FLINK: MODEL-DRIVEN DEPOLYMENT

Manasjyoti Bhuyan-893800, Poornima Jagannathan- 897800

DEIB – Politecnico Di Milano, Italy

I. Introduction

New models of computing have been growing up rapidly and are become widely popular with the amount of data growing at an exponential rate.
For the computation of these data in the present era, we need high speed processing techniques, which in turn leads to a curtail the process of software development cycle. Big Data analytics has become an indispensable tool in transforming science, engineering, healthcare, finance and ultimately business itself, due to the unprecedented ability to extract new knowledge and automatically find correlations in massive datasets that naturally accumulate in our digital age.

Flink is a tool developed with the goal of supporting the deployment and management of Big Data applications. It leverages Infrastructure-as-Code (IaC) paradigm. Flink is similar to Spark, both are used conversely for batch and iterative processing but Flink is a bit better it performance and accuracy in data streaming. Still it is in its initial stage. We are trying to deploy it and test it.

The problem here is how we can define precise rules, and try modifying the DICE profile for Flink and generate model for it in DDSM. Which is followed by generation of TOSCA blueprint for Flink and then check if it supports profile extension.

In this project we extended the dice profile to support Flink as like Spark and then generate the model for flink and generate the TOSCA blueprint from the available chef recipes of flink.

II. Background

DICER is a tool developed with the goal of supporting the deployment and management of Big Data applications. Main goal of DICER is to exploit deployment models specified in accordance with the DICE Deployment Specific metamodel, in order to speed up the deployment process.
DICER is a tool developed in the context of the DICE H2020 European Project enabling the model-driven deployment of data-intensive applications (DIAs)

leveraging the Infrastructure-as-Code (IaC) paradigm. More specifically, DICER adopt the OASIS Topology and Orchestration Specification for Cloud Applications standard and is able to automatically generate IaC for DIAs in the form of TOSCA blueprints from stereotypes UML models.

Using Eclipse UML Papyrus and the DICER Eclipse plugins, we are able to design the UML Deployment Diagram of its DIA using the DICER **UML profile** to enrich the model with DIA-specific deployment concepts by applying standard UML stereotypes. By using the UML stereotypes, we can configure various properties of our model, both at the infrastructural level (i.e. on some number of VMs) and also at the platform level.

How to extend profile - In order to create a profile, it has to be done with the support of the papyrus. After the creation of the profile, it is now time to populate this latter with UML extensions, i.e. stereotypes, and their related concepts such as properties, extensions, and metaclasses. A stereotype is created as any other UML model elements in Papyrus. Once we have created a stereotype we can add the required parameters and then, we need to import the UML2 metaclasses we want to extend. Next the importing of the metaclasses has to be done. Once the stereotype is created and the meta-class is imported, we may then model the extension relation from the stereotype to the meta-class. The extension relationship is modeled using the extension tool in the profile diagram palette "-> "(extension). Note when designing a profile, it is possible to reuse existing stereotypes defined in other existing profiles. Stereotypes can indeed be generalized enabling to create child stereotypes that inherit features of one or more generalized stereotype define either locally in the profile or externally other profiles.

In this project we are trying to deploy **Flink** which is an open-source framework for distributed stream processing. It allows exactly-once semantics for stateful computations along with that it supports stream processing and windowing with event time semantics, supports flexible windowing based on time, count, or sessions in addition to data-driven windows. It's lightweight fault tolerance structure allows the system to maintain high throughput rates and provide exactly-once consistency guarantees at the same time. Even processing of large volume can be done at faster way using Flink because of it high throughput and low latency. Flink can run in the cloud or on premise and on a standalone cluster or on a cluster managed by YARN or Mesos.

To deploy the models that has been generated in the DDSM into the TOSCA blueprint we need to generate TOSCA code. To generate these codes, we can use the **ATL Transformation language**. ATL-Transformation Language is a model transformation language and toolkit. In the field of Model-Driven Engineering (MDE), ATL provides ways to produce a set of target models from a set of source models. Developed on top of the Eclipse platform, the ATL Integrated Environment (IDE) provides a number of standard development tools (syntax highlighting, debugger, etc.) that aims to ease development of ATL transformations. It provides a way to produce a number of target models from a set of source models. An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models.

For the creation and support of profiles and the generation of the UML models, Papyrus is used. Papyrus is an open source project to provide an integrated environment for editing UML. It's is implemented in the eclipse as a plugin. Papyrus allows to create Profiles and Stereotypes in any kind of models. But, if you want to define Profiles and Stereotypes in order to apply them to UML elements, you need to define a Profile in a Papyrus profile resource (file).

III. Profile Extension and Model Generation

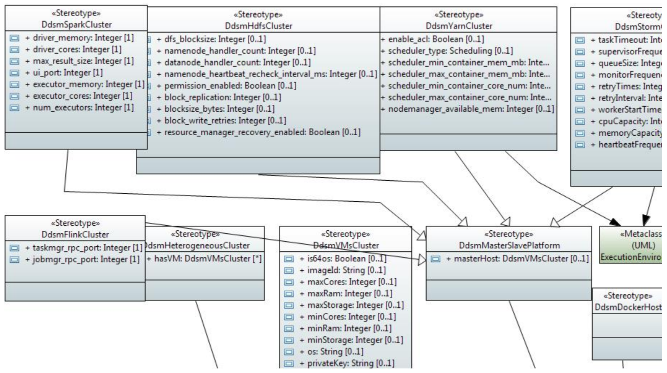

«Stereotype»
DdsmSparkCluster
+ driver_memory: Integer [1]
+ driver_cores: Integer [1]
+ max_result_size: Integer [1]
+ ui_port: Integer [1]
+ executor_memory: Integer [1]
+ executor_cores: Integer [1]
+ num_executors: Integer [1]
«Stereotype»
DdsmHdfsCluster
+ dfs_blocksize: Integer [0..1]
+ namenode_handler_count: Integer [0..1]
+ datanode_handler_count: Integer [0..1]
+ namenode_heartbeat_recheck_interval_ms: Integer [0..1]
+ permission_enabled: Boolean [0..1]
+ block_replication: Integer [0..1]
+ blocksize_bytes: Integer [0..1]
+ block_write_retries: Integer [0..1]
+ resource_manager_recovery_enabled: Boolean [0..1]
«Stereotype»
DdsmYarnCluster
+ enable_acl: Boolean [0..1]
+ scheduler_type: Scheduling [0..1]
+ scheduler_min_container_mem_mb: Inte...
+ scheduler_max_container_mem_mb: Inte...
+ scheduler_min_container_core_num: Inte...
+ scheduler_max_container_core_num: Inte...
+ nodemanager_available_mem: Integer [0..1]
«Stereoty
DdsmStorm
+ taskTimeout: Int
+ supervisorFreque
+ queueSize: Integ
+ monitorFrequen
+ retryTimes: Integ
+ retryInterval: Inte
+ workerStartTime
+ cpuCapacity: Int
+ memoryCapacity
+ heartbeatFreque
«Stereotype»
DdsmFlinkCluster
+ taskmgr_rpc_port: Integer [1]
+ jobmgr_rpc_port: Integer [1]
«Stereotype»
DdsmHeterogeneousCluster
+ hasVM: DdsmVMsCluster [*]
«Stereotype»
DdsmVMsCluster
+ is64os: Boolean [0..1]
+ imageId: String [0..1]
+ maxCores: Integer [0..1]
+ maxRam: Integer [0..1]
+ maxStorage: Integer [0..1]
+ minCores: Integer [0..1]
+ minRam: Integer [0..1]
+ minStorage: Integer [0..1]
+ os: String [0..1]
+ privateKey: String [0..1]
«Stereotype»
DdsmMasterSlavePlatform
+ masterHost: DdsmVMsCluster [0..1]
«Metaclass
(UML)
ExecutionEnviro
«Stereotype»
DdsmDockerHost

Fig. 1

The VMsCluster stereotype can be defined as a specialization of ExternalComponent, since renting computational resources such as virtual machines is one of the main services offered by Cloud providers. VMsCluster is applied to the Device UML meta-class. A cluster of VMs logically represents a single computational resource with processing capabilities, upon which applications and services may be deployed for execution.
We then define two DIA-specific deployment abstractions, i.e. the MasterSlavePlatform abstract stereotype, as further specializations of InternalComponent. The MasterSlavePlatform stereotype allows indicating a dedicated host for the master node, which might require more resources. By extending the MasterSlavePlatform stereotypes we implemented a set of technology-specific concrete stereotype – FlinkCluster. DIA execution engines (Flink) extend UML ExecutionEnvironment, so to identify those platforms which DIA jobs can be submitted to. The JobSubmission stereotype, which extends UML Deployment, is used to specify additional deployment options for a DIA job. It allows to specify job's scheduling options, such as how many times it has to be submitted and the time interval between two subsequent submissions. In this way the same DIA job can be deployed in multiple instances using different deployment options. Additionally, an OCL constraint requires each DiaJob to be connected by means of JobSubmission to an execution engine, i.e. an ExecutionEnvironment holding a sub-stereotype of MasterSlavePlatform.

In order to overcome the overloading of the big data platforms within the same cluster, we can host the Apache FlinkCluster in dedicated VMs. The resulting model is shown Fig. 2. The DICER user can easily derive it from the original one and generate the corresponding blueprint. In this case, we could extend the deployment model into the one in Fig. 2 where we have added a new VMCluster and a new ExecutionEnvironment element with the FlinkCluster stereotype (flink).

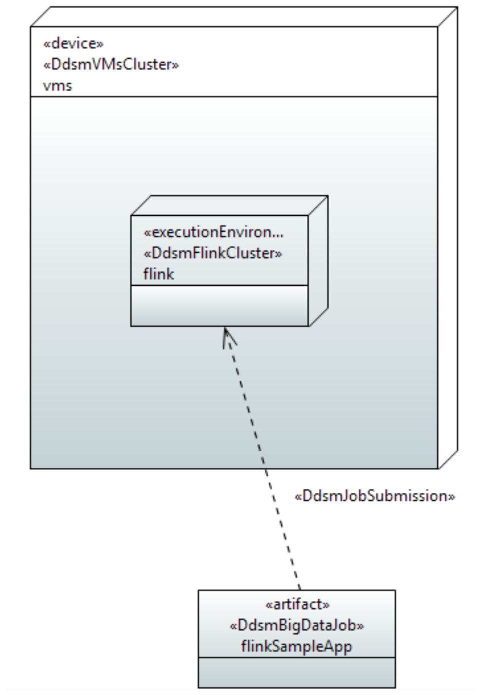


Fig. 2

IV. Model transformation extension

In this work, when we refer to deployment model of a data-intensive application, this includes the deployment of all the required services, technologies and of the jobs to be run over resulting platforms. In particular we defined a configurable node specific to Big Data technology- Apache Flink. We also enabled configuring client nodes, which can submit the user's custom workfows or jobs on top of the Big Data techonologies, making the framework suitable for modeling data-intensive applications. Once we drafted the initial model, this is transformed into a TOSCAcompliant model using a model-to-model transformation, developed in the ATLAS Transformation Language (ATL), called the Deploy Transformation. This transformation is designed over the TOSCA meta-models. By extending the TOSCA, the we can easily specify deployable Cloud-based deployment models. The TOSCA model output of the Deploy Transformation is finally serialized, using an Xtext model-to-text transformation, into a deployable TOSCA YAML blueprint, which is in turn automatically sent to the Deployment Service in order to be processed.

Sample TOSCA Codes-

```
------------------- FLINK -----------------------
-> select(device |
device.hasStereotype('DdsmVMsCluster')) ->
collect(device | device.nestedNode -> select(nested |
nested.oclIsTypeOf(MMUML!ExecutionEnvironment) and
nested.hasStereotype('DdsmFlinkCluster')) ->
collect(flink |
thisModule.flinkToTosca(flink))).flatten()).union(MMUML!Device.allInstances()
-> select(element |
element.oclIsTypeOf(MMUML!Artifact) and
element.hasStereotype('DdsmBigDataJob')) ->
collect(job |
thisModule.getDiaJobNodeTemplate(job)).flatten()).union(MMUML!Device.allInstances()
```

V. Workflow

Steps for profile extension –

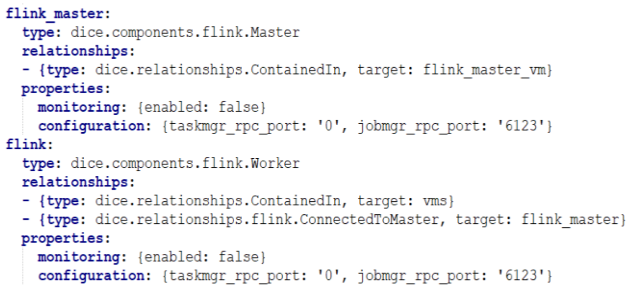
i. Modify DICE-profile.di by creating a stereotype as DdsmFlinkCluster and extend the stereotype to MasterSlavePlatform. Add the required parameters for the Flink and set the appropriate default values to the parameters.
ii. In order to run the profile, we used the DICE Simulation SDK Target Platform.
iii. After running the profile, create the model i.e., deployment diagram as flink.di as shown in Fig 2 in the second eclipse.
iv. Followed this, reload the DICE-genmodel and generate the model code.
v. Create flink,yaml file representing the TOSCA definitions i.e., chef-recipes for Flink which is similar to Spark.
vi. Transformation code for generating TOSCA blueprint is done by modifying uml2tosca.atl
vii. The input file and the output file path are specified in Dicer.java. The type of output files to be generated are specified in this file.
viii. The output files generated are in the format of flink_out.json, flink_out.xmi and flink_out.yaml.
ix. The TOSCA blueprint generation is available in flink_out.yaml

VI. Conclusion

Result-

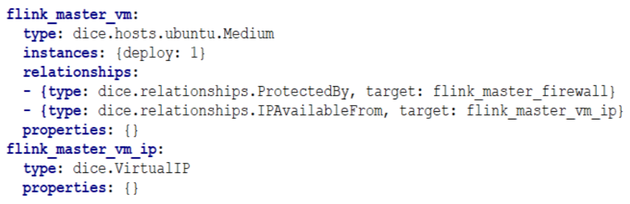
TOSCA Blueprint

Flink master and config.-



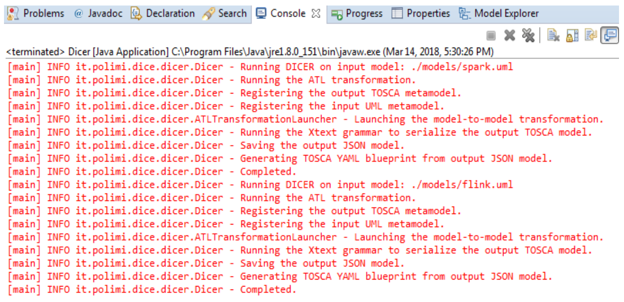
```
flink_master:
  type: dice.components.flink.Master
  relationships:
  - {type: dice.relationships.ContainedIn, target: flink_master_vm}
  properties:
    monitoring: {enabled: false}
    configuration: {taskmgr_rpc_port: '0', jobmgr_rpc_port: '6123'}
flink:
  type: dice.components.flink.Worker
  relationships:
  - {type: dice.relationships.ContainedIn, target: vms}
  - {type: dice.relationships.flink.ConnectedToMaster, target: flink_master}
  properties:
    monitoring: {enabled: false}
    configuration: {taskmgr_rpc_port: '0', jobmgr_rpc_port: '6123'}
```

Flink_Master VMs-



```
flink_master_vm:
  type: dice.hosts.ubuntu.Medium
  instances: {deploy: 1}
  relationships:
  - {type: dice.relationships.ProtectedBy, target: flink_master_firewall}
  - {type: dice.relationships.IPAvailableFrom, target: flink_master_vm_ip}
  properties: {}
flink_master_vm_ip:
  type: dice.VirtualIP
  properties: {}
```

Output-



```
<terminated> Dicer [Java Application] C:\Program Files\Java\jre1.8.0_151\bin\javaw.exe (Mar 14, 2018, 5:30:26 PM)
[main] INFO it.polimi.dice.dicer.Dicer - Running DICER on input model: ./models/spark.uml
[main] INFO it.polimi.dice.dicer.Dicer - Running the ATL transformation.
[main] INFO it.polimi.dice.dicer.Dicer - Registering the output TOSCA metamodel.
[main] INFO it.polimi.dice.dicer.Dicer - Registering the input UML metamodel.
[main] INFO it.polimi.dice.dicer.ATLTransformationLauncher - Launching the model-to-model transformation.
[main] INFO it.polimi.dice.dicer.Dicer - Running the Xtext grammar to serialize the output TOSCA model.
[main] INFO it.polimi.dice.dicer.Dicer - Saving the output JSON model.
[main] INFO it.polimi.dice.dicer.Dicer - Generating TOSCA YAML blueprint from output JSON model.
[main] INFO it.polimi.dice.dicer.Dicer - Completed.
[main] INFO it.polimi.dice.dicer.Dicer - Running DICER on input model: ./models/flink.uml
[main] INFO it.polimi.dice.dicer.Dicer - Running the ATL transformation.
[main] INFO it.polimi.dice.dicer.Dicer - Registering the output TOSCA metamodel.
[main] INFO it.polimi.dice.dicer.Dicer - Registering the input UML metamodel.
[main] INFO it.polimi.dice.dicer.ATLTransformationLauncher - Launching the model-to-model transformation.
[main] INFO it.polimi.dice.dicer.Dicer - Running the Xtext grammar to serialize the output TOSCA model.
[main] INFO it.polimi.dice.dicer.Dicer - Saving the output JSON model.
[main] INFO it.polimi.dice.dicer.Dicer - Generating TOSCA YAML blueprint from output JSON model.
[main] INFO it.polimi.dice.dicer.Dicer - Completed.
```

Future Work –

The Deployment Service can be operated using its simple web user interface, but the principal use of the tool is to be integrated in a larger DevOps workflow. Normally, it will be invoked from a Continuous Integration tool (e.g., Jenkins or Apache Continuum). The service operates with uniquely identifiable logical unit in the deployment environment, simply called virtual containers. A container can receive an application blueprint expressed in standard TOSCA YAML format. An administrator will normally create one or more empty containers and share their identifiers with the developers, or use them to associate containers with specific Continuous Integration jobs. When a user or a client sends a blueprint to be deployed in an empty container, the Deployment Service will deploy the Big Data technologies according to the specifications in the TOSCA blueprint, and launch the user's job within the deployment. If a client later submits another blueprint to the same container, the tool will first assure that the previous deployment has been purged.