Beyond Generic Lifecycles: Reusable Modeling of Custom-Fit Management Workflows for Cloud Applications

Conference Paper · July 2018

DOI: 10.1109/CLOUD.2018.00048

6 authors, including:

Merlijn Sebrechts
Ghent University
8 PUBLICATIONS **15** CITATIONS
SEE PROFILE

Gregory Van Seghbroeck
Ghent University
32 PUBLICATIONS **128** CITATIONS
SEE PROFILE

Tim Wauters
Ghent University
101 PUBLICATIONS **602** CITATIONS
SEE PROFILE

Bruno Volckaert
Ghent University
114 PUBLICATIONS **544** CITATIONS
SEE PROFILE

Beyond Generic Lifecycles: Reusable Modeling of Custom-Fit Management Workflows for Cloud Applications

Merlijn Sebrechts*, Cory Johns†, Gregory Van Seghbroeck*, Tim Wauters*, Bruno Volckaert* and Filip De Turck*
* *Ghent University - imec, IDLab, Department of Information Technology*
Technologiepark-Zwijnaarde 15, B-9052 Ghent, Belgium;
Email: merlijn.sebrechts@ugent.be
† *Canonical Ltd, London, SE1 0SU, UK. Email: cory.johns@canonical.com*

***Abstract*—Automated management and orchestration of cloud applications have become increasingly important, partly due to the large skills shortage in IT operations and the increasing complexity of cloud applications. Cloud modeling languages play an important role in this, both for describing the structure of a cloud application and specifying the management actions around it. The TOSCA cloud model standard recently defined *declarative workflows* as the preferred way to specify these management actions but, as noted in the standard itself, this is far from ideal. This paper draws lessons from six years of using declarative workflows in Juju for deploying and managing complex platforms such as OpenStack and Kubernetes in production. This confirms the limitations: declarative workflows are inflexible, hard to reuse, and allow for related components to become silently incompatible. This paper proposes the *reactive pattern* to solve these issues by enabling the creation of *emergent workflows* using declarative *flags* and *handlers*, which can be easily grouped into reusable layers. After more than two years of using this pattern in production as part of our *charms.reactive* framework, it is clear that it enables reusability and ensures compatibility: 67% of reactive charms share parts of the management workflow and 73% of reactive charms share a relationship workflow.**

I. Introduction

Due to the large skills shortage in IT operations [1] and the increasing complexity of cloud applications [2], automated management and orchestration of cloud applications have become increasingly important. The OASIS *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [3] is a front-runner in this field: a standard with large backing from both the industry and academia. It provides a specification to create self-contained cloud models that describe the structure of a cloud application in a topology model, as well as the surrounding management and orchestration processes in a workflow model. The order in which these processes are to be executed is either explicitly defined in an imperative workflow model or implicitly in a declarative workflow model. The latter type is of great importance to this research because it allows capturing the knowledge on how to manage a cloud application in a reusable way, which is crucial to solving the skills shortage and the complexity of cloud applications [2]. Consequently, the Juju cloud modeling language [4], which closely resembles the TOSCA standard, has been using declarative workflows since its inception. Section II introduces these concepts and related work in greater detail.

However, as noted in the TOSCA specification itself, declarative workflows are inherently inflexible: every workflow needs to adhere to a single lifecycle defined by the cloud modeling language. The real-world implications of this issue have become painfully clear during 6 years of managing complex platforms in production with Juju. Section III reflects on this experience and identifies the main shortcomings of the declarative approach.

This paper proposes the reactive pattern in Section IV to address the limitations of declarative workflows. Specifically, the reactive pattern allows creating custom-fit workflows, not limited by lifecycles, and enables more fine-grained reuse than declarative workflows. Its implementation is discussed in Section V and the evaluation of two years of production use is discussed in Section VI. The reactive pattern has resulted in widespread code reuse and increased compatibility. Consequently, it forms a great battle-tested foundation for improved workflow support in the TOSCA standard.

II. Background and Related Work

A. Cloud Modeling Languages

Model-based management of cloud applications is ubiquitous [2] and can be traced back to the 1995 paper of Burgess et al. where the idea of converging towards a predefined end-state was proposed [5]: a system administrator declaratively specifies what the desired end-state of the cloud application is, and an orchestrator interprets that specification and iteratively executes the required actions to get the application into that state. This idea has evolved over the years and has resulted in the creation of topology-based cloud modeling languages to enable better portability and reusability [3]: the model of the cloud application consists of a graph of components which are connected by their dependencies. Each component is a self-contained description of part of the cloud application and new models can be created by rearranging the components thus making each component reusable.

As said in the introduction, OASIS TOSCA [3] provides a specification to create self-contained cloud models that describe both a) the structure of a cloud application in a topology model, and b) the management and orchestration processes surrounding it in a workflow model. The topology consists of a number of **nodes** connected to each other using **relationships**. Relationships denote dependencies between two nodes. A web app, for example, might have a relationship with a database to denote that the web app uses the database for storage. The types of relationships possible between nodes are defined by their **node type** in the form of requirements and capabilities: a relationship connects a node that requires a certain dependency to a node that provides the same dependency. The structure of the workflow model has changed over the past few years. TOSCA 1.0, released in 2013, does not enforce a specific workflow language but favors BPMN. TOSCA Simple YAML profile 1.0, released in 2016, lost the ability to specify a workflow and only with the 2018 release of TOSCA Simple YAML profile 1.1 have workflows been included again, now in two forms: imperative and declarative workflows [6].

B. Imperative Workflows

Imperative workflows are often depicted as a set of *activities* linked by a *control flow*. Each activity is a piece of work that forms one logical step of a process. The control flow describes the order in which the individual activities are performed. These can be represented as a directed graph where each node represents an activity and the vertices describe the control flow. In an imperative workflow, the order of execution is explicitly defined as part of the workflow definition e.g. as a flow diagram. Many IT service management practices such as Information Technology Infrastructure Library (ITIL) [7] use imperative workflows as high-level descriptions of IT business processes. Thus, the use of such workflows in cloud modeling languages makes it easy to align IT services with business needs. The imperative workflows in a cloud model define how to deploy, manage and undeploy a topology. Each activity is a management action such as "install MySQL", and the control flow describes when each management action needs to be performed. During deployment, the orchestrator executes the workflow step by step until the entire topology is deployed.

The downside of using imperative workflows is that they are defined for a specific topology instead of for one component. Thus, when the topology is changed, the workflow needs to be recreated. This is an inherent limitation of imperative workflows: changing a constraint in an imperative workflow description requires a complete rewrite of the control flow [8]. Having to rewrite the workflow every time a component in the topology changes, goes against the modular nature of topology-based cloud modeling languages. Wagner et al. propose to define the imperative workflows on the level of the individual nodes, and interconnect the workflows of all the components in a topology using a choreography. This approach however still requires manual creation of the choreography because the orchestrator cannot know how the individual workflows should be connected [9].

C. Declarative Workflows

Declarative workflows provide optimal reusability: the declarative workflow for each component of a topology is contained inside the description of that node. Adding the node to the topology will automatically add all the management activities to the global workflow. This is because the control flow is not explicitly specified but rather implicitly derived from the constraints of each activity. In TOSCA, the constraints specify which lifecycle phase the activity is part of, for example *installing*, *configuring* or *starting*. It is then up to the orchestrator to decide when each lifecycle phase for each component needs to be executed, so the orchestrator "generates" an imperative workflow by merging all activities from all nodes in the topology [10]. The lifecycles themselves are however defined by the orchestrator which presents the biggest drawback of declarative workflows in TOSCA: workflows are limited to the states and transitions defined in the orchestrator's lifecycle.

Furthermore, this also restricts the types of dependencies possible between nodes. TOSCA specifies a number of normative relationships that each carry specific meaning about the dependency between related nodes. As an example, the *DependsOn* relationship means that the target node needs to be started before the source node is created. This directly translates into how the orchestrator connects the declarative workflows of these two components: the deployment workflow of the source node is executed when the target node reaches the *started* state. As a result, declarative workflows can only model dependencies which are explicitly defined by the orchestrator. The current TOSCA specification, for example, does not support circular dependencies [11], i.e. dependencies where the control flow jumps back and forth between two nodes multiple times.

III. Lessons learned: history of declarative workflows in Juju

Juju [4] is a cloud modeling language and orchestrator created by Canonical that closely resembles the TOSCA standard. Since its inception in 2012, Juju has been used in production to deploy big software such as OpenStack and Big Data clusters [12], and is at the core of BootStack and the Canonical Distribution of Kubernetes.

A Juju **charm** is similar to a TOSCA node type: it represents one service in the cloud application and defines which relationships it supports using **requires** and **provides** statements. Juju also uses declarative workflows: the orchestrator defines a number of lifecycle stages such as *install*, *start* and *config-changed*, and executes a program called a **hook** during each lifecycle stage. A hook is a workflow activity

and its name defines which lifecycle transition it performs. Thus, the Juju orchestrator decides *when* hook code gets executed, and the charm developer decides *what* operation should be performed. A deployed instance of a charm is called a **unit** and adding a unit to a model automatically adds the hooks of its charm to the topology-wide declarative management workflow. This approach resulted in a number of issues.

The lifecycle provided by Juju does not match the actual lifecycle of the managed services. Juju's provided lifecycle is too simple for most services, which require many more lifecycle phases and transitions. This results in a frequently used anti-pattern where all lifecycle stages execute the exact same code which implements a rudimentary state machine with *if-then* statements that mimics the real lifecycle of the application. The state machine figures out which actual lifecycle stage the application is in, and executes the required actions. Expanding the lifecycle of Juju's orchestrator is not a good solution because each service requires its own specialized lifecycle so a one-size-fits-all lifecycle is simply not sufficient. Moreover, it should not be up to the orchestrator to define what the lifecycle of a service is, this should be defined by the service.

Reusing parts of the lifecycle of a single service is difficult. Many services share components, and many lifecycle steps are the same for multiple services. Many services are installed using the distribution package manager, for example, and need to be updated when security fixes are released. Encapsulating this functionality in a way that it allows being reused in other lifecycles is not possible. Over the years, a number of charm helper libraries have been created in order to increase code reuse, but the issue with a library is that it only encapsulates *how* to do a certain lifecycle action, not *when* that action should be performed.

The relationship lifecycle provided by Juju does not match the actual relationship lifecycle of the managed services. As explained in Section II, the use of declarative lifecycles restricts the types of dependencies between two nodes to the ones supported by the orchestrator. Juju supports only one type of dependency in which the lifecycles of both units run concurrently. After the *start* hook of both units, the relationship lifecycle runs and the units exchange configuration values. In reality, however, many services require knowing configuration values, such as the IP address of a database, before starting. This causes developers to create a state machine that completely ignores hooks such as *config-changed* and *start*, and waits until the *relation-changed* hook to actually configure and start the service. This results in a discrepancy between the state that the orchestrator thinks a service is in, and the actual state a service is in.

Silent incompatible relationships. Because of the previous issue, the relationship lifecycles are actually implemented by the charm, instead of by the orchestrator. The orchestrator has therefore no way of verifying that two ends of a relationship actually implement a compatible lifecycle. This has resulted in many semi-compatible charms that implement the same relationship according to the orchestrator, but differ in subtle incompatible ways in practice.

IV. The reactive pattern

This paper proposes the reactive pattern as a fundamentally new approach to managing services using cloud modeling languages. Such pattern allows the creation of flexible and reusable emergent workflows that manage the entire lifecycle of a modeled cloud application including dependency management, initial deployment, second day operations, topology changes and node type upgrades. Although it was initially created for the Juju cloud modeling language, the pattern itself is generic enough so that it can be used in different cloud modeling languages such as TOSCA or as the *service engine* in a Distributed Service Orchestrator [13]. This section gradually introduces all the primitives of the reactive pattern and explains their role and how they address the shortcomings of declarative workflows.

Just like with regular declarative workflows, the actual management operations are encapsulated in activities which are part of the node definitions. The novel part of this pattern is how the control flow gets created: the orchestrator does not define a lifecycle, it only defines a number of events. Developers create custom event-based workflows for each service and hook them into these events. These workflows are created using constraint-based modeling [14]: each activity defines a set of constraints which need to be satisfied in order for them to execute. Unlike approaches like DECLARE [15], these constraints are not explicitly tied to events regarding the execution of other activities. Rather, the constraints use semantic flags that can also represent a number of different types of events such as the arrival in a certain state, a change in the topology and service events e.g. a crash.

A. Handlers and Flags

The reactive framework is based on the idea of handlers reacting to flags. Handlers are the activities of the workflow: pieces of code that perform management actions on the cloud service. The control flow, the order in which handlers get executed, is driven by flags: each handler defines which flags it reacts to, i.e. which flags need to be set and/or unset for the handler to execute. The framework executes a handler when its preconditions are met, during which it modifies the service, and can set and clear flags. This triggers other handlers to run, until there are no more handlers whose preconditions are met. In this sense, the reactive pattern uses constraint-based modeling with arbitrary events.

The power of a flag is that it can represent almost anything, from internal state such as "the service is running" and "disk utilization is critically high" to topology modifications

such as “a new relation is established” or “this node has been removed”. The following is a non-exhaustive list of what semantic meaning a flag might hold.

- **Lifecycle Stage:** The orchestrator itself defines a number of flags that represent which lifecycle transition it requests such as *install*, *config-changed*, and *stop*. These are the reactive pattern’s counterpart to the hooks and lifecycles of declarative workflows.
- **Service state:** Developers can define a number of flags that represent low-level state of the service such as *“the webserver is installed”* and *“the SSL certificate is registered”*.
- **Service events:** A flag can also represent events that happened in the past, and that might need to be handled, such as *“the service has crashed”*, which might require notifying a system administrator, even when the service has successfully been restarted.
- **Topology state and events:** Flags can also represent the topology or changes to it. A flag can indicate that a new relation was created in the model or that a related service in the topology has entered a certain state.
- **Day 2 operations:** A flag can signal that a backup is requested, that an update is required, or that an SSL certificate needs to be renewed.

Note that not all flags need to be set by the handlers themselves. The operating system itself can set a flag when a service crashes or when a certain time has passed, and the orchestrator sets flags to indicate which lifecycle stage the application is in and what the current state of the topology is. This for example allows the workflow to hook into the lifecycle provided by the orchestrator.

*Definition 1: A **handler** is an activity that manages a cloud resource, accompanied by a set of preconditions that, using flags, states when that activity should execute.*

*Definition 2: A **flag** is a boolean identified by a unique string that is a semantic representation of an event to be used in a handler’s preconditions.*

Figure 1 shows a custom workflow that emerges from a set of handlers and their preconditions. Each activity is a handler and the control flow emerges from their preconditions. The orchestrator starts the workflow and sets the appropriate flags when the domain name config is set and changed. The operating system itself sets flags when 20 days have passed and when an update is available to the packages. The pseudocode for the handlers and their preconditions is available online [16].

The resulting workflow shows that some activities such as deploying the web app and registering the SSL certificate can be executed in parallel. This however only regards the control flow dependencies, not the actual dependencies of the activities themselves: it does not matter which action is run first, but the actions might not be able to run at the same time.

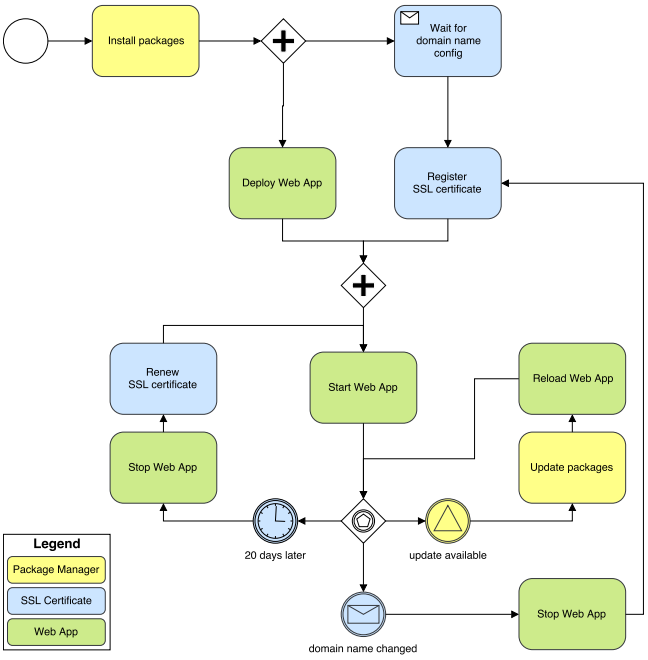

Figure 1. This workflow emerges from the handlers, their preconditions and their flags. Each handler is represented by an activity. The “stop web app” handler is represented by two different activities because the next activity depends on which activity was executed previously to “stop web app”. The handlers are colored according to which aspect of the service they manage.

In summary, flags and handlers allow the construction of emergent workflows that hook into and expand the lifecycle provided by the orchestrator. Because the emergent reactive workflow hooks into the lifecycle provided by the orchestrator using flags, it can leverage the existing techniques to combine the declarative workflows of multiple components into a single workflow that manages the entire cloud application. Just like with declarative workflows, reactive emergent workflows are shipped as part of the node type of a component. When that component gets added to a model, the accompanied workflow will be hooked into the model’s global workflow. This approach thus eliminates the downsides of TOSCA’s imperative workflows while allowing for a greater level of flexibility.

B. Scope

A big advantage of cloud modeling languages stems from the separated scope between nodes. A node can only access information about another component if there is an explicit relationship that shares that piece of information. This property is also present in the reactive pattern. Flags in the reactive pattern are unit-scoped: each instance of a node type has its own set of flags. Handlers themselves are node-type scoped: all instances of a single node-type have the same set of handlers. This means that the emergent workflow of each unit will be the same, but the current position in the

workflow might be different. The web app example from Figure 1 is a single service that consists of a number of components: an SSL encrypt certificate, a webserver and a web app. When the web app scales out into multiple instances, each instance will have its own set of flags, but the handlers will be the same over each unit.

C. Layers

As mentioned previously, the web app example service can be divided into three components: the webserver, the SSL certificate and the web app. The emergent workflow in Figure 1 shows each activity colored based on which component it manages. As one can see, it is not possible to divide the emergent workflow into three sub-workflows, one for every component. With the reactive pattern, this becomes possible since the workflow itself is just an emergent property from the handlers and their preconditions: it only exists at runtime. At design time, the handlers can be divided into arbitrary groups because there are no explicit dependencies between activities: the only dependencies are implicit with the flags as an intermediary.

In the reactive pattern, each set of grouped handlers is called a "layer". Figure 2 shows the handlers from the web app example divided into three layers. Each layer contains the handlers that manage a specific part of the service: the web app, the SSL certificate and the webserver. Adding a layer to a node type results in the handlers of that layer being added to the emergent workflow of that node type. This thus greatly improves the reusability and allows developers to focus on the components that they are an expert in, instead of having to code the entire service. In a sense, this is aspect-oriented programming: each layer contains the activities that manage one specific aspect of the service. The flags define the "cut points", the points in which the aspects get injected into the program code.

A layer is one level below a TOSCA node type: multiple layers combined form one node type. From the viewpoint of the orchestrator, all layers of the same node type share the same lifecycle. The orchestrator does not coordinate the lifecycles of each layer individually since the control flow of a service is defined by the flags and the preconditions of handlers. This also has the advantage that a model designer does not come into contact with layers, the model designer only sees a single node and layers are an "implementation detail" of the node. Finally, this makes it possible to use layers without needing any changes to TOSCA itself since layers are "compiled" into a TOSCA node type, and the orchestrator only interacts with the node type.

In order for layers to be reusable, it is important that each layer defines what the semantic properties are of each flag. Some flags might be for internal use in a layer itself, while other flags are to be used by other layers to signal this layer or to use in the preconditions of their handlers. It is also important to define how these flags will be managed

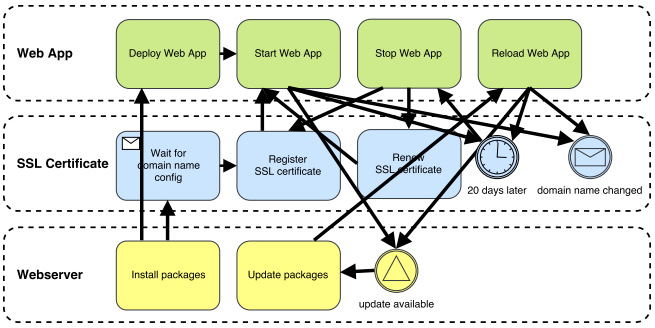

Figure 2. Since the dependencies between handlers are implicit, the handlers can be grouped by which aspect of the service they manage, even though the emergent control flow goes back and forth between the layers in an erratic manner.

by a layer: whether or not flags will be automatically set or removed when certain conditions happen.

Furthermore, it is important to avoid conflicts between layers. An example of a conflict is when two handlers, A and B, react to the same flag, *config.changed*, and both clear that flag during execution. Though it is not immediately obvious, this results in non-deterministic behavior because the workflow is executed sequentially and the preconditions of flags are rechecked after execution of every handler. If handler *A* runs first, it will clear the *config.changed* flag and handler *B* will not even run. This in itself is wanted behavior: a handler is never allowed to run if its preconditions are not met. If handler *A* clears a flag during its execution, it signals that the event that set the flag is *handled*, indicating that no other handlers which handle the same event should run.

It is however entirely possible that multiple layers handle the same event. Layers do not have explicit dependencies on each other, so a handler cannot know, at the time of clearing the flag, if the event is actually handled by every layer. Triggers are used to avoid such conflicts.

*Definition 3: A **trigger** is a causal, directed dependency between two flags that sets or clears a flag immediately when the other flag is set or cleared.*

Immediately in this context means that when a flag changes, the execution of handlers is paused until all triggers are processed.

Using a trigger, a layer links the *config.changed* flag to a custom flag for example *layer-a.config.changed*, such that the custom flag is set immediately after *config.changed* is set. This custom flag is meant for internal use in that layer only and is thus prefixed with the layer's name. Since a trigger is one-way, the custom flag will *not* be cleared when another layer clears *config.changed*. The developer can thus safely use the custom flag in the preconditions of a handler without having to worry that it will be cleared by another layer before the handler has a chance to run.

D. Interface layers and Endpoints

Much like layers contain reusable handlers to manage individual services, interface layers contain reusable handlers to manage the relationship *between two nodes*. Unlike regular layers, a single interface layer contains handlers for two nodes because an interface layer implements both sides of the relationship. An interface layer is thus a declarative model of the communication between two nodes. This again makes it possible for the orchestrator to know whether a relationship between two nodes is possible: if two nodes share the same interface layer, the relationship is possible.

Relationships in TOSCA serve two purposes during orchestration: they are used to connect the control flow of two nodes in a way that the dependency is resolved, and they are used to exchange information such as IP addresses in order to configure both services correctly.

The control flow of a single component in the reactive pattern is defined by the flags. It is however not desirable to share all the flags of one node with another node, since that creates deep dependencies between nodes, loses the modularity and limits the reusability of a layer. Thus, all sharing of state and data happens explicitly by the handlers of the interface layer so that the dependencies between the implementation of two nodes are limited to the handlers of the interface layer, which is not an issue since the interface layer is already shared between two nodes.

Each time the relationship and its data gets changed, the orchestrator notifies the interface layer by setting flags that denote lower-level relationship events such as **endpoint.x.joined**, when a relationship is established, **endpoint.x.changed**, when relationship data changes, and **endpoint.x.departed** when a relationship is removed. These lower-level flags are the internal API of an interface, they are only to be used by relationship handlers which react to these events, read and write relationship data and manage higher-level flags. Regular layers should only react to the higher-level flags since those are regarded as the "external API" of the interface.

As an example, in the MySQL case of an interface which is used to connect a node that provides a MySQL database to a client, the MySQL side will have the higher-level flags **table.requested**, to denote that a client has requested the creation of a table, and **user.requested** which is set when a client requests the creation of a user account. The layer that manages the MySQL database will contain a number of handlers that react to these flags to create the requested tables and users, and will call back to the endpoint object to notify that the requests are executed.

Endpoints are the key to the second purpose for relationships: sharing information between nodes. An endpoint is an object that represents one side of the relationship. It publishes and reads the relationship data to communicate with the endpoint at the other side and it translates the raw relationship data into high-level objects to be used by handlers.

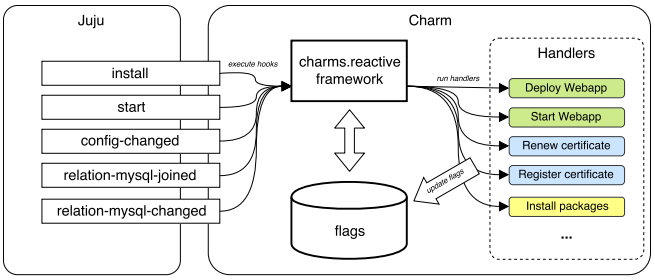

Figure 3. The architecture of the charms.reactive framework: when the orchestrator executes a hook, the reactive framework initiates and runs the handlers whose preconditions are true.

V. Implementation

A. The "charms.reactive" framework

The *charms.reactive* framework is our implementation of the reactive pattern built on top of Juju's declarative workflows[1]. It is written in Python 3. Handlers are decorated python functions or executable files that implement the *external handler api*.

A reactive charm is built from layers. Each layer is a directory with a number of handlers and a *layer.yaml* file that holds metadata such as the name of the layer, and the dependencies of this layer i.e. what other layers this layer uses. The *charm build* tool is used to *compile* a layer and its dependencies into a charm. It downloads all the dependencies from the *layer-index* [2], merges all the layers and packages the result into a deployable charm.

Figure 3 shows the architecture of the framework. Much like the state machines mentioned in Section II, the framework ties into the Juju hooks so that any hook simply executes the framework. It then decides which handlers to run based on the preconditions of the handlers. When there are no more handlers to run, the framework exits the hook. According to the Juju orchestrator, a reactive charm is thus no different from a regular charm.

At the start of each hook, the reactive framework loads the flags from persistent storage, sets and clears the managed flags based on the hook and the information from the orchestrator, and starts to execute the handlers whose preconditions are met, as shown in Listing 1. A handler is considered *matching* if the preconditions are true and the handler did not yet run in the current hook or the flags referenced in its preconditions have changed since the last time it ran.

All handlers on the same unit are executed sequentially, even if the emergent control flow allows concurrent execution, since the reactive pattern does not provide a way for handlers to define whether or not two handlers can actually

[1]https://charmsreactive.readthedocs.io/en/latest/

[2]https://github.com/juju/layer-index

Listing 1. Pseudocode for a run of the reactive framework

```
set and clear managed flags
add matching handler to the queue

while queue is not empty:
    for each handler in queue:
        run handler
        if handler failed:
            revert flag changes
            fail hook
        remove handler from queue
        remove not matching handlers from queue
    add matching handler to queue
    if max iterations reached:
        revert flag changes
        fail hook
```

run concurrently. This is however still an improvement over TOSCA's declarative workflows, where even the activities of related nodes are run sequentially. The order in which the reactive framework runs handlers when multiple handlers match is undefined but deterministic: every run will result in the same order, but a charm developer should not rely on any order.

From Juju's standpoint, a hook is transactional: if a hook fails, Juju will rollback the state changes of that hook and try the hook again. This fixes transient failures. For this reason, the reactive framework itself also rolls back all changes to flags when a handler fails. This protects against transient failures as shown by Wettinger et al. [17]. Juju's approach to this does not eliminate the need for idempotency because the orchestrator does not roll back the actual changes to the service so the service might be in an inconsistent state, and handlers might run multiple times when the hook is retried.

B. Lessons learned

In the initial version of the *charms.reactive* framework, **flags were called *states***, which confused developers because they thought they were building finite state machines (FSMs). It is possible to build an FSM with the reactive pattern, but the pattern is a lot more powerful since it also allows event-based programming. Flags are a much more neutral term which does not imply any specific model of computation. As an example, some of the relationship flags represent events instead of states. The *relationship.{name}.joined* flag is a state: it is set when the relationship reaches the *joined* state, and is cleared when the relationship leaves that state. However, the *relationship.{name}.departed* flag represents an event: it is set every time a unit departs from a relationship and is manually cleared by a handler that "handled" the departure. In contrast, if this event were a state, it would be set when the first unit departs a relationship and never be cleared since that unit remains departed, even when that departure has been "handled".

In the current implementation of the reactive framework, handlers whose preconditions are true are re-executed in every hook. This however turned out to be counter-intuitive for developers, especially new developers without experience writing non-reactive charms. Since the reactive framework is a layer on top of Juju's declarative workflow, and hooks are thus hidden, having such a reliance on their lifecycle adds unnecessary complexity for developers. It is however not possible to change this behavior currently because this will break backwards compatibility.

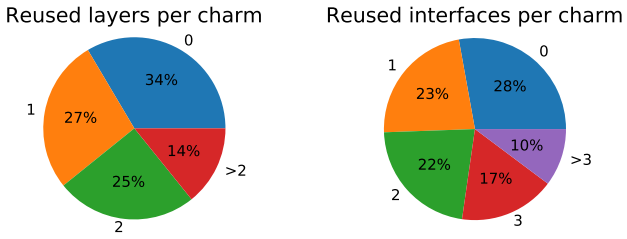

Figure 4. Number of reused layers and interfaces per charm.

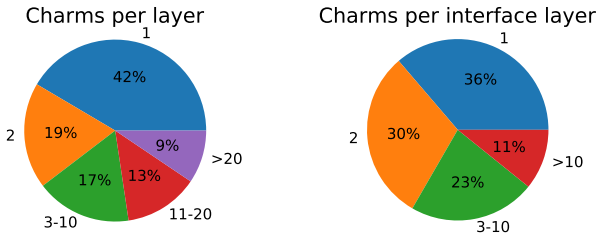

Figure 5. Number of times each layer is used.

VI. In practice

This section shows the results of using the reactive pattern in Juju for more than two years, since our implementation has become available. The results shown in this section are obtained using the public Juju charm store api [3]. A cached copy of the data and the full code to download and process it is available on Github [16].

The charm store contains a total of 529 active charms: charms that have been downloaded in the last month. Of those, only 176 or 33% use the reactive framework. The relatively young age of the framework plays a big role in this: many charms were built before the reactive framework, and porting these charms to the reactive framework is not trivial, since it requires a complete rewrite of the charm code.

Figure 4 shows the number of *reused layers and interfaces* per charm, i.e. the number of layers and interfaces which are also used by another charm. This shows that the reactive framework has indeed made it possible to reuse workflow code across charms: two-thirds of actively-used reactive charms share at least one layer with another charm. This is an incredibly high number compared to the workflows in TOSCA, where node templates simply can not share any

[3]https://github.com/juju/charmstore/blob/v5-unstable/docs/API.md

workflow code. However, there is a lot of unused potential because 41% of layers are used in only one charm as shown in Figure 5.

Figure 4 also shows that interface layers are also heavily reused: 73% of charms use at least one interface layer that is shared with another charm, which improves the compatibility of charms implementing the same interface. However, not all interface layers have this benefit: 36% of interface layers are only used once, as shown in Figure 5. This is because of the high number of non-reactive charms: these interface layers are used to connect reactive charms to non-reactive charms.

VII. Conclusion

Six years of managing cloud applications in production using declarative workflows shows that their inflexibility limits their usefulness. Moreover, they don't provide enough opportunity for code reuse, causing duplicated effort and allowing connected workflows managing different services to become silently incompatible. The reactive pattern proposed in this paper addresses these issues by allowing declarative specification of workflows that match the actual lifecycles of the services and by enabling aspect-based grouping of workflow activities into reusable layers.

The results of two years of production use show the reactive pattern's benefits: 67% of reactive charms use shared layers and 73% of reactive charms use shared interfaces. This shows that the reactive pattern solves the issues of declarative workflows and even though it originated from the Juju ecosystem, it is generic enough so that it can form the basis for improved workflow support in other cloud modeling languages such as TOSCA.

Acknowledgment

This work was supported by the Research Foundation Flanders (FWO) under Grant n G059615N - "Service-oriented management of a virtualised future internet".

Special thanks to Alex Kavanagh and Stuart Bishop for many insightful discussions on the reactive pattern and their contributions to the charms.reactive framework.

References

[1] D. Russel and M. Chuba, "Top Challenges Facing I&O Leaders in 2017 and What to Do About Them," Gartner, Tech. Rep. G00324370, Feb. 2017.

[2] D. Weerasiri, M. C. Barukh, B. Benatallah, Q. Z. Sheng, and R. Ranjan, "A Taxonomy and Survey of Cloud Resource Orchestration Techniques," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 26:1–26:41, May 2017.

[3] TOSCA Technical Committee, "OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) Technical Committee | Charter," Dec. 2013. [Online]. Available: https://www.oasis-open.org/committees/tosca/charter.php

[4] "Ubuntu Juju: Operate big software at scale on any cloud," https://jujucharms.com/. Accessed October 3, 2017. [Online]. Available: https://jujucharms.com/

[5] M. Burgess and O. College, "Cfengine: a site configuration engine," in *in USENIX Computing systems, Vol*, 1995.

[6] M. Rutkowski and L. Boutier, "TOSCA Simple Profile in YAML Version 1.1," Jan. 2018.

[7] Cabinet Office, *ITIL Service Strategy 2011 Edition*. Norwich: The Stationery Office, 2011.

[8] R. R. Mukkamala, "A Formal Model For Declarative Workflows: Dynamic Condition Response Graphs," PhD Thesis, IT-Universitetet i Kbenhavn, Denmark, 2012.

[9] S. Wagner, U. Breitenbcher, O. Kopp, A. Wei, and F. Leymann, "Fostering the Reuse of TOSCA-based Applications by Merging BPEL Management Plans," in *Cloud Computing and Services Science*, Apr. 2016.

[10] D. Calcaterra, V. Cartelli, G. D. Modica, and O. Tomarchio, "Combining TOSCA and BPMN to Enable Automated Cloud Service Provisioning," Feb. 2018, pp. 187–196.

[11] T. A. Lascu, J. Mauro, and G. Zavattaro, "Automatic deployment of component-based applications," *Science of Computer Programming*, vol. 113, pp. 261–284, Dec. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642315001409

[12] K. Tsakalozos, C. Johns, K. Monroe, P. VanderGiessen, A. Mcleod, and A. Rosales, "Open big data infrastructures to everyone," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec. 2016, pp. 2127–2129.

[13] M. Sebrechts, T. Vanhove, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck, "Distributed Service Orchestration: Eventually Consistent Cloud Operation and Integration," in *2016 IEEE International Conference on Mobile Services (MS)*, Jun. 2016, pp. 156–159.

[14] M. Pesic, M. Schonenberg, N. Sidorova, and W. M. van der Aalst, "Constraint-based workflow models: Change made easy," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, 2007.

[15] M. Pesic, H. Schonenberg, and W. M. P. v. d. Aalst, "DECLARE: Full Support for Loosely-Structured Processes," in *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, Oct. 2007.

[16] Merlijn Sebrechts, "Code and results of reactive pattern." May 2018, original-date: 2018-05-07T12:37:20Z. [Online]. Available: https://github.com/IBCNServices/reactive-pattern-results

[17] J. Wettinger, U. Breitenbcher, and F. Leymann, "Compensation-Based vs. Convergent Deployment Automation for Services Operated in the Cloud," in *Service-Oriented Computing*, X. Franch, A. K. Ghose, G. A. Lewis, and S. Bhiri, Eds., Nov. 2014.