Reusing cloud-based services with TOSCA*

Antonio Brogi and Jacopo Soldani
Department of Computer Science, University of Pisa, Italy
{*brogi,soldani*}*@di.unipi.it*

Abstract: The OASIS TOSCA specification [OAS13b] aims at enhancing the portability and interoperability of cloud-based applications by providing a language to describe and manage them across heterogeneous clouds. A cloud-based application is modelled by a service template, an orchestration of typed nodes which can be in turn instantiated by matching [BS13] other service templates. In this paper we present a methodology to non-intrusively adapt a service template S into a new service template $newS$ which exactly matches a node type N, and hence to easily reuse any actual service modelled by S to deploy cloud-based applications that rely on N.

1 Introduction

Current cloud technologies suffer from a lack of standardization, with different providers offering similar resources in a different manner. Hence, migrating (parts of) an application from one cloud to another is a costly and error-prone process. As a result, cloud users tend to end up locked into the cloud platform they are using since it is practically unfeasible for them to migrate (parts of) their application across different clouds platforms [PMPC13].

In this scenario, OASIS recently released the *Topology and Orchestration Specification for Cloud Applications* (TOSCA), which aims at enhancing the portability of cloud-based applications by defining a language to describe and manage complex applications across heterogeneous clouds. More precisely, TOSCA [OAS13b] defines a XML-based language that permits to specify —in a vendor-agnostic way— topology and behaviour of complex composite applications as service templates that orchestrate typed nodes.

As stated in the TOSCA primer [OAS13a], a node type can be instantiated by substituting it with a "matching" service template. In our previous work [BS13], we provided a formal definition of when a service template exactly or plug-in matches a node type. Informally speaking, a service template S *exactly* matches a node type N if S has the same capabilities, requirements, properties, policies and interfaces of N. S instead *plug-in* matches N if S has "less" requirements and "more" capabilities, properties, and interfaces of N. It is worth noting that, while [BS13] formally defines the conditions for type-checking whether a node type N may be instantiated by a service template S, no methodology is provided in [BS13] to actually perform such an instantiation when S does not exactly match N.

In this paper we present a methodology which permits to non-intrusively adapt a service template S into a new service template $newS$ which exactly matches a node type N, thus allowing (the adaptation of) S to be (re)used in place of N [OAS13a]. We first consider the case of a service template S that plug-in matches a node type N. As we will see, $newS$ is obtained by extending —in a non-intrusive way— S by (artificially) adding requirements

*Work partly supported by project EU-FP7-ICT-610531 SeaClouds.

of N that are not requirements for S, and by hiding capabilities, properties, policies and interfaces of S that N does not have.

We then show that also a service template S which does *not* plug-in match a node type N may be non-intrusively adapted into a new service template $newS$ which exactly matches N. More precisely, we will show how the above mentioned adaptation can be properly extended so as to be successfully performed (i) when S exposes all the capabilities and properties as N, but in a *syntactically* different way, and/or (ii) when N declares all the requirements which are exposed by S, but in a *syntactically* different way, and/or (iii) when N features one or more interface operations which are not matched by any operation featured by S, while they can be matched by some compositions of S's operations.

It is important to observe that the adaptation of a TOSCA service template S to match a node type N does suffice to reuse any actual service modelled by S to deploy cloud-based applications that rely on N. This is thanks to the powerful way in which TOSCA supports the deployment of cloud-based applications. TOSCA permits to pack in a CSAR (*Cloud Service ARchive*) file an application specification together with the actual executable files to be deployed on a cloud platform. When a CSAR file is given in input to a TOSCA container, the latter takes care of deploying and executing the application specification contained in the CSAR file [OAS13a]. Therefore, in order to adapt an actual service modelled by a service template S to deploy an application that relies on a node type N, it suffices to adapt S into a new service template $newS$ that matches N — without having to generate an implementation of the adaptation specified by S.

It is also worth observing that, since the aforementioned adaptation methodology is non-intrusive, it can be performed both by the cloud provider offering the service S and by the application developer who needs the node N and discovers the availability of S.

Note that the adaptation works also in the case in which the CSAR of S should not be available, for instance when S is a proprietary service offered by a third party. In such cases it suffices to develop a simple proxy of the remote service modelled by S, and to pack it in a new CSAR file together with the application specification containing $newS$ (and the executable files associated with such a specification).

Finally, it is worth highlighting that, thanks to the features of TOSCA, the simple adaptation methodology that will be described in this paper considerably reduces the work needed to reuse cloud-based services if compared to the alternative of explicitly devising adapters as in traditional software adaptation approaches (e.g., [BCP06, GMMC13, KMNB$^+$09]).

The rest of paper is organised as follows. TOSCA and the matching between service templates and node types are introduced in Section 2. The methodology for adapting service templates is presented in Section 3. Finally, related work and some concluding remarks are discussed in Sections 4 and 5, respectively.

2 Background

In this section we briefly recall the main notions of the *Topology and Orchestration Specification for Cloud Applications* (TOSCA)[1] and the notions of matching between TOSCA services [BS13].

[1] Interested readers may refer to [OAS13b] and [OAS13a] for a comprehensive presentation of TOSCA, or to [BSW14] for a quick introduction to TOSCA.

2.1 TOSCA

The main aim of TOSCA is to enhance the portability of multi-cloud applications by enabling an interoperable description of application and infrastructure cloud services, of the relationships between service parts, and of the operational behaviour of services, independently of the supplier creating the service and of any particular cloud provider or hosting technology.

Syntactically speaking, TOSCA is an XML-based language for describing service templates. All definitions are contained in the XML *Definitions* element (the root of a TOSCA XML document). The *ServiceTemplate* element (Figure 1) defines all the topology and

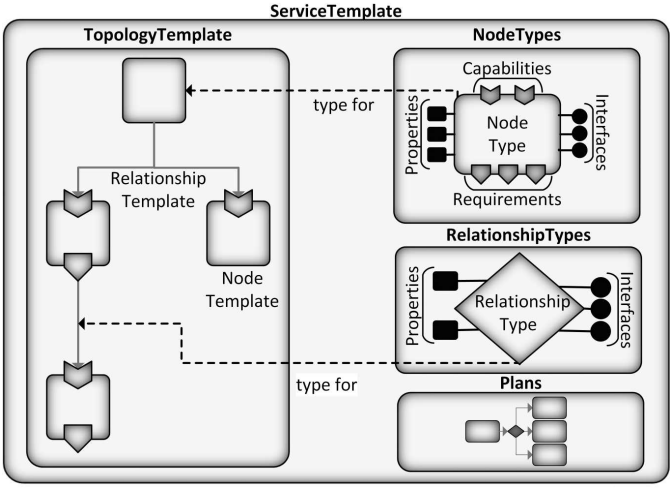

Figure 1: TOSCA *ServiceTemplate*.

management aspects of a service by means of *TopologyTemplate* and *Plans* elements. A *TopologyTemplate* specifies the topological structure of a service as a directed graph, whose nodes are *NodeTemplate*s and whose arcs are *RelationshipTemplate*s. *Plans* contains *Plan* elements that specify how to manage the associated service template during its whole lifetime. *BoundaryDefinitions* can be used to define which features are externally exposed by a *ServiceTemplate*[2].

*NodeTemplate*s are typed by means of *NodeType*s, which define the structure of the features whose values are specified in a *NodeTemplate*. Such features include properties, interfaces, requirements, capabilities, and policies. While properties and interfaces can be defined internally, requirements and capabilities must be typed by referring to external *RequirementType*s and *CapabilityType*s. Finally, *NodeTemplate*s and *ServiceTemplate*s can declare QoS information by exposing *Policy* elements, which must be typed by referring to *PolicyType*s. A *PolicyType* defines the structure of policy (and the *NodeType*s it is applicable to), while a *Policy* assigns actual policy values.

[2]For instance, a property p_{ext} can be defined on the boundaries of a service template as a reference to an internal property p_{int} (which can have a different name but must have a compatible type).

2.2 Matching service templates with node types

As stated in the TOSCA primer ([OAS13a], page 35): *"node types can be made concrete by substituting them by a service template"*. While the matching between *ServiceTemplate*s and *NodeType*s is mentioned with reference to an example, no definition of *matching* is given either in [OAS13b] or in [OAS13a]. In our previous work [BS13] we formally defined when a service template *exactly* or *plug-in* matches a node type.

A service template S *exactly* matches a node type N ($S \equiv N$) if the capabilities, requirements, properties, policies and interfaces exposed by S exactly match those of N, namely: (i) the requirements, capabilities and properties of S and N have the same name and type, and they are in a one-to-one correspondence, (ii) the policies exposed by S are applicable to N, and (iii) the interfaces of S and N have the same name, contain the same operations, and are in a one-to-one correspondence.

On the other hand, a service template S *plug-in* matches a node type N ($S \simeq N$) if, intuitively speaking, the former "requires less" and "offers more" than the latter. Namely, (i) for each requirement r of S there exists a requirement of N which has the same name as r and whose type is a sub-type of r's type, (ii) for each capability c of N there exists a capability of S which has the same name as c and whose type is a super-type of c's type, (iii) for each property p of N there exists a property of S which has the same name as p and whose (XML) type is a sub-type of p's type, (iv) the policies exposed by S are applicable to N, and (v) for each interface operation o of N there exists an interface operation of S which exactly matches o.

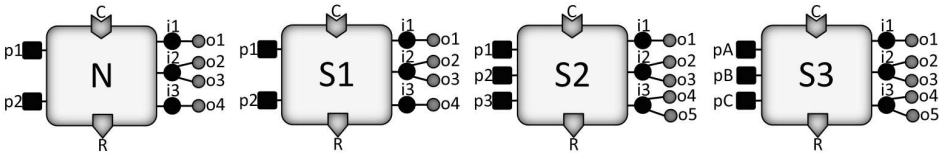

Figure 2: Example of node type and service templates to be matched.

Consider for instance the node type N and the service templates $S1$, $S2$ and $S3$ in Figure 2, where C is a capability of type $CType$, R is a requirement of type $RType$, $p1$, $p2$ and $p3$ are string properties, $i1$, $i2$ and $i3$ are interfaces, and $o1$, $o2$, $o3$, $o4$, and $o5$ are operations[3]. It is easy to see that $S1$ exactly matches N ($S1 \equiv N$) since $S1$ has the same capabilities, requirements, properties, and interfaces of N. The same does not hold for $S2$ and $S3$ ($S2 \not\equiv N$ and $S3 \not\equiv N$). $S2$ plug-in matches N ($S2 \simeq N$) because $S2$ and N expose the same requirements and capabilities, and $S2$ features "more" properties and interface operations than N. On the other hand, $S3$ does not plug-in match N ($S3 \not\simeq N$) since property names differ.

3 Adaptation Methodology

In this section we present a methodology to non-intrusively adapt a service template S into a new service template $newS$ which exactly matches a node type N, and hence to easily reuse an actual service modelled by S in the deployment of cloud-based applications that rely on N.

[3]For the sake of simplicity, we assume that S has no policies and that operations with the same name have the same input and output parameters.

3.1 Adaptation of plug-in matching services

Figure 3 describes the steps to perform when adapting a service template S, which does not exactly match ($S \not\equiv N$) but plug-in matches N ($S \simeq N$), into a new service template $newS$ that exactly matches N ($newS \equiv N$).

(1) Create a new service template $newS$ which initially contains S as the only node template in its topology.

(2) For each capability (property) of N
- (i) define a capability (property) with the same name and type on the boundaries of $newS$, and
- (ii) map the defined capability (property) onto the corresponding one of S.

(3) For each interface i of N
- (i) declare a new interface with the same name and operations on the boundaries of $newS$, and
- (ii) map each operation of the new interface onto the corresponding operation of i.

(4) Add a dummy node template *NoBe* (whose capabilities satisfy the requirements of S and whose requirements are the same of N) to the topology of $newS$.

(5) For each requirement of N
- (i) define a requirement with the same name and type on the boundaries of $newS$, and
- (ii) map the defined requirement onto the corresponding one of *NoBe*.

(where mapping f to f' simply means that f is a reference to f')

Figure 3: Adaptation of plug-in matching service templates.

We now employ the *GameScoreService* in Figure 4 to illustrate the adaptation described in Fig 3. *GameScoreService* is a service which manages score data of various games by associating to each *nick*name a list of ⟨*game*,*score*⟩ pairs. We assume that *GameScoreService* exposes a *DataConfidentialityPolicy*, that the *GameScoreApp* capability is of type *RecordingCapabilityType*, that the requirements *OSMySQL* and *OSApache* are of *OSRequirementType*, that the property *Version* is a string, and that the *nick* and *game* parameters (both representing names) are strings, while the *score* parameter is an integer.

Figure 4: Available *GameScoreService*.

Figure 5: Target *GScoreNodeType*.

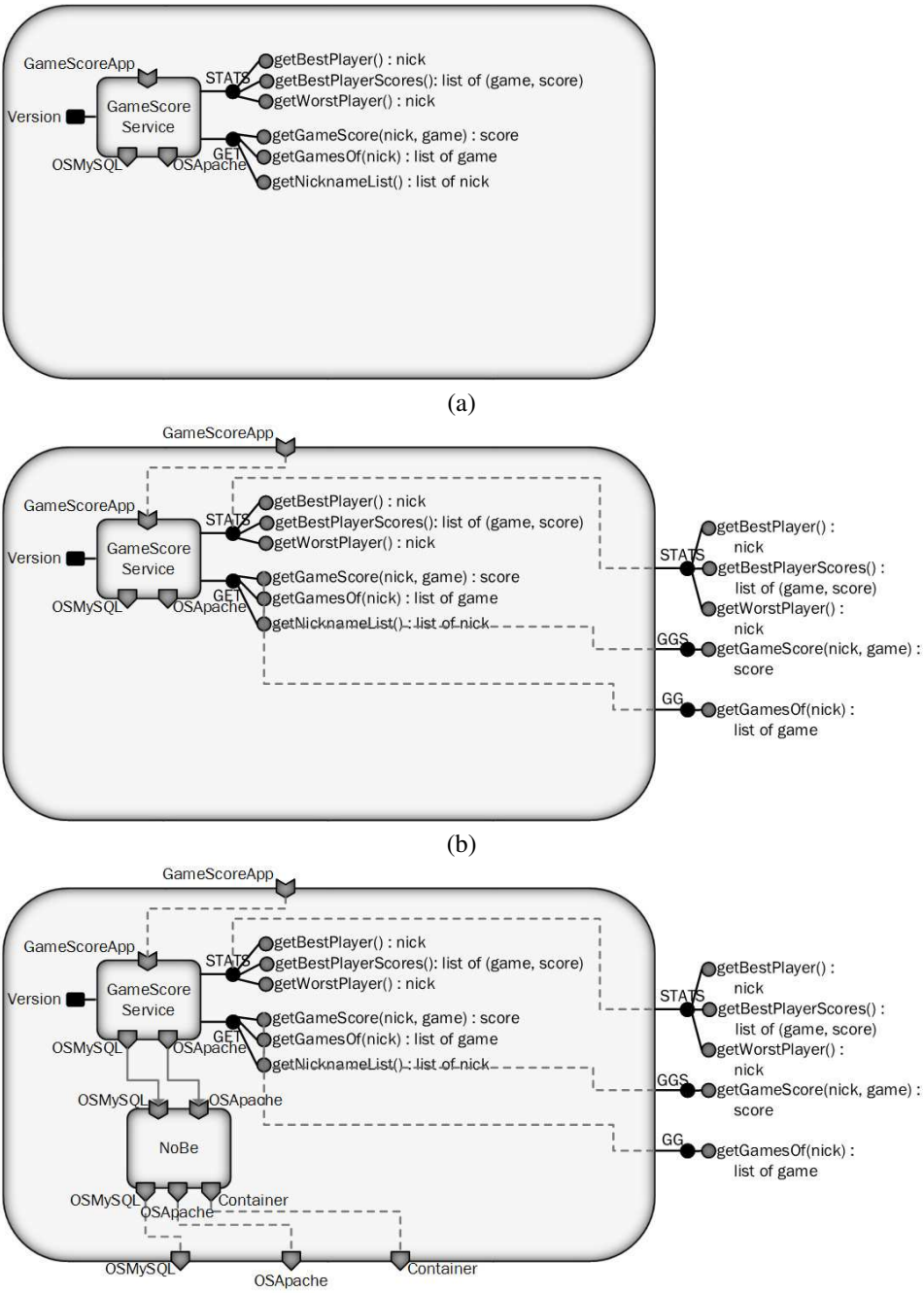

Figure 6: Example of adaptation of a plug-in matching service template: (a) after step 1, (b) after steps 2 and 3, (c) after steps 4 and 5.

Consider now the target *GScoreNodeType* (Figure 5). Suppose that the *DataConfidentialityPolicy* is applicable to *GScoreNodeType*, that the *GameScoreApp* capability is of type *GameScoreApplicationCapabilityType* (which is sub-type of *RecordingCapabilityType*), that *Container* is of type *ContainerRequirementType*, while *OSMySQL* and *OSApache* are both of type *LinuxRequirementType* (which is sub-type of *OSRequirementType*), and that the parameters *nick* and *game* are strings, while *score* is an integer. We can observe that, according to the (informal) definitions given in Sect. 2.2, *GameScoreService* does not exactly match, but it plug-in matches *GScoreNodeType*.

Figure 6 illustrates how the *GameScoreService* service template can be adapted into a new service template *GScoreService* which exactly matches *GScoreNodeType*, by applying the steps in Figure 3. Namely, (a) we create a new service template *GScoreService* which contains *GameScoreService* as the only node template, (b) we suitably define the *GameScoreApp* capability and the interface operations on the boundaries of *GScoreService*, and (c) we add the *NoBe* node to the topology of *GScoreService* so as to expose the desired requirements on its boundaries.

3.2 Adaptation of non plug-in matching services

We now extend the adaptation methodology introduced in the previous section so as to show also how a service template S which does *not* plug-in match a node type N may be non-intrusively adapted into a new service template $newS$ which exactly matches N.

(1) Create the adapted service template $newS$ which initially contains S as the only node template in its topology.

(2) For each capability c (property p) of N

(i) define a capability (property) with the same name and type on the boundaries of $newS$, and

(ii) map the defined capability (property) to that of S which is type-compatible with c (p) and whose name can be considered equivalent to the name of c (p).

(3) For each interface exposed by N, define an interface with the same name on the boundaries of $newS$. For each operation op exposed by N,

(i) define an operation with the same name and parameters in the corresponding interface exposed by $newS$, and

(ii) map the defined operation to
- an operation of S which you consider semantically equivalent to op (e.g., to an operation which exposes the same input and output parameters), or
- a plan which combines the operations of S so as to obtain an operation which you consider semantically equivalent to op.

(4) Add a dummy node template *NoBe* (whose capabilities satisfy the requirements of S and whose requirements are the same of N) to the topology of $newS$.

(5) For each requirement r of N

(i) define a requirement with the same name and type on the boundaries of $newS$, and

(ii) map the defined requirement to that of *NoBe* which is type-compatible with r and whose name can be considered equivalent to r's name.

(where mapping f to f' simply means that f is a reference to f')

Figure 7: Adaptation of *non* plug-in matching service templates.

Figure 7 describes how such an adaptation can be successfully performed (i) when S exposes all the capabilities and properties as N, but in a *syntactically* different way, and/or (ii) when N declares all the requirements which are exposed by S, but in a *syntactically* different way, and/or (iii) when N features one or more interface operation which is not matched by any operation featured by S, while it can be matched by some composition of S operations. The adaptation described in Figure 7 implements the relaxed matching conditions that were defined in [BS13] (in terms of ontology-based name equivalences). According to [BS13], the adaptation process cannot succeed if capability, property or requirement mismatches are not just syntactic or if operation mismatches cannot be solved by means of operation compositions. In other words, the adaptation in Figure 7 fails if (at least) one of the steps cannot be performed, while it succeeds if all steps are performed.

Consider for instance the target *GmsScoreNodeType* (Figure 8), where the *GmsScoreApp* capability is of type *GmsScoreApplicationCapabilityType* (which is sub-type of *RecordingCapabilityType*), *Container* is of type *ContainerRequirementType*, while *OS4MySQL* and *OS4Apache* are both of type *LinuxRequirementType* (which is sub-type of *OSRequirementType*), and the *nick* and *game* parameters are strings, while the *score* parameter is an unsigned integer. We observe that, according to the (informal) definition given in Sect. 2.2,

Figure 8: Target *GmsScoreNodeType*.

the service template *GameScoreService* (Figure 4) does not plug-in match the node type *GmsScoreNodeType*.

Figure 9 illustrates how the *GameScoreService* service template can be adapted so as to exactly match *GmsScoreNodeType*. First, (1) we create a new service template *GmsScoreService* which contains *GameScoreService* as the only node template. Since *GmsScoreApp* and *GameScoreApp* are type-compatible, (2) we adapt the capabilities by adding a *GmsScoreApp* to the boundaries of the adapted service and by mapping it to the *GameScoreApp* of the available service. Analogously, (3) we adapt the unmatched operations *getGmsScore* and *getGmsOf* by mapping them respectively on the *getGameScore* and *getGamesOf* operations. We also adapt the unmatched *getWorstPlayerScores* operation by generating the *WPScoresPlan* which suitably combines the available *getWorstPlayer*, *getGameScore* and *getGamesOf* operations. Finally, we adapt the unmatched requirements by (4) adding the *NoBe* node to the *GmsScoreService*'s topology and by (5) suitably defining the needed requirements on its boundaries.

Once all adaptation steps have been performed (Figure 9(c)) the obtained *GmsScoreService* exposes all the features exhibited by *GmsScoreNodeType*. This implies that they exactly match and hence that *GmsScoreService* can be employed to instantiate *GmsScoreNodeType*.

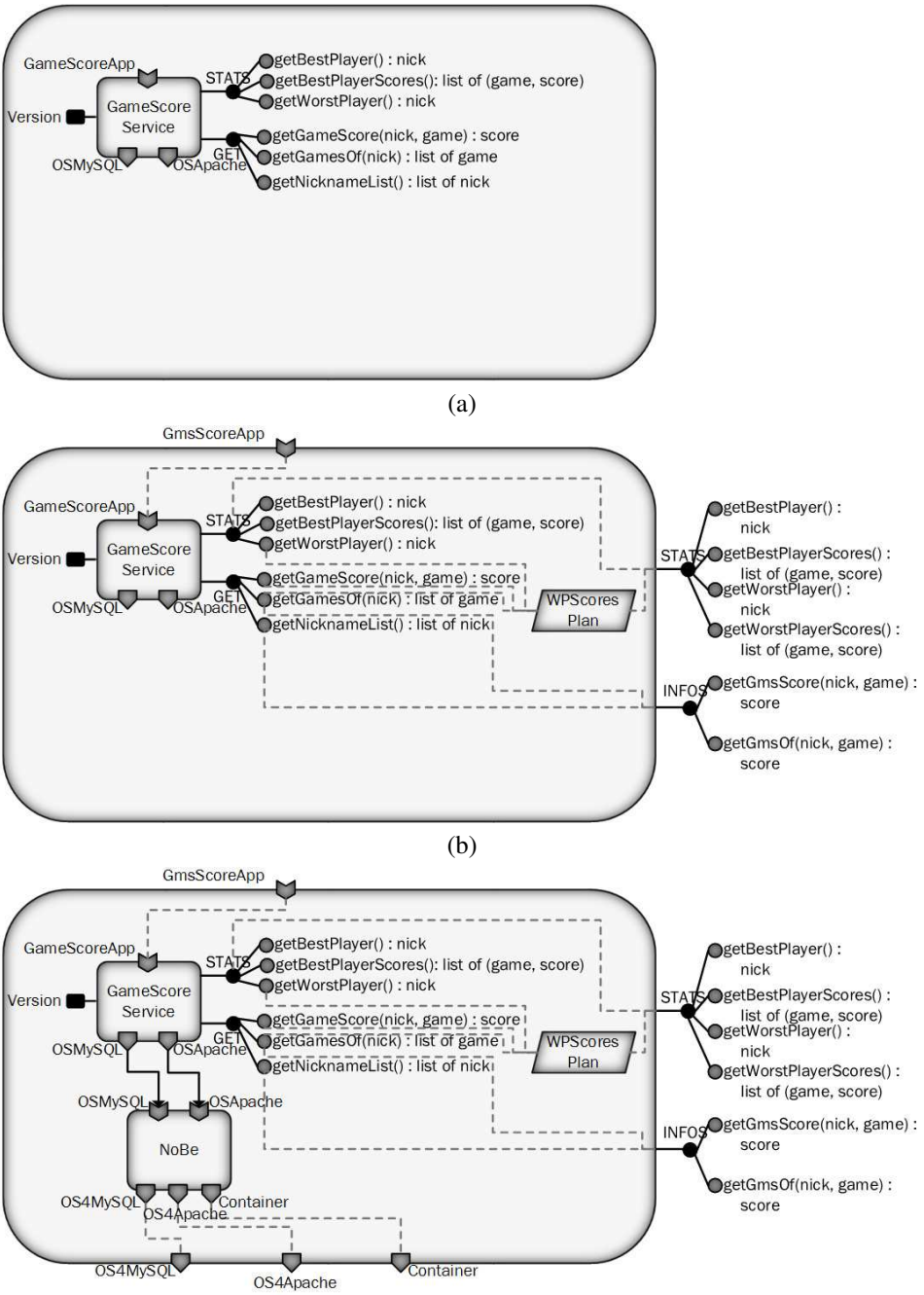

Figure 9: Example of adaptation of a non plug-in matching service template: (a) after step 1, (b) after steps 2 and 3, (c) after steps 4 and 5.

4 Related Work

The development of systematic approaches to reuse existing software is widely recognized as one of the crucial problems in system integration [XW05]. In spite of the increasing availability of cloud services, currently platform-specific code often needs to be manually modified to (re)use services in cloud-based applications. This is obviously an expensive and error-prone activity, as pointed out in [TKLF11], both for the learning curve and for the testing phases needed.

Various efforts have been recently oriented to to try devising systematic approaches to reuse cloud services. For instance, [MPC+11] and [HLT11] propose two approaches to transform platform-agnostic source code of applications developed with a model-driven methodology into platform-specific applications. In contrast, our approach does not restrict to applications developed with a specific methodology, nor it requires the availability of applications' source code, and it is hence applicable also to non open-source, third-party services.

[GMMC13] proposes a framework which allows developers to write the source code of cloud-based services as if they were "in-house" applications. Cloud deployment information must be provided in a separate file, and a middleware layer employs source and deployment information to generate the artifacts to be deployed on cloud platforms. It is worth noting that in [GMMC13] the reuse of a cloud service requires (always) invoking the middleware layer, while in our approach adaptation is performed only once. Moreover, [GMMC13] always requires to write source code, while our approach only requires to edit the application specification.

In general, most existing approaches to the reuse of cloud services support a from-scratch development of cloud-agnostic applications, and do not account for the possibility of adapting existing (third-party) cloud-based services. To the best of our knowledge, ours is the first approach which proposes a methodological approach for adapting existing cloud applications, by relying on TOSCA [OAS13b] as the standard for cloud interoperability, and to support an easy reuse of third-party services.

It is worth noting that the novelty of our approach does not reside in the type of adaptation techniques that we employ to adapt service templates. Indeed, our methodology exploits well-know adaptation patterns (e.g., [GHJV95, BBG+06]) to adapt TOSCA templates. The novelty of (applying) our approach is rather that, in contrast with traditional adaptation approaches (e.g., [BCP06, GMMC13, KMNB+09]), no additional code must be developed to reuse existing cloud-based services. This is thanks to the powerful way in which TOSCA supports the deployment of cloud-based applications. Indeed we exploit the possibility provided by TOSCA of mapping exposed features onto internal ones [OAS13b], and of entirely delegating the management of such mappings to TOSCA containers [OAS13a]. Last, but not least, it is also worth noting that our approach can be fully automated by employing ontologies to resolve syntactic differences, as suggested in [BS13].

5 Concluding remarks

In this paper we presented a methodology to non-intrusively adapt a service template S into a new service template $newS$ which exactly matches a node type N, and hence to

easily reuse any actual service modelled by S to deploy cloud-based applications that rely on N. Notably, as we already observed, the methodology can also be applied to adapt and reuse third-party services, whose source code is not available.

According to the TOSCA specification [OAS13b], we consider that a service template S can be used to substitute a node type N if S has "less" requirements, namely if for each requirement r of S there exists a requirement of N which has the same name as r and whose type is a sub-type of r's type. However, it is worth noting that there may be situations in which ignoring requirements of N that are not requirements of S may lead to loosing some relevant constraints on N (e.g., that a node template with type N should be linked to a private network). Interested readers may refer to [BSW14] for a discussion of this and other possible research directions for TOSCA.

We conclude the paper by mentioning some directions for our future work. One is obviously the implementation of a tool supporting the matching described in [BS13] as well as the adaptation methodology described in this paper. This is scope for our immediate future work, together with experimenting the methodology via such tool. Moreover, the matching and adaptation tool may also be fruitfully integrated in a plug-in for the TOSCA implementations that are currently under development (e.g., the Winery editor [KBBL13] and the OpenTOSCA engine [BBH+13]) as soon as a high-level API will be available

Another interesting direction for future work starts from the observation that cloud applications do share management infrastructure. The possibility of reusing fragments of available service templates is hence of clear practical interest. Extending the proposed methodology in this direction is also scope for our immediate work.

A further possible direction for future work is to investigate the applicability to the problem of adapting TOSCA service templates of more complex adaptation techniques, like those that input also an explicit specification of the desired adaptation (e.g., like [BCP06]).

References

[BBG+06] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an Engineering Approach to Component Adaptation. In *Architecting Systems with Trustworthy Components*, volume 3938, pages 193–215. Springer, 2006.

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner. OpenTOSCA – A Runtime for TOSCA-based Cloud Applications. In *11th International Conference on Service-Oriented Computing*. Springer, 2013.

[BCP06] A. Brogi, C. Canal, and E. Pimentel. On the semantics of software adaptation. *Science of Computer Programming*, 61(2):136 – 151, 2006.

[BS13] A. Brogi and J. Soldani. Matching Cloud Services with TOSCA. In C. Canal and M. Villari, editors, *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Science*, pages 218–232. Springer, 2013.

[BSW14] A. Brogi, J. Soldani, and P. Wang. TOSCA in a nutshell: Promises and perspectives. In *ESOCC 2014 - Proceedings of the 3rd European Conference on Service-Oriented and Cloud Computing*, Lecture Notes in Computer Science. Springer, 2014. *[In press]*.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GMMC13] J. Guilln, J. Miranda, J. M. Murillo, and C. Canal. A service-oriented framework for developing cross cloud migratable software. *Journal of Systems and Software*, 86(9):2294 – 2308, 2013.

[HLT11] M. Hamdaqa, T. Livogiannis, and L. Tahvildari. A reference model for developing cloud applications. In F. Leymann, I. Ivanov, M. van Sinderen, and B. Shishkov, editors, *CLOSER 2011 - Proceedings of the 1st International Conference on Cloud Computing and Services Science*. SciTePress, 2011.

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann. Winery – Modeling Tool for TOSCA-based Cloud Applications. In *11th International Conference on Service-Oriented Computing*. Springer, 2013.

[KMNB+09] W. Kongdenfha, H. R. Motahari-Nezhad, B. Benatallah, F. Casati, and R. Saint-Paul. Mismatch Patterns and Adaptation Aspects: A Foundation for Rapid Development of Web Service Adapters. *IEEE Transactions on Services Computing*, 2(2):94–107, 2009.

[MPC+11] B. Martino, D. Petcu, R. Cossu, P. Goncalves, T. Mahr, and M. Loichate. Building a Mosaic of Clouds. In *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science*, pages 571–578. Springer, 2011.

[OAS13a] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer, Version 1.0. "`http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.pdf`", 2013.

[OAS13b] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA), Version 1.0. "`http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf`", 2013.

[PMPC13] D. Petcu, G. Macariu, S. Panica, and C. Craciun. Portable Cloud applications - From theory to practice. *Future Generation Computer Systems*, 29(6):1417 – 1430, 2013.

[TKLF11] V. Tran, J. Keung, A. Liu, and A. Fekete. Application Migration to Cloud: A Taxonomy of Critical Factors. In *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, SECLOUD '11, pages 22–28. ACM, 2011.

[XW05] X. Xiong and Z. Weishi. The Current State of Software Component Adaptation. In *Semantics, Knowledge and Grid, 2005. SKG '05. First International Conference on*, pages 103–103, 2005.