# On Formalizing and Identifying Patterns in Cloud Workload Specifications

Christos Tsigkanos and Timo Kehrer
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano, Italy

***Abstract*—Managing, configuring and deploying complex applications in the cloud are emerging problems in contemporary cloud computing. Cloud workload specifications, as portable abstractions of cloud computations, are an important means to deal with these problems on an architectural level. They focus on defining the components of an application and their structural relations, workflows regarding their initialization and management, along with configuration and artifacts required for application operation. The prevalence of cloud computing drives demand for applications that rely on systematic engineering and smell-free architectures, thus quality assurance techniques for cloud workload specifications are strongly required. In particular, cloud workload specifications exhibit characteristics of software architectures such as patterns or anti-patterns which state desired or undesired quality aspects. To facilitate formal reasoning about latent qualities of a workload design, we propose a static bigraphical semantics for the modeling language defined by the emerging Topology and Orchestration Specification for Cloud Applications (TOSCA) standard. Thereupon, we illustrate how to check for the presence (absence) of (anti-)patterns expressed as logical formulae over bigraphical predicates.**

## I. Introduction

Managing, configuring and deploying complex applications in the cloud are emerging problems in contemporary cloud computing. A cloud workload consists of the configuration of an application, data or code artifacts supporting it as well as the configuration of underlying computing resources and network connectivity. Cloud workloads are defined in architectural specifications which include components and their structural interrelations, workflows regarding their initialization and management, along with configuration and artifacts required for application operation. Approaches include orchestration specifications CAMP [1], [2], Open-CSA, SOA-ML and USDL, and on the industrial side solutions such as Amazon CloudFormation, OpenStack Heat, Cloudify and Alien4Cloud. To consolidate and enable interoperability within this variety of approaches, a technical committee by OASIS [3] defined a standard for the Topology and Orchestration Specification of Cloud Applications (TOSCA) [4], [5], which defines guidelines and facilities for the complete specification, orchestration and configuration of complex workloads, addressing portability in heterogeneous clouds.

The prevalence of cloud computing leads to workloads which are becoming bigger and harder to manage, and thus, besides the basic requirement for interoperability as addressed by TOSCA, drives demand for applications that rely on systematic engineering of cloud workload specifications. Since cloud workload specifications exhibit characteristics of software architectures, methods which have been shown to be successful in the context of traditional software architectures [6] should be adopted in the cloud computing domain. Amongst others, styles [7] and patterns [8], [9], [10] describing possible solutions for recurring problems are important instruments for software architects. Likewise, anti-patterns have been shown to be a successful method for engineering smell-free architectures [11], [12], [13]. Consequently, we reasonably assume that the use of patterns and anti-patterns has the potential to substantially improve upon the systematic engineering of cloud workload specifications. Our goal is to provide cloud engineers with a suitable technique to specify and identify patterns in cloud workload specifications, thus improving their architecting capabilities. In essence, specifications of patterns or anti-patterns state desired or undesired quality aspects in workload specifications, and pattern identification is a means to reason about these quality aspects.

Although there is extensive literature on software architecture patterns and anti-patterns, existing approaches cannot be adopted immediately for reasoning on a broad range of latent design qualities in cloud workload specifications: they target specific modeling languages such as the UML [8] or traditional Architecture Description Languages (ADLs) [14], dedicated domains such as distributed object middleware [9], [10] Java EE architectures [13], or very specific quality aspects such as performance [12], [13]. However, topological information inherent in workloads—a static semantics of workload specifications—have not been utilized for checking the presence (absence) of patterns (anti-patterns), which is already needed for rather simple checks such as ensuring that "all database nodes in the workload are hosted on different computing instances than in-memory cache nodes". Thus, in order to enable various forms of analysis as typically performed in software engineering and verification, such semantics need to be provided. This facilitates formal reasoning on a wide range of qualitative properties, which in this context can be considered as structural (anti-)patterns. If there is no confusion, we will hereafter refer to both patterns (i.e. desirable features) and anti-patterns (i.e. undesirable ones) with the term *patterns*.

Our fundamental intuition is that a cloud workload specification is a set of components and connectors, upon which essential notions of locality and connectivity can be expressed through a meta-calculus. Our approach grounds on previous work [15], [16] in which we advocated that the topology of cyber-physical spaces, i.e. their structure in terms of key elements and their relationships, can provide a system with both structural and semantic awareness of contextual characteristics, especially with respect to security. The setting presented here is rather different, but some abstract concepts and the underlying modeling formalism suggest that the formalization and analysis techniques can be transferred to this entirely different domain. In this paper, we lift these metaphors to support verification of cloud workload specifications in order to facilitate reasoning on a variety of properties.

In particular, starting from a TOSCA description of a cloud workload, we show how its static semantics can be provided through a graph structure in terms of a *bigraph* [17], which consists of two graphs; a place forest that captures notions of locality, and a hyper-graph that models linking among entities. Patterns can then be represented using bigraphical matching properties [18], [19], [20] expressing configurations of interest. Bigraphical predicates and bigraph matching are concise and flexible techniques for the specification and identification of patterns, and thus provide a fundamental basis to achieve the main goal addressed in this paper. Moreover, although not in the paper's main scope, we outline further analysis potentials regarding dynamics of complex cloud workloads as manifestations of continuously evolving software architectures.

The rest of the paper is structured as follows: Section II describes a motivating example. Section III briefly discusses cloud workloads and introduces the TOSCA cloud topology model. Section IV refers to a topological static semantics of TOSCA in terms of bigraphs. Section V illustrates how to specify and finally verify properties reflecting (anti-)patterns, and Section VI discusses reconfigurations of cloud workload specifications. Section VII concludes the paper along with an outlook on future work.

## II. Motivating Example

As a simple example, we consider the setting of a data-intensive workload which is part of a larger cloud application deployment. Figure 1 shows architectural deployment and data flow diagrams in a combined way. The workload consists of several architectural components; a master data store, a series of slave databases and client components. Components are arranged in backend, frontend and search parts. The backend solely provides data storage, while the frontends both store data and are responsible for core business logic. Since the master database may execute blocking read queries which may have negative effects on overall system performance, dedicated search and temporary data retention functionalities are provided in the architecture.
The example workload of Figure 1 is intended to exhibit data consistency among slave nodes, something which we consider a *data consistency pattern*. Application logic implemented by clients must only read data from slave databases and must only execute write operations upon the master datastore, which in turn is responsible for updating slave database nodes with fresh data; data is consistent among slave nodes.

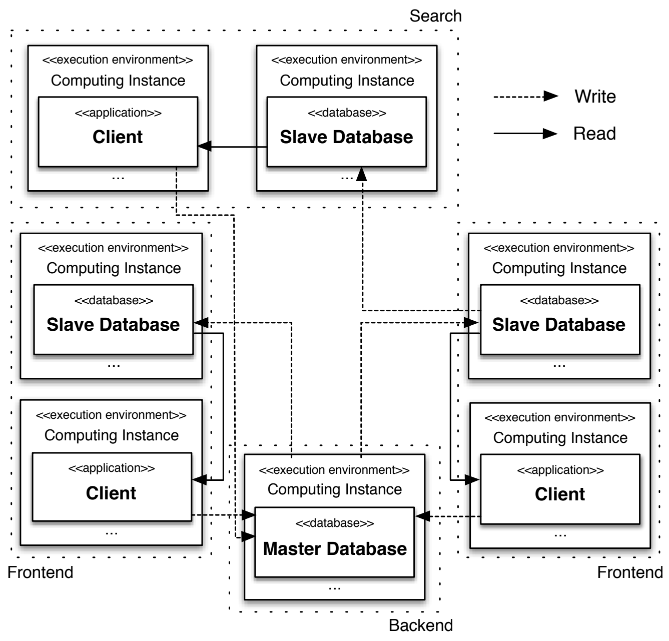

Fig. 1: Combined deployment and data flow diagrams.

## III. Cloud Workloads and TOSCA

The notion of a workload is central to cloud computing; it is basically an abstraction of the actual computation (or work) that a set of cloud instances are going to perform when deployed. Portability is a key concept involved in a workload, as it is essentially an abstract description of the capability or amount of work that could conceivably run in different environments and underlying resources. As such, cloud workloads encompass refinements of traditional notions of software engineering such as *architecture* and *requirements*. Workload requirements may indicate resources needed, or constraints referring to compliance with regulations or budget. Moreover, components involved in a cloud computation may have requirements or dependancies that need to be satisfied. In turn, they may provide functionality to other components. Relationships among components can indicate how they must be connected, deployed or interfaced. Naturally, computing components of a workload form a structure which can be considered as the *topology* of a workload.

In Section III-A, we will briefly introduce the TOSCA topology modeling approach which will be later applied to our running example in Section III-B. For a detailed introduction to TOSCA, the interested reader is kindly referred to [4], [5].

### A. Topology Modeling in TOSCA

In the TOSCA topology modeling approach, a *ServiceTemplate* describes the components of a cloud workload as well as their interrelations. Fundamental topological concepts and building blocks are illustrated in Figure 2.

*Application Topology* is modeled by a *TopologyTemplate*, essentially a typed graph whose nodes represent components while edges represent relationships between them.

*Components* of an application are modeled by *NodeTemplates*. A *NodeType* defines the type of a component, its observable properties, requirements needed for its operation, and capabilities it may offer to other components. In fact, properties are described by *PropertyDefinitions*, while operations are specified using *Interface* and *Operation* elements. Moreover, requirements are described by *RequirementDefinitions*, and capabilities are exposed by *CapabilityDefinitions*. A *NodeTemplate* may add additional usage constraints or assign concrete values to properties defined by its type.

*Relationships* between components are modeled by RelationshipTemplates; each refers to a RelationshipType which defines properties as well as operations which can be performed on its interfacing components. Analogously to *NodeTemplates* and *NodeTypes*, *RelationshipTemplates* and *RelationshipTypes* are defined separately for reuse purposes, while *RelationshipTemplates* may refine or constrain general definitions of their associated type.

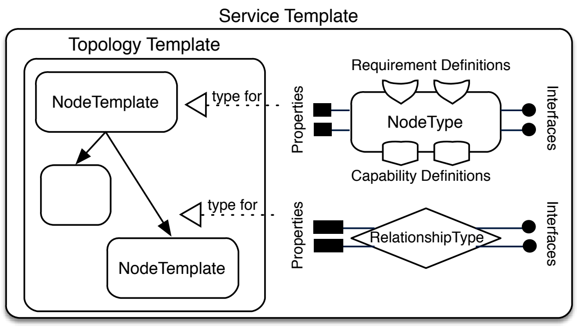

Fig. 2: Topology in a TOSCA Service Template.

In addition to the topology of cloud workloads, TOSCA also addresses workflows regarding the initialization and management of components as well as their configuration and artifacts required for application operation. Therefore, a component may have associated *Plans*, referring to a workflow description of application deployment or management aspects. Moreover, *Artifacts* realize specific deployment particulars of components or relationships. They can be of various types in order to describe operations needed. In this paper, we focus on structural topological aspects of a workload affecting architectural qualities and thus some particulars of topology nodes such as *Plans* or *Artifacts* will be not considered. Moreover, generally nodes in TOSCA templates are either concrete, in the sense that all deployment particulars and artifacts of a node are provided in the specification, or abstract, where the template describes the node type along with its required capabilities and properties that must be satisfied. We abstract from such functionalities (such as node substitutions), as they do not concern functional features of the core of our approach. An engineer can specify a configuration making use of templating facilities available within TOSCA, typically with XML/YAML formats; in this paper, we will use the more compact graphical representation instead of the actual textual form defined by the standard.

### B. A TOSCA Topology Model of Our Example

A TOSCA workload specification reflecting the example introduced in Section II is shown in Figure 3. Each application component of Figure 1 appears in the topology as a node, defined by a *NodeTemplate* and its respective *NodeType*. Consider for example the node identified by symbolic identifier $s1$ in Figure 3; it is of type $SlaveDB$ which defines an interface called $exec$ and a property named $mem\_size$. The *NodeTemplate* assigns concrete values to these general definitions, namely the shell script $consistent.sh$ which refers to the interface definition $exec$ and the memory of $8gb$ for property $mem\_size$, respectively.

Nodes in the topology model get related to each other when one node has a requirement against some capability provided by another node. For example, database software needs to be hosted on a computing resource; thus, node $m1$ of type $MasterDB$ has a requirement called $Container$ which needs to be fulfilled by pointing to another node offering this capability. In our example, this capability is provided by the $Compute$ node $c1$, connected by the relationship $rel2$ of type $Contains$. In a similar way, nodes of type $SlaveDB$ are connected to $Compute$ nodes serving as hosts for the database software. Nodes representing master and slave databases have different types, namely *MasterDB* and *SlaveDB*, since they exhibit, among others, different requirements and capabilities.

## IV. Static semantics of TOSCA Specifications

A modeling formalism expressing the static semantics of TOSCA specifications through their topological relationships should allow expression of locality and linking among entities in a specification; these notions will essentially comprise the semantic domain of our approach. In this section, a static semantics of TOSCA specifications is given in terms of *bigraphs* [17]. We draw inspiration from [21], which utilizes a form of bigraphs to define formal semantics of reconfiguration operations in the context of a traditional ADL.

### A. Bigraphs as the Semantic Domain

Bigraphs are an emerging formalism for structures in ubiquitous computing, consisting of two graphs. A *place graph* is a forest, a set of rooted trees defined over a set of nodes. A *link graph* is a hypergraph over the same set of nodes and a set of edges, each linking any number of nodes to names; this graph represents generic many-to-many relationships among nodes. Connections of an edge with nodes are called ports. Place and link graphs are orthogonal, and edges between nodes can cross locality boundaries. What follows is an informal presentation; the interested reader is referred to [17] for complete definitions and proofs of the bigraphical theory.

$$\begin{aligned}
&P.Q && \textit{Nesting } (P \textit{ contains } Q) && \text{(1a)}\\
&P \mid Q && \textit{Juxtaposition of nodes} && \text{(1b)}\\
&-_i && \textit{Site numbered } i && \text{(1c)}\\
&K_w.(U) && \textit{Node with control } K \textit{ having ports} && \text{(1d)}\\
& && \textit{with names in } w.\ K \textit{ contains } U && \\
&W \parallel R && \textit{Juxtaposition of bigraphs} && \text{(1e)}
\end{aligned}$$

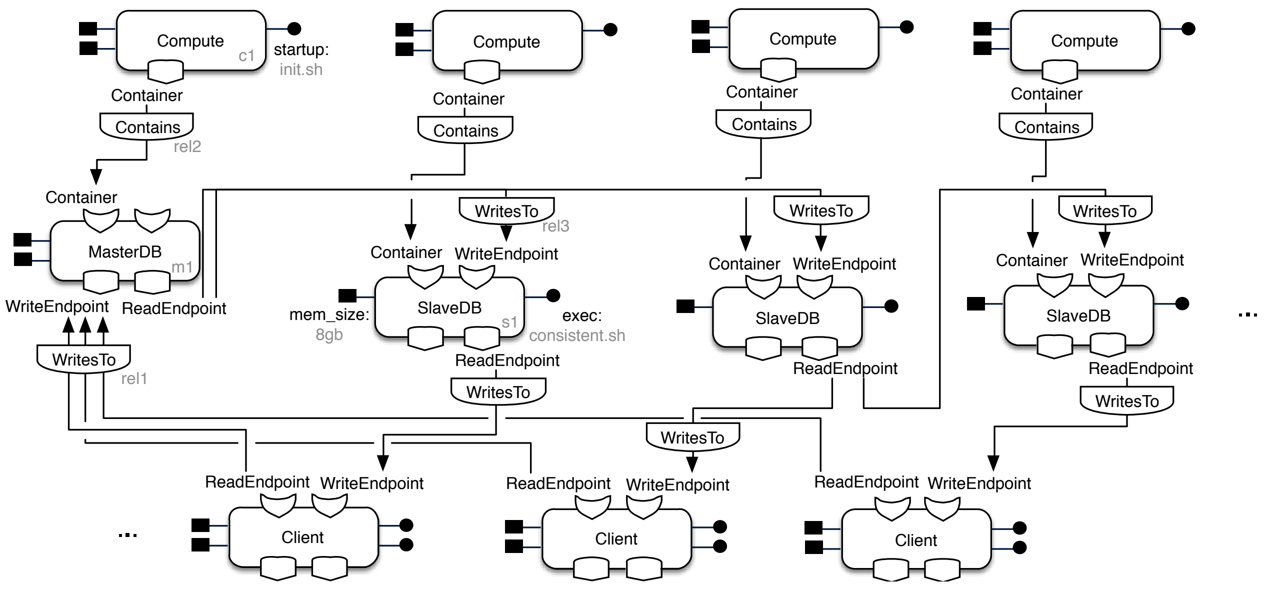

Fig. 3: Fragment of a workload TopologyTemplate of the motivating example.

Bigraphs can be described through algebraic expressions (Formulae 1a-1e), in a process calculi fashion. The containment relationship is expressed as in Formula 1a. Bigraphs can contain sites (Formula 1c) that can be used to denote placeholders; sites can be used to indicate presence of unspecified nodes. Controls are names that define a node's type; each node control can be associated with a number of named ports. $P$, $Q$, and $U$ are controls of bigraph nodes. If a single instance node of that type exists in the bigraph, the control also uniquely identifies that node. Otherwise, port names are used as a way to uniquely identify it. In Formula 1d the node identified by control $K$ and port name $w$ also contains $U$. Ports that appear in a formula with the same name are connected, forming a hyperedge with that name, called *link* in the sequel. Bigraphs can be contained in roots that delimit different hierarchical structures; in Formula 1e, $W$ and $R$ are different roots.

## B. Bigraphical Encoding of TOSCA Specifications

Our objective is expressing topological information inherent in a TOSCA specification through containment (expressing locality) and linking relations, mapping it to a bigraph placing and linking structure. The idea is to map types of *NodeTemplates* found in a *TopologyTemplate* to bigraph controls; *NodeTemplates* themselves are mapped to nodes typed over the respective control. Identifiers of *NodeTemplates* will correspond to port names uniquely identifying nodes in the bigraphical structure; for example, a port named $m1$ will identify a node of type $MasterDB$ in the bigraphical representation of the TOSCA specification of Figure 3. Analogously, RelationshipTypes are mapped to controls, and *RelationshipTemplates* are mapped to bigraphical nodes typed over the respective control and having a port whose name corresponds to the relationship's identifier. Subsequently, we formulate the placing structure as follows.

A TOSCA *TopologyTemplate* is conceived as a bigraph root, containing juxtaposed nodes representing *NodeTemplates* and *RelationshipTemplates* obtained as described above. These in turn, include juxtaposed nodes representing definitions. Definition nodes are typed over a fixed set of controls characterizing the definition type, i.e. we distinguish capabilities, requirements, interfaces and properties. Definition nodes may contain further nodes representing attributes of a definition.

To populate the linking structure, we consider how (i) names in attributes take values, (e.g. property attribute $exec$ may point to name $consistent.sh$), and (ii) how nodes in a topology template are connected through relationships. For example, a bigraph formula partially representing topology template nodes $c1$ and $m1$ of Figure 3 will be as follows:

$$Compute_{c1}.(Cap.(-_0)|Iface.(startup_{init.sh})|-_1)|MasterDB_{m1}.(-_2)$$

After parsing a TOSCA specification, a $TopologyTemplate$ holds $NodeTemplates$ and $RelationshipTemplates$ ($RelTemplates$) of types $NodeType$ and $RelationshipType$ ($RelType$), respectively. These in turn may contain definitions ($Def : \{Cap \mid Req \mid Iface \mid Prop\}$). Definitions have types ($DefType$), and they contain attributes ($Attr$) which point to concrete values ($val$). Function $\mathtt{T}$ translates a workload specification to an equivalent bigraphical representation.

$$\mathtt{T}(TopologyTemplate) = \mathtt{T}(NodeTemplates) \parallel \mathtt{T}(RelTemplates)$$
$$\mathtt{T}(NodeTemplates) = \underset{NodeTemplate+}{|} \mathtt{T}(NodeTemplate)$$
$$\mathtt{T}(RelTemplates) = \underset{RelTemplate+}{|} \mathtt{T}(RelTemplate)$$
$$\mathtt{T}(NodeTemplate) = NodeType_{name}.\Big(\underset{Def+}{|} \mathtt{T}(Def)\Big)$$
$$\mathtt{T}(RelTemplate) = RelType_{name}.\Big(\mathtt{T}(Iface) \mid \mathtt{T}(Prop)\Big)$$
$$\mathtt{T}(Def) = DefType.\Big(\underset{Attr+}{|} \mathtt{T}(Attr)\Big)$$
$$\mathtt{T}(Attr) = Attr_{val}$$

By consecutive applications of function T, Formula 2 partially encodes the following fragment of our TopologyTemplate of Figure 3; the capability $Container$ of $Compute$ node $c1$ satisfies the respective requirement of the $MasterDB$ node $m1$ through a $Contains$ relationship $rel2$.

$$\begin{aligned}&Compute_{c1}.(Cap.(Container_{rel2}) \mid -_0) \mid MasterDB_{m1}.( \\ &Cap.(-_1) \mid Req.(Container_{rel2}) \mid -_2) \parallel Contains_{rel2}\end{aligned} \tag{2}$$

We finally note that a TopologyTemplate may also specify inputs it expects and outputs that it will provide when being instantiated [3]. Inputs and outputs will be preserved in the bigraphical representation, placed under the root node representing a topology template. However, inputs and outputs are insignificant for our analysis and are thus not considered here.

## V. Patterns as Bigraphical Predicates

In the context of this paper, we refer to architectural patterns within a broader scope; a pattern (or anti-pattern) aims to address issues such as performance, high availability and minimization of business risk. Its realization in a workload reflects specific structure and connectivity among architectural components that address such issues. As such, the specification of a pattern is left to the engineer, to maintain generality of the approach. Thus, domain specifics and expert knowledge can be taken into account in pattern specification; an engineer tackling security concerns of a workload may be concerned with different patterns than a performance analyst. The specification of a pattern that a configuration may exhibit, can be done either starting from the same source language of TOSCA or directly using bigraphical predicates; its satisfaction or violation on a target configuration will be checked by evaluating the set of predicates that describe the pattern, as we will see in the remainder of this section.

A configuration described by a bigraph satisfies a property if the bigraph specifying the property can be matched against it, meaning that it exhibits containment and connectivity relations among the same entities. Failure of matching the bigraph representing the property means instead that the property is not satisfied. The utilization of sites in the bigraph specifying the property indicates that the portion of the configuration that matches a site does not affect satisfaction. Given that variables $a, b, c$ range over different names of relationships and $x, y, z$ over names of nodes and that relationships associate always pairs of nodes, utilizing boolean connectives and elementary predicates expressed in terms of bigraphs the property representing the data consistency requirement of the example presented has the form of Formula 3.

$$\begin{aligned}&SlaveDB_x.(Interface.(exec_{consistent.sh}) \mid -_0) \Rightarrow \\ &MasterDB_y.(Cap.(ReadEndpoint_a \mid WriteEndpoint_b) \mid -_1) \\ &\mid Client_z.(Req.(ReadEndpoint_b \mid WriteEndpoint_c) \mid -_2) \\ &\mid SlaveDB_x.(Req.(WriteEndpoint_a) \mid Cap.(ReadEndpoint_c) \mid -_3) \\ &\parallel WritesTo_a \mid WritesTo_b \mid WritesTo_c\end{aligned} \tag{3}$$

Formula 3 states that should a node of type $SlaveDB$ with an Interface definition of $exec : consistent.sh$ exist in the model under consideration, it should always have a Capability definition ($Cap$) $ReadEndpoint$ pointing to name $c$, which is connected through a $WritesTo$ relationship to the $WriteEndpoint$ ($Req$) of a $Client$ node. Moreover, the $Client$ must satisfy a requirement definition $ReadEndpoint$ pointing to the $WriteEndpoint$ ($Cap$) capability of a $MasterDB$ node through another $WritesTo$ relation ($b$). The $MasterDB$ node must be in turn connected through a ($Cap$) $ReadEndpoint$ to the $SlaveDB$ node's $WriteEndpoint$ through another relation ($a$). Through binding of values to the variable names, the property varies over all $SlaveDB$ nodes, thus ensuring the data consistency requirement. Due to the presence of sites in the property specification, other definitions of each NodeTemplate do not affect satisfaction. Truth values of bigraphical predicates are evaluated over the configuration; their satisfaction over the property can be checked automatically through bigraph matching [18], [20]. For the example of Figure 3, checking will succeed.

Note that instead of providing parametric bigraphical formulae, workload configurations that reflect patterns can also be specified by providing TOSCA specifications, utilizing the automatic translation facilities presented. However, to support pattern specification a placeholder concept needs to be established to enable parametric statements; essentially a way to allow the engineer to abstract from unimportant details and specify pattern elements of interest. To achieve this, the source language could be extended with an extra entity, introducing bigraphical sites at the specification language level.

## VI. Towards Reconfigurations

The next conceptual step would be to consider how workload specifications may be reconfigured. This reconfiguration (or change) may be manifested both at run-time and at design-time. Reasoning about change refers to the dynamics of a system; in the context of the bigraphical theory, dynamics are expressed in terms of Bigraphical Reactive Systems [17], which augment a bigraphical representation with parametric rewriting rules. The theoretical underpinnings of such a formalism enable reasoning on the *evolution* of a bigraphical configuration. A wide range of analyses can then be performed by interpreting the BRS over some form of a Labelled Transition System [22] (LTS), a modeling formalism that concretely describes systems and their evolution in terms of states and transitions; an LTS is essentially a state machine description, enabling reasoning about temporal properties.

Utilizing the BRS mechanism, the cloud engineer can provide elementary reconfigurations reflecting change primitives desired for the analysis of the dynamics that the workload will exhibit when in operation. These can include, for instance, application-specific configuration changes, management operations, or (de-)allocations of cloud instances. Such a state machine description can then be used for both design-time and run-time analysis of a variety of properties. Simulation or model checking methods can be utilized in such regard. Moreover, analysis of reconfigurations of a workload at the design phase can also reason on editing operations performed by an engineer, ensuring that patterns (or anti-patterns) are (not) present in the final configuration, or that reconfigurations applied by the designer are valid [21]. Output from such analyses can also be utilized to provide feedback.

Furthermore, keeping models of workloads alive at runtime can enable reconfigurations with respect to non-functional requirements [23] such as performance or cost, especially relevant for the cloud. Changes in a workload have a direct impact on such requirements; for example, allocating virtual machines to cope with increased load can increase throughput but also incur cost. Overall, providing dynamic semantics through a BRS interpreted as an LTS can serve as a basis for reasoning on a variety of such requirements.

## VII. Conclusions and Outlook

In this paper, we presented ongoing work on formalizing and identifying patterns and anti-patterns occurring in cloud workload specifications. In particular, we showed how static semantics of workload specifications can be provided through a meta-calculus, paving the way for more advanced reasoning. The proposed approach operates directly on TOSCA specifications, providing a translation back and forth a bigraphical representation and thus avoiding the need for an explicit ADL to be used. Moreover, the cloud engineer can specify patterns for checking using familiar concepts from the cloud, such as containment and linking of components. The formal foundations and technique proposed in this paper provide the basis for an industrial-level realization of our approach. We plan to formalize the semantics showcased and realize the approach presented as tool support for TOSCA specifications; this will enable evaluation upon a substantive case study, focusing on the expressiveness of the pattern specification as well as the accuracy of the pattern identification. To attain a desired industrial impact, however, a framework which would include a catalog of patterns, best practices as well as relevant methodology would be additionally needed. The resulting industrial impact would then have to be evaluated as a whole with respect to issues encountered in practitioner's use. Moreover, on a foundational level this provides a setting in which we want to evaluate whether bigraph composition [17] is an appropriate means to compose larger patterns from smaller patterns or architectural primitives [8].

Regarding future work on formal reasoning about workload specifications, the next logical step to realize an extensive quality assurance framework would be to consider how workloads change, namely their dynamics. This refers to both design-time and run-time forms of change. On one hand, understanding editing operations by the cloud engineer and using their semantics could facilitate design. On the other hand, maintaining a workload model at runtime would open novel avenues for run-time adaptation of cloud application deployment and management. Relations with orchestration engines backing TOSCA realizations [24] would have to be further investigated and more refined notions of workload component matching [25] should also be integrated. We hope that the present paper can stir up discussion on such matters.

## Acknowledgment

This work was partially supported by ERC Advanced Grant no. 227977 (SMScom).

## References

[1] OASIS, "CAMP 1.0 (Cloud Application Management for Platforms), V1.0," docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html.

[2] J. Carrasco, J. Cubo, and E. Pimentel, "Towards a flexible deployment of multi-cloud applications based on tosca and camp," in *Advances in Service-Oriented and Cloud Computing*. Springer, 2014, pp. 278–286.

[3] OASIS, "Topology and Orchestration Specification for Cloud Applications (TOSCA)," https://www.oasis-open.org/committees/tosca/.

[4] A. Brogi, J. Soldani, and P. Wang, "Tosca in a nutshell: Promises and perspectives," in *Service-Oriented and Cloud Computing*. Springer.

[5] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "Tosca: Portable automated deployment and management of cloud applications," in *Advanced Web Services*. Springer, 2014, pp. 527–549.

[6] O. Vogel, I. Arnold, A. Chughtai, and T. Kehrer, *Software architecture: a comprehensive framework and guide for practitioners*. Springer Science & Business Media, 2011.

[7] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Prentice Hall Englewood Cliffs, 1996, vol. 1.

[8] U. Zdun and P. Avgeriou, "Modeling architectural patterns using architectural primitives," in *ACM SIGPLAN Notices*, vol. 40. ACM, 2005.

[9] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2013, vol. 2.

[10] M. Völter, M. Kircher, and U. Zdun, *Remoting patterns: foundations of enterprise, internet and realtime distributed object middleware*. John Wiley & Sons, 2013.

[11] W. H. Brown, R. C. Malveau, and T. J. Mowbray, "Antipatterns: refactoring software, architectures, and projects in crisis," 1998.

[12] V. Cortellessa, A. Martens, R. Reussner, and C. Trubiani, "A process to effectively identify guilty performance antipatterns," in *Fundamental Approaches to Software Engineering*. Springer, 2010, pp. 368–382.

[13] T. Parsons, "A framework for detecting performance design and deployment antipatterns in component based enterprise systems," in *Proc. of the 2nd Intl. Doctoral Symposium on Middleware*. ACM, 2005.

[14] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *Software Engineering, IEEE Transactions on*, vol. 26, no. 1, pp. 70–93, 2000.

[15] C. Tsigkanos, L. Pasquale, C. Menghi, C. Ghezzi, and B. Nuseibeh, "Engineering Topology Aware Adaptive Security: Preventing Requirements Violations at Runtime," in *Proc. of the 22nd Int. Requirements Engineering Conf.*, 2014, pp. 203–212.

[16] C. Tsigkanos, L. Pasquale, C. Ghezzi, and B. Nuseibeh, "Ariadne: Topology aware adaptive security for cyber-physical systems," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, 2015, pp. 729–732.

[17] R. Milner, *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[18] L. Birkedal, T. C. Damgaard, A. J. Glenstrup, and R. Milner, "Matching of Bigraphs," *Electronic Notes in Theoretical Computer Science*, vol. 175, no. 4, pp. 3–19, 2007.

[19] A. J. Glenstrup, T. C. Damgaard, L. Birkedal, and E. Højsgaard, "An Implementation of Bigraph Matching," 2008.

[20] M. Sevegnani, C. Unsworth, and M. Calder, "A SAT Based Algorithm for the Matching Problem in Bigraphs with Sharing," University of Glasgow, Tech. Rep., 2010.

[21] A. Sanchez, L. S. Barbosa, and D. Riesco, "Bigraphical modelling of architectural patterns," in *Formal Aspects of Component Software*. Springer, 2012, pp. 313–330.

[22] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT press, 1999.

[23] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model evolution by run-time parameter adaptation," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 111–121.

[24] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "Opentosca–a runtime for tosca-based cloud applications," in *Service-Oriented Computing*. Springer, 2013, pp. 692–695.

[25] A. Brogi and J. Soldani, "Matching cloud services with tosca," in *Advances in Service-Oriented and Cloud Computing*. Springer, 2013.