StratusML: A Layered Cloud Modeling Framework

Mohammad Hamdaqa, Ladan Tahvildari
Software Technologies Applied Research (STAR) Group
Department of Electrical and Computer Engineering
University of Waterloo, Canada
{mhamdaqa, ltahvild}@uwaterloo.ca

***Abstract*—The main quest for cloud stakeholders is to find an optimal deployment architecture for cloud applications that maximizes availability, minimizes cost, and addresses portability and scalability. Unfortunately, the lack of a unified definition and adequate modeling language and methodologies that address the cloud domain specific characteristics makes architecting efficient cloud applications a daunting task. This paper introduces StratusML: a technology agnostic integrated modeling framework for cloud applications. StratusML provides an intuitive user interface that allows the cloud stakeholders (i.e., providers, developers, administrators, and financial decision makers) to define their application services, configure them, specify the applications' behaviour at runtime through a set of adaptation rules, and estimate cost under diverse cloud platforms and configurations. Moreover, through a set of model transformation templates, StratusML maintains consistency between the various artifacts of cloud applications. This paper presents StratusML and illustrates its usefulness and practical applicability from different stakeholder perspectives. A demo video, usage scenario and other relevant information can be found at the StratusML webpage [1].**

I. Introduction

It is widely acknowledged within the software engineering community that architectural languages are defined by stakeholder concerns [2], [3], and that a single "universal" notation (e.g., UML) is impractical to address all the concerns of a particular domain [3]. A domain language is usually needed. In the cloud domain, an application evolves *at runtime* to meet performance, availability, and scalability targets under changing conditions; a routine task that involves continuous changes to their service models and deployment artifacts. While each stakeholder may conduct a certain type of change to address a specific concern, the impact of a change may span multiple models and influence the decisions of several stakeholders. This paper presents StratusML, a modeling framework and domain specific modeling language for cloud applications. The aim of StratusML is to satisfy the cloud stakeholders need to model and evolve applications that leverage cloud computing for maximum value with minimum conflicts. The original contributions of StratusML: (i) it provides multiple views and different layers to address the various cloud stakeholders concerns (ii) it facilitates visual modeling of adaptation rules and actions, in order to specify the dynamic behaviour of applications at runtime, and (iii) it enables generating the configuration space artifacts of a selected target platform, by utilizing template-based transformation.

The rest of this paper is organized as follows. Section II motivates the need for a domain specific modeling language for the cloud configuration space. Section III provides an overview of the StratusML modeling framework. Section IV highlights the framework capabilities using an example. Section V covers the related work. Finally, Section VI concludes the paper .

II. Preliminaries

Cloud computing shifted the control to the provider's side to help organizations focus on business functionalities. However, organizations still need to utilize the platform components to design reachable, scalable and available applications.

A. Domain Challenges

The main challenges facing cloud adopters include:

(a) The Vendor Lock-in Problem: The cloud "pay per use" model promotes business agility and promises cost saving. However, for this model to make sense, customers should have the ability to port their applications between providers to maximize business opportunities. Without portability, migrating applications will require partially modifying or even rewriting the application, which is difficult and costly. Therefore, customers will be locked within a particular vendor. This problem is usually referred to as the "vendor look-in" problem [4].
(b) Deployment Architecture Mismatch: Cloud computing promotes reuse at all levels (i.e., Infrastructure, Platform and Software). Unfortunately, "as the level of reuse and the complexity of assumption increase, architectural mismatch become more of an issue [5]" that requires advanced software engineering solutions. Mismatch occurs due to hidden assumptions about (i) the application domain, (ii) components at the same level of abstraction, and (iii) the infrastructure. Cloud dynamics represented by the multiple providers and continuous updates creates a mismatch at the architectural deployment level. Therefore, an architecture description language that makes these assumptions explicit is needed.
(c) Lack of Domain Concepts and Methodologies: Cloud applications have their own identity that developers need to understand. A cloud application should be designed from modular components that have the ability to be re-used, re-configured, re-combined, and re-composed. A cloud modeling language must provide the concepts to specify a cloud architecture deployment model for a multi-tenant and web-farm friendly application so that the application can be distributed, parallelized, and hosted in multiple locations [6]. A modeling language should also provide patterns that enable loosely coupled asynchronous interactions and late binding and specifying auto-scaling rules and availability zones and scenarios.
(d) The Gap Between Cloud Stakeholders: Existing cloud modeling frameworks address the requirements of each stakeholder individually. An efficient modeling framework should

provide a holistic view that facilitates collaboration between the different stakeholders at the various cloud service levels.

To address the preceding challenges, we built StratusML.

B. A Language for the Cloud Configuration Space

Deploying an application on a cloud platform requires specifying how the application service model will use the platform resources of that particular provider. This involves specifying (i) the service model, which defines the structure of the service itself in terms of the software modules that compose the service and how those modules are communicating, (ii) the runtime deployment configurations that specify how the service model modules are instantiated and replicated, and (iii) the behaviour of the application at runtime under diverse conditions. This behaviour is usually specified using an adaptation model, which is a set of rules and actions.

Listings 1, 2 and 3 show respectively a service definition (service model), a configuration file (runtime deployment configuration), and an adaptation model that are used to deploy an application on Windows Azure platform. The syntax of these files conforms to the azure platform schemas. The service definition file in Listing 1 describes a cloud application that uses one *role*. A *role* in Windows Azure refers to a virtual appliance that is prepared with the required software stack to run a certain family of applications (i.e., web, or back-end). The service configuration file further specifies the service definition by assigning values to the configuration settings defined in the service definition file. For example, the service configuration in Listing 2 specifies the number of instances of the worker role. Finally the adaptation model in Listing 3 shows a reactive rule “ScaleUp”; which is used to scale the role “ShoppingCartProcessing” when the average CPU utilization exceeds 75%. While this example is based on Windows Azure application packaging specifications; the information required to specify a cloud application deployment is essentially the same (e.g., the previously described role is equivalent to Amazon AWS beanstalk and GAE Module [7]). It is apparent from this example that managing all these related artifacts and maintaining consistency between them requires a holistic view

```
<ServiceDefinition>
  <WorkerRole name="ShoppingCartProcessing" vmsize="
   Small"=>
    <ConfigurationSettings>
      <Setting name.."DataConnection" />
    </ConfigurationSettings>
  </WorkerRole>
</ServiceDefinition>
```

Listing 1: Example of Service Definition File.

```
<ServiceConfiguration>
  <Role name="ShoppingCartProcessing">
  <Instances count="2" />
    <ConfigurationSettings>
      <Setting name="DataConnection"
        value="UseDevelopmentStorage=true" />
    </ConfigurationSettings>
  </Role>
</ServiceConfiguration>
```

Listing 2: Example of Service Configuration File.

```
<rules>
  <reactiveRules>
    <rule name="ScaleUp" description="Increases
     instance count" enabled="true" rank="3">
      <when>
        <greaterOrEqual operand="Avg_CPU" than="75"
          />
      </when>
      <actions>
        <scale target="ShoppingCartProcessing" by="
         1" />
      </actions>
    </rule>
  </reactiveRules>
  <operands>
    <performanceCounter alias="Avg_CPU"
      performanceCounterName="\Processor(_Total)\%
       Processor Time" aggregate="Average" source="
       RoleB" timespan="00:10:00" />
  </operands>
</rules>
```

Listing 3: Example of Windows Azure Adaptation Model.

that integrates them. Moreover, in order to deploy the same application on multiple providers and facilitate its migration, there is a need to provide a layer of abstraction that captures the domain concepts then identifies the mapping between the domain independent concepts and the deployment-description artifacts concepts of the different providers. This paper shows an example of a language that provides this abstraction and a framework that facilitates model integration and visualization. The paper also explains how this framework can be used to address the cloud domain challenges.

III. The Stratus Modeling Language and Framwork

This section presents the StratusML features, meta-model, implementation and users. More details can be found on the StratusML web page [1].

A. StratusML Features

StratusML is a modeling framework for cloud applications. It enables the design of high-quality distributed applications that are tailored to be deployed on the cloud. Through layers StratusML empowers the cloud stakeholders to view the models each from their perspective. This ability to separate between concerns makes working with complicated models more efficient. A layer can be turned on/off at any time providing a holistic or partial view. StratusML supports visual modeling of adaptation rules and constraints, it also automates the generation of the corresponding artifacts for the target adaptation manager. StratusML supports the generation of complete platform specific artifacts based on template-based transformation.

Using templates and layered modeling, generating complete platform specific artifacts, and the ability to visually model adaptation rules and actions are the main distinctive advantages of StratusML over existing frameworks. However, StratusML provides several other features that make developing and managing cloud applications a seamless experience. The StratusML allows users to (i) define a cloud application deployment model, and partition components into groups based

on geolocation, scaling factors, and/or functionality, (ii) specify a cloud application configurations and adaptation rules (e.g., auto-scaling rules). (iii) select a cloud provider or create a custom one (iv) estimate the applications' running cost, and (v) use templates to transform the model into platform specific artifacts (e.g., Azure definition files).

B. The StratusML Framework

As shown in Fig.1, the StratusML framework covers model creation, validation transformation, and adaptation. Its architecture adheres to the Model-View-Controller (MVC) style. This aims at maintaining the models consistent by performing the required model *transformations*, *validation*, and *analysis* whenever the models are updated. Both model *validator* and *editor/viewer* use the *StratusML meta-model*.

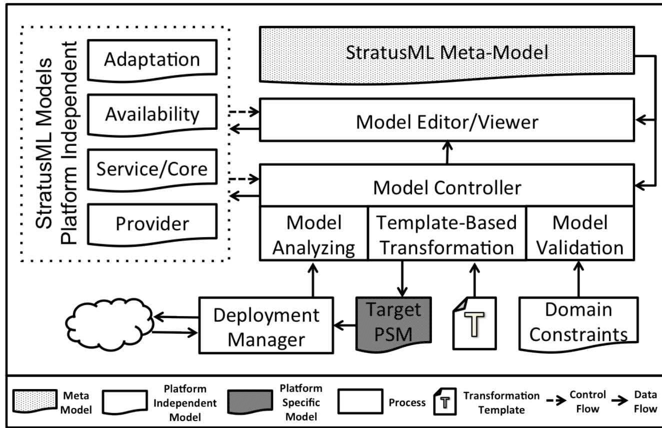

Fig. 1: The StratusML Framework Architecture

The StratusML meta-model integrates five different meta-models with one *core* meta-model to address five different, but interleaved concerns of the cloud applications (i.e., *core* (service composition), *availability* (distibution and replication), *adaptation* (elasticity and dynamic behaviour), *provider*, *performance* and *workflow*). The usage scenario explained in this paper focuses on the integration of the core model with three of its prespectives (i.e., availability, adaptation, and provider), leaving the performance and workflow meta-models for the future. The StratusML meta-models have been explained in detail in [8] and [9] and made available online for reference in [1]. Fig. 2 shows a part of the StratusML metamodel that focuses on the aforementioned three prespectives. In a nutshell, the *core* meta-model acts as a pivot model that captures the concepts of most cloud providers and describes the deployment architecture of the cloud application in terms of *tasks* and interactions, where a cloud *Task* refers to a software module that is composed of sub-processes called *activities* and wrapped with the required software stack in virtual machine image. The core meta-model facilitates specifying cloud *service* definitions and initial configurations. A cloud *service* is composed of several components (i.e., *GroupableCoreComponents*) that can be nested into *Groups* so that certain properties such as location constraints (i.e., *Availability Groups*) or scaling ratios (i.e., *Scalability Groups*) would be applied to all of them. Each of the other meta-models are used to further enrich the semantics of the core model. The *adaptation* meta-model is used to specify the adaptation *rules* and *actions* for each task or group of tasks in the core model. The meta-model enables specifying two types of actions, *predefined* and *custom* actions. While the process of predefined actions is known a head of time (e.g., scaler action), custom actions allow assigning external processes. An *action* is triggered based on a constraint or a reactive *adaptation rule*. A *ConstraintRule* is a predefined (static) constraint, such as the minimum or maximum number of instances allowed at a certain time; whereas, a *ReactiveRule* is a dynamic rule based on evaluating some runtime environment parameters against a set of key performance indicators (e.g., CPU utilization, queue length, response time), such indicators are collected by the platform through diagnostic API's (e.g., Azure diagnostic monitor[1]). The *availability* meta-model provides the components required to instantiate the core model and distribute its instances into different geographic locations. Finally, the *provider* meta-model is used to create or import a provider profile, which consists of the provided service templates and the pricing profile. As shown in Fig. 3, each of these meta-models has its own layer to be viewed on the modeling IDE. This integration of all the meta-models is what gives the cloud models their unique characteristics.

The StratusML framework supports two types of transformations, a *model transformation* that is used to specialize the Platform Independent Models (PIM) into Provider Specific Models (PrSM), and a *template-based transformation* [10], which is used to generate the Platform Specific Models (PSM) for the target platforms. Unlike other approaches that incorporate model-driven engineering to solve the vendor lock-in problem, StratusML employs template-based transformation that is supported with an automatic schema matching technique [7] to generate the different cloud model artifacts. The template-based transformation is a key feature in the StratusML framework. The transformation engine uses the validated StratusML model, and applies template transformation to it to generate a target model. The model encapsulates the essential data about the entities that need to be generated, while the template dictates the syntax of the target model. The template transformation engine produces the target model by replacing the template internal references with real data coming from the model according to the transformation rules specified in a procedural way in the template. Template-based transformation provides flexibility; as you can generate all types of models without changing the transformation engine, and portability; as the data model and engine are not touched. Moreover, the transformation syntax is simple, which facilitates reusability and productivity. A sample template that generates Windows Azure configuration from StratusML models is provided in Section V.

C. Implementation

The StratusML framework is built as an extension of Microsoft Visual Studio 2012. In particular, we used the Microsoft DSL toolkit [11] to design the StratusML visual designers and Microsoft Text Template Transformation Toolkit (T4) to produce the different artifacts generators. Microsoft DSL model designer was used to define the different meta-models and then map each concept in these meta-models to

[1] http://msdn.microsoft.com/en-us/library/dn535595.aspx

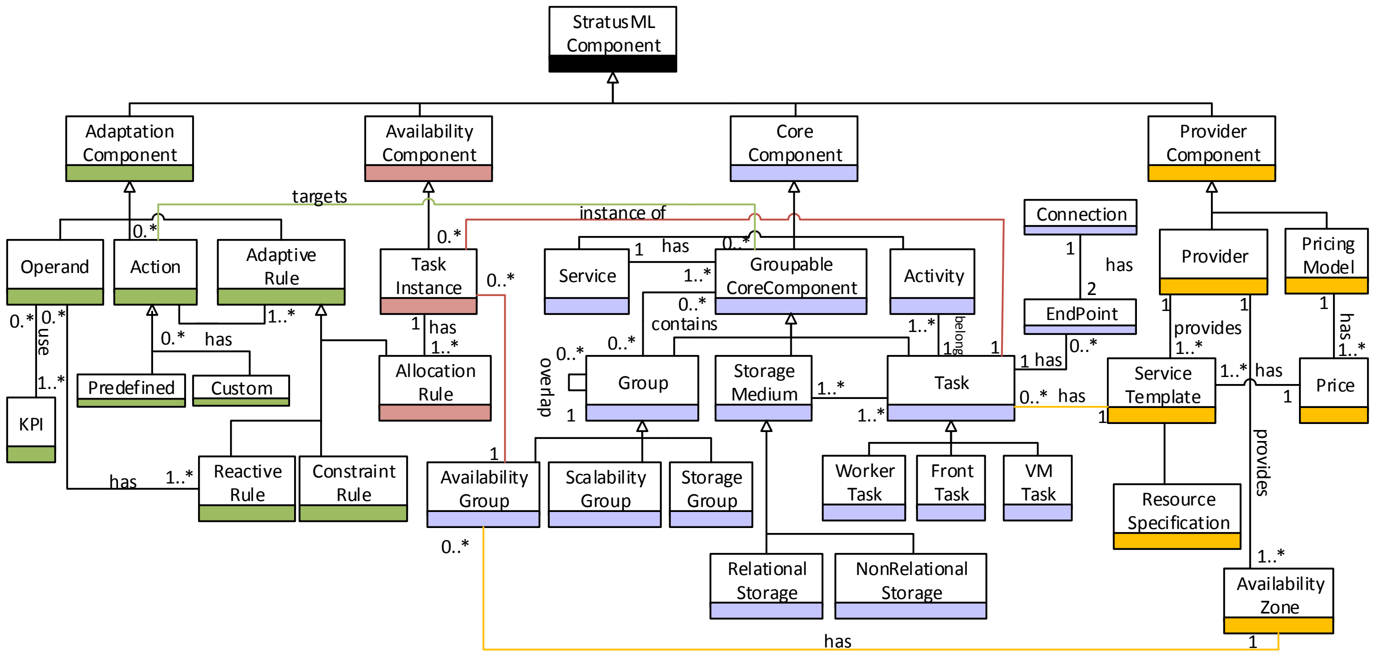

Fig. 2: Part of The StratusML Meta-Model

its corresponding visual component shapes and decorators. A custom code has been used for creating the layering feature and to provide advanced validation. By combining T4 and StratusML meta-models, the user of the StratusML framework can easily generate artifacts for any target platform. Creating a new transformation template is no different than writing a simple procedural program. This alleviates the need to learn complex transformation syntax (e.g., XSLT). StratusML enables stakeholders to add new files of (.stratus) extension that enable them to build their model, and build or utilize the provided transformation templates to transform the models into the desired output files.

Fig. 3 shows a snapshot of the design window of the StratusML modeling framework. The tool box on the left side can be used to explore categories, and expose the different components. Users can drag and drop components as needed in the model design window, connect them and view and change their properties using the property window on the far right hand side. StratusML offers more than 60 domain concepts and more than 150 domain constraints. These constraints can be used to ensure model completeness and to steer the stakeholders decisions to the right design. In the middle right hand side, you can find the model information tab. The upper part is used to calculate the components availability (e.g., low, medium , high) based on the number of instantiated instances of each module and how they are distributed into different geographic locations. While the lower part is used to estimate the price of the current configuration based on a selected provider. Finally, on the bottom is the views bar that allows the users to toggle between partial and holistic views that represent different aspects of the model. StratusML installation instructions, and a detailed step-by-step usage scenario is provided in the StratusML web page [1], which also presents and explains the different meta-models and validation rules that represent the StratusML syntax and semantics.

D. *The StratusML Envisioned Users*

The following list describes how four different cloud stakeholders can utilize StratusML in their job: (i) Providers can specify the resources and services they provide. (ii) Developers can define their application services. (iii) Administrators can configure the services for deployment, specify the application runtime behaviour through a set of adaptation rules, and evaluate their performance. (iv) Financial Managers can estimate the cost of deployment onto different platforms. Next section provides an example of how StratusML can be utilized to address the aforementioned cloud stakeholders' challenges.

IV. Stratus Framework Capabilities

A platform provider (e.g., Windows Azure) should empower the developers to build high quality applications. A modeling language (e.g., StratusML) assists stakeholders to make the right decisions to fully utilize the platform. This section shows by example how StratusML contributes to solve the typical challenges that face any organization adopting cloud computing.

A. *The CoupoNet Scenario*

Let's consider CoupoNet; a fictitious startup company that offers coupon services on the cloud. The software is a multi-tier and multi-tenant application that works as follows: CoupoNet tenants obtain a free trial or paid subscription that allows them to design and post coupons and publish them based on the target customers geolocation. The application stores the buying and selling data and performs sophisticated analytics to rank and position the offers, provides statistical data to the subscribers, and analyzes the users interactions to dynamically updates the CoupoNet business model. The CoupoNet team decided to host their application on the cloud

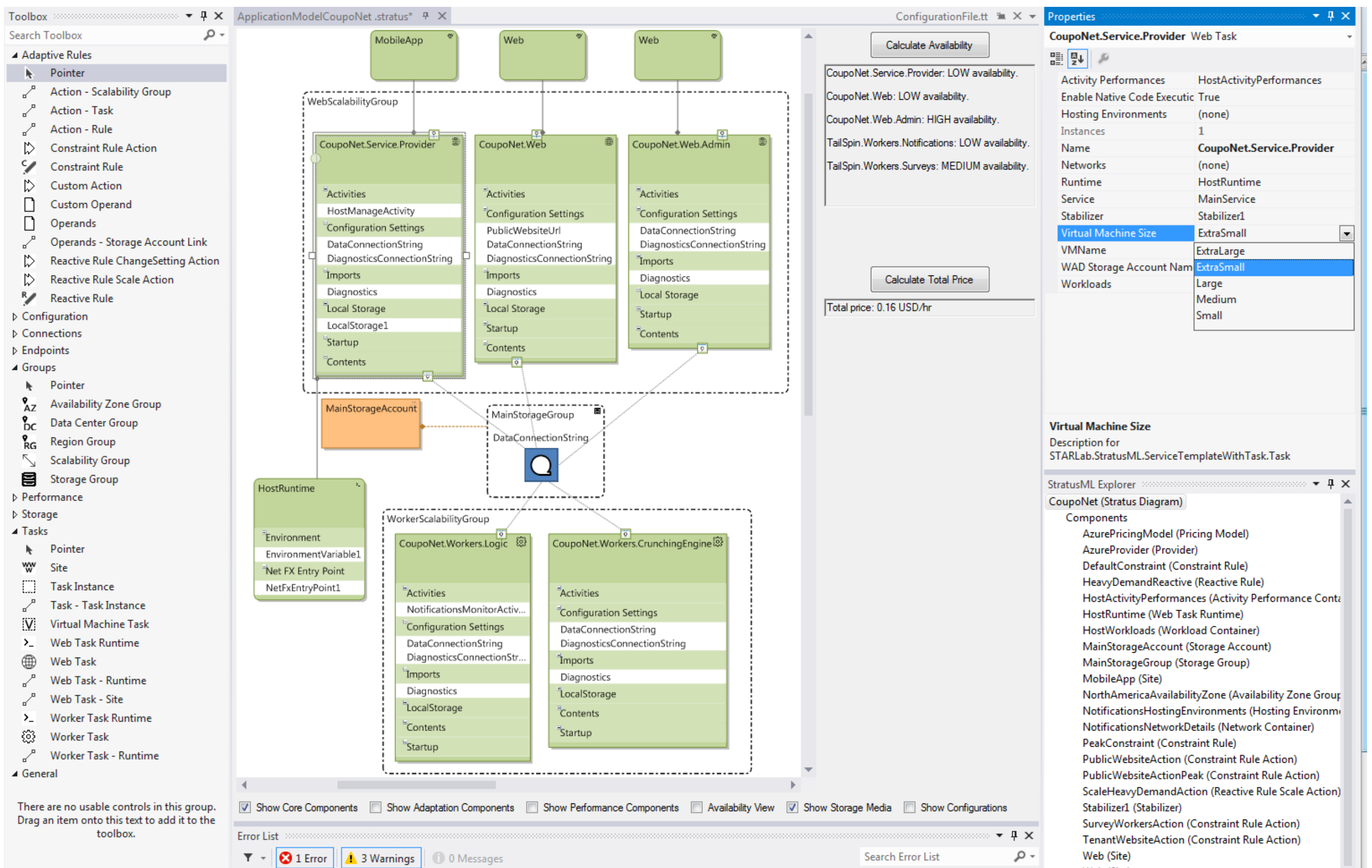

Fig. 3: A Snapshot of the Design Window of the StratusML Modeling Framework

to harness its benefits; cut initial costs, provide reliable services, and minimize administrative and configuration tasks. As a start-up, there are many challenges to be met. The CoupoNet team is unsure how much resources they need. The coupons market is vibrant; coupons can be seasonal, time limited and susceptible to slash dot effect; no one can predict when a coupon will become popular. The CoupoNet site should be able to respond quickly to increasing demand. CoupoNet is also unsure of how to distribute the services geographically to insure the highest availability and minimize the traffic overhead, or how to assure security and compliance. Cost wise, CoupoNet is unsure which provider to select; while currently provider X provides the cheapest services, CoupoNet developers are familiar with provider Y's technologies. Moreover, CoupoNet expects a more competitive offer next year from CloudKick a third provider. From a technical point of view, CoupoNet's architect designed the application to ensure components have lowest coupling and highest cohesion. However, at the time of the deployment, the system administrator, who is unaware of the architect design decisions, may redistribute the components to minimize cost and initialize them with arbitrary ratios based on his best estimation. As the demand fluctuates the administrator needs to update the deployment model and reconfigure the different services.

B. *Using the StratusML Framework Modeling Features*

This section demonstrates the capabilities of StratusML using the CoupoNet example. Table I summarizes how StratusML can be used to address the CoupoNet team challenges. The first column shows the challenges that CoupoNet team faces. The second column shows the StratusML layer that is used to address each challenge, while column three provides the specific features that address the challenge. The table demonstrates the benefits of using the layering feature, and how the different views can foster collaboration between the different cloud stakeholders.

StratusML provides a number of features. However, to make the paper concise and to adhere to the page limits, the rest of the paper will only focus on how to use StratusML to capture the application deployment configuration and to generate the required artifacts to deploy a cloud application on a target platfrom (e.g. Windows Azure), by transforming the StratusML provider independent models into provider specific configurations. Capturing the application deployment configuration includes: (i) the application static structure (a.k.a, service model) (ii) the application runtime model (i.e., how the different components are instantiated, distributed and replicated), and (iii) the application dynamic behaviour at runtime (a.k.a., adaptation model). Here is how StratusML can be used to specify such information:

TABLE I: Addressing CoupoNet Challenges

CoupoNet Challenges	Modeling Layer	StratusML Solutions
Model multi-tenant applications	Core	Provide different groups (i.e., storage, availability, and scalability groups) that address multi-tenancy by applying different service/data partitioning strategies
Model multi-tier applications	Core	The core meta-model provides platform independent task-templates for frontend, backend and cloud-storages. It also provides connections to describe the different interactions between Tasks.
Communicate architectural decisions to administrators.	Core	StratusML groups can overlap with each other. Using these groups architects can ensure their original decisions, which aim to reduce coupling and increase cohesiveness, are maintained. Grouped components will always stay, migrate and scale together.
A service distribution into multi geo-graphic locations	Availability	Provide availability groups that facilitate managing the service instances locations and counts
Uncertainty of required resources	Adaptation	Facilitate modeling for adaptation rules and actions with focus on scalability actions. A user can specify constraint and reactive rules, and associate them to task or scaling groups. The framework generates the required rule-based configurations to automate resources provisioning
Evaluate different provider offerings	Provider	A user can select one of the available providers or create a custom provider. The system will estimate the cost of deploying the application on the selected provider.
Migrating between different providers	Provider	Provide a set of templates that can be customized to any platform in order to automatically generate all the target platform artifacts
Minimize administration and configuration tasks. /model co-evolution	Provider	Provide a set of templates that can be customized to any platform in order to automatically generate all the target platform artifacts

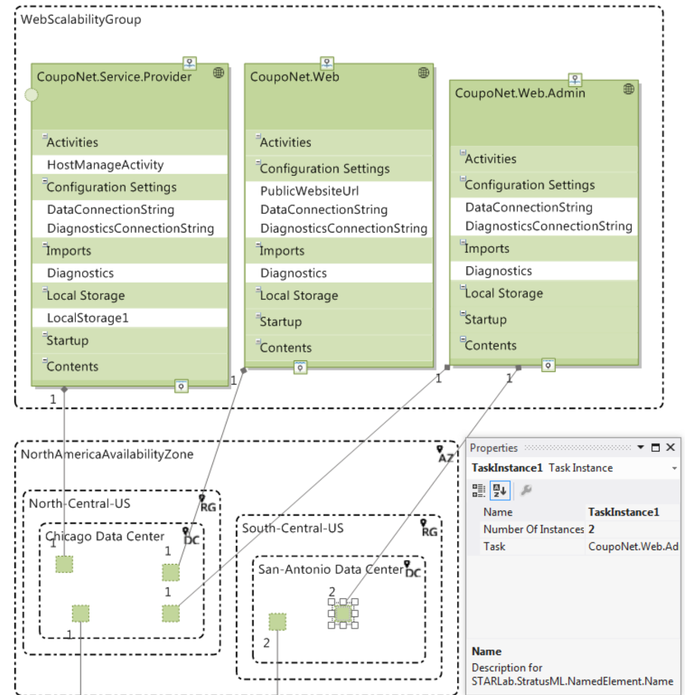

Fig. 4: StratusML Availability View Excerpt

Define a Cloud Application Service Model (Structure): The core layer provides a set of visual components that corresponds to the StratusML core meta-model elements. Using the *core* meta-model service developers can describe the structure of a cloud service composed of one or more *Tasks*, the types of tasks, and their relationships. For example, the model in the design window of Fig. 3 shows a provider independent deployment model that corresponds to the CoupoNet example. The model is composed of one service (not shown in the model as it is part of the hidden configuration view) with three web tasks (i.e., frontend modules) and two worker tasks (i.e., backend modules). Each web task has at least one external endpoint that must be secure and is a frontend web MVC-style application to be accessed by specific user groups (i.e., Coupon Providers, Coupon Buyers, Admins and Marketing Researchers). The first worker task corresponds to the application backend that handles all operations (logic tier). The second is the analytics/data-crunching engine, which processes buy/sell data. There is also a storage tier, which consist of blobs for storing data collected from buy/sell dumps, and queues for asynchronous communication between worker and web tasks. The core layer furnishes various groups (i.e., storage, availability, and scalability) that assist in modeling multi-tenancy using different service/data partitioning strategies.

Specifying the Modules Replication and Distribution: The availability of the system depends on how the different components and modules are replicated and distributed into different regions. StratusML provides a modeling view that makes it easier to manage the locations and counts of cloud task instances. This helps administrators model cloud service distribution to multi-geographic locations. Fig. 4 shows how the CoupoNet tasks are replicated and distributed. For example three instances have been instaniated for task "CoupoNet.Web.Admin" in two different regions both in North America, one instance in the North Central "Chicago" datacenter and two instances in the South Central "San Antonio" datacenter as shown from link cardinality. An administrator can easily instruct the cloud fabric to relocate the instance into a different location within the same provider or even a different provider by changing the instance properties.

Specifiying the Application Behaviour (Adaptation Model): Using the adaptation layer, StratusML facilitates modeling adaptation rules and actions, with a focus on scalability actions. A user can specify constraints and reactive rules, and associate them with a task or scaling group. The framework generates the rule-based configurations required to automate resource provisioning. This solves the problem when required resources cannot be estimated at the beginning of the project.

Fig. 5 shows a screenshot of a task (*CoupoNet.Workers.Logic*) that is associated with an adaptation action (*ScaleHeavyDemandAction*) that is activated based on a reactive rule (*HeavyDemandReactive*). The reactive rule has the rank 2. It can be enabled or disabled when needed. The activation of the reactive rule depends on the value of the operand (*HeavyDemandOperand*) and activated when the demand is greater than 75%. The value of the operand is collected at runtime. The *MainStorageAccount* component is used to set the configuration parameters of the diagnostic API.

Creating and Selecting a Cloud Provider: StratusML users can use one of the supported providers or create a new

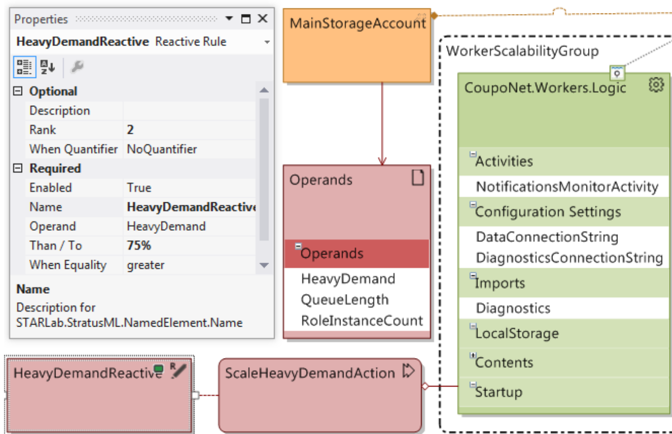

Fig. 5: StratusML Adaptation Rule and Action Excerpt

one. The provider layer provides a set of components to describe providers' specifications. This includes; specifying (i) the *service templates*, which describe the resources provided in bundles (i.e., CPU speed, number of cores, memory size, disk space), (ii) the *availability zones* that represent the physical locations of provider data-centers, and (iii) the *pricing profiles* which specify the cost of using a VM with a specific service template under the designated provider. The information specified can be used to estimate the system performance and calculate the cost of deployment under various providers. Using and generating analytical performance models from the specified information in the deployment and service models is out of the scope of this paper and will be discussed in detail in a separate research paper.

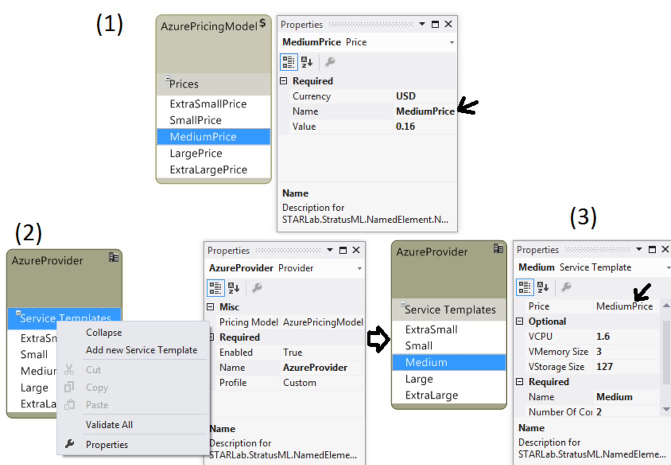

Fig. 6: StratusML Provider Snapshot

Fig.6 corresponds to Windows Azure provider model. The figure shows the list of service templates offered (e.g., small, medium) by Azure, the pricing model of Azure and how each of the service templates can be assigned a pricing profile.

C. Model Validation

There is always a need to check the correctness and completeness of the models created or generated using a DSML. Validating a model includes checking:

(a) **The model structural constraints and well-formedness rules**: Structural and well-formedness rules can be specified by superclasses together with the multiplicity and type information expressed in the language meta-model. These are *hard* constraints that are normally verified automatically by the modeling framework. An example of a structural constraint is setting the lower cardinality of a component to one in the meta-model. This constraint enforces that a model must be created with at least one component of this type (i.e., an empty model is invalid).

(b) **The model semantic constraints**: Semantic constraints can be checked by defining *invariants* to ensure model correctness and domain conformance. These *invariants* can be implemented as soft or hard constraints (e.g., each task must have a unique name). Moreover, depending on the validation context there may also be a need to define *pre-conditions* and *post-conditions* as in the case of *method* validation to ensure context state validity before and after the execution of the method.

(c) **The transformation constraints**: Those are the constraints used to verify the correctness and completeness of the information needed to generate the target model before a transformation.

StratusML utilizes Microsoft DSL framework to define the validation rules required to ensure that the specified model satisfies the basic domain requirements and provides the information required to generate the target platform specific artifacts. Microsoft DSL automatically generates the validation methods for the structural constraints (e.g., checking minimum multiplicity based on the defined meta-model). However, for semantic and transformation constraints, the validation constraints must be explicitly defined by adding validation methods to the domain classes or relationships of the DSL.

In Microsoft DSL, validation constraints can be classified into hard and soft constraints. Hard constraints are those that the user can never violate (i.e., it prevents the user from making modeling mistakes). In contrast, soft constraints are allowed to be violated, but still create warnings and errors to guide the user to the correct decisions. For example, a constraint that enforces that no two tasks in a model should have the same name could be implemented either as a hard constraint or as a soft constraint. If it is implemented as a hard constraint, then a user will not be allowed "at any point in time" to create a task that has the same name as another task. However, if it is implemented as a soft constraint, then a user is allowed to create a task with the same name as another task, but an error will be generated to instruct the user of the problem. Hard and soft constraints provide different usability experiance. While hard constraints make the design environment rigid, lots of soft constraint may result in deferring errors discovery to the time of saving the model. This cloud lead to error accumulation and hence user frustration. StratusML mixes between soft and hard constraints, a StratusML model cannot be saved if errors still exist. For example, in the case of CoupoNet each of the web tasks external endpoints must be secure. The

model validator ensures this by checking if each of these endpoints is using SSL for communication and that an SSL certificate is assigned for each external port. If an external endpoint is not using SSL or does not have a certificate, an error message will be displayed and the model will not be saved unless the error is resolved. In addition to hard and soft constraints, StratusML also differentiates validation rules based on generality into domain specific or generic rules. Both domain specific and generic rules are used to specify semantics of the cloud concepts beyond the meta-model, such as the rules of groups nesting. Domain specific constraints are classified in StratusML based on the layer of the concepts it validates (e.g., availability, adaptation). On the other hand, generic rules are the rules used to enhance the modeling experience in general, such as those checking for name duplication.

Each validation rule is applied to a *scope* (i.e., class, relationship). The rule scope specifies the *context* (i.e., name of the class or relationship), which the rule is validating and its attributes or methods. Running a validation, either by a user or under program control, executes some or all of the validation methods. Each method is applied to each instance of its class. Moreover, there can be several validation methods in each class. Validation is executed as a response to an event *trigger*. The types of triggers supported in Microsoft DSL and used in StratusML are: file *Open*, *Load*, *Save*, and *Custom* triggers. A *Custom* trigger is normally implemented as an event handler. Each rule validation method is implemented using a list of invariants, which are the conditions the rule validates. Whether the rule invariant is *valid* or *invalid* a set of actions can be activated. The actions can be *error* or *warning* messages, or custom actions.

The following are two examples of the validation rules used in StratusML (i.e., a soft and a hard rule). The complete list of the validation constraints and the validation code can be found in the StratusML webpage [1].

Example 1 - A Hard Validation Rule: Fig. 7 is an example of a validation code for a hard rule. The first line represents the context where the rule is applied (i.e., *Group* class). As shown in lines three and four, the rule is triggered in response to a custom event, which is a model element creation of the type *GroupableComponent*. Line five is the invariant, which validates that if the created element has a *Group*, and the element is of type *Storage* then the *Group* must be of the type *StorageGroup*. As a result, if the invariant is invalid then an error will be shown and the component created will be automatically deleted. This is an example of hard constraint as it prevents the user of creating the component by deleting the created component, in order to preserve the context initial state.

```
1. [RuleOn(typeof(GroupableComponent))]
2. public sealed class OnlyStorageInStorageGroupRule : AddRule{
3.   public override void ElementAdded(ElementAddedEventArgs e){
4.     GroupableComponent component = e.ModelElement as
       GroupableComponent;
5.     if (component.Group != null && !(component is Storage) &&
           component.Group is StorageGroup) {
6.         Helpers.ShowErrorMessage
           ("Only a storage can be nested in a storage group.");
7.         component.Delete();}}} //reverse action
```

Fig. 7: Example of Validation Code for a Hard Rule

Fig. 8 is the documentation of the rule in Listing7. In which, we specify the rule name, description, context, trigger event, invariants, and pre- and post- conditions. StratusML rules have been documented by utilizing the same template in Fig. 8. Notice that the rule *pre-conditions*, *invariants* and *post-conditions* are declarative (i.e., do not change the rule context state). StratusML does not support validation rules with imperative actions with the exception of hard rules that validate the creation of components. These rules support delete actions to reverse the creation action effect in order to maintain the context initial state.

```
Rule:             OnlyStorageInStorageGroupRule
Decryption:       checks that a StratusDiagram class contains at least one Task class.
Context:          <class> GroupableComponent
Trigger Event:    <custom> adding GroupableComponent.
Pre-conditions:   GroupableComponent exists, GroupableComponent.Group exists
Invariants:       ¬((GroupableComponent.Group is StorageGroup)
                  ∨ (GroupableComponent.Group is StorageGroup)
                  ∧ (Component is Storage)) ⇒ valid.
Post-conditions:  On invalid
                  display <error> "Only a storage can be nested in a storage group".
                  delete GroupableComponent.
```

Fig. 8: Example of Hard Validation Rule

Example 2 - A Soft Validation Rule: Fig. 9 is an example of a soft validation rule that is applied to the adaptation *action* concept. The rule is triggered by the model *save* event. The rule validates that an action is associated to a *Task* or *ScalabilityGroup* as a target. If this invariant is invalid, an error message will be generated. Moreover, for better user experience a *set focus* action will be used to direct the user to the *component* (i.e., the action) that generated the error.

```
Rule:             ActionMustHaveAtLeastOneTargetRule
Decryption:       checks if an Action class references at least one
                  ScalabilityGroup or Task
Context:          <class> Action
Trigger Event:    < Save>
Pre-conditions:   Action exists
Invariants:       ReactiveRuleScaleAction.TaskTargets.Count > 0 ||
                  ReactiveRuleScaleAction.ScalabilityGroupTargets.Count > 0
                   ⇒ valid
Post-conditions:  On invalid
                  display <error> "Actions must have at least one target (task or scalability group)".
                  Set Focus <Action> A
```

Fig. 9: Example of Soft Validation Rule

D. Artifacts Generation

The purpose for designing a visual DSML is to generate artifacts (e.g., code, configuration) from the models written in that language to reduce the efforts of manually creating them. By generating the cloud configuration space artifacts, StratusML reduces the administration and configuration efforts. StratusML facilitates application migration between different cloud providers, by modeling the service structure and configuration independently from the platform specifications, then generating the configuration artifacts (i.e., text XML files) required to run the application on the target platfrom. There are multiple approaches with DSL tools to transform models into code and text files [12]. StratusML uses transformation templates. Particularly, StratusML utilizes the Text Templating Transformation Toolkit (T4), which is a text transformation technology developed by Microsoft for artifacts generation.

Models created using visual editors, such as StratusML are already loaded in memory. This eliminates the need to parse and serialize the model. A template-based transformation uses text files, called transformation templates. A transformation template contains code to import the model loaded in memory, navigates through the model, and generates textual artifacts based on the transformation rules (i.e., code) specified in the template. A transformation template consists of three parts: a static part that represents the structure of the output file, a dynamic part that corresponds to the code logic used for model navigation, patterns identification and transformation rules, and template directives which specify how the template should be processed. T4 organizes these parts into blocks. These blocks are distinguished by their opening control markers. The folwoing are the main blocks provided in T4:

(a) Standard control blocks '$< \# \; statements \; \# >$': Contain statements written in C# or Visual Basic to control the flow of processing in the text template.
(b) Expression control blocks '$< \# = \; expressions \; \# >$': Contain expressions that are evaulated and automatically converted to strings to be inserted into the output file.
(c) Class feature control blocks '$< \; \# + \; methods \; \# \; >$': Contain methods, fields and properties. It is used to add reusable pieces of template, such as transformation helper functions.
(d) Directive blocks '$< \; @ \; directives \; \# \; >$': It provides instructions to the T4 template engine such as, specifying that the output file should have a '.cscfg' extension, or it should be splitted into multiple files.

Anything written outside the boundaries of these blocks are considered part of the static content, which will be emitted to the output file without processing.

Selecting a platform provider calls a hard imperative rule that transforms the model into provider specific, but platform independent model. If the model passes validation with no errors, then the save action will trigger the template based transformation to generate the actual platform specific artifacts. StrartusML provides a set of transformation templates out of the box and allows creating custom ones for a new providers.

Listing 4 is an example for part of a template that is used to generate Windows Azure service definition file from the StratusML model. The complete Azure defenition template as well as the other templates required to fully generate all the configurations and artifacts to deploy any StratusML model into Windows Azure is available in the StratusML web page [1]. The code in Listing 4 starts with a set of directives that specifies the output file name extension '*csdef*', defines the name of a class that makes the link between our model and the T4 engine '*StratusMLDirectiveProcessor*' and loads the source instance model '*ApplicationModelCoupoNet.stratus*' of the application as modeled using the StratusML language.

The code from line 7 to 20 of the standard control block represents the primitive transformation function of the template. It utilizes three supportive transformation helper functions that are implemented within class feature control blocks to be reusable. Those are: *GenerateBasicRoleNodes*, *GenerateWebRoleNodes*, and *GenerateWorkerRoleNodes*. The *GenerateBasicRoleNodes* helper method generates the essential elements that are presented in all three types of tasks.

```
1  <#@ template inherits="Microsoft.VisualStudio.TextTemplating.VSHost.
     ModelingTextTransformation"#>
2  <#@output extension=".csdef" #>
3  <#@StratusML processor="StratusMLDirectiveProcessor" requires="fileName='
     ApplicationModelCoupoNet.stratus'" #>
4  <#@ import namespace="System.Collections.Generic" #>
5  <# Write("<ServiceDefinition name=\"{0}\" ", StratusDiagram.Name);
6    ... //omitted for paper presentation
7    List<Component> tasks = GetAllTasks();
8    foreach (Task task in tasks) {
9      if (task is WebTask) {
10       WriteLine("<WebRole name=\"{0}\" enableNativeCodeExecution=\"{1}\"
           vmsize=\"{2}\">", task.Name, (task as WebTask).
           EnableNativeCodeExecution, task.VirtualMachineSize);
11       GenerateWebRoleNodes(task as WebTask);
12       WriteLine("</WebRole>");
13     } else if (task is WorkerTask) {
14       WriteLine("<WorkerRole name=\"{0}\" enableNativeCodeExecution
           =\"{1}\" vmsize=\"{2}\" >", task.Name, (task as WorkerTask).
           EnableNativeCodeExecution, task.VirtualMachineSize);
15       GenerateWorkerRoleNodes(task as WorkerTask);
16       WriteLine("</WorkerRole>");
17     } else if (task is VirtualMachineTask) {
18     ... //omitted for paper presentation
19     } else
20     { throw new Exception("Invalid task type for task: " + task);}
21   }
22   GenerateNetworkTrafficRulesNode(tasks);
23   PopIndent();
24   Write("</ServiceDefinition>");
25 #>
26 <#+ void GenerateWorkerRoleNodes(WorkerTask role) {
27     GenerateBasicRoleNodes(role);
28     PushIndent(" ");
29     if (role.Runtime != null) {
30       WriteLine("<Runtime executionContext=\"{0}\">", role.Runtime.
           ExecutionContext);
31       PushIndent(" ");
32       GenerateEnvironmentNode(role.Runtime.Environment);
33       if (role.Runtime.NetFxEntryPoint != null && role.Runtime.NetFxEntryPoint.
           Count == 1) {
34         WriteLine("<EntryPoint>");
35         PushIndent(" ");
36         foreach (NetFxEntryPoint netFxEntryPoint in role.Runtime.NetFxEntryPoint)
37           {
38           WriteLine("<NetFxEntryPoint assemblyName=\"{0}\"
               targetFrameworkVersion=\"{1}\" />", netFxEntryPoint.AssemblyName,
               netFxEntryPoint.TargetFrameworkVersion);
39           break; // Only output one
40             }
41         PopIndent();
42         WriteLine("</EntryPoint>");
43           } else if (role.Runtime.ProgramEntryPoint != null && role.Runtime.
               ProgramEntryPoint.Count == 1) {
44       ... // //omitted for paper presentation
45             }
46         PopIndent();
47         WriteLine("</EntryPoint>");
48           }
49       PopIndent();
50       WriteLine("</Runtime>");
51         }
52     if (role.Startup != null && role.Startup.Count > 0) {
53       ... //omitted for paper presentation
54       }
55       PopIndent();
56       WriteLine("</Startup>"); }
57     if (role.Contents != null && role.Contents.Count > 0) {
58     ... // omitted for paper presentation
59   }
60 #>
61 <#+ List<Component> GetAllTasks() {
62       List<Component> tasks = StratusDiagram.Components.FindAll(x => x is Task);
63     foreach (ScalabilityGroup scalabilityGroup in StratusDiagram.Components.FindAll(
         x => x is ScalabilityGroup)) {
64       tasks.AddRange(GetAllTasks(scalabilityGroup)); }
65     return tasks;
66     }
   #>
```

Listing 4: T4 Transformation (StratusML To Azure Definition).

The *GenerateWebRoleNodes* and *GenerateWorkerRoleNodes* helper methods generate nodes that are specific to web tasks and worker tasks, respectively. Both of these methods call *GenerateBasicRoleNodes*, before they start running. The template then checks that all the values were added correctly in the template. We used this template along with the templates provided in the StratusML webpage to generate the complete configurations required to deploy the CoupoNet example on Windows Azure platfrom.

V. Related Work

Using model driven engineering for managing and configuring software systems at runtime to satisfy desired quality attributes is not new [13]. Example of approaches that address this problem are surveyed in [14]–[16] . The shortcomings of these approaches are: (i) most of them are limited to performance, (ii) they focus on domains other than cloud computing, and (iii) they normally capture the dynamics of the software components only without considering the dynamics of the underlying resource model. StratusML addresses these issues. While recently, there has been several proposals to exploit model-driven engineering to address the cloud modeling concerns, most of the current frameworks are not comprehensive enough; they either focus on portability [4], [17], [18], or security [19]. The most comprehensive cloud modeling frameworks, in terms of the number of cloud concerns they address, that we are aware of are MODAClouds [20], CloudML [21] and the cloud DSML by Caglar et. al. [22]. Those are the most relevant to StratusML. These modeling frameworks can be differentiated based on the features they provide. What distinguishes StratusML from both MODACloud and CloudML is (i) its ability to provide partial and holistic views of the different cloud application concerns through utilizing the concept of layers, (ii) its ability to visually model adaptation rules and actions, (iii) its comprehensive validation constraints that covers a wide range of the cloud application requirements, and most importantly, (iv) its template-based transformation that can automatically generate ready to deploy platform specific cloud application artifacts.

VI. Conclusions

This paper presented StratusML, a modeling framwork and a layered modeling language for cloud applications. StratusML differentiates itself, by using layers to provide partial and holistic views for the different cloud application concerns, facilitating visual modeling of adaptation rules, and using template-based transformation to deal with platforms heterogeneity. StratusML captures the components needed to model platform independent quality cloud applications and simplifies the applications' management through "model once deploy everywhere" model driven approach by minimizing assumptions about the target cloud platform. It supports the cloud applications development process, by covering the main architectural views that represents the application structure, behaviour and configuration. StratusML promotes flexibility, portability, reusability and productivity.

Immediate future directions consist of conducting an empirical study to evaluate the usefulness of the proposed framework to address the different cloud stakeholders, and to show quantitatively the effort saved using the tool.

References

[1] M. Hamdaqa. (2014) The stratus modeling language. Online. University of Waterloo. [Online]. Available: http://www.stargroup.uwaterloo.ca/ mhamdaqa/stratusml/

[2] N. Medvidovic, E. Dashofy, and R. Taylor, "Moving architectural description from under the technology lamppost," *Information and Software Technology*, vol. 49, no. 1, pp. 12–31, 2007.

[3] I. Malavolta, H. Muccini, P. Pelliccione, and D. Tamburri, "Providing architectural languages and tools interoperability through model transformation technologies," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 119–140, 2010.

[4] D. Petcu, B. Di Martino, S. Venticinque *et al.*, "Experiences in building a mosaic of clouds," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 2, no. 1, p. 12, 2013.

[5] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: Why reuse is still so hard," *IEEE Software*, vol. 26, no. 4, pp. 66–69, 2009.

[6] M. Hamdaqa and L. Tahvildari, "Cloud computing uncovered: A research landscape," *Advances in Computers*, vol. 86, pp. 41–85, 2012.

[7] M. Hamdaqa and L. Tahvildari, "Prison-break: A generic schema matching solution to the cloud vendor lock-in problem," in *the International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based System*, 2014, pp. 37–46.

[8] M. Hamdaqa and L. Tahvildari, "The (5+1) architectural view model for cloud applications," in *the IBM Centers for Advanced Studies Conference*, 2014, pp. 46–60.

[9] M. Hamdaqa, T. Livogiannis, and L. Tahvildari, "A reference model for developing cloud applications." in *the International Conference on Cloud Computing and Services Science*, 2011, pp. 98–103.

[10] D. S. Kolovos, L. M. Rose, and J. R. Williams, "Using model-to-text transformation for dynamic web-based model navigation," in the International Workshop on Models@run.time, 2011, pp. 1-12.

[11] S. Cook, G. Jones, S. Kent, and A. C. Wills, *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2008.

[12] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[13] G. Blair, N. Bencomo, and R. France, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[14] V. Cortellessa, A. Di Marco, and P. Inverardi, *Model-based software performance analysis*. Springer Berlin Heidelberg, 2011.

[15] M. Isa, M. Zaki, and D. Jawawi, "A Survey of Design Model for Quality Analysis: From a Performance and Reliability Perspective," *Computer and Information Science*, vol. 6, no. 2, pp. 55–70, 2013.

[16] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance Evaluation*, vol. 67, no. 8, pp. 634–658, 2010.

[17] J. Guillén, J. Miranda, J. M. Murillo, and C. Canal, "A service-oriented framework for developing cross cloud migratable software," *Journal of Systems and Software*, vol. 86, no. 9, pp. 2294–2308, 2013.

[18] A. H. Ranabahu, E. M. Maximilien, A. P. Sheth, and K. Thirunarayan, "A domain specific language for enterprise grade cloud-mobile hybrid applications," in *the workshops on Domain-Specific Modeling*. ACM, 2011, pp. 77–84.

[19] M. Almorsy, J. Grundy, and A. Ibrahim, "Adaptable, model-driven security engineering for saas cloud-based applications," *Automated Software Engineering*, vol. 21, no. 2, pp. 1–38, 2013.

[20] D. Ardagna, E. Di Nitto, P. Mohagheghi *et al.*, "Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds," in *the ICSE Workshop on Modeling in Software Engineering*, 2012, pp. 50–56.

[21] G. Gonçalves, P. Endo, M. Santos, D. Sadok *et al.*, "CloudML: An integrated language for resource, service and request description for d-clouds," in *the International Conference on Cloud Computing Technology and Science*, 2011, pp. 399–406.

[22] F. Caglar, K. An, S. Shekhar, and A. Gokhale, "Model-driven performance estimation, deployment, and resource management for cloud-hosted services," in *the Workshop on Domain-specific Modeling*, 2013, pp. 21–26.