential blocks or iterations and conditionals (branching). By just representing these as

objects, we preserve the verbose nature of our initial DSL. We could use decorators in the same way as we use them for Actions and Workflows, but that would mess up our scoping, since function scopes are local and only available inside the function body. For now that could do (and it did for a short while during development), since we still use *string expressions*, but if we manage to replace them (which we will in section 4.4), decorators wouldn't work. Using decorators to do many things could also be confusing compared to only using it on function-like objects. We do, however, want some of the same functionality decorators provide, especially something that lets us setup and teardown the scope of the block.

Context managers do just that. They have an `__enter__` and an `__exit__` function, that gets called when entering and exiting the context. The most well-known example of a context manager in Python is `open`. It takes a path and a mode and returns a file. If used in conjunction with `with` (e.g. `with open(path) as f:`), it automatically closes and cleans up the IO on either a context exit or on exceptions. This gives a better overview of where the file is accessible. We do, however, not get the same way of returning data from the context body as we did with decorators (unless we exploit exceptions, injecting data into the `__exit__` function).

By making each class into a context manager and augmenting it in order to add statements dynamically (since we can't return them from the context body), we can create scopes local to the context. The improvement stemming from this is not that great yet, but the artificial scope will be key in making this language succesful, as described in the forthcoming chapters. See example 4.7 for an example of a context manager.

```
1  class Loop(Statement):
2      # Left out model class implementation
3
4      def __enter__(self):
5          return self
6
7      def __exit__(self, exc_type, exc_val, exc_tb):
8          self.validate()
9
10     def add_statement(self, statement):
11         self.sequential.append(statement)
12
13 @workflow()
14 def deploy(service, clusters):
15     with Loop('cluster', clusters) as loop:
16         loop.add_statement(message(message="'deploying to {}
           ↪  now'.format(cluster)"))
17     return [loop], None
```

Figure 4.7: Context manager for Loop

4.3 Stack

So far we have achieved model classes, that provides us with validation and suggestions (through tooling), a better representation of procedural programming and functions (through an implementation of `__call__` and decorators) and better visibility and control of scoping (through context managers and decorators). We still, however, have to manually add every step to a list in order to define the program flow as seen in section 4.2.2 and mentioned as one of the major problems in figure 3.3.

We can take advantage of the new-found control of scope, since we know whenever we enter and exit a level of scope (i.e. for context managers through their `__enter__` and `__exit__` and for decorators through manually calling the decorated function). We want to automatically collect everything that happens in a body and store it in our list. When entering the body, we can make note of a new level or frame, then during the execution of the body append newly created objects to the frame, and then when exiting take everything from the note till now and attach it to the parent object. This is basically a call stack[10]. See figure 4.8 for a sequence diagram of the stack interaction of an example Workflow.

We will make a globally available stack, that we can add to and pop from. When entering a new level of scope, the parent structure will announce that it is using the stack (using a global semaphore) and store the beginning of the frame (the current size of the stack). When statements (objects of our model classes except Workflows and Actions, as seen in figure 4.1) are created, they will add themselves to the stack (if it is being used). When exiting the current level of scope, the parent to the level will pop the corresponding stack frame from the stack, using the previously stored stack pointer, and announce that it isn't using the stack anymore. Now we don't need to explicitly tell what the sequential looks like to each object, but it is instead inferred from the order of execution. See figure 4.9 for a simplified implementation of the stack.

4.4 Serializable Python

Having the stack, it isn't necessary to manually define program flow anymore - it will be created according to the order of the statements, solving problem 2 in figure 3.3. All interaction (except for named assignments, described in section 4.4.3) with model objects has been hidden from the user, essentially removing the need for variables (at least as long as *string expressions* continue to be the only way to reference output from other Workflows or Actions). As mentioned in 3.1.1, one of the goals of WL2 is to fix Python integration. Section 3.2.1 mentions 3 goals (ease of use, seamless integration and testability) that should be accomplished, while still preserving the power of the *string expressions*. Looking at approaches to achieving these goals in WL2, two challenges are posed:

- expressions have to be lazily evaluated (in contrast to eagerly evaluation), since we don't have the scope on compile time
- expressions have to be be serializable, since Workflows are transferred around and executed distributedly

Considering the first challenge (lazy evaluation), the obvious approach is to pass the expression as a lambda or function. Functions are lazily evaluated and ascope can easily be injected. They are, however, not innately serializable. There are workarounds for this (either through the Dill and Pickle packages or through serialization and reconstruction of a function's deconstructed code object), but they are hacky and hard to debug. Code objects aren't version independent either, so updating Python could break all existing Workflows. It seems fair to say functions aren't feasible.

What we can do instead is to create a class hierachy that captures and collects information instead of executing it. That way we can reenact what should have happened from that information, but do it server-side during an execution where a scope is actually present. Since almost everything in Python can be overridden, this is possible. `obj.__getattr__('attribute')` is used to look up an attribute (field) when doing `obj.attribute` and `obj.__getitem__('item')` is used to look

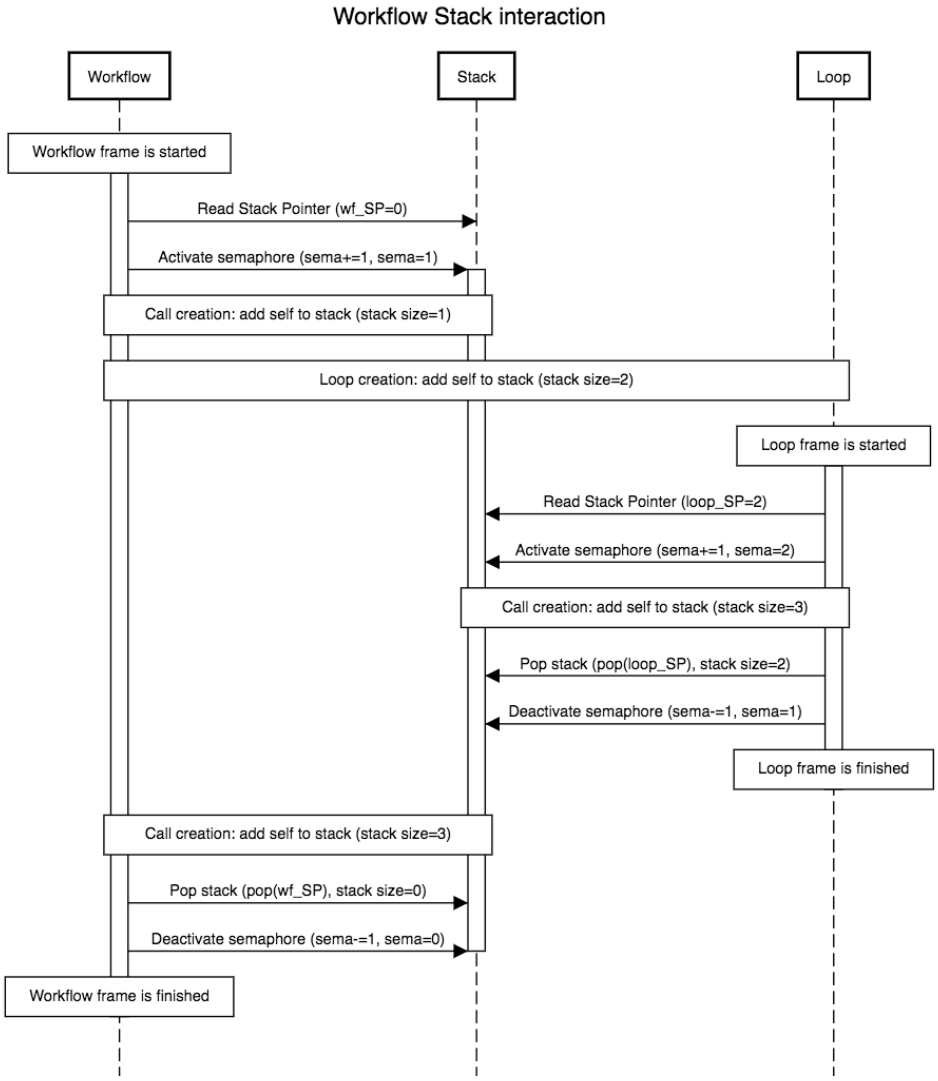

Figure 4.8: Sequence diagram of Workflow and stack interaction

```
stack = []
semaphore = 0

class Statement(Model):
    def __init__(self)
        if semaphore > 0:
            stack.append(self)

class BaseContextManager(object):
    def __enter__(self):
        self._offset = len(stack)
        grab()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        release()
        self.sequential = stack[self._offset:]
        del stack[self._offset:]

def workflow(description=None):
    def decorator(f)::
        parameters = f.__code__.co_varnames[:f.__code__.co_argcount]
        offset = len(stack)
        grab()
        output = f(*parameters)
        release()
        sequential = stack[offset:]
        del stack[self._offset:]
        w = Workflow(name=f.__name__, description=description,
         ↪  parameters=parameters, workflow=sequential, output=output)
        w.validate()
        return w
    return decorator

@workflow()
def deploy(service, clusters):
    with Loop('cluster', clusters):
        message(message="'deploying to {} now'.format(cluster)")
    return None
```

Figure 4.9: Stack interaction

up an item when doing `obj[item]`. Both of these can be overridden in order to capture the necessary information, i.e. the arguments and the type, and return a serializable object instead. The most basic case is referencing a variable from scope, in which situation, we only need to know the name and which case we're in. An example of a more advanced case could be a call, where we have to serialize named and listed parameters. An example of this can be seen in figure 4.10.

```
class Expr(object):
    def __init__(self, name):
        self._name = name

    def __getattr__(self, item):
        return AttributeExpr(self, item)

    def serialized(self):
        return {EXPR_TYPE: EXPR,
                'name': self._name}

class AttributeExpr(Expr):
    def __init__(self, caller, method):
        super(AttributeExpr, self).__init__(None)
        self._caller = caller
        self._method = method

    def serialized(self):
        return {
            EXPR_TYPE: ATTRIBUTE_EXPR,
            'caller': self._caller.serialized(),
            'item': self._method
        }
```

```
>>> expression = Expr('expression')
>>> expression.this.that.serialized()
{'__exprtype': 2,
 'caller': {'__exprtype': 2,
  'caller': {'__exprtype': 0, 'name': 'expression'},
  'item': 'this'},
 'item': 'that'}
```

Figure 4.10: Capturing class

4.4.1 Uncollectible data

This example only shows how to collect data from `__getattr__`. Almost any operation can be overridden (`__getitem__`, `__call__`, `__eq__` e.g.), except for a few reserved words ({`and`, `or`, `not`, `in`, `is`}), inline boolean expressions (`expr if cond else expr`) and {`list`, `dict`, `set`} comprehensions (`[i for i in lst]`). For these, we have to create helper functions, since they will always be eagerly evaluated.

4.4.2 Scope

So what can we do with these lazily evaluated expressions represented by information capturing objects? Example 4.10 shows an explicit and manual instantiation of the `Expr` class, which can capture and serialize data. Compared to *string expressions*, we can now write actual Python code, but at the expense of instantiating the class manually and deserializing the serialized expression. For now, let's focus on improving the manual instantiation.

There are four main cases we need to look at. Three, where we are introducing a new variable into the scope of a Workflow, and one, which is in regards to the builtin variables. The first is the input that a Workflow or Action takes. The second is the return value of a call (action or Workflow). The third is when we loop over a list. The fourth is in terms of establishing the initial scope with builtin variables. If we can make these four cases automatically introduce an `Expr` object, then there is no need to manually create it or even expose the class to the user. Luckily, we have a lot of control in all these areas, since we have made conscious decisions to control our flow and scope as much as possible (with regard to `__call__`, context managers and decorators).

In the first case, we're looking to inject `Expr` objects into our decorated functions. As seen in line 3-5 in example 4.6a, we inject strings representing the variable names (since we also need them for our model object). If we instead change `parameters` to `[Expr(i) for i in parameters]`, we will be injecting `Expr` objects with the right reference into the function without the user knowing or caring.

In the second case, we are already creating a `Call` object in our overridden `__call__` that will be put on the stack, as seen in example 4.4. As mentioned earlier in this chapter, we have nothing to return in `__call__`, but if we keep returning the `Call` object and then make it possible to add a referencing name with `.returns(name)`, we can make `returns` return an `Expr` object referencing that particular name. We will find an even better way to do this in section 4.4.3.

In the third case, we are using context managers as seen in example 4.7. Before adding the stack, we needed to return the model object, so that we could add statements to its sequential. Now with the stack, we don't need to return anything. In the particular loop cases, this means we can return an `Expr` object referencing the loop variable. An iteration would instead look like `with Loop('cluster', clusters) as cluster:`.

In the fourth case, we have to do some hacking. We basically want to replace the global builtins in the entire scope of our Workflows/Actions, but only while inside their function body (so the rest of the compiler isn't affected by the distorted global scope). Decorators make this sort of easy, since we have the function available and can change its object. Our way of doing this is shown in example 4.11.

This means that we have replaced all the occurences, where we would otherwise have to manually create an `Expr` object, with automatic creation instead. When referencing a variable, we can now easily see what scope it is part of and where it was introduced. One thing to note is, that this is only possible due to control flow statements being modelled by context managers instead of decorators, since an `Expr` object created inside a decorated function isn't in scope outside of the function body. Objects created in a managed context are, however, available in scope.

4.4.3 Named Assignments

As mentioned with both the `Loop` variable and the variable storing the returned value from calls, we still have to explicitly declare the variable using respectively the `var` parameter in the `Loop` constructor and `.returns(name)` when using calls. This is similar to how it's done in YAML, where it is declared in a field in the particular structure. This works well, but it feels strange when the rest of the language has started to be similar to normal Python. Unfortunately, the only way to connect a variable name to the object it's referencing or pointing to is by using

```
builtin_vars = dict(response_code=Expr('response_code'),  # Everything in Expr is
 ↪  exposed by uOrchestrate
                    execution_id=Expr('execution_id'),
                    body=Expr('body'),
                    uuid4=Expr('uuid4'),
                    str=GlobalsExpr('str'),
                    any=GlobalsExpr('any'),
                    all=GlobalsExpr('all'),
                    set=GlobalsExpr('set'))  # A lot of builtins left out in this
                     ↪  example

def call_func_with_builtin_vars(f, *args, **kwargs):
    temp_builtins = {}
    for k, v in builtin_vars.iteritems():
        temp = f.__globals__.get(k)
        if temp is not None:
            temp_builtins[k] = temp
        f.__globals__[k] = v
    try:
        offset = len(stack)
        grab()
        res = f(*args, **kwargs)
        release()
        seq = stack[offset:]
        del stack[offset:]
    finally:
        for k, v in builtin_vars.iteritems():
            temp = temp_builtins.get(k)
            if temp is not None:
                f.__globals__[k] = temp
            else:
                del f.__globals__[k]
    return res, seq
```

Figure 4.11: Calling function with temporarily overwritten builtins

code introspection. This is very unreliable since it depends a lot on the formatting of the source file, so we want to avoid that.

If we instead throw away the readable name and generate a UUID[1], then we can get rid of the explicit variable declaration. This makes it harder to get insight (e.g. when debugging) since variable names become indistinguishable, but we can sort of work around that by appending the call name to the UUID. Debugability are provided through other means as well (such as through the testing framework described in section 4.7).

New Assignments

Using UUIDs does present a challenge, we need to fix. Since the language has branching, we can end up in cases, where a variable with the same name stores different things depending on which branch it went into. Using UUIDs, assigning something to the same variable will make it reference a new `Expr` with a new UUID, essentially overwrite the existing UUID, meaning the variable will always reference the latest call.

In our `Expr` class, we have overwritten all non-assigning operations (that is, everything except augmented arithmetic assignments[26]). We can't have assignments in our expressions, so we left them out. We can, however, maybe solve our current predicament by overriding one of these augmented arithmetic assignments, since

[1] universally unique identifier

that lets us modify the object our variable is currently referencing.

We chose the left shift augmented arithmetic assignment operator (<<= or `__ilshift__(self, other)`, since it was the one that looked the most like an actual assignment. See figure 4.12 for an example of a bad and a good assignment.

```
@workflow()
def create_or_upgrade(service):
    depl_exists = deployment_exists(service)
    with If(depl_exists):
        state = create(service)
    with Else():
        state = upgrade(service)
    return state
```

(a) Bad assignment - state will always reference upgrade call return value

```
@workflow()
def create_or_upgrade(service):
    depl_exists = deployment_exists(service)
    with If(depl_exists):
        state = create(service)
    with Else():
        state <<= upgrade(service)
    return state
```

(b) Good assignment - state will keep UUID created at 'create' call and attach it to 'upgrade' call aswell

Figure 4.12: Good and bad assignment

4.4.4 Deserialization

So far, everything done to improve the language has been done client-side in order to improve the syntax and compilation process. The two first problems in figure 3.3 only interferred with the client-side compiling of WL2, while the third overlaps between the client and the uOrchestrate runtime (described in section 1.4.4). Creating lazily-evaluated serializable expressions happens client-side as well, but they are executed remotely, meaning some work has to be done there. Once expressions have been serialized, they are transferable. Where *string expressions* are easily executed in Python using its internal interpreter, serialized expressions needs to be deserialized - in practice this would be an execution of the expression. As described in section 4.4 and example 4.10, we construct call hierarchies by having calls point to their caller. When serializing a call hierarchy, the outermost layer will be the last call, which will point to whatever did the call. We cannot do the call before having resolved the caller, meaning we cannot peel off layer by layer, but instead have to recurse down the hiearchy until we reach the original caller (the innermost layer, dictionary or `Expr`), resulting in an inside-out way of parsing the hiearchy.

The deserialization algorithm recursively parses through the layers, resolving them when their children have been resolved. It's loosely based on the Visitor pattern[15, p. 331], in the sense that the algorithm visits an `Expr` implementation based on its type.

It takes a serialized expression (as a dictionary) and a scope (a dictionary as well) and returns the result of the execution. Simple references are looked up in the scope and actions are performed on the caller.

Figure 4.13 shows a simplified version of the deserializer.

```
def handle_simple_expr(expr, scope):
    return scope[expr['name']]


def handle_action_expr(expr, scope):
    caller = _parse_expr(expr['caller'], scope)
    args = [_parse_expr(arg, scope) for arg in expr['args']]
    kwargs = expr['kwargs']
    return caller.__call__(*args, **kwargs)


def handle_attribute_expr(expr, scope):
    caller = _parse_expr(expr['caller'], scope)
    item = _parse_expr(expr['item'], scope)
    return caller.__getattribute__(item)


visit_parser = {EXPR: handle_simple_expr,
                ACTION_EXPR: handle_action_expr,
                ATTRIBUTE_EXPR: handle_attribute_expr}


def handle_list_type(expr, scope):
    return [_parse_expr(v, scope) for v in expr]

def handle_simple_type(expr, _):
    return expr

type_visitor = {int: handle_simple_type,
                list: handle_list_type}

def _parse_expr(expr, scope):
    v_type = type(expr)
    if v_type in type_visitor:
        return type_visitor[v_type](expr, scope)
    expr_type = expr[EXPR_TYPE]
    if expr_type in visit_parser:
        return visit_parser[expr_type](expr, scope)
    raise CodeExecutionError('Bad expression %s' % str(expr_type))
```

Figure 4.13: Simplified version of the serialized expression deserializer

Reconstruct *string expressions*

A similar algorithm was made to reconstruct the serialized expressions into the strings that they previously would have been in WL1. The algorithm has the same flow, but without the scope to look up in. The reproduced string is used for multiple purposes, primarily to make a serialized expression readable, which is very useful when debugging. They can also be used to check whether expressions are valid without having a scope.

4.5 Bundles

With WL1, uOrchestrate would ingest Workflows and Actions as part of bundles passed to it as described in section 1.4.6. A bundle is a collection of Workflows with some logical connection and some metadata to describe it. It consists of YAML files representing the Workflows and Actions and a JSON manifest with the metadata. Using a client, this will all be compressed into a ZIP archive and sent to uOrchestrate, which will validate it to the best of its ability. Once uploaded, the bundle needs to be activated by its version and name, and then the main Workflows

will be executable. An example of a bundle can be seen in figure 4.14.

In WL2, we need to figure out a way to structure bundles, since uOrchestrate's main point of entry is through them. We can try to do it the same way as with WL1. Have a Python file containing all Workflows in WL2, but separate each Workflow or Action into its own JSON (serialized) file, preserve the JSON manifest and then reuse the original point of entry. Doing it this way will introduce the following issue:

We have to figure out a smart way to build our objects, serialize and store them in each their JSON file before we can upload. This can be done in many ways (e.g. by adding a `save()` method, that will write the serialized object to a file), but all of them add another step to the build process, since a bundle has to be rebuilt everytime it needs to be uploaded.

If we look away from doing it the exact same way as with WL1, we can try to get rid of the JSON files and the manifest, but only on the surface (so that we don't have to change the entry point). An example can be seen in figure 4.15. A bundle will be represented mainly by a Python file, which will contain all of the bundles resources (Workflows and Actions, either directly or imported from another Python file) and the metadata (which can be stored in a module docstring as some sort of easily parseable format, e.g. YAML). The archive will then be created dynamically by extracting the resources and the metadata from the bundle file, so it can be sent to the entry point. To do this, we only need to augment the client, not uOrchestrate itself.

4.6 Translator

The language has come to a state, where the majority of the intended functionality has been achieved. However, a large part of the power in WL2 comes from the way it interacts with other WL2 bundles and imports. Importing bundles written in WL1 is possible, but not nearly as nice as doing it in WL2. It might be harder to get people to adopt WL2 if WL1 stays prevalent. If people aren't adopting it, Workflows won't get translated/migrated (a Catch-22). We could do it manually, but with over 300 Workflows, manual translation is not an option.

Since we designed the language to be as equal a representation as possible, it should be fairly easy to translate one-to-one. We wrote a translator to make the migrations easier. We also wrote a program to compare the old representation to the new one.

4.7 Testing framework

As mentioned in section 1.4.5, WL1 and uOrchestrate has orchestratorman as testing framework. Workflows and Actions can be tested on a local uOrchestrate instance. orchestratorman also makes it possible to stub and mock Workflows and Actions. orchestratorman primarily facilitates integration-tests, since the goal and the test strategy is to upload, activate and run the Workflow in question. It is very useful and a very important feature, but it is possible to get far into the test process to discover something unit-tests would have surfaced instantly, meaning we have a chance to minimize impact of errors here (as described in 1.2d).

WL2 makes it easier to define inline Workflows, since the framework uses pytest, i.e. tests are written in Python. WL2 also does local compilation and validation, which makes it possible to extract that part from the original orchestratorman test strategy instead of having to spin up a uOrchestrate instance (which can take time, if the bundle in question has a lot of dependencies). In addition, WL2 has introduced the `@expr()` decorator, which aims to make it easier to unit-test expressions.

```yaml
name: health_check
action:
    request:
        url: http://localhost:19074/
         ↪  health_check?service={service}
        method: GET
    input:
        - service
    output:
        - status: response_code
```

(a) health_check.yaml Action

```yaml
name: health_check_service
input:
    - service
output:
    - alive: status == 200
workflow:
    - health_check:
        name: status
        args:
            service: service
```

(b) health_check_service.yaml Workflow

```json
{
  "name": "health_check",
  "description": "Workflow for checking services health",
  "maintainers": [
    "eplatz@uber.com"
  ],
  "version": 1,
  "resources": [
    {
      "name": "health_check_service",
      "type": "main",
      "file": "health_check_service.yaml"
    },
    {
      "name": "health_check",
      "type": "lib",
      "file": "health_check.yaml"
    }

  ],
  "imports": []
}
```

(c) health_check bundle manifest

Figure 4.14: Health check bundle in WL1

4.7.1 expr

Functions decorated with `@expr()` are expressions that can be both serialized and evaluated locally. If the decorated function is called with the named argument `test_scope`, then it will be executed locally, otherwise it will be serialized, so that it can be executed remotely in uOrchestrate.

```
"""
name: health_check
description: Workflow for pinging services
maintainers: [eplatz@uber.com]
version: 1
"""
from workflow_lang.frontend import (
    action,
    workflow
)
from workflow_lang.globals import response_code

@action():
def health_check(service):
    action.request.url = 'http://localhost:19074/health_check?service={service}
    action.request.method = 'GET'
    return {'status': response_code}

@workflow(main=True)
def health_check_service(service):
    status = health_check(service)
    return {'alive': status == 200}
```

Figure 4.15: health_check bundle in WL2

Chapter 5

Evaluation

In this section, we will take a look at how the final result has been received. In order to get an idea of the improvement, three ways of evaluation has been used. An experiment has been conducted with a variety of participants. A questionnaire has been sent out to the particular users. The adoption within Uber has also been looked at.

5.1 Workshop

A small workshop was held with 5 different Uber employees of different skill[1] (3 experts, 1 intermediate and 1 beginner). It consisted of a small presentation, a live demo and then an experiment. The participants had different amounts of experience with the original Workflow language, ranging from zero experience to experts. The intermediate user was the only one having tried WL2. The idea of the presentation and the live demo was to give the participants an introduction to the language before having them work in it. The experiment they had to do was based on the example deployment strategy described in section 1.4.7.

5.1.1 Introduction to Language

The participants were given a small presentation outlining the core concepts and ideas behind the language and a few examples illustrating them. The presentation lasted approximately 10 minutes. The demo lasted 10 minutes and consisted of a writing a simple Action (doing a POST request to localhost) and a simple Workflow (calling the Action), uploading the bundle and starting the Workflow. The code can be seen in example 5.1.

5.1.2 Setup

The participants were asked to clone a git repository containing scaffolding, stubs (see figure 5.2 for a stubbed bundle with a build Workflow, that will send a message to a chat room) and setup instructions for the experiment. The instructions can be seen in figure 5.3. The participants were shown the same diagram as in figure 1.7. An introduction to what the diagram meant was given and then the participants had to recreate Workflows to represent the Workflow in the diagram. See figure 5.4 for a sample solution. Some leeway were given in regards to the participants solution, in since it could be solved in many ways (e.g. without conditionals or splitting it

[1] experts having written >10 Workflows, intermediates having written 3-9 Workflows and beginners having written 1-2 Workflows

```
"""
name: hello_world
version: 1
maintainers: [eplatz@uber.com]
"""
from workflow_lang.frontend import (
    action,
    base,
    workflow
)
from workflow_lang.globals import *


@action(base)
def call(obj):
    action.request.url = 'http://localhost:19074'
    action.request.body = {'payload': obj['value']}
    action.request.method = 'POST'
    return {'message': response}


@workflow(main=True)
def do_call():
    call({'value': 'Hello World'})
```

Figure 5.1: Live demo bundle

Participant	Experience Level	Time	Errors	Documentation Lookups
1	Expert	13m 45s	0	0
2	Expert	14m 30s	0	3
3	Expert	15m 00s	1	1
4	Intermediate	10m 30s	0	0
5	Beginner	17m 00s	3	5
Baseline	Expert	33m 15s	3	5

Table 5.1: Experiment results

up into 4 Workflows). The same experiment was performed on an expert Workflow writer to get a baseline to compare to. Instead of the presentation on WL2, he was given 5 minutes to read the documentation (which can be seen in appendix B), brush up on keywords etc.

5.1.3 Results

The results can be seen in table 5.1. They were in clear favor of WL2. People wrote the Workflows much faster with fewer errors (an error being a failed upload), even though only one of them had previous experience with WL2. The results indicate that WL2 is twice as fast to write in compared to WL1, even when having no prior experience with Workflows or the language. The amount of errors were fewer in WL2 and the documentation (which can be seen in appendix C) wasn't as necessary due to IDE suggestions and familiar syntax.

```
"""
name: mubuild
version: 1
maintainers: [eplatz@uber.com]
imports:
  - bundle: chat
"""
from workflow_lang.frontend import (
    LegacyCall,
    Format,
    workflow
)
from workflow_lang.globals import (
    utcnow,
    uuid4,
    workflow_user
)


@workflow()
def build(service, gitref):
    LegacyCall('chat.message', room='workflow-language-v2-workshop',
     ↪  message=Format('{wfuser} tried to build {svc} from {gitref} at time
     ↪  {utc}', gitref=gitref, svc=service, wfuser=workflow_user, utc=utcnow()),
     ↪  color='green')
    return {'build-ref': str(uuid4())}
```

Figure 5.2: Stubbed ubuild bundle example

```
1. git clone {repository}
2. git checkout -b {name}_incremental_deployment
3. make bootstrap
4. source env/bin/activate
5. uorchestrate-cli bundles initialize-bundle
6. {name}_incremental_deployment
7. Open {name}_incremental_deployment.py in your favorite Python IDE/editor
8. Fill out remaining bundle metadata (version, maintainers)
9. import mudeploy
10. import mubuild

11. Write Workflows

12. uorchestrate-cli bundles create-n-upload --python
 ↪  {name}_incremental_deployment.py
13. uorchestrate-cli bundles activate {name}_incremental_deployment {version}
14. uorchestrate-cli workflows start -p arg1=val -p arg2=val
 ↪  {name}_incremental_deployment.incremental_deployment
```

Figure 5.3: Setup instructions for experiment

```
"""
name: incremental_deployment
version: 1
maintainers: [eplatz@uber.com]
"""
from workflow_lang.frontend import (
    If,
    Loop,
    LoopAsync,
    workflow
)

import mubuild
import mudeploy


@workflow()
def upgrade_deployment(service, deployment, build_ref, cluster):
    transfer_res = mudeploy.transfer(service, deployment, cluster, build_ref)
    with If(transfer_res['success']):
        upgrade_res = mudeploy.upgrade(service, deployment, cluster, build_ref)
        with If(upgrade_res['success']):
            mudeploy.monitor(service, deployment, cluster)


@workflow()
def rollout_cluster(service, cluster, build_ref):
    with Loop(['prod1', 'prod2', 'prod3', 'prod4']) as deployment:
        upgrade_deployment(service, deployment, build_ref, cluster)


@workflow()
def rollout(service, build_ref):
    with Loop([['DC1', 'DC2'], ['DC3', 'DC4']]) as parallel_clusters:
        with LoopAsync(parallel_clusters) as cluster:
            rollout_cluster(service, cluster, build_ref)


@workflow(main=True)
def incremental_deployment(service, gitref):
    build_res = mubuild.build(service, gitref)
    rollout(service, build_res['build-ref'])
```

Figure 5.4: Sample solution for experiment

5.2 Questionnaire

In order to get an idea of how users felt when writing in WL2, a questionnaire was sent out to existing users and experiment participants. Questions were designed around the hypothesis (section 1.5).

5.2.1 Setup

The questionnaire consists of a section for users with experience with WL1 (11 questions) and a section for users with no previous experience (5 questions) and then 2 questions for both user groups. All answers are given a rating from 1 to 5, 1 being the worst and 5 being the best. 7 people answered the questionnaire.

5.2.2 Results

The results can be divided into two sections: One focusing on users with previous Workflow experience and one focusing on users without. In regards to questions, they can be divided into 5 areas: Tooling, Syntax, Python Integration, Language Comfortability, Efficiency and Ease of Learning.

Users with previous Workflow experience

5 out of the 7 questionnaire participants had previous experience with writing Workflows.

In regards to Tooling, WL1 scored an average of 1,8/5. WL2 scored an average of 3,25/5.

In regards to Syntax, WL1 scored an average of 2,4/5. WL2 scored an average of 3,75/5.

In regards to Python Integration, WL1 scored an average of 1,6/5. WL2 scored an average of 3,75/5.

In regards to Language Comfortability, WL1 scored an average of 3,4/5. WL2 scored an average of 3,5/5.

In regards to Efficiency, WL1 scored an average of 2/5. WL2 scored an average of 3,5/5.

In regards to Ease of Learning, WL2 scored an average of 3,25/5.

WL2 scored higher than WL1 in all areas. How much higher depended on what area. Python Integration, for instance, scored over twice as high in WL2 as the old. Language Comfortability, on the other hand, was very close, which makes sense, since the answers came from users that were experienced in WL1 and not WL2. The same could be said for Efficiency, but WL2 still scored significantly higher in that area.

Users without previous Workflow experience

2 out of the 7 questionnaire participants didn't have any experience with writing Workflows.

In regards to Tooling, WL2 scored an average of 4/5.

In regards to Syntax, WL2 scored an average of 4,5/5.

In regards to Python Integration, WL2 scored an average of 4/5.

In regards to Language Comfortability, WL2 scored an average of 3/5.

In regards to Efficiency, WL2 scored an average of 4/5.

In regards to Ease of Learning, WL2 scored an average of 3,5/5.

Without having WL1 as a frame of reference, WL2 scores higher. Comfortability is still fairly low, which makes sense, since the language is completely new to the participants.

5.3 Adoption

Three projects have adopted WL2 so far and there are efforts to migrate existing Workflows in queue. The language was released close to the end of the year, which means that people are looking to finish their current projects before deadlines and performance evaluation. Due to this, people haven't been looking very much outward, but instead inward. Not that many Workflows have been written in general, but the majority have been written in WL2.

5.3.1 Unsolicited Feedback

Early adopters of WL2 have in general had a small need to reach out for help in regards to functionality, syntax etc. The fact that users have been able to create >300 LOC bundles independently indicates that the language is fairly intuitive to use. In the few occasions where users have reached out, they have given compliments about the general use of the language, e.g. how it has improved writing Workflows greatly in terms of efficiency.

5.4 Compared to hypothesis

Looking back at our hypothesis in section 1.5, we can take a look at each QAS in table 1.2 and compare it to our results.

In regards to 'Learn to use system' (1.2a), we can see from our questionnaire results, that experienced users prefer tooling, syntax and Python integration in WL2 twice as much as in WL1. New users also felt very good about those three (scoring 4/5, 4,5/5 and 4/5 respectively), even though they had no frame of reference. Both groups of users thought the language was fairly easy to learn as well. Unfortunately, it wasn't possible to get data on how hard it was to learn WL1, but it has generally been an issue during informal discussions on the language. From the experiment, we can see that users in general looked in the documentation very few times, indicating that using the language felt intuitive to them.

In regards to 'Use system efficiently' (1.2b), we can see that the task time was indeed down by approximately 50% in all cases, compared to the baseline. Number of errors varied between users of different experience, but in general, the number of errors were down by atleast 50%. Looking at the questionnaire, both experienced and new users felt very efficient in WL2.

In regards to 'Feel comfortable' (1.2c), both experienced and new users expressed neutrality on comfortability of WL2. This makes sense, since it usually takes time to get comfortable with a new language. It could have been interesting to compare this to language comfortability with new users of WL1. This would be a better indicator of whether or not the language is an improvement.

In regards to 'Minimize impact of errors' (1.2d), no particular experiments have been made. This basically depends on good practice, but WL2 does provide a bunch of tools to minimize time lost due to error. This comes primarily from the fact, that a uOrchestrate instance isn't needed to validate the bundle or Workflow.

Conclusion

Looking at the hypothesis and evaluating whether it holds, we see that WL2 does indeed improve on a lot of the circumstances surrounding WL1. Considering how early in WL2's lifecycle this is and that almost all of the evaluation was based on inexperienced users, this is very promising. Once it gets a larger userbase, the benefits will accrue, since one of the greater improvements is the interaction between Workflows and bundles, i.e. we need existing Workflows and bundles in WL2 to take advantage of this improvement.

Chapter 6

Conclusion

In this thesis, we have attempted to redesign uOrchestrate's Workflow Language (WL1) with the aim of fixing usability issues relating to design choices made early in its development.

In order for us to really understand WL1, we took a step back to look at Uber's technological architecture and infrastructure and then continuously zoomed in on particular challenges until WL1 became needed, making the context and influences of its development clear. The context is described using Uber's microservice architecture and its internal deployment system, uDeploy, as the primary example of a WL1 consumer.

The philosophy and implementation of WL1, along with its context, was analysed using theory on DSLs and Software Architecture resulting in identification of three main issues: Syntax, Toolchain and Python integration - all relating to WL1's syntax, a subset of YAML.

A solution to these issues was proposed through a redesign of WL1's syntax based on the findings of the analysis. The redesign replaces YAML as the syntax with an internal DSL written in Python, solving all three main issues by respectively improving the syntax, providing the user with access to Python's toolchain and having the syntax be Python, making it possible to do actual integration.

A new frontend of the language (the Python DSL) has been implemented based on the redesign (WL2). WL2 uses Model classes to represent the old YAML structures. It takes advantage of specific Python features, such as language constructs and object callability, in order to support System Initiative by using IDEs and static analysis. An actual scope, implemented using context managers, makes the syntax easier to keep track of and it also makes it possible to use a stack in order to get rid of explicit and manual ordering of statements. It integrates with Python expressions through a functionality-collecting, serializable class, that overrides as much primitive functionality as possible, meaning Python expressions can now be lazily evaluated or deserialized server-side. WL2 is completely compatible with WL1. It is also capable of replacing WL1 in the long run.

In order to evaluate WL2, we have made two user studies: A workshop with a small experiment and a questionnaire. We have held these up against our hypothesis, which have shown WL2 to solve many of the issues we set out to fix. The results show improvements on 3 out of the 4 main statements in the hypothesis.

6.1 Future Work

There are still potential improvements to be done in regards to WL2.

The stack (described in section 4.3) is still very primitive and only used early in the compilation pipeline in order to keep track of program flow. In order to get better debuggability, the stack could be augmented to collect more data in order to provide it to the runtime. This could open up for the possibility of having line numbers or a stacktrace.

Serializable Python (described in section 4.4) still isn't used seamlessly, when looking at list and dictionary comprehensions, inline boolean expressions and reserved words such as `and` or `or`.

Bibliography

[1] Pytest. https://docs.pytest.org/en/latest/. Accessed: 2018-01-11. 1.4.5

[2] Python Data Model. https://docs.python.org/2/reference/datamodel.html. Accessed: 2018-01-03. 4.2

[3] Schematics. https://schematics.readthedocs.io/. Accessed: 2018-01-03. 4.1.1

[4] Spring Expression Language (SpEL). https://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/html/expressions.html. Accessed: 2018-01-10. 2.2.1

[5] J. Barr. Amazon Simple Workflow Service. https://aws.amazon.com/blogs/aws/amazon-simple-workflow-cloud-based-workflow-management/, 2012. Accessed: 2017-10-16. 2.1

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012. 1.5, 1.6, 3.1.1

[7] O. Ben-Kiki and C. Evans. YAML Ain't Markup Language (YAML) Version 1.1. http://www.yaml.org/spec/1.2/spec.html. 3.2.4

[8] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016. 1.2.1

[9] H. B. Christensen, A. Corry, and K. M. Hansen. The 3+ 1 approach to software architecture description using uml revision 2.3. 2012. 4.1

[10] E. W. Dijkstra. Recursive Programming. *Numer. Math.*, 2(1):312–318, Dec. 1960. 4.3

[11] T. Donaldson. Python as a first programming language for everyone. In *Western Canadian Conference on Computing Education*, volume 547, page 2015, 2003. Accessed at https://www.cs.ubc.ca/wccce/Program03/papers/Toby.html. 3.2.3

[12] R. Fletcher. Spinnaker Orchestration. https://medium.com/netflix-techblog/spinnaker-orchestration-19e7f7b88d33, 2017. Accessed: 2018-01-10. 2.2.1

[13] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? 2005. Accessed at http://www.martinfowler.com/articles/languageWorkbench.html. 1.4, 1.4.1, 1.4.5, 1.6, 3.1.1, 3.1.1, 3.2.2, 3.2.3

[14] M. Fowler and J. Lewis. Microservices. *ThoughtWorks.* *http://martinfowler.com/articles/microservices.html*, 2014. Accessed: 2018-01-10. 1.1.1

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. 1.6, 4.1, 4.4.4

[16] E. Haddad. Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow. `https://eng.uber.com/soa/`, 2015. Accessed: 2018-01-09. 1.1.1

[17] R. Hamadi and B. Benatallah. A petri net-based model for web service composition. In *Proceedings of the 14th Australasian database conference-Volume 17*, pages 191–200. Australian Computer Society, Inc., 2003. 1.4.1, 1.4.3

[18] C. N. Johnsen. Architects of Infrastructure: Meet Uber Aarhus Engineering. `https://eng.uber.com/aarhus-engineering/`, 2017. Accessed: 2018-01-14. (document)

[19] A. Jordens. Netflix Spinnaker. `https://blog.spinnaker.io/scaling-spinnaker-at-netflix-part-1-8a5ae51ee6de`, 2016. Accessed: 2017-11-02. 2.2

[20] J. Joseph, M. Ernest, and C. Fellenstein. Evolution of Grid Computing Architecture and Grid Adoption Models. *IBM Syst. J.*, 43(4):624–645, Oct. 2004. 1.1.1

[21] C. A. Petri and W. Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008. revision #91646, accessed at `http://www.scholarpedia.org/article/Petri_net`. 1.3

[22] D. Sato. CanaryRelease. `https://martinfowler.com/bliki/CanaryRelease.html`, 2014. Accessed: 2018-01-02. 1.2.1

[23] R. Schmidt. Project Mezzanine: The Great Migration at Uber Engineering. `https://eng.uber.com/mezzanine-migration/`, 2015. Accessed: 2017-12-21. 1.1

[24] M. Schwarz. μDeploy: Deploying daily with confidence. `https://eng.uber.com/micro-deploy/`, 2016. Accessed: 2017-10-16. 1.2.1

[25] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms.* Pearson Prentice Hall, 2007. 1.1, 1.2, 1.2.1, 1.3, 1.6

[26] T. Wouters. CanaryRelease. `https://www.python.org/dev/peps/pep-0203/`, 2000. Accessed: 2018-01-02. 4.4.3

[27] R. Zienert. Codifying your Spinnaker Pipelines. `https://blog.spinnaker.io/codifying-your-spinnaker-pipelines-ea8e9164998f`, 2017. Accessed: 2018-01-08. 2.2.1

Appendices

Appendix A

Using WL2

In order to try out WL2, the code (excluding tests) has been extracted from uOrchestrate and uploaded to a public repository. Only code directly related to the new language has been extracted (into the directory workflow_lang), which represents the first part of the compile pipeline, Lexing & Parsing (described in section 1.4.2). This is enough code to write in WL2 and try out some of the improvements, but not enough to actually run workflows (which would also require a uOrchestrate instance). Code to run serialized expressions (described in section 4.4) and translate WL1 workflows (described in section 4.6) has also been extracted (into respectively the workflow_runtime and translator directories). The repository contains a README.md with instructions on how to setup the environment and use the language, along with a few demos:

- sample bundles similar to the ones described in section 1.4.7 and section 5.1.2
- a Python script running a few serialized expressions

The repository can be found at:
`https://bitbucket.org/elloarmy/workflow_lang_v2`

Appendix B

WL1 Documentation

The following is the internal documentation of WL1. It has been redacted in order to remove internal and unnecessary details.

THE ORCHESTRATOR WORKFLOW LANGUAGE

Orchestrator workflows consist of actions that are bound together by control structures. Workflows are simple programs that string together calls to actions in a way that is similar to a classical, imperative programming language. Workflows are, however, executed in a fault tolerant manner so that each action will be retried until it succeeds. An execution of a workflow can be distributed over several orchestrator instances and execution is fault tolerant in that execution continues even if one or more of the participating orchestrator instances terminate.

A SIMPLE HELLO WORLD EXAMPLE

The following is a simple example workflow, which will pause immediately after it has been started. It will announced this in the given chat channel and wait to be resumed.

```
input:
    - room

workflow:
  - confirm_to_continue:
      args:
          room: room
```

STATE

Each workflow execution has a state. Actions write output to this state and commands can read from it. The state is a python dictionary that can be accessed when writing Python expressions.

Command output

Commands such as actions may define output. This output is accessible in the state if the command has been named in the workflow. A `name` key is used to name a command. In the example above, we could name the `confirm_to_continue` command output `confirm` by setting a name:

```
input:
    - room
```

```
workflow:
  - confirm_to_continue:
      name: confirm
      args:
          room: room
```

Actions declare their output using an `output` map as shown in the next section.

ACTIONS

Actions are the basic building block of the workflow language. Each action defines a type of interaction with a REST endpoint. Actions should be idempotent in that they may be executed more than once to retry on failures (which could be network failures).

The super type of all actions is `base`:

```
#
# Base action
#
# This is the arc type action - sub-types may only "overwrite" fields spef
# everything else will be ignored
#

name: base          # name used in error reporting
extends:

action:
    condition:
        pre:            # python assertion
        post:           # python assertion
        post_feedback:  # optional expression that gives a human readable
    request:
        url:            # url
        parameters:     # a python map expression or a list of mappings to
        method:         # GET, PUT, POST, DELETE, HEAD
        body:           # a python map expression or a list of mappings to
        headers:        # a python map expression or a list of mappings to
        returns:        # expected HTTP response code
        retry:
            wait:         # ms to wait before a retry
            max-retries:  # positive integer, number of retries before rep
    output:
    input:
```

Concrete actions can override any of the properties of base. The `input` clause is a list of parameter names that are expected and the `output` clause defines a map of output from the action. The wait time can be set to any positive integer, but there is a natural latency in the orchestrator between runs of XXms.

As an example, the following action definition describes the REST call needed to start a build in uDeploy:

```
name:               build-start
extends:            base
action:
  request:
    url:            http://localhost:{port_num}/api/service/build
    method:         POST
    body:
      - svc_id:     service_id
      - git_ref:    gitref
      - description: description
    returns:        200
    retry:
      wait:         50
      max-retries:  4
  input:
    - name: service_id
      type: service_id
    - name: gitref
      type: gitref
    - name: description
  output:
    - build_revision: response.items()[0][1].items()[0][1]['result']['rev_
```

In the example, the request url is set to `http://localhost:{port_num}/api/service/build` and the request method is POST. The body is a JSON map containing three values which are just simple lookups of input values. The action expects the return code to be 200. If it is for example 503, the request will be retried after a while (but at least 50ms). The `post` entry can furthermore contain a Python expression that checks whether a request has succeeded. If this expression is false, the request will also be retried.

When calling the actions, the caller must set the values for `service_id`, `gitref`, and `description`. A call could look like this:

```
input:
    - service

workflow:
  - build-start:
      args:
          service_id: service
          gitref: "'origin/master'"
          description: "'Build something'"
```

This would build origin/master for some service. Notice that since the values are Python expressions, we need to enclose strings in quotes. Other workflows can be called in the same way that actions are called.

SUPPORTED CONTROL STRUCTURES

The workflow language supports a simple conditionals, iteration as well as implicit fork-join. These control structures can be combined with each other as well as with actions to create workflows.

do if

The `do if` construct executes an action if the given condition is satisfied. The condition is a Python expression:

```
do:
  if: expr
  sequential:
    - Workflow command
    ...
```

The sequence is only executed if the expression has the value true.

parallel

The `parallel` construct executes a list of commands in parallel, waiting for the last one to finish before moving on to the next command after parallel:

```
parallel:
  - Workflow command
  - Workflow command
```

iterate

The `iterate` construct iterates through a list, executing a list of commands for each member of this list:

```
iterate:
      for: var
      in: list
      sequential:
            - Workflow command
            - Workflow command
```

Run concurrently

The `iterate` construct has a variant which allows authors to iterate over a list but run the associated block concurrently.

```
run_concurrently:
      for: var
      in: list
      sequential:
            - Workflow command
            - Workflow command
```

The control-flow automatically forks and joins without additional work required. Note that all of the parallel executions must successfully reach the end of the block before execution can proceed.

```
run_concurrently:
      for: var
      in: list
      output:
          - <variabel>: <expr>
      sequential:
            - Workflow command
            - Workflow command
```

When processing items in parallel it is often useful to be able to collect output. To do that you can specify an *output* section to the header of the iteration. By defining this, the list of provided variables will be available in scope after successful completion of the block. Each variable will map to a dictionary with the value of the expression indexed under the list item.

Appendix C

WL2 Documentation

The following is the internal documentation of WL2. It has been redacted in order to remove internal and unnecessary details.

THE ORCHESTRATOR WORKFLOW LANGUAGE V2

The Orchestrator workflow language v2 is a new approach to defining workflows in Python by using decorators, context managers, classes and functions. It is fully interoperable with the existing workflow language and in general it is structured the same way too, so any existing know-how should be transferable. In any case, this documentation will cover the exact same material as “The Orchestrator workflow language” and “Workflow bundles”, but with the new language.

INTRODUCTION

In the new language Python modules acts a bundles. This means that if you import a bundle, it is going to be actively available (instead of just being a namespaced reference) in the compilation process and that any IDE support or tooling (such as auto-completion or suggestions) works in regards to its members. The only important distinction is that if you import another module, it is the same as importing the bundle, but if you import specific members of a module, it won’t count as importing the bundle. In that case, those members will be added as resources to the bundle that’s importing them. Using this you can split up bundles into multiple files to keep an overview. The bundle metadata will primarily be extracted from the module docstring, which is formatted as YAML and has the same attributes as the current JSON manifest, except resources and as mentioned imports. Resources are extracted from the members of the module, more specifically ActionDefinitions and WorkflowDefinitions.

Actions and Workflows are defined by decorating a function with respectively the action and the workflow decorator. A decorated function is callable, using either named or ordered parameters, meaning you just call the workflow or action inside other workflows as you would call a normal Python function. An IDE will even suggest the parameters and their types (if written in the functions docstring) and the compiler will validate the call. Actions can inherit from other actions which means that anything you don’t override will keep the parents attribute, including parameters.

Inside workflows, you can call existing workflows and actions, but you can also use different context managers as control structures:

- Sequential
- Parallel
- Loop
- LoopAsync

- If and Else (Else can only be used in succession of an If)

When using Python expressions (a feature in the old Workflow language), you no longer have to pass them as strings. Most Python integrates completely with the new Workflow language, except for different types of comprehensions (list, dict etc.), boolean expressions, a few operations (and, or, in, is, not and combinations of them) and builtin string functions (format and join etc.). For these, utility functions are exposed (Join, Format, List etc.).

While reading the following documentation, you should keep a few wins of the new language in mind, since it especially simplifies the particular flow around those features. The wins are:

- Better modularity through imports
- IDE support
- Python integration
- Feels closer to programming than before

Orchestrator workflows consist of actions that are bound together by control structures. Workflows are simple programs that string together calls to actions in a way that is similar to a classical, imperative programming language. Workflows are, however, executed in a fault tolerant manner so that each action will be retried until it succeeds. An execution of a workflow can be distributed over several orchestrator instances and execution is fault tolerant in that execution continues even if one or more of the participating orchestrator instances terminate.

Disclaimer

Disclaimer: This language isn't Python. This is a DSL, meaning the rules have been bent to make them fit into uOrchestrate's universe. An example of this is ad hoc object creation: If you want to use a dictionary, you have to make sure that it's in scope. It won't be if you just define it and reference it, however, if you return it as output or pass it into uOrchestrate as input, it will be.

A SIMPLE HELLO WORLD EXAMPLE

The following is a simple example workflow, which will pause immediately after it has been started. It will announced this in the given chat channel and wait to be resumed.

```
@workflow()
def hello_world(room):
    confirm_to_continue(chat_room=room)
```

STATE

Each workflow execution has a state. Actions write output to this state and commands can read from it. The state is a python dictionary that can be accessed when writing Python expressions.

Python Expressions

Python expressions are no longer passed as a string, but instead integrated into the current language more seamlessly. Expressions can be defined in two ways. Either by constructing them directly where you need them or by extracting them into a function annotated by the @expr decorator. This way you can reuse the code and even unit test them by importing them into your tests. The last approach is preferable because it encourages good engineering practices, such as code reutilization. You can then import the expression into multiple modules and even unit test it.

The following shows how to construct a serializable Python expression, that takes two numbers and returns the range from 0 to their sum.

```
@expr()
def expression(variable1, variable2):
    return range(int(variable1) + int(variable2))

expression(4, 5)  # results in lazily evaluated serializable Python expres
expression(4, 5, test_scope={})  # eagerly evaluates the expression, inten
```

Command output

Commands such as actions may define output. This output is accessible in the state if the command has been named in the workflow. In the example above, we could assign the `confirm_to_continue` command output to `confirm`:

```
@workflow()
def hello_world(room):
    confirm = confirm_to_continue(chat_room=room)
```

Actions declare their output using an `output` map as shown in the next section.

ACTIONS

Actions are the basic building block of the workflow language. Each action defines a type of interaction with a REST endpoint. Actions should be idempotent in that they may be executed more than once to retry on failures (which could be network failures).

The super type of all actions is `base`:

```
@action()
def base():  # Set parameters
    """
    Base action
    This is the arc type action - sub-types may only "overwrite" fields sp
    everything else will be ignored
    """   # Set parameter types and descriptions
    action.condition.pre = None              # python assertion
    action.condition.post = None             # python assertion
    action.condition.post_feedback = None    # optional expression that gi
    action.request.url = None                # url
    action.request.parameters = None         # a python map expression or
    action.request.method = None             # GET, PUT, POST, DELETE, HEA
    action.request.body = None               # a python map expression or
    action.request.headers = None            # a python map expression or
    action.request.returns = None            # expected HTTP response code
    action.request.retry.wait = None         # ms to wait before a retry
    action.request.retry.max_retries = None  # positive integer, number of
    return None  # Set output map
```

Concrete actions can override any of the properties of base. Input parameters are set in the function signature and the return value defines a map of output from the action. The wait time can be set to any positive integer, but there is a natural latency in the orchestrator between runs of XXms.

As an example, the following action definition describes the REST call needed to start a build in uDeploy:

```
@action(base)
def build_start(service_id, gitref, description):
    """
    :type service_id: service_id
    :type gitref: gitref
    """
    action.request.url = 'http://localhost:{port_num}/api/service/build'
    action.request.method = 'POST'
    action.request.body = {'svc_id': service_id,
                           'git_ref': gitref,
                           'description': description}
    action.request.returns = 200
    action.request.retry.wait = 50
    action.request.retry.max_retries = 4
    return {'build_revision': action.response.items()[0][1].items()[0][1][
```

In the example, the request url is set to `http://localhost:{port_num}/api/service/build` and the request method is POST. The body is a JSON map containing three values which are just simple lookups of input values. The action expects the return code to be 200. If it is for example 503, the request will be retried after a while (but at least 50ms). The `post` entry can furthermore contain a Python expression that checks whether a request has succeeded. If this expression is false, the request will also be retried.

When calling the actions, the caller must set the values for `service_id`, `gitref`, and `description`. A call could look like this:

```
@workflow()
def workflow_name(service):
    build_start(service_id=service, gitref='origin/master', description='B
```

This would build origin/master for some service. Other workflows can be called in the same way that actions are called.

SUPPORTED CONTROL STRUCTURES

The workflow language supports simple conditionals, iteration as well as implicit fork-join. These control structures can be combined with each other as well as with actions to create workflows.

do if

The `If` context manager executes an action if the given condition is satisfied. The condition is a Python expression:

```
with If(expr):
    # Workflow command
```

The sequence is only executed if the expression has the value true.

parallel

The `Parallel` context manager executes a list of commands in parallel, waiting for the last one to finish before moving on to the next command after parallel:

```
with Parallel():
  # Workflow command
  # Workflow command
```

iterate

The `Loop` context manager iterates through a list, executing a list of commands for each member of this list:

```
with Loop(list) as var:
    # Workflow command
    # Workflow command
```

Run concurrently

The `LoopAsync` context manager has a variant which allows authors to iterate over a list but run the associated block concurrently.

```
with LoopAsync(list) as var:
    # Workflow command
    # Workflow command
```

The control-flow automatically forks and joins without additional work required. Note that all of the parallel executions must successfully reach the end of the block before execution can proceed.

```
with LoopAsync(list) as var:
    # Workflow command
    # Workflow command
    var = LoopAsync.outputs(expr)
```

When processing items in parallel it is often useful to be able to collect output. To do that you can specify *outputs* by using LoopAsync.outputs. By defining this, the list of provided variables will be available in scope after successful completion of the block. Each variable will map to a dictionary with the value of the expression indexed under the list item.

PHASES

Phases are created using the utility functions phase_{start, complete, fail, cancel, rollback, pause, update}. All these functions take a key representing the phase name and the context as mandatory inputs. Additionally, it is possible provide notification and display_data as optional input. The utility functions return a Phase object, that you can then add signals to.

```
phase = phase_start('build', [service],
                    display_data={'service': service,
                                  'build-revision': upgrade_config.get('ne
phase.add_signal('status', status(service, build_start['build_revision']))
```

WORKFLOW BUNDLES

As of 2017 workflows are to be managed via bundles. Bundles allows simple packaging of workflows which belong together, as well as service (or domain) specific actions which are used by these.

A bundle contains any number of resources (workflows and actions) and metadata which declares the content and the dependencies required by the included resources.

THE BUNDLE MANIFEST

In Workflow language v2, the whole bundle is contained in a Python module (or multiple, if resources are split out). The Python module describes the content of the bundle, as well as some required metadata, such as the version of the bundle.

The bundle lets you declare which of the embedded workflows are intended for execution and which are intended as “library” workflows. It also let’s you specify the version of the *imported* bundles i.e. the bundles required by your workflows. By explicitly providing the version of the bundle you require, you ensure that the workflows you call from your workflows do not change underneath you (similarly to what you would do with any other software to freeze you dependencies)

The format is simple. It’s a basic module docstring (YAML dict) with the following fields:

- name (alpha numeric with underscore)
- description (utf8 string)
- maintainers (list of strings)
- version (can be number, but will be stored as a string)

Resources and imports are automatically added when defined or imported, i.e. it is no more required to define resources and imports in the bundle metadata as you had to in the past. A resource is either an ActionDefinition or a WorkflowDefinition made by decorating a function with the respective decorator. Actions will always be libs, and workflows are libs by default unless you specify it to be main in the decorator. The resource name is the function name, which is used by you when you need to run the workflow or when you want to refer to the workflow from another workflow (see more below)

The type admits *main* and *lib* values, used by uOrchestrate to restrict the number of workflows exposed in the “workflow list” which are the workflows can can be run (see API or CLI). Workflows designated as

main may also be used as libs, but only mains are compiled on upload, and checked for syntax and linking problems. If all libs are used by a main you can be confident that they are syntactically correct, if not, you will have to test it by using the orchestratorman test framework.

The bundle docstring is a simple YAML document:

```
"""
name: some_name
description: this is what my bundle can do for you
maintainers: [
"somebody@uber.com"
]
version: 0
"""
from udeploy_orchestrator.lib.workflow_lang.builder.frontend import (
action,
workflow
)
from udeploy_workflows import chat

@action()
def some_action():
    action.request.url = '{some_host}:{some_port}/{some_endpoint}'

@workflow()
def some_workflow()
    color = some_action()
    chat.message('some_room', 'some message', color)
```

REFERRING TO RESOURCES AND IMPORTS

In order to refer to workflows in your bundles or imports, we've adopted a simple namespacing scheme. Resources in your own bundle may be referenced (called) by simply using their *name* while imports are referenced by bundle and name using '.' as a namespace separator. E.g. *udeploy.upgrade_deployment* where *udeploy* is an import in your bundle (e.g. *import udeploy_workflows.udeploy as udeploy*), and *upgrade_deployment* is a resource in the corresponding bundle. The version of the bundle you import is frozen to the one you're referencing at upload time.

IMPORTING AND USING YAML BUNDLES

If you want to import and use a bundle that hasn't been ported to the new language, you can add it to an *imports* list in the module docstring as a dictionary containing bundle and version (optional, defaults to newest). When wanting to call a resource, you have to use the LegacyCall class, which takes a name

(which has to be a string with the same format of how a call would normally be, e.g. *'udeploy.upgrade_deployment'*) and arguments as keyword arguments, e.g. *varname = LegacyCall('udeploy.upgrade_deployment', arg1=arg1, arg2=arg2)*:

```
"""
name: some_name
description: this is what my bundle can do for you
maintainers: [
"somebody@uber.com"
]
version: 0
imports:
  - bundle: chat
"""
from udeploy_orchestrator.lib.workflow_lang.builder.frontend import (
LegacyCall,
action,
workflow
)

@workflow()
def some_workflow()
    LegacyCall('chat.message', room='some_room', message='some message', c
```

USING OTHER BUNDLES VIA IMPORTS

In order to know which lib (or main) workflows in a given manifest you did not author, you will need to download the bundle or read documentation provided by the maintainers.

MANAGING BUNDLES

The lifecycle of a bundle is fairly straight forward.

- write
- upload
- activate
- run
- rollback (if needed)

Authoring a bundle is reduced to writing and testing your workflow. You might want to initiate the bundle in your current directory which you can do with the following command:

```
uorchestrate-cli bundles initialize-bundle
```

Upload can be as simple as running

```
uorchestrate-cli bundles create-n-upload --python /some/path/to/bundle
```

If you are using orchestratorman to test your workflow (as you should) the *uorchestrate-cli* will already be in your virtual environment. You may want to use the different options to point to the tool to an appropriate environment e.g. staging, dca, or sjc (it's easy to clone and run uOrchestrate locally too).

Activating is a release mechanism, it means to change to *default* version of your bundle. The default is used to determine which version is run by your users by default (your users can always choose to run your workflows by explicitly specifying the bundle version and not relying on the default version). The default is also used by uOrchestrate when others import your bundle in theirs without specifying the version. Activation is done using the command:

```
uorchestrate-cli bundles activate bundle_name version_to_activate
```

Finally, activation means that an uploaded bundle does not take effect before you decide it should. You can upload and test (in production) and then activate it once you are happy with the result. This allows you to change you mind and **rollback** the default version to the previous version if you discover an issue at a later stage. uOrchestrate will keep track of the activation history and all you need to do is call the rollback endpoint or use the *uorchestrate-cli* to rollback.