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 Java 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 [1]. 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 [2]. 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 [3]. 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 [4]. 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[6] 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.

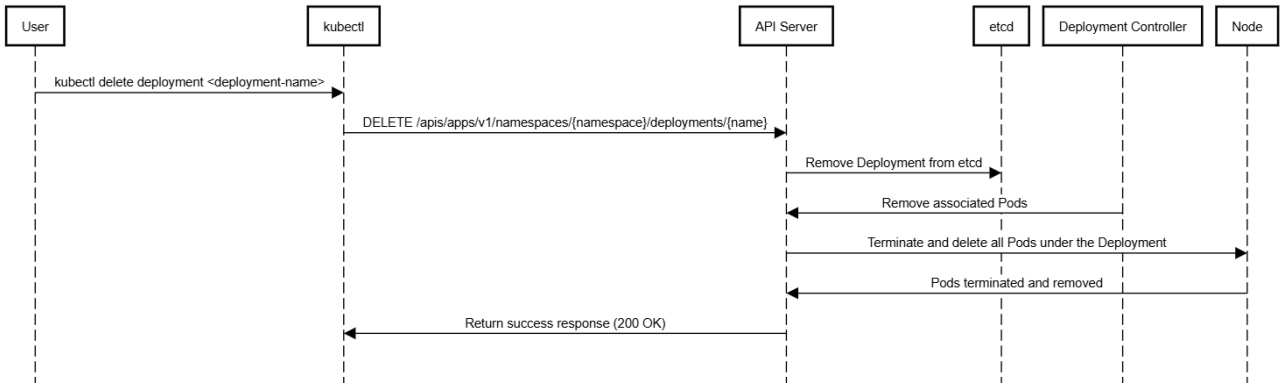

Figure 1: Creating a Deployment

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.

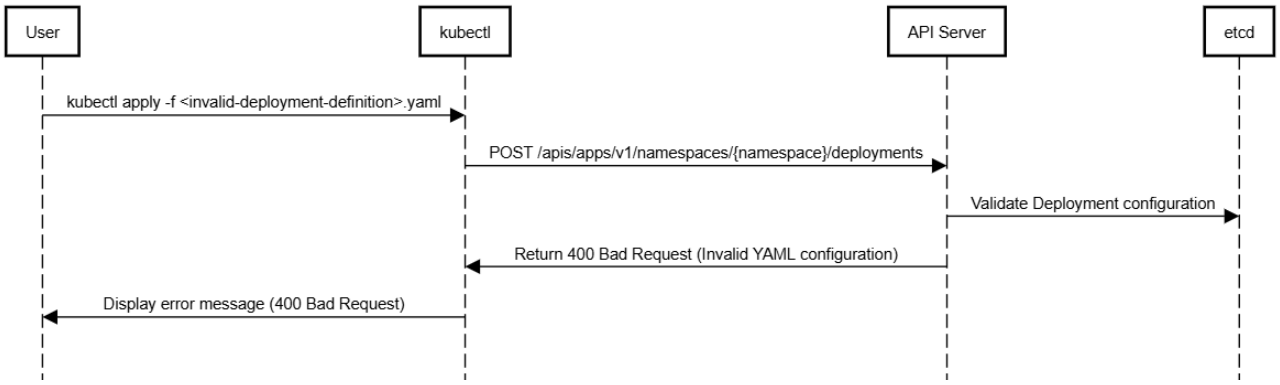

Figure 2: Error During Creating a Deployment

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:

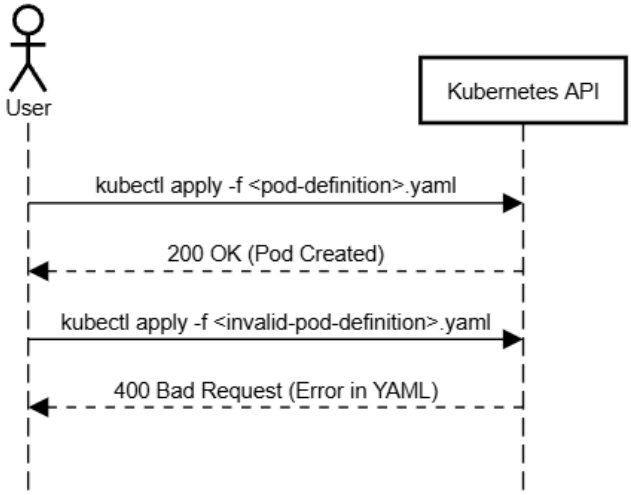

Figure 3: Apply pod definition

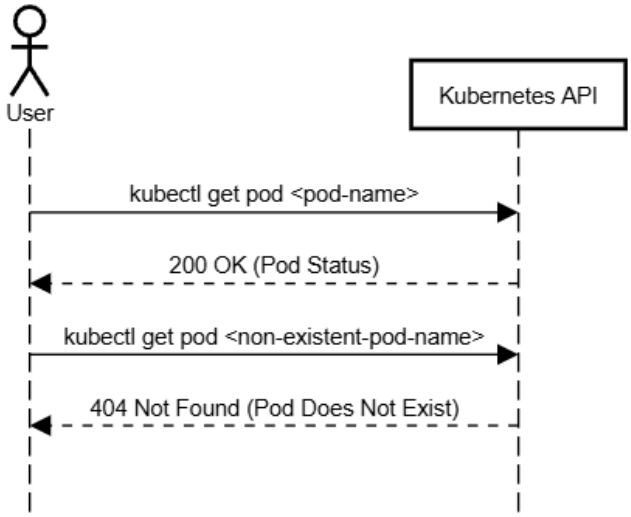

Figure 4: Get the status of the pod

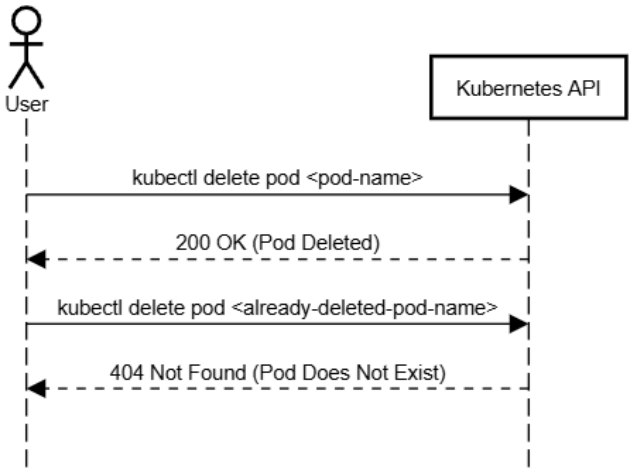

Figure 5: Delete a pod

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

The following figure is the extended version of the model, made by the MTR transition tour test generation algorithm with `--graphviz` flag. It also contains the input and output symbols too.

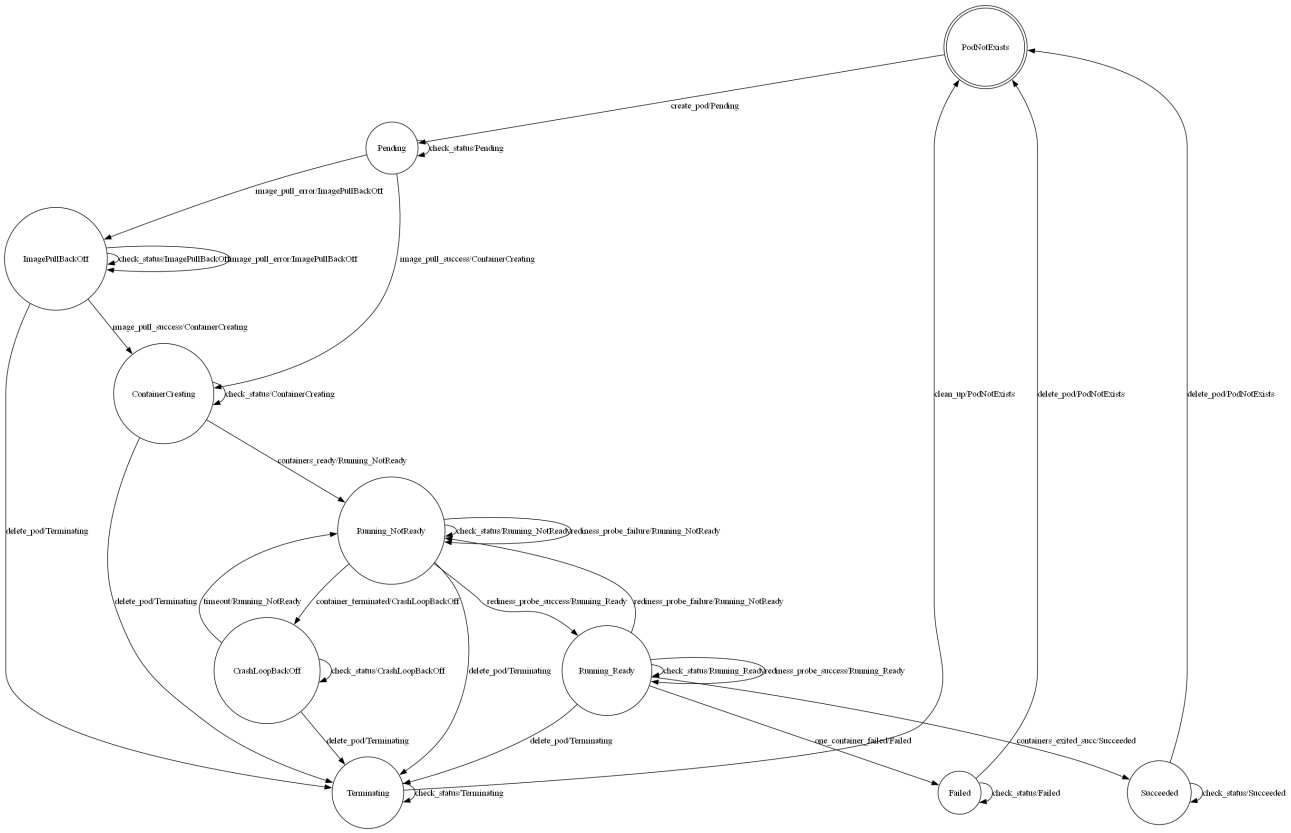

Figure 7: Grapviz result

4 Test generation with MTR

For test generation, the ModelTestRelax(MTR, Version: 3.5.16) framework [5] was used with different parameters.

4.1 Random Walk

The *Random Walk* algorithm generates a test sequence by randomly selecting a transition from the current state at each step. This process continues until one of the specified stop conditions is met. The algorithm supports two stop conditions:

- A given percentage of states has been visited.
- A given percentage of transitions has been covered.

While *Random Walk* is useful for exploratory testing, it is less efficient for systematic functional testing of large models due to its potentially long test sequences. However, its randomness makes it suitable for uncovering unexpected behaviors.

4.1.1 Execution and Parameters

The algorithm is executed using the following parameters:

- Coverage type: States or transitions.
- Coverage percentage: Desired percentage of coverage.
- Reset transitions: Optionally include reset transitions in coverage calculations.

4.1.2 Results

For this analysis, *Random Walk* was run with 100% state and transition coverage and also with 75% state and transition coverage as the stop condition. The last one was repeated five times to account for variability in sequence length and coverage.

Results: After running the Random walk(75% transition coverage) 5 times, the following statistics can be made.

	Test Generation Time	**Test Sequence Length**	**Reached coverage**
1	0.001434 s	264	77.42%
2	0.000823 s	41	80%
3	0.0008907 s	58	77.42%
4	0.0010825 s	75	77.42%
5	0.0008158 s	88	77.42%
Average	**0,0010092 s**	**105**	**77.94%**

Table 1: Summary of Random Walk Algorithm (5 Runs)

Here we can see that there was one case (namely the first one) where it took much longer time and test sequence to reach the desired coverage or exceed it, which was 77.42%. This value then was coming up 4 out of 5 times overall. The outstanding long first run is not that surprising since it is a randomized algorithm, if it "chooses worng" multiple times, it will finish later.

4.2 Transition Tour

The *Transition Tour* (TT) algorithm is designed to generate the shortest possible test sequence that ensures every transition in a reduced, deterministic, and strongly connected FSM is visited at least once, and returning to the initial state. This approach guarantees full state and transition coverage, making it particularly effective for detecting output faults, though it may not uncover all transfer faults.

The process involves two main stages:

1. Graph Adjustment: The FSM graph is transformed into an Eulerian graph by duplicating certain transitions to balance the number of incoming and outgoing arcs for each state. This step relies on solving a minimum matching problem in a bipartite graph.

2. Eulerian Path Construction: Using the augmented graph, an Eulerian path is determined by creating a spanning tree where edges point back toward the initial state.

With a time complexity of $O(n^3 + m)$ and test sequence length proportional to the number of transitions ($O(m)$), the TT algorithm offers an efficient solution for achieving comprehensive coverage in FSM-based testing.

The following figure is an example run on the FSM model.

```
C:\Users\annas\Documents\IK\modelling_and_testing\MTR-3-5-16\MTR-3-5-16>MTR -m TT -f  k8s_fsm_model_source.json
[U-CTRL ] [info] Version: 3.5.16 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 Transition Tour test generation
[TG-TT  ] [info] Not Eulerian, starting augmentation
[TG-TT  ] [info] Building bipartite graph
[TG-TT  ] [info] Bipartite graph successfully built
[TG-TT  ] [info] Creating minimum weight perfect matching
[TG-TT  ] [info] Matching done
[TG-TT  ] [info] Duplicating transitions according to matching
[TG-TT  ] [info] Augmentation over, checking if graph is Eulerian
[TG-TT  ] [info] Augmenting to Eulerian graph successful
[TG-TT  ] [info] Building spanning tree
[TG-TT  ] [info] Ordering edges
[TG-TT  ] [info] Edges ordered
[TG-TT  ] [info] Generating test sequence
[TG-TT  ] [info] Tour length: 54
[TG-TT  ] [info] Finished computation at 2024-11-20 23:20:51.6131919
        elapsed time: (real time) 0.0063979 s
        elapsed time: (user time) 0.006000 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 8: All-state

4.3 All-State

The *All-State* algorithm aims to visit every state in the FSM at least once. Using the Nearest Neighbor heuristic, it constructs a test sequence by repeatedly selecting the shortest path to an unvisited state until all states are covered.

Key Characteristics:

- Guarantees 100% state coverage, but not transition coverage.
- Time complexity: $O(n^2)$, where n is the number of states.
- Sequence length: $O(m)$, where m is the number of transitions.

4.3.1 Execution and Parameters

No additional parameters are required for FSMs, but the `--consider_reset_transitions` flag can be used for including reset transitions. The following figure is an example run on the FSM model.

```
C:\Users\annas\Documents\IK\modelling_and_testing\MTR-3-5-16\MTR-3-5-16>MTR -m AS -f  k8s_fsm_model_source.json
[U-CTRL ] [info] Version: 3.5.16 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 All-Transition-State (ATS)

4.4.1 Algorithm Overview

The *All-Transition-State* (ATS) algorithm generates a test suite that meets two criteria:

1. Each transition must be included in a sequence that reaches all FSM states.
2. Each state must be reachable without using a specific transition (if feasible).

The algorithm operates in two parts:

- Preamble: Covers all transitions using the *Transition Tour* method.

- Postamble: Covers all states using the Nearest Neighbor heuristic.

ATS has three variations:

- ATS0: Standard version.
- ATSa: Iterative version with no limit.
- ATSx: Iterative version with a specified limit.

Key Characteristics:

- Time complexity: $O(n^3 + m)$ for ATS0; higher for iterative versions.
- Sequence length: Comparable to *Transition Tour*.

4.4.2 Execution and Parameters

For this analysis, the standard version (ATS0) is used.

4.5 N-Switch Coverage

The *N-Switch Coverage* algorithm generates test sequences that cover all consecutive $N + 1$ transitions in the FSM. It constructs a list of all such paths, selects uncovered paths, and augments sequences as needed to achieve full coverage.

Key Characteristics:

- Higher N values increase fault coverage but also lengthen test sequences.
- Time complexity: $O((N + 1) \cdot m^{N+1})$.
- Sequence length: $O((N + 1) \cdot m^{N+1})$.

4.5.1 Execution and Parameters

For this analysis, 1-switch coverage is used as the default configuration.

```
C:\Users\annas\Documents\IK\modelling_and_testing\MTR-3-5-16\MTR-3-5-16>MTR -m NS -f  k8s_fsm_model_source.json
[U-CTRL ] [info] Version: 3.5.16 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 NSwitchCoverage test generation
[TG-1-SC] [info] Generating all possible, consecutive N+1 transitions
[TG-1-SC] [info] Starting test sequence generation from the ordered list L of consecutive 2 long transitions
[TG-1-SC] [info] Result:
        length of test sequence: 200,
        number of elements in ordered list L (that contains all possible, consecutive N+1 transitions): 99
[TG-1-SC] [info] Finished computation at 2024-11-20 23:22:07.2223080
        elapsed time: (real time) 0.0125292 s
        elapsed time: (user time) 0.013000 s
[TG-TGR ] [info] Test generation summary written: test_summary/ns_result.csv
[U-TW   ] [info] Test suite written: test_suites/Kubernetes_Pod_Lifecycle-NS-1-test_suite.json
```

Figure 10: All-state

4.6 Comparison of Test Generation Algorithms

Table 2 summarizes the performance of the selected algorithms based on the FSM model of Kubernetes pod lifecycle management.

Table 2: Comparison of Test Generation Algorithms

Algorithm	Test Sequence Length	Elapsed Time	Coverage
Random Walk (100% transition c.)	965	0.0052706 s	100% transition coverage
All-State	18	0,0001648 s	100% state coverage
Transition Tour	54	0.0063979 s	100% state- and transition coverage
All-Transition-State	193	0.0244396 s	100% state- and transition coverage
N-Switch Coverage (N=1)	200	0.0125292 s	100% state- and transition coverage

4.7 Conclusions

The Random Walk (100% transition coverage) algorithm generated the longest test sequence, consisting of 965 steps, with a moderate elapsed time of 0.00527 seconds, emphasizing complete transition coverage but at the cost of producing lengthy sequences. In contrast, the All-State algorithm created the shortest test sequence of only 18 steps and was the fastest, with an execution time of 0.000165 seconds, as it focuses solely on state coverage, resulting in minimal generation. The Transition Tour algorithm generated a moderate sequence of 54 steps but took slightly longer, with an elapsed time of 0.00640 seconds, reflecting its method of traversing transitions optimally to ensure coverage. The All-Transition-State (ATS) algorithm achieved comprehensive state and transition coverage, producing a sequence of 193 steps but with the highest elapsed time of 0.02444 seconds due to its detailed coverage requirements. Lastly, the N-Switch Coverage (N=1) algorithm provided a balance, generating a sequence of 200 steps with a reasonable elapsed time of 0.01253 seconds, focusing on transitions within a specified depth. These results demonstrate a clear trade-off between coverage, sequence length and execution time.

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

5.1 Adaptation code

In this section, the adaptation code is described that was written to execute the test suites generated by the MTR tool for the Kubernetes pod lifecycle model. The purpose of the adaptation code is to interface with the Kubernetes cluster using 'kubectl ' commands and to monitor pod states based on the inputs from the test suite and the expected results.

To parse the generated test suites, a JSON test suite parser was written. It returns the input and the output symbols in a list, so that the test runner class can easily iterate through them and execute each transition. The adaptation code contains two important mappings, which also contain the main logic of the program:

- **Input Mappers**: The input mapper maps the current input symbol to the corresponding function call in the program. This is where it is decided which Kubernetes action needs to be taken (e.g., creating or deleting a Pod). The input is mapped to the relevant functions that handle the Kubernetes API interactions through 'kubectl'. Each transition was performed with `kubectl apply -f` command, where different Pod definitions were provided for the different test scenarios. For example to test a Pod getting into the ImagePullBackOff state, the Pod definition intentionally contained a faulty image name.

  Notes on limitations: Since it is not permitted by the Kubernetes API to modify certain properties of a Pod such as Liveness probes while running, it was only possible to carry out each transitions if every time the Pod was deleted and then recreated with a new Pod definition. This actually caused longer runtime, especially because before some actions like asserting that the pull of a Docker image was successfull it was necessary to include some waiting times.

- **Output Mappers**: Since the FSM model was created in a way that the output symbols correspond to the next state of the model, these could be used for assertions too. For this, the output mapper translates the states into assertions, where we check the Pod states after each transitions. So in the end, the result of these checks are compared to the current expected output symbol of the generated test suite.

In summary, the adaptation code works as follows:

- **Test Suite Parsing**: The test suite JSON is parsed to extract the input and expected output list.
- **Kubernetes Interaction**: Using 'kubectl', the code performs the corresponding actions (e.g., creating a Pod with different definitions, deleting a Pod).
- **State Monitoring**: The code monitors the pod's state after each operation and compares it against the expected output.

5.2 Prerequisites

The adaptation code was developed and executed on a Windows machine with the following setup:

- **Docker Desktop**: Version 4.19.0 (or the latest version available) - used for running a local Kubernetes cluster in conjunction with Minikube.
- **Minikube**: Version 1.30.0 (or the latest version available) - used to create and manage a local Kubernetes cluster.
- **kubectl**: Version 1.25.0 (or the latest version available) - the Kubernetes command-line tool used to interact with the Kubernetes API for managing pods, deployments, etc.
- **Java**: Version 17 (or the most recent version of the OpenJDK) - the programming language used for writing the adaptation code and handling the test execution.

For being able to run the test, the above mentioned tools are needed, and the Docker Desktop and Minikube must be started.

5.3 Test Execution Report

The test execution phase was carried out using the adaptation code, which interfaces with the Kubernetes cluster and executedthe generated test suites. The results of these executions, including any errors or issues encountered, are summarized in the following sections.

All of the test suites were executed that was generated by the algorithms described in the previous chapter: Random walk (100% state coverage), All State, Transition Tour, All Transition State, N-switch coverage. Neither of them discovered any fault, all test passed. The differences were only the test execution times, which were expected because of the various test sequence lengths. Furthermore, as mentioned above, some transitions like Docker image pull, or waiting for a Pod to become ready required more waiting time, so the test suite that used more of these types of transitions were naturally longer to execute. The following table summarizes the execution times:

Algorithm	**Elapsed Time**	**Test sequence length**
Random Walk (100% state c.)	232 s	34
All-State	141 s	18
Transition Tour	330 s	54
All-Transition-State	1510 s	193
N-Switch Coverage (N=1)	1523 s	200

Table 3: Comparison of Test Executions

5.4 Conclusion

Based on the results of the test executions, we conclude that the Kubernetes pod lifecycle management is functioning as expected. All test cases passed, and no issues were identified during the testing phase.

6 Summary

In this study, the Kubernetes Pod lifecycle management process was tested where an FSM model was created and the MTR model-based testing tool was used. The FSM captured the extended states and transitions of the Pod lifecycle, enabling a model-based approach to test generation. Various algorithms were employed to generate test suites, including Random Walk, All-State, Transition Tour, All-Transition-State, and N-Switch Coverage, each offering distinct trade-offs between coverage, test sequence length, and generation time.

The generated test suites were executed using a Java-based adaptation code that interacted directly with the Kubernetes cluster via kubectl. This approach allowed precise control over pod states and transitions, ensuring alignment between the model and the actual system under test. During execution, the adaptation code successfully applied all test cases without encountering any faults, indicating that the Kubernetes pod lifecycle management implementation adhered to the expected behavior defined in the FSM model.

References

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

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

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

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

[5] *ModelTestRelax*. URL: `https://gitlab.inf.elte.hu/nga/ModelTestRelax`.

[6] *YAML*. URL: `https://yaml.org/`.