Model-driven Engineering for the Configuration and Deployment of Data Processing Applications

Hui Song, Nicolas Ferry, Jakob Høgenes and Arnor Solberg
SINTEF, Oslo, Norway

Keywords: Big Data, Model-driven Engineering.

Abstract: This position paper reports our current endeavour towards a model-driven engineering framework to support the dynamic configuration and deployment of complex data processing applications. In particular, our approach includes a domain-specific modelling language that abstracts the data processing tasks and at the same time exposes the control of how these tasks are deployed on specific platforms and resources. A modelling framework of model transformation and models@runtime engines realises the semi-automatic configuration and deployment of the applications based on the abstract models.

1 INTRODUCTION

Data processing applications are playing an important role in industry, partly as a result of the big data movement. These applications typically combine heterogeneous, distributed and dedicated software solutions. In order to reduce operation costs, they are often deployed over cloud infrastructures that provide on demand access to a virtually infinite set of computing, storage and network resources. However, in practice the development and operation of data processing applications typically face two challenges: *(i)* the complexity and time required to learn as well as to design and integrate the many existing general purpose frameworks and *(ii)* the complexity of operation, maintenance and evolution of the applications.

As a result of the first challenge, a large proportion of the data engineers' effort is dedicated to configuration and deployment activities. It is complex to properly configure and connect the many general purpose frameworks (*e.g.*, Hadoop, Storm, Spark, Kafka, *etc.*) and to deploy them on the adequate infrastructure (*e.g.*, OpenStack VM, Amazon DynamoDB). Part of this challenge can be addressed by providing a proper abstraction hiding some platforms specificities as well as by facilitating the reuse and sharing of part of the system. Some cloud providers offered graphical data processing environments, such as Amazon Big Data[1] and Microsoft Azure Machine Learning[2]. Thanks to such environment, engineers can quickly prototype their ideas into a runnable application. However, these solutions prevent developers from understanding and controlling how the applications are deployed and run on cloud infrastructures, and also implies vendor lock-in.

Moreover, as these systems must inevitably evolve, it is challenging to maintain and evolve them whilst minimizing down-time. In order to shorten delivery time and to foster continuous evolution, we should reconcile development and operation activities (Hüttermann, 2012). The DevOps movement advocates to not only automate configuration and deployment of the application, but also the capability to monitor and control the application, in order to improve the both the efficiency and flexibility.

In this paper we introduce a framework to foster the continuous design, deployment, operation and evolution of data processing applications. Our solution, called DAMF (Data Modelling Framework), is composed of *(i)* a domain specific modelling language to specify both the data flow and the deployment of the data processing tasks and *(ii)* a toolset with model transformations and models@runtime engines for the automatic deployment and adaptation of such data processing applications.

The remainder is organised as follows. Section 2 introduces a motivating example. Section 3 introduces DAMF and Section 4 discusses our future plans. Finally, Section 5 discusses related approaches and Section 6 concludes the paper.

[1]aws.amazon.com/training/course-descriptions/bigdata/
[2]azure.microsoft.com/services/machine-learning/

Song H., Ferry N., HÃÿgenes J. and Solberg A.
Model-driven Engineering for the Configuration and Deployment of Data Processing Applications.
DOI: 10.5220/0006266505230528
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 523-528
ISBN: 978-989-758-210-3

2 MOTIVATING EXAMPLE

We use a experimental application called WHEPET (WHEre PEople Tweet)[3] to illustrate the typical activities to create, run and evolve an application exploiting streams of real-time data. The objective of WHEPET is to show where people have recently posted tweets.

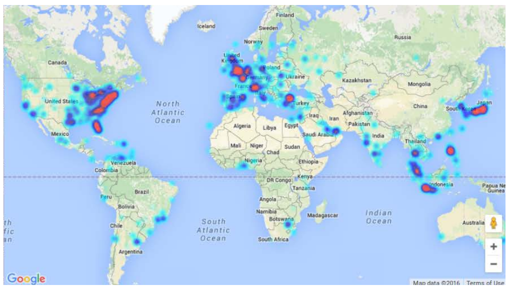
Figure 1: Screenshot of WHEPET heatmap.

The data processing comprises the tasks of obtaining the real-time tweets from the Twitter Public Streaming API [4], filtering tweets with coordinates, extracting these coordinates, and rendering them on the heatmap as depicted in Figure 1. Since how to analyse the data is not part of the challenges we address, the algorithms are deliberately simple.

The data retrieval tasks involve heavy data load (*i.e.*, about 500 tweets per second during our experiment), and we selected the Storm platform [5] to process them for scalability purpose. Following Storm's concepts, we wrap the tweets listening, filtering and extraction tasks as a *Spout* and two *Bolts*, respectively, and define a *Topology* to connect them. The Storm topology is deployed on a Storm cluster, which consists of four types of nodes, *i.e.*, Supervisor, UI, Nimbus and Zoopkeeper. The heatmap is realised in HTML using Google Map Widgets. A Kafka [6] message queue with a WebSocket wrapper plays as a mediator between Storm and the browsers.

We deployed all the required platforms, *i.e.*, Storm, Kafka and Kafka-websocket, on a virtual machine from Amazon EC2. When initally running this deployment, we noticed a big sudden drop in the rate of real-time tweets. To solve this, we exploited the Storm's built-in scaling support, to add a new Storm Supervisor node on a new virtual machine. At a later stage, we evolved the application to provide accurate statistics about the number of geo-tagged tweets posted from different countries. We developed a converter from coordinates to country names. The converter reads input from Kafka, and writes the output back via a different topic. A counter program consumes the output and updates a Redis database. For the scenario we wanted to deploy and integrate this new feature at runtime (*i.e.*, without stopping the running application).

[3]https://github.com/songhui/whepet

[4]https://dev.twitter.com/streaming/public

[5]http://storm.apache.org

[6]https://kafka.apache.org

The scenario illustrates the following requirements for DAMF:

Abstraction (R_1): DAMF should provide an abstract way to describe data flows and their deployments in a platform and cloud provider-independent and -specific way, *i.e.*, support the configuration of data processing flows, the implementation on specific platforms, and the deployment on provider resources respectively.

White- and Black-Box Control on Platform and Infrastructure (R_2): As for the example above, for some features it is possible to quickly obtain a running application, without knowing any details of the supporting platforms. However, for some features, we need the capability to configure the platforms themselves, *e.g.*, exploiting the Storm scaling out feature.

Modularity and Reusability (R_3): It should provide a modular, loosely-coupled specification of the data flow and its deployment so that the modules can be seamlessly substituted and reused. Elements or tasks should be reusable across scenarios, for example, deploying a Storm application.

Automation (R_4): We expect the automatic deployment of data processing applications. Indeed, experimental scenario requires tedious manual work. For example, we need to check the IP address and port of KafkaWS after deployment, and use them to reset the WebSocket server.

Reconcile Design- and Run-Time Activities (R_5): DAMF should support the continuous development and operation and frequent switching between design, implementation, and deployment activities.

3 THE APPROACH

DAMF leverages model-driven engineering techniques to support developers and operators in developing and operating data processing applications.

3.1 Approach Overview

Figure 2 illustrates the overview of our approach. The core of the approach is the DAML modelling language. The language encompasses three views (addressing R_1): *(i)* the platform-independent design of data flows, *(ii)* the platform-specific deployment of

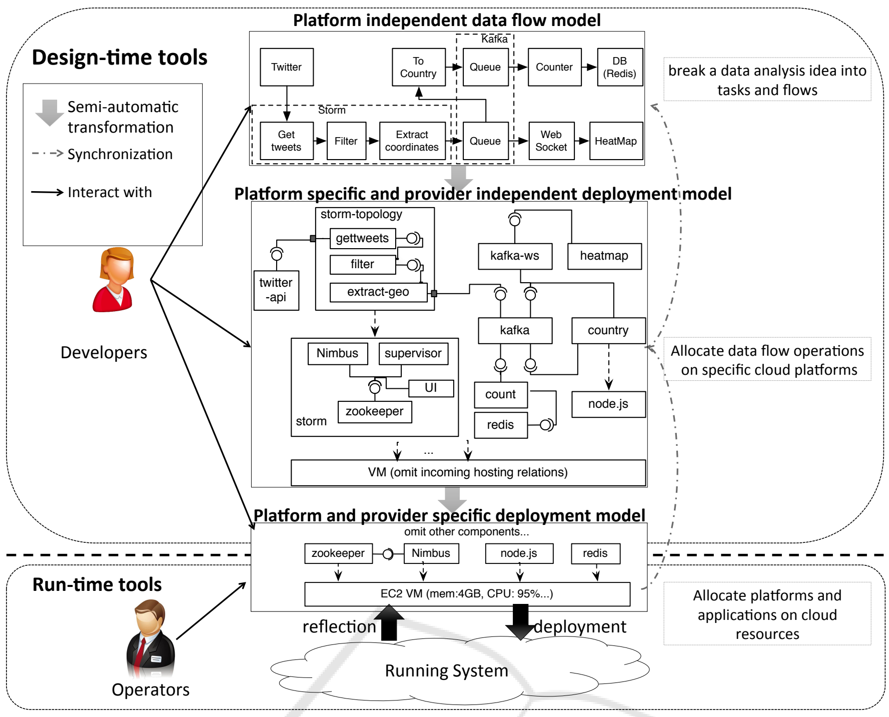

Figure 2: Approach Overview.

the application, and *(iii)* the platform-specific and cloud provider-specific deployment, which also includes runtime information (*e.g.*, public and private addresses, status) once the system is deployed.

Developers start the overall design of a data processing application from the platform-independent model, focusing on splitting the data processing task into sub-tasks. After that, they select the platforms to implement the sub-tasks, and lead the semi-automatic transformation from the data flow model into a platform-specific deployment model. The deployment model comprises deployable components and how they are hosted by the abstract resources. The component-based approach which better isolate concerns (addressing R_1) and ease reuse (addressing R_3). Developers can further adjust the generated deployment model (addressing R_2). If the adjustment impacts the original overall design, the changes will be synchronized to the data flow. Once the deployment model properly defined, developers choose the provider of the resources, and then launch the automatic deployment of the system on the selected resources (addressing R_4).

At runtime, the platform-specific model will be enriched with runtime informations and automatically synchronized with the running system (addressing R_5). Thus, the model represents the running system and any change in the system (*e.g.*, the Storm master crashes) is automatically reflected in the model. Similarly, any change on the model will trigger incremental deployment and cause the corresponding changes on the system.

3.2 Platform-independent Modelling

We support the platform-independent view of DAML by providing a data flow modelling language named DAFLOW. DAFLOW helps data engineers break a data analysis idea into tasks that are connected by flows. In addition, it also allows data engineers to annotate the tasks and flows with high level design decisions (*e.g.*, what platform will be used to implement a task).

As a platform-independent language used for early design, we keep DAFLOW simple and generic. The meta-model consists of only 3 essential data flow concepts, *i.e.*, `source`, `task`, and `flow`. We also introduce two auxiliary concepts, *i.e.*, `group` to encapsulate tasks, and `annotation` to add additional information to data flow elements. These concepts are platform independent. The annotation mechanism ensures the flexibility of DAFLOW. Data engineers can annotate, in an open style, any model elements with the relevant information such as the platform to use, the initial scale, *etc.* These annotations can be human-readable, or machine readable to determine how the platform-independent models are automatically translated into platform-specific ones.

```
data flow WhePeT {
  source TwitterStream
  task TwitterListener, Filter, ExtractCoord,
        HeatMap
  flow TwitterStream => TwitterListener
  flow TwitterListener => FilterGeoTagged
  flow FilterGeoTagged => ExtractCoord
  flow ExtractCoord => HeatMap
  group Storm : (platform="Storm")(initsize="1")
    {TwitterListener FilterGeoTagged ExtractCoord}
}
```

Figure 3: Initial DAFLOW model of WHEPET.

Figure 3 is an excerpt of the DAFLOW model that captures the initial design of WHEPET (See Section 2). In this round, we, playing the role of data engineers, first define the `sources` and `tasks` for analysing twitter data, and the `flows` between them, using the model elements marked with corresponding keywords. The textual model corresponds to an earlier version of the graphical data flow diagram shown in the top square of Figure 2. After defining the data flow, we record our early technical decisions by annotations, *e.g.*, the `group` of three processes that handle tweets will be hosted by the Storm platform.

High-level refinement and evolutions on the data flow level are also performed on the DAFLOW model, such as replacing the mismatched flow with two tasks related to the message queue and the WebSocket wrapper, and adding new tasks to convert coordinates to countries and count the appearance of each country. After these iterations, the final data flow evolves into the one as illustrated in the top of Figure 2. We omit the concrete textual model.

3.3 Platform-specific Modelling

The data flow model will be transformed into a deployment model as shown in the middle part of Fig 2. Data engineers can tune the deployment model concerning platform-specific parameters and configurations, and the infrastructures to host the platform.

Figure 2 illustrates the main concepts of DADEPLOY model. The core concept is *component*. A component can be a running service operated by a third party (such as the Twitter Streaming API, or an AWS-EC2 virtual machine), or a software artefact hosted by a service (such as a Kafka message broker). Such *hosting relationship* is represented by dashed arrows. A component may also exposes provided or required *ports*. A pair of matched *ports* can be connected by a *dependency* relationship, which means that the component with the required port “knows” how to access the component with the provided port, and therefore the former can invoke the latter to pull or push data. Finally, a *composite* component contains other components. The model depicted in Figure 2 includes one composite component representing a Storm topology, which consumes data from the Twitter Stream API, and is hosted by a Storm platform. The platform itself is in turn composed by 4 different Storm nodes. It is worth noting that in this case the hosting relationship is between two composite components, which means that the developers do not need to care about how the components within a storm topology is distributed into the different storm nodes - this is automatically handled by the storm platform. The last component inside the Storm topology will publish the extracted coordinates to Kafka. In the same time, the WebSocket wrapper subscribes to the same topic and sends the wrapped coordinates via WebSocket messages to the heatmap.

DADEPLOY provides a formal concrete syntax in a textual format. Figure 4 shows a sample model which defines two of the components depicted in Figure 2, the heatmap and the WebSocket wrapper. The example involves a key concept in DADEPLOY, *i.e.*, *prototype*, borrowed from the JavaScript object-oriented language, which also facilitates reusability (R_3) and abstraction (R_1). A component can be derived from another component as its prototype. The new component inherits all the features (*i.e.*, attributes and ports) from its prototype, as well as the values already bound to these features. Inside the definition of the new component, we add new features or override values of features defined by the prototype. For example, in Line 1 of Figure 4, we first defined a component to implement a HTTP server that can host one simple HTML file. The component is inherited from `dockercomp`, a predefined component for any Docker images. Inside the `one-page-httpd`, we set the actual image (the official python image) and the command associated to this image to download an HTML file and start a built-in python http server to host it.

Finally, the `configuration` part defines the component assembly of the application. It contains a `heatmap` component inherited from `one-page-http`, with a specific port 80 and a concrete page, and another component for the WebSocket wrapper. The two components are connected by a link between the required and provided ports from the two components, respectively. The components will be connected automatically during deployment: According to the link in Line 19, the tool will check where `ws` is deployed, in order to set the address and port values inside the required port `heatmap.wsport`. These values will be assigned to an environment variable `ws` inside the docker container (Line 14), for the http page to access the

```
component one-page-httpd prototype dockercomp{
  image: "python:2"
  command: "wget $this.resource;
    python -m SimpleHTTPServer 80"
  portmaps: {$this.httpport:80}
  httpport: None
  resource: None
}
configuration WhePeT{
  component heatmap prototype one-page-httpd{
    httpport:80
    resource: "https://github.com/songhui/\
      bigml-attempt/blob/master/vsempl/index.html"
    environment: "ws=$this.wsport.ip: \
      $this.wsport.port"
    required port wsport
  }
  component ws prototype kafka-ws
  component vm prototype ec2-big
  link heatmap.wsport -> ws.wsport
  host ws on vm
}
```

Figure 4: Excerpt of the DADEPLOY textual syntax.

WebSocket server.

The DADEPLOY model is machine-readable, and can be automatically deployed by DAML engine into a running application. The engine utilizes the mechanisms pre-defined along with a number of predefined root components, such as `dockercomp`, executable JAR file, EC2 virtual machine, etc. For each of these root components, the engine has built-in logics to automatically deploy it on the host component. Considering the `heatmap` component as an example, the engine will use the docker deployment logic defined with the root component `dockercomp` and compose the following docker commands, to be executed via ssh on the virtual machine.

```
docker run -d -p 80:80 python2 \\
  -e "ws=172.0.2.15:7080" \\
  bash -c "wget https://.../index.html && \\
    python -m SimpleHTTPServer 80"
```

Once the configuration model is completed, we specify the resources providers, *e.g.*, using the VM from EC2. The runtime engine will automatically deploy the components based on the generated scripts, and return a runtime model as shown in the last part of Figure 2. The runtime model maintains the same structure as the deployment model if the deployment succeeds (we omit most of the components in Figure 2 for the sake of simplicity), but its resource components carry the provider information and the runtime status of the resources, such as the actually memory allocated to the VM, the dynamic IP address, the current CPU load, *etc.*

3.4 Support for Model-driven DevOps

The DAML modelling language is the basis for the model-driven DevOps of data processing applications. Depending on the focus, developers and operators can work on any of the three views. Their changes on the model will be quickly implemented as evolutions of the running system, thanks to the transformations and the models@runtime engines.

The model-based DevOps of data processing applications is composed of a set of agile iterations of developing and evolving the application, each of which ends with visible effects on the running application. Take our WHEPET scenario as an example, our first iteration starts from a very simple DAFLOW model with data retrieving and text-based representing tasks, as shown in Figure 3. The second iteration is to scale out the location extraction component on the DADEPLOY model. The third iteration is to introduce new tasks into the DAFLOW model for country converting. Each iteration leads quickly to changes of the deployment of WHEPET.

The DevOps iterations are powered by the DAMF engines which implements the transformation between model views, and the causal-connection between the model and the system, as shown in Figure 2.

The *flow-deployment transformation* (the grey arrow on the top of Figure 2) takes as input the DAFLOW model with annotations of platforms and reused component types in DADEPLOY, and generates the DADEPLOY configurations to implement the data flow. It is a bidirectional model-to-model transformation, with traceability support to reflect subsequent changes on the deployment model into the original data flow. The *deployment-provider transformation* (the arrow in the middle of Figure 2 keeps the component configuration, but generates the provider-specific setting of the resources.

The *models@runtime engine* maintains the bidirectional causal connection between the provider-specific deployment model and the system running on the provider's resources. In one direction, the models@runtime engine deploys the configuration changes on the DADEPLOY model into system changes. The automatic incremental deployment is based on the OS-level provider-independent deployment tools, such as DockerCompose, as well as the provider-specific deployment APIs. The engine supports adaptive planning algorithms to design and optimize the order to invoke these built-in capabilities. In the other direction, the engine uses the provider's monitoring APIs to collect the system changes at runtime and reflects them into the model.

4 FUTURE WORK

On the basis of the DAML model, we will construct the complete DAMF framework to enable the semi-automatic DevOps supports of data processing system. The research around DAMF are mainly along three directions, *i.e.*, the modelling language, the engines and the applications.

For the language, we will provide both textual and graphical model editors, together with auxiliary tools such as model validation and auto-completion. We will further elaborate on both our prototyping approach and the type-object pattern (Atkinson and Kühne, 2002) to improve the reuse.

We will keep improving the model-to-model transformations and the models@runtime engines, in order to: *(i)* improve the synchronization between the models at the different levels of abstraction and *(ii)* support the mainstream data processing platforms and the cloud resource providers. We will also work on the model-to-text transformation to support the third-party deployment engines such as chef (www.chef.io).

Finally, we will extend DAMF with support for the continuous delivery of trustworthy data processing applications. The model will be used to reveal and visualize the provenance and how the data are actually flowing through the different tasks and resources.

5 RELATED WORK

This work is an extension to the approaches on model-driven DevOps in cloud. CloudML (Ferry et al., 2015) provides domains-specific languages and and engines to support the vendor-independent modeling and automatic deployment of cloud applications. Artist (Menychtas et al., 2014) provides model-based support for the migration of legacy software into cloud. Our approach follows the same direction but are specific to data-processing applications, with special views on data processing and also focuses on the control of data processing at different levels.

The DICE project (Casale et al., 2015) also leverage model-driven engineering to manage data processing applications, and aims at providing a new UML profile for developers to understand and analyse the applications. Different from our approach, DICE focuses on the non-functional perspectives of the applications, such as reliability and performance, whereas we focus on the rapid construction of data processing from a functional point of view.

The approach is based on a set of advanced model-driven engineering research topics, *i.e.*, domain-specific language engineering (Kelly and Tolvanen, 2008), bidirectional model transformation (Czarnecki et al., 2009) and models@runtime (Blair et al., 2009).

6 CONCLUSION

In this position paper we present a model-driven approach to the development and operation of data processing applications. Using the sample case,we reveal the possibility and potential benefits of designing and managing data processing applications from a high abstract level using modelling languages, and automating the DevOps processes though transformation and models@runtime mechanisms.

ACKNOWLEDGEMENTS

This research has received funding from the European Community's H2020 Programs under grant agreement number FoF-2015.680478 (MC-Suite), and the Norwegian Research Council under grant agreement 257153 (Open Transport Data)

REFERENCES

Atkinson, C. and Kühne, T. (2002). Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation*, 12(4):290–321.

Blair, G., Bencomo, N., and France, R. B. (2009). Models@ run. time. *Computer*, 42(10):22–27.

Casale, G., Ardagna, D., Artac, M., et al. (2015). Dice: quality-driven development of data-intensive cloud applications. In *7th MiSE workshop*, pages 78–83.

Czarnecki, K., Foster, J. N., and Hu, Z. (2009). Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations*, pages 260–283.

Ferry, N., Chauvel, F., Song, H., and Solberg, A. (2015). Continuous deployment of multi-cloud systems. In *QUDOS workshop*, pages 27–28. ACM.

Hüttermann, M. (2012). *DevOps for developers*. Apress.

Kelly, S. and Tolvanen, J.-P. (2008). *Domain-specific modeling: enabling full code generation*. John Wiley & Sons.

Menychtas, A., Konstanteli, K., Alonso, J., Orue-Echevarria, L., Gorronogoitia, et al. (2014). Software modernization and cloudification using the artist migration methodology and framework. *Scalable Computing: Practice and Experience*, 15(2):131–152.