Model-Based Testing of Kubernetes Pod Lifecycle Management

Abstract

This document describes the automated testing of Kubernetes Pod lifecycle management using a model-based testing (MBT) approach. To represent the lifecycle states and the transitions of Kubernetes Pods, a finite state machine (FSM) model was developed using Graphwalker. For test generation based on the model, the Model-TestRelax (MTR) framework was used. All of the files in connection with this project can be found under this repository: `https://github.com/annasz11/kubernetes-mbt`.

The testing environment was set up on Minikube, with `kubectl` serving as the primary command-line tool to interact with the Kubernetes API. Adaptation code written in Python was integrated to translate generated test cases into actual commands executed on the Kubernetes cluster, bridging the gap between model-based tests and real-world execution.

The document covers the following areas:

- **Introduction to Kubernetes**: A foundational overview of Kubernetes and its architecture, including critical components that facilitate container orchestration, such as the API server, nodes, Pods, and control plane.
- **Tested Features of Kubernetes**: Detailed analysis of each Pod lifecycle state, explaining how they can be tested. Specific attention is given to interaction patterns with `kubectl` and the API server, with UML diagrams illustrating key transitions.
- **Test Generation with MTR**: A discussion on how tests are generated using Model-based Testing (MBT) with the MTR tool, demonstrating how the FSM model translates into executable test cases.
- **Adaptation Code and Test Execution**: This section describes the adaptation code used to interface with Kubernetes, as well as the process of executing and evaluating the tests in a live Kubernetes environment.

Contents

1 Introduction to Kubernetes

Kubernetes, also known as k8s, is an open-source platform for automating the deployment, scaling, and management of containerized applications [2]. Originally developed by Google and now maintained by the Cloud Native Computing Foundation, Kubernetes has become a critical tool in modern software infrastructure, allowing organizations to manage large-scale, distributed applications with resilience and efficiency.

Kubernetes provides tools to orchestrate containers, enabling developers to focus on code rather than managing infrastructure. By automating tasks like deployment, scaling, and fault-tolerance, Kubernetes allows applications to run reliably in clusters across multiple machines, improving scalability and resilience.

1.1 Core Concepts and Components

Clusters A Kubernetes cluster is a collection of machines, or nodes, managed as a single unit. This structure allows workloads to be distributed and balanced across multiple nodes, providing high availability and fault tolerance. Clusters can include physical servers, virtual machines, or cloud-based resources.

Nodes Each cluster consists of nodes, which are individual machines where applications run. There are two main types of nodes:

- Control Plane: This node manages the cluster and coordinates tasks, ensuring that desired states are met. It contains several components:
  - API Server: The API Server serves as the gateway for communication with Kubernetes. It provides a RESTful interface where users and automated tools interact with the cluster.
  - Controller Manager: Ensures that the current state matches the desired state by monitoring resources and managing replication, scaling, and health checks.
  - Scheduler: Assigns workloads (Pods) to nodes based on resource needs and availability.
  - etcd: A distributed key-value store that Kubernetes uses to store all data related to the cluster, including configuration and state.
- Worker Nodes: These nodes handle application workloads and contain the actual running containers managed by Kubernetes.

1.2 Kubernetes Objects

Kubernetes uses objects to represent and manage resources. Some core Kubernetes objects include:

Pods A Pod is the smallest deployable unit in Kubernetes and represents one or more containers that share networking and storage resources [3]. Pods encapsulate the application container(s), along with networking configurations, and serve as the basic building block for Kubernetes applications. Each Pod follows a distinct lifecycle, and its state depends on the type of workload it is associated with. Here are the main states that a Pod can transition through:

- **Pending**: This is the initial state when a Pod has been created but is not yet running on any node. The Pod remains in the `Pending` state while waiting for the Kubernetes Scheduler to assign it to a node based on available resources and other constraints. Once assigned, the Pod will transition to `Running` if all initial setup checks are successful.
- **Running**: In this state, the Pod has been scheduled to a node, and at least one of its containers is active and operational.
  - For **Deployments** and other long-running workloads, Pods in the `Running` state are expected to remain active and available for the duration of the Deployment. The Deployment controller ensures that if a Pod in `Running` fails, it will be automatically restarted or replaced to meet the desired replica count.
  - For **Jobs** and **CronJobs**, the `Running` state is typically temporary and ends when the task completes.
- **Succeeded**: This is a terminal state indicating that the Pod has successfully completed its task. This state primarily applies to Pods created by Jobs, where the containers within the Pod have exited successfully without any errors. Pods in the `Succeeded` state are not restarted by Kubernetes.

- **Failed**: This terminal state indicates that one or more containers in the Pod have exited due to errors or failures. Like `Succeeded`, `Failed` primarily applies to Job-related Pods, which will not restart once they reach this state. However, for Pods managed by Deployments, a failure typically results in the Pod being replaced or restarted to maintain availability.
- **Terminating**: The `Terminating` state represents the phase when a Pod is in the process of shutting down. This state can apply to both Deployment and Job Pods and is triggered by a user action (such as `kubectl delete pod`) or by the Kubernetes system when scaling down a Deployment. While in `Terminating`, Kubernetes stops all running containers and removes the Pod from the node.

In addition to these main states, our testing will also involve examining other transient states such as `CrashLoopBackOff` and `ImagePullBackOff`, which reflect specific issues encountered during the Pod lifecycle.

Deployments A Deployment is a higher-level controller that manages Pods, ensuring they are available, up-to-date, and replicable. Deployments automate scaling, updates, and rollbacks, making it easy to manage multiple instances of an application.

Services Kubernetes Services provide stable endpoints to Pods, enabling consistent access even as Pods are created or terminated. Key service types include:

- ClusterIP: Only accessible within the cluster.
- NodePort: Opens a specific port on each node to make the service accessible outside the cluster.
- LoadBalancer: Integrates with cloud provider load balancers for external access.

ConfigMaps and Secrets ConfigMaps store non-sensitive data configurations, such as environment variables, that can be injected into Pods. Secrets provide a way to store sensitive information, like passwords and API keys, which are kept encrypted and secure.

Namespaces Namespaces create isolated virtual clusters within a single physical cluster. This feature is essential for large projects, providing logical separation and access control.

1.3 Workloads and Scaling

ReplicaSets A ReplicaSet ensures a specified number of Pod replicas are running at all times, improving availability and fault tolerance. If a Pod crashes, the ReplicaSet automatically recreates it to maintain the desired number.

Horizontal Pod Autoscaling Kubernetes supports autoscaling to adjust the number of Pods dynamically based on metrics such as CPU or memory usage. This feature allows applications to scale up during high demand and scale down to save resources during low demand.

1.4 Kubernetes API and kubectl

Kubernetes API The Kubernetes API is the central interface for interacting with the cluster [4]. Every action within Kubernetes—creating, modifying, or deleting resources—goes through the API. Users can programmatically manage resources by sending HTTP requests to the API, which serves as the cluster's command center.

kubectl `kubectl` is Kubernetes' command-line tool that allows users to interact with the cluster [5]. It's essential for managing, inspecting, and troubleshooting Kubernetes resources. Commands issued by kubectl are translated into HTTP requests to the API, which then processes these requests. Examples of basic commands include:

- `kubectl get pods -n <namespace>`: Lists Pods in a namespace or cluster-wide. It translates to `GET /api/v1/namespaces/{namespace}/pods/`.
- `kubectl apply -f <file>.yaml`: Applies a configuration to create or update a resource. It translates to `POST /api/v1/namespaces/{namespace}/pods`.
- `kubectl delete pod <pod-name>`: Deletes a specified Pod from the cluster. It translates to `DELETE /api/v1/namespaces/{namespace}/pods/{name}`

1.5 Kubernetes Architecture for Resilience and Fault Tolerance

Kubernetes is designed to keep applications available and stable even during failures or changes in demand.

Self-Healing Kubernetes continuously monitors the health of Pods, restarting containers that fail unexpectedly to maintain the system's desired state.

Load Balancing Kubernetes Services balance traffic across Pods, distributing requests to ensure reliability and performance.

Rolling Updates and Rollbacks Kubernetes Deployments support rolling updates to introduce new changes with minimal disruption. If an update causes issues, Kubernetes can perform a rollback to revert to a previous, stable version.

1.6 Readiness Probe

A readiness probe is a mechanism in Kubernetes that determines whether a Pod's container is ready to accept traffic. This probe can be configured with various parameters, including the initial delay before the first check, the frequency of subsequent checks, and the criteria for determining success or failure. This configuration allows Kubernetes to manage the Pod's availability effectively by ensuring that only ready Pods receive traffic.

2 Tested features of Kubernetes

This section details the Kubernetes Pod lifecycle states tested in this document and describes each transition in terms of its purpose, how it can be reached, and testing methods.

2.1 States

- **PodNotExists**: The initial or end state when a pod is absent in the cluster. It appears before a pod is created or after it is deleted. **Note:** The PodNotExists state used in this FSM model is not an actual Kubernetes Pod state. It is included solely for modeling purposes to represent when a Pod has not yet been created or has already been deleted.
- **Pending**: The pod is created but has not yet been scheduled onto a node, awaiting resource assignment.
- **ContainerCreating**: The container image is being pulled, and the pod is initializing.
- **ImagePullBackOff**: The pod is stuck due to an image pull error, which prevents the containers from starting.
- **Running_NotReady**: The pod has started, but one or more containers are not fully ready, potentially failing readiness probes.
- **Running_Ready**: The pod is fully functional with all containers running and ready, passed rediness probes.
- **Succeeded**: The pod has completed its task successfully, and all containers have exited without issues. It is possible only if the pod is a job, otherwise its containers won't typically exit.
- **Failed**: The pod has terminated due to a failure in one or more containers.
- **CrashLoopBackOff**: The pod is restarting repeatedly due to a failure in one or more containers.
- **Terminating**: The pod is in the process of being deleted from the cluster.

2.2 Transitions

2.2.1 Create Pod (PodNotExists → Pending)

In Kubernetes, creating a Deployment is typically preferred over creating individual Pods. A Deployment defines the desired state of Pods and manages their lifecycle, ensuring that the specified number of replicas are running, handling restarts, and managing updates. When a Deployment is created, Kubernetes automatically schedules and manages the Pods to maintain the Deployment's specified state.

The user provides a YAML configuration for the Deployment, which includes metadata (e.g., name, namespace), and specification (e.g., desired replicas, Pod template with containers, images, and resources). Then the user runs `kubectl apply -f <deployment-definition>.yaml`. `kubectl` sends a POST request (`POST /api/v1/namespaces/namespace/pods`) to the Kubernetes API Server with the Deployment specification. The API server validates the YAML configuration, checking for required fields (such as metadata and Pod template). The API ensures that resource quotas, namespace permissions, and other requirements are met. The API server records the Deployment in etcd, marking the desired number of replicas. The Deployment controller in the control plane reads the Deployment spec and begins creating the necessary Pods to reach the desired state. The Scheduler assigns each Pod to an available node based on resource availability. Each Pod enters the Pending state initially, moving to Running once its containers start successfully.

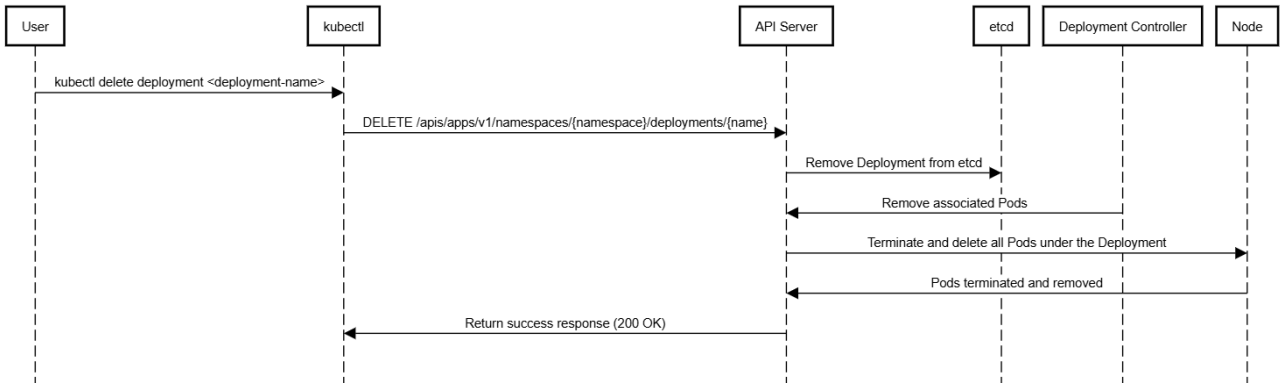

Figure 1: Creating a Deployment

During validation, if the YAML file has missing or invalid fields (e.g., no container image specified), or if the user lacks the necessary permissions, the API detects this error, and returns one the following error codes:

- 400 Bad Request: If the YAML syntax or required fields are incorrect.
- 403 Forbidden: If the user lacks permissions to create resources in the specified namespace.
- 409 Conflict: If a Pod with the same name already exists in the namespace.

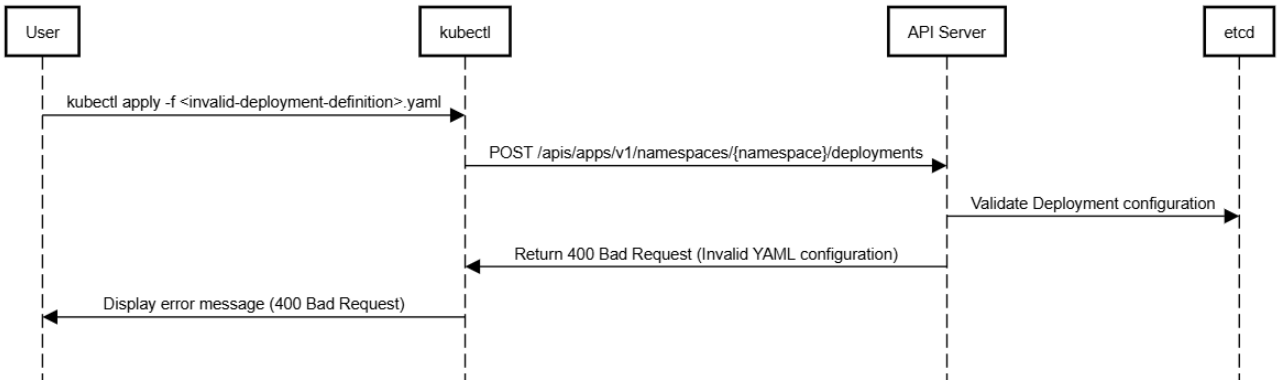

Figure 2: Error During Creating a Deployment

For the purposes of our testing, we simplify this process by focusing on individual Pod creation rather than a full Deployment. This involves defining a YAML file with metadata and container specifications for the Pod and running `kubectl apply -f <pod-definition>.yaml`. This request places the Pod in the Pending state as it waits for resources, without the additional management and scaling features provided by a Deployment. The Pod's status can be checked with `kubectl get pod <pod-name>`.

Note: From now on, our focus is primarily on the user-level interactions involved in creating and managing Pods, particularly the modifications to their YAML descriptions. While Kubernetes encompasses complex internal interactions, such as scheduling and resource allocation, our analysis centers on the direct commands issued by the user and the resulting states of the Pods. Therefore, UML diagrams will not be provided for every transition tested, as many involve similar user interactions: modifying YAML configurations followed by applying them with `kubectl apply -f <pod-definition>.yaml`.

The following figures illustrate the most important user-level interactions, including both positive and negative call flows:

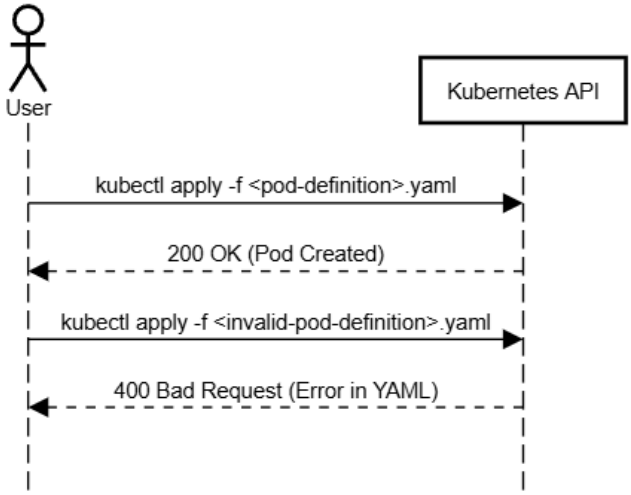

Figure 3: Apply pod definition

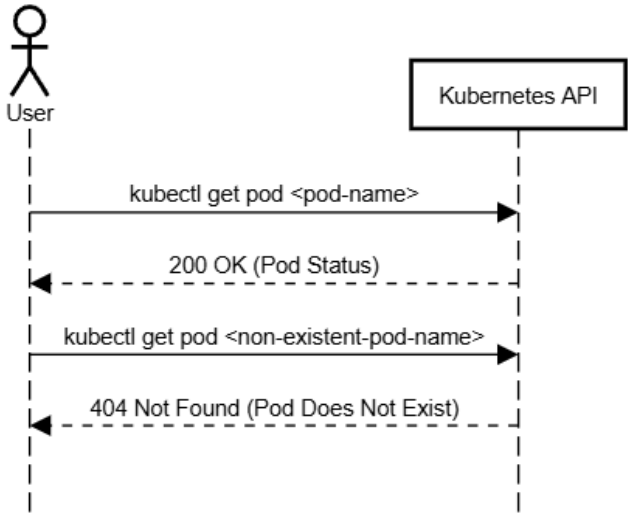

Figure 4: Get the status of the pod

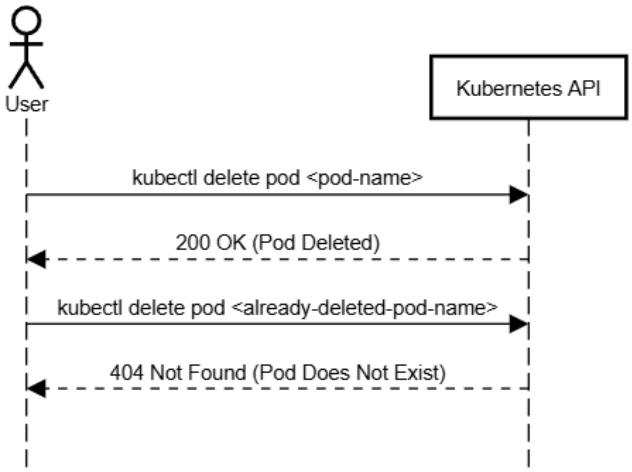

Figure 5: Delete a pod

2.2.2 Image Pull Error (Pending → ImagePullBackOff)

If a Pod is unable to pull the specified container image, it transitions to the ImagePullBackOff state. This can happen due to reasons such as the image not existing or the user not having the necessary permissions to access the image registry. To trigger this transition for testing, we can create a Pod with an invalid image name. Upon executing `kubectl apply -f <pod-definition>.yaml`, the Pod will enter the ImagePullBackOff state, which can be verified by running `kubectl get pod <pod-name>`. The status will indicate that the Pod is in ImagePullBackOff.

2.2.3 Image Pull Success (Pending or ImagePullBackOff → ContainerCreating)

When the container image is successfully pulled, the Pod moves from the Pending (or the ImagePullBackOff) state to the ContainerCreating state. During this state, Kubernetes prepares the environment for the container, setting up the network, storage, and any other necessary configurations. This transition can be tested by creating a Pod with a valid image. Upon running `kubectl apply -f <pod-definition>.yaml` and checking the status with `kubectl get pod <pod-name>`, the Pod should transition to ContainerCreating.

2.2.4 Containers Ready (ContainerCreating → Running_NotReady)

After the environment is set up, the containers within the Pod are started. If they are not yet ready to serve requests, the Pod is in the Running_NotReady state. To test this, we can deploy a Pod with a readiness probe configured to fail initially. After running `kubectl apply -f <pod-definition>.yaml` and checking its status, it will show the Pod in Running_NotReady when we check its status.

2.2.5 Readiness Probe Success (Running_NotReady or Running_Ready → Running_Ready)

Once the readiness probe passes, the Pod transitions to the Running_Ready state (or stays in Running_Ready state), indicating that it is fully operational. This can be tested by modifying the readiness probe of the previous Pod to succeed after a short delay. Once the Pod has started and the readiness probe passes, running `kubectl get pod <pod-name>` will show the status as Running_Ready.

2.2.6 Readiness Probe Failure (Running_Ready or Running_NotReady → Running_NotReady)

If the readiness probe fails, the Pod returns to the Running_NotReady state (or stays in Running_NotReady state). This can occur if the application within the container is unable to respond correctly. To test this transition, we can modify the application to simulate a failure in response to the readiness probe. After deploying the Pod with this failing application, the status can be checked with `kubectl get pod <pod-name>` to confirm the transition.

2.2.7 Pod Success (Running_Ready → Succeeded)

When a Pod completes its task successfully and all containers exit without errors, it moves to the Succeeded state. This is used for batch jobs rather than for average pods. We can test this by creating a job-type pod that runs a command that exits with a zero status. After deploying the Pod, checking its status with `kubectl get pod <pod-name>` will show that it has moved to the Succeeded state.

2.2.8 Pod Failure (Running_Ready → Failed)

If a container within a Pod exits with a non-zero status, indicating an error, the Pod transitions to the Failed state. We can test this by deploying a Pod that is designed to fail during execution. After the Pod runs, checking its status with `kubectl get pod <pod-name>` will show that it has moved to the Failed state.

2.2.9 Container Crash Loop (Running_NotReady → CrashLoopBackOff)

When a container in a Pod fails repeatedly during startup, the Pod enters the CrashLoopBackOff state. This transition indicates that the system is trying to restart the container but is unable to do so successfully. To test this, we can deploy a Pod with a configuration that intentionally causes the container to fail on startup. Observing the Pod's status with `kubectl get pod <pod-name>` will reveal the transition to CrashLoopBackOff.

2.2.10 Crash Loop Timeout (CrashLoopBackOff → Running_NotReady)

If a Pod remains in the CrashLoopBackOff state for too long without a successful start, it will transition back to the Running_NotReady state. This indicates that the system is no longer attempting to restart the container. To test this transition, we can simulate a prolonged crash loop and wait for the timeout. After the timeout, checking the Pod's status will confirm its return to Running_NotReady.

2.2.11 Delete Pod (All States → Terminating)

Issuing a delete command for a Pod in any state causes it to transition to the Terminating state. This transition ensures that the Pod is in the process of being removed from the cluster. To test this, we can run the command `kubectl delete pod <pod-name>` and then check the Pod's status. It should show that it is in the Terminating state.

2.2.12 Final Deletion (Terminating → PodNotExists)

Once a Pod has been marked for termination, Kubernetes performs automatic cleanup processes. After these processes are complete, the Pod no longer exists in the cluster. To verify this transition, we can issue the delete command using `kubectl delete pod <pod-name>` and then attempt to check the status with `kubectl get pod <pod-name>`. If the Pod has been successfully removed, we will receive an error indicating that the Pod no longer exists, confirming the transition from Terminating to a non-existent state.

3 The FSM model

To create the FSM model based on the identified states and transitions from the previous chapter, the Graphwalker Studio was used, a straightforward tool for setting up models with a drag-and-drop interface. After organizing the model visually, it can be exported to JSON format and manually edited to define inputs and outputs for each transition. The final result is the FSM model shown below.

Figure 6: FSM model

4 Test generation with MTR

For test generation, the ModelTestRelax(MTR) framework [1] was used with different parameters.

4.1 Random walk - 100% transition coverage

```
C:\Users\annas\MTR-3-4-0>MTR -m Random --random_coverage_type transition --random_coverage_percent 100 -f k8s-testing\k8s_model.json
[U-CTRL ] [info] Version: 3.4.0 R4: Belted kingfisher, profile: DEFAULT, verbosity: 3, debug mode: off
[U-LMI  ] [info] Model Name: Kubernetes_Pod_Lifecycle, Type: FSM, Reset: No
[U-CTRL ] [info] Running Random test generation, coverage percent: 100.000000, coverage type: transition
[TG-RW  ] [info] Desired transition coverage reached: 100.0%,  Walk length: 425
[TG-RW  ] [info] Finished computation at 2024-11-02 16:12:14.6540093
        elapsed time: (real time) 0.0019165 s
        elapsed time: (user time) 0.002000 s
[TG-TGR ] [info] Test generation summary written: test_summary/random_result.csv
[U-TW   ] [info] Test suite written: test_suites/Kubernetes_Pod_Lifecycle-Random-transition-100-test_suite.json
```

Figure 7: Random walk - 100% transition coverage

4.2 Random walk - 75% transition coverage

```
C:\Users\annas\MTR-3-4-0>MTR -m Random --random_coverage_type transition --random_coverage_percent 75 -f k8s-testing\k8s_model.json
[U-CTRL ] [info] Version: 3.4.0 R4: Belted kingfisher, profile: DEFAULT, verbosity: 3, debug mode: off
[U-LMI  ] [info] Model Name: Kubernetes_Pod_Lifecycle, Type: FSM, Reset: No
[U-CTRL ] [info] Running Random test generation, coverage percent: 75.000000, coverage type: transition
[TG-RW  ] [info] Desired transition coverage reached: 77.41935483870968%,  Walk length: 86
[TG-RW  ] [info] Finished computation at 2024-11-02 16:13:57.9694059
        elapsed time: (real time) 0.0009165 s
        elapsed time: (user time) 0.001000 s
[TG-TGR ] [info] Test generation summary written: test_summary/random_result.csv
[U-TW   ] [info] Test suite written: test_suites/Kubernetes_Pod_Lifecycle-Random-transition-75-test_suite.json
```

Figure 8: Random walk - 75% transition coverage

4.3 All-state

```
C:\Users\annas\MTR-3-4-0>MTR -m AS -f k8s-testing\k8s_model.json
[U-CTRL ] [info] Version: 3.4.0 R4: Belted kingfisher, profile: DEFAULT, verbosity: 3, debug mode: off
[U-LMI  ] [info] Model Name: Kubernetes_Pod_Lifecycle, Type: FSM, Reset: No
[U-CTRL ] [info] Running All-State test generation
[TG-TGR ] [info] Test generation summary written: test_summary/as_result.csv
[U-TW   ] [info] Test suite written: test_suites/Kubernetes_Pod_Lifecycle-AS-test_suite.json
```

Figure 9: All-state

4.4 Transition tour

```
C:\Users\annas\MTR-3-4-0>MTR -m TT -f k8s-testing\k8s_model.json --graphviz
[U-CTRL ] [info] Version: 3.4.0 R4: Belted kingfisher, profile: DEFAULT, verbosity: 3, debug mode: off
[U-LMI  ] [info] Model Name: Kubernetes_Pod_Lifecycle, Type: FSM, Reset: No
[U-CTRL ] [info] Created graphviz dot file of original model: ./graphviz-TT
[U-CTRL ] [info] Running Transition Tour test generation
[TG-TT  ] [info] Not Eulerian, augment
[TG-TT  ] [info] Building bipartite graph
[TG-TT  ] [info] Bipartite graph successfully built
[TG-TT  ] [info] Creating matching
[TG-TT  ] [info] Matching done
[TG-TT  ] [info] Duplicating transitions according to matching
[TG-TT  ] [info] Augmenting to Eulerian graph successful
[TG-TT  ] [info] Ordering edges
[TG-TT  ] [info] Edges ordered
[TG-TT  ] [info] Generating test sequence
[TG-TT  ] [info] Starting traversal
[TG-TT  ] [info] Tour length: 54
[TG-TT  ] [info] Augmented model graphviz file created: graphviz/graphviz-TT/Kubernetes_Pod_Lifecycle_augmented.dot
[TG-TT  ] [info] Finished computation at 2024-11-02 16:18:37.6916783
        elapsed time: (real time) 0.0088502 s
        elapsed time: (user time) 0.009000 s
[TG-TGR ] [info] Test generation summary written: test_summary/tt_result.csv
[U-TW   ] [info] Test suite written: test_suites/Kubernetes_Pod_Lifecycle-TT-test_suite.json
```

Figure 10: Transition tour

This is the Graphviz output:

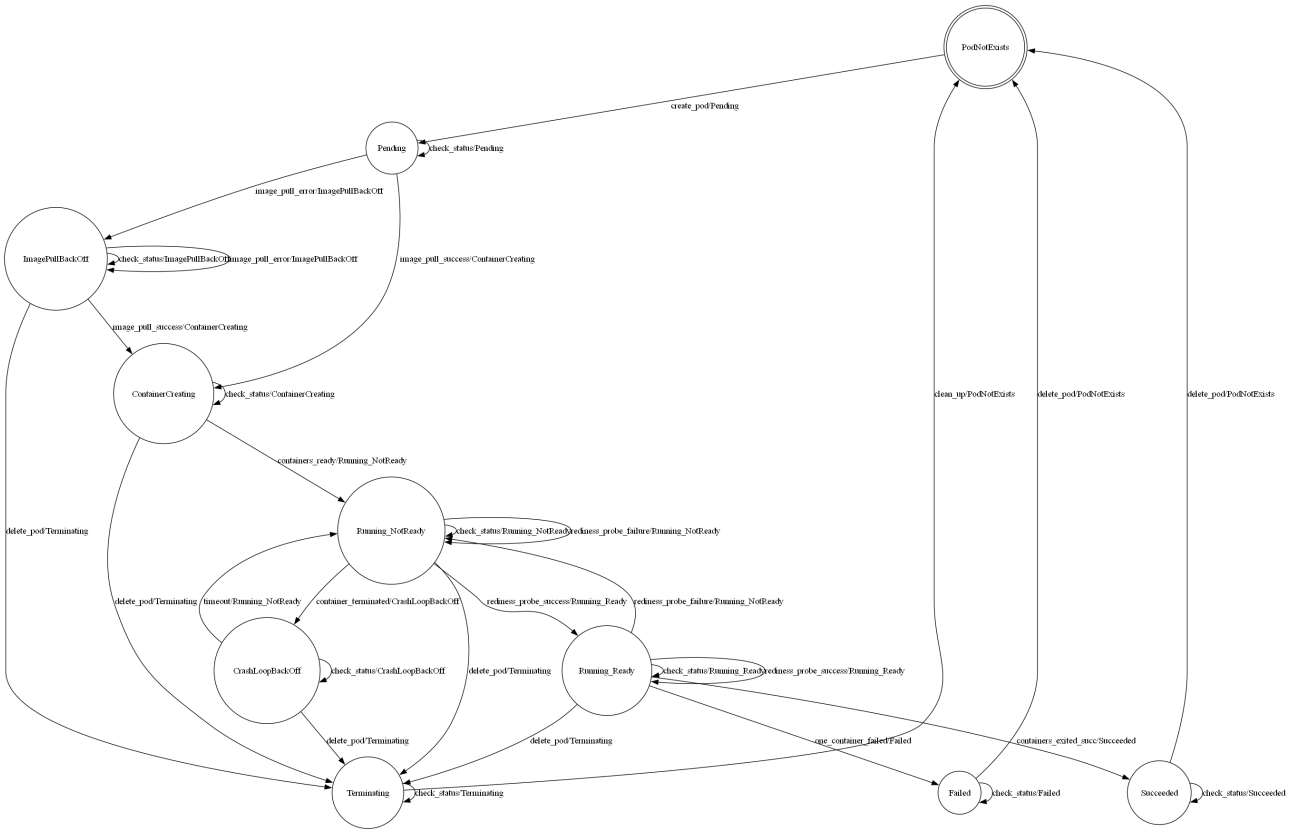

Figure 11: Grapviz result

The generated test suites and result can be found in the github repository: `https://github.com/annasz11/kubernetes-mbt`.

5 Adaptation code and test execution

Coming soon...

6 Summary

Coming soon...

References

[1] Németh Gábor Árpád. *ModelTestRelax*. URL: `https://gitlab.inf.elte.hu/nga/ModelTestRelax`.

[2] Kubernetes Authors. *Kubernetes Documentation*. Accessed: 2024-11-02. 2024. URL: `https://kubernetes.io/`.

[3] Kubernetes Authors. *Kubernetes Documentation*. Accessed: 2024-11-02. 2024. URL: `https://kubernetes.io/docs/concepts/workloads/pods/`.

[4] Kubernetes Authors. *Kubernetes Documentation*. Accessed: 2024-11-02. 2024. URL: `https://kubernetes.io/docs/concepts/overview/kubernetes-api/`.

[5] Kubernetes Authors. *Kubernetes Documentation*. Accessed: 2024-11-02. 2024. URL: `https://kubernetes.io/docs/reference/kubectl/`.