Web Service Migration with Migration Decisions Based on Ontology Reasoning

M. Mohanned Kazzaz and Marek Rychlý
Department of Information Systems, Faculty of Information Technology, Brno University of Technology,
Božetěchova 2, 612 66 Brno, Czech Republic,
Email: {ikazzaz, rychly}@fit.vutbr.cz

***Abstract*—The advancement in ubiquitous computing requires more effort to cope with contextual changes in the environment and to adapt to these changes in regular and efficient ways. A context model and context-awareness are proposed to provide a model of the contextual information and to describe its impact on individual behaviour of system's components and on rules and relationships which control their interactions. This paper extends the Web service migration framework which provides software support for services migrating between potential service providers in service-oriented architecture according to the contextual changes. The extension utilises ontology-based context modelling in OWL/RDF and reasoning by means Jena reasoners and rules to make necessary migration decisions.**

Keywords*—*Service-oriented Architecture, Service Migration

I. Introduction

Applications of ubiquitous computing in diverse environments and needs for automatic adaptation to cope with changing contextual information in the environments lead to emerging *context awareness* in the ubiquitous systems. The context awareness enables a system to process contextual information, to be aware of contextual changes and consequent interactions between system components during runtime.

Context-aware computing was introduced at the first time by [1] as the ability of a mobile user's applications to discover and react to changes in the environment. The mobile users uses these applications to monitor and derive information on the surrounding environment. This information has meaningful observations which will be used to adapt the system according to its current status. The *context* term was defined by [2] as "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered to be relevant to the interaction between a user and an application, including the user and applications themselves".

The contextual information available to a system or its components is modelled by a *context model*. In the case of service-oriented architecture (SOA), context models describe contextual information available to individual services, which may extend a functional description of the services provided by usual means (e.g., their description in *Web Services Description Language* (WSDL)), and the actual state of service providers' environment affecting their availability (e.g., available resources or quality of network connections). In this way, context modelling makes services more accessible and discoverable. Moreover, sharing context models between different environments allows to establish a common contextual base for system or component mobility.

In our previous work [3], we proposed a concept of the Web service migration framework. The framework utilises semantic description of characteristics of services and service providers in the service migration process through defining rules and conditions of the migration. The semantic description helps a migration controller to make reasoning on the contextual data of entities in the service-oriented architecture by evaluating the rules and conditions for a particular service migration, and finally, to make a migration decision (i.e., to decide which services will be migrated and which providers will be the most suitable as the migration targets).

In this paper, we propose an ontology for semantic description of the contextual information on migratable services and relevant service providers and describe its integration and implementation into the Web service migration framework. We describe ontology reasoning by means the underlying Jena reasoner/rules engine to support migration decisions.

The remainder of this paper is organized as follows: Section II describes related work and give reasons for our approach. In Section III, we outline the previously proposed Web service migration with focus on its architecture and a migration decision process. Section IV deals with the proposed ontology and its representation. In Section V, we describe utilisation of the ontology in the migration decision process. Section VI gives details on the implementation of the Web service migration framework with reasoning based on the proposed ontology. Finally, we draw conclusions in Section VIII.

II. Related Work

Service mobility have been proposed as a very promising approach to leverage the interoperability and reusability characteristics of SOA. Multi-agent system approaches, such as JADE [4], Mobile-C [5], and AgentScape [6], provide middle-wares to host and migrate mobile agents in a distributed system. A mobile agent is an autonomous composition of code, state, and data, that can be transported from one environment to perform agent's tasks. MobiGo [7], which is another middleware system for seamless mobility, provides a mechanism to migrate services according to user's needs using a simple service description. The description contains information about service name, service type, and I/O devices which can be used to run that service. User can select the desired service from the list of available services on that particular device.

Service description is considered the main backbone supporting the reusability and interoperability of services. Ontologies have been proposed to provide service description with

modelling and reasoning techniques on context in pervasive environments to make the related decision in the system.

CONON [8] is an OWL-based ontology which provides a formal context model and implements a Descriptional Logic (DL) reasoning. A reasoning rules were used to reason over a low-level (explicit) context to derive a high-level (implicit) context based on the proposed ontology and by means of DL and RDF reasoners. Another similar work [9] uses the Web Ontology Language (OWL) [10] ontology and the Semantic Web Rule Language (SWRL) [11] rules to model context in a context-aware system using Rule-Based Inference engine.

Ejigu et al. [12] similarly proposed Context Management Model (GCoMM) to provide reasoning and decision making in context-aware system. They used an ontology-based context model and defined rules on given data instances.

An OWL-based device ontology was provided by Bandara et al. [13] to describe devices and their hardware and software components. However, the proposed ontology lacks for full service descriptions as it only provides an initial representation for devices' services using a relationship called *hasService* without any detailed description of the services' attributes. On the other hand, in [14], an ontology has been used to provide a service's description and preferences and to allow match-making techniques on these descriptions.

In this work, we favor to use the ontology-based approach for service migration not only because it describes a system semantically with a proper definitions of the relationships between its components, but also regarding to its capability to reason with the Semantic Web. For example, Ontology Based Language (OWL) uses DL functionalities to reason the semantic description and helps in making the migration decision.

III. Web Service Migration Framework

Traditional models of SOA assume that services are provided permanently by service provides which are predefined at a system's deploy-time or found in service registries at its runtime [15]. This approach is sufficient for the most applications of SOA where services implement fixed functionality for a given application running in a stable environment. However, in the cases of volatile runtime environments (the environments with variable resources, unreliable providers, etc.), services have to be deployed and redeployed at runtime according to environmental and contextual changes.

The service migration enables services to be transparently moved across various network nodes that act temporarily as service providers according to their availability and their resources. Through the service migration, the system is able to cope with the inherent environmental dynamics and to keep functionality and quality of its services (e.g., to employ temporarily available mobile devices as service providers, to react to possible failures of service providers with unreliable resources, etc.).

To support migration of Web services in SOA, we proposed a concept of the Web service migration framework [3]. The framework describes an overall service-oriented architecture supporting the service migration and defines interfaces which can be implemented to adapt the framework to a particular Web service implementation technology. It also provides extension points for user-defined migration decision strategies, i.e., the strategies deciding when the migration of a particular service is needed and how it will be performed.

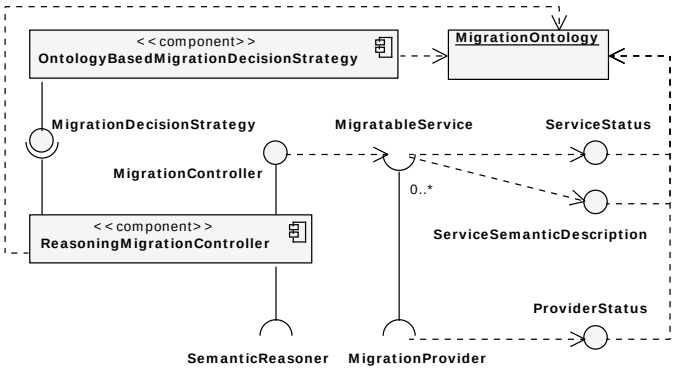

Fig. 1. Components of the Web service migration framework, their interfaces, and interfaces which have to be provided or utilised by user components, e.g., by migratable services and service providers.

In the framework, service migration is controlled by a *migration controller* component. The controller checks statuses of migratable services and their actual and potential service providers and evaluate them in cooperation with *migration decision strategy* component to decide which services will be migrated and which providers will be the most suitable as the migration targets. Basically, the framework itself does not take the migration decisions, it relies on a particular migration decision strategy component provided by a user utilising the framework, while the framework just defines an interface and a protocol, i.e. service specification and orchestration, to plug-in and to use the component, respectively.

However, to simplify the utilisation of the framework, we decided to integrate the migration decision component into the framework's core. The framework, not its user, should implement a migration decision algorithm, yet it should be able to provide the user with control over the migration decisions based on current properties of services, actual and potential service providers, and their environment. The migration decision should be controlled by the user in declarative way, e.g., by addressing the context required by the service and provided by the service providers, without bothering about implementation details of the migration decision algorithm (i.e., about the implementation of the migration decision component).

The above mentioned goals can be reached by using semantic description (ontologies) of migratable services and their providers (supplied by a user) and by employing ontology reasoning in migration decisions (ensured by the framework).

The framework's core components, their interfaces, and interfaces which have to be provided or utilised by user components (e.g., by migratable services and service providers) are described in Figure 1. The framework implements the migration decision algorithm in *ReasoningMigrationController* and *OntologyBasedMigrationDecisionStrategy* components. Both components use *MigrationOntology* describing the domain of service-oriented architecture with migrating services and their providers. The services and providers are linked to the framework's components via required interfaces *MigratableService* and *MigrationProvider* (a provider can provide "zero to many" services, which is indicated by the association of

theses interfaces). To provide semantic description of migratable services and their actual or potential providers, the services have to implement *ServiceStatus* and *ServiceSemanticDescription* interfaces and the providers have to implement *Providerstatus* interface, with respect to *MigrationOntology*. Then, *ReasoningMigrationController* can get the semantic descriptions and utilise *SemanticReasoner* and *OntologyBasedMigrationDecisionStrategy* to make migration decisions.

IV. Ontology representation in SADL

Many ontology languages, such as OWL, SWRL, or Resource Description Framework (RDF) [16], were developed to provide formal semantics for Web domains and to apply rules on these semantics to derive new meaningful data. In these languages, a user has to define the semantics of system components in XML or RDF triples which may become more complicated and not suitable for demonstration purposes, especially in the cases of systems with complex ontologies.

In this paper, we use the Semantic Application Design Language (SADL, [17]) to describe presented ontologies. SADL provides an English-like language for building semantic models and authoring rules. It enables us to describe the relationships between a domain's defined concepts and also to define conditional statements as rules on the concepts. SADL is provided as a plug-in in the Eclipse environment which automatically converts a defined SADL model into a set of OWL schemas and Jena rules. Then, OWL and Jena reasoners can be used to deduce new facts and implicit context through querying and inferencing on OWL and RDF data.

The basic element in SADL model is a *class*. Each class can have *properties*. A property of a class can have a single or many values of one of SADL data types (it is a data-type property) or previously declared classes (it is an object property, which describes a relationship between classes).

The core of our ontology, illustrated in Figure 2, consists of *Service* and *ServiceProvider* classes. *Service*, which is based on Web Service OWL-S model, is specialised to a *ProvidedService* and *FrameworkService*. A *ProvidedService* is considered as a *MigratableService* iff it is *presentedBy* a *MigratableServiceProfile*. A *ServiceProvider*, which is usually hosted by a *Device* (also described in the ontology), is identified by *hostname* and *protocol* properties. The relationship between *ServiceProvider* and its *ProvidedService* is called *provides*.

Both of *MigratableServices* and *ServiceProviders* may have operating preferences, which is indicated by *false* value of their *noPreferenceRules* property. Each preference is described as a simplified Jena rule expressing the operating conditions that satisfy service or provider needs. These rules are evaluated by Jena reasoner to check whether each service can be a *possibleProvidedService* by some *ServiceProvider* or not. Based on this reasoning, each *MigratableService* which cannot be *possibleProvidedService* by its current provider is marked as a *CandidateForMigrationService* and its provider as a *CandidateOriginServiceProvider*. The same reasoning is used later, to find a suitable *CandidateDestinationServiceProvider* for each *CandidateForMigrationService* and in case it is found a *MigrationDecision* will be taken (the service can be migrated).

A set of auxiliary services were implemented to support the migration process. The services are instances of *FrameworkService* and operate either on origin or destination providers to collect informations about *CandidateForMigrationService*, *CandidateOriginServiceProvider*, and *CandidateDestinationServiceProvider*, and to help in performing the migration process.

```
Device is a top-level class.
/* a provided service is an OWL-S service provided by a service provider */
ProvidedService (alias "provided service") is a type of
OWLS_Service:Service.
/* a migratable service and its specification by an OWL-S profile */
MigratableService is a type of ProvidedService.
MigratableServiceProfile is a type of OWLS_Profile:Profile.
relationship of MigratableService to MigratableServiceProfile is
OWLS_Service:presentedBy.
CandidateForMigrationService is a type of MigratableService.
/* a service provider */
ServiceProvider (alias "service provider") is a top-level class
described by hostname with a single value of type string
described by protocol with a single value of type string.
protocol of ServiceProvider always has value "SOAP".
relationship of ServiceProvider to Device is hostedBy.
/* types of ServiceProvider participating in migration decision process */
CandidateOriginServiceProvider is a type of ServiceProvider.
CandidateDestinationServiceProvider is a type of ServiceProvider.
/* attributes indicating empty preference rules set */
noPreferenceRules describes { MigratableService or
CandidateDestinationServiceProvider } with a single value of type
boolean.
/* a migration decision, a mapping of services which will/can be migrated to
other providers */
MigrationDecision is a top-level class
described by migratedService with a single value of type
CandidateForMigrationService
described by destinationProvider with a single value of type
CandidateDestinationServiceProvider.
```

Fig. 2. A fragment of the Web service migration ontology in SADL, defined classes, properties and relationships.

V. Ontology Reasoning and Migration Decisions

Migration decisions are justified by a set of rules. A rule can represent a service or provider preferences, or a migration decision making condition. In case of a preference rule, specific details about service or provider requirements are identified to ensure an ideal operation circumstances for them. For example, the rule can specify a limit of the free memory or battery level that must be available during system operation. These rules are checked by the controller whether these preferences are satisfied or violated. As a result of rules' reasoning, new contextual facts are concluded describing which provider can provide which service. In other word, which provider is considered *possibleDestinationProvider* for some *MigratableService*.

The controller periodically investigates about any possible migration in a system through the following rules presented in Figure 3: First, *LookForCandidateForMigrationServiceDueToProvidersPreferences* and *LookForCandidateForMigrationServiceDueToServicesPreferences* are used to find *CandidateOriginServiceProvider* and *CandidateForMigrationService* suitable for migration based on contextual facts derived by the aforementioned preference rules. Then, *LookForMigrationDestinationsForEachMigratingService* provides all possible *MigrationDecision* suggestions of *migratedService* and *destinationProvider* (representing "what" and "where" will be migrated).

VI. Implementation Details

As it was mentioned before, migration decisions taken by the migration controller in the Web service migration framework are based on semantic description of services and providers, on their status, and on their contextual information. The decisions

```
/* 1st step of migration decision making */
Rule LookForCandidateForMigrationServiceDueToServicesPreferences
      given  service is a MigratableService
             origin is a CandidateOriginServiceProvider
      if     origin provides service
             origin is a CandidateDestinationServiceProvider
             service has possibleDestinationProvider not origin
      then   service is CandidateForMigrationService.
Rule LookForCandidateForMigrationServiceDueToProvidersPreferences
      given  service is a MigratableService
             origin is a CandidateOriginServiceProvider
      if     origin provides service
             origin is a CandidateDestinationServiceProvider
             origin has possibleProvidedService not service
      then   service is CandidateForMigrationService.
/* 2nd step of migration decision making */
Rule LookForMigrationDestinationsForEachMigratingService
      given  service is a CandidateForMigrationService
             origin is a CandidateOriginServiceProvider
             destination is a CandidateDestinationServiceProvider
      if     origin provides service
             origin is not destination
             service has possibleDestinationProvider destination
             destination has possibleProvidedService service
      then   a MigrationDecision
                    with migratedService service
                    with destinationProvider destination.
```

Fig. 3. The migration decision rules defined in SADL.

are based on reasoning with the ontology described in the Section IV and the domain rules described in Section V.

For the reasoning with the ontology and the rules, we use tools from the Apache Jena project[1], a Java-based framework for building Semantic Web applications. More specifically, Apache Jena is utilised for the following purposes:

1) to model the services and providers, their status and contextual information – by describing the ontology with classes, object and data-type properties, and their restrictions in the *OWL 2 Web Ontology Language* (OWL 2) with *OWL 2 RDF-Based Semantics* [18],
2) to explicitly describe facts about the services and providers, their status and contextual information in accordance with the model – by building graphs in *Resource Description Framework* (RDF, [16]) describing migratable services and providers (as RDF resources), their relationships (as RDF object properties interconnecting the resources) and status (as RDF literals assigned to data properties of the resources),
3) to derive unstated facts about the services and providers, their status and contextual information from the explicitly described facts together with the model – by utilising Jena's *OWL* reasoner [19] to derive additional information, which may not be explicitly contained in the RDF graphs and which can be consequences of the knowledge in the ontology in the *OWL/Lite* subset of the *OWL/Full* language [20],
4) to derive additional facts about the services and providers, their status and contextual information from the all known facts and given axioms and rules – by defining Jena hybrid rules (combinations of forward and backward rules, [19]) with premises and conclusions containing the previously defined facts and built-in primitives and by utilising Jena's *general purpose rule engine* in a hybrid execution model.

[1] https://jena.apache.org/

The OWL 2/RDF model (purpose (1) in the list above) of migratable services and service providers, their status and contextual information, is static, i.e., it does not depend on a particular application of the framework, and it is given by the ontology (see Section IV). Contrary to the model, the facts (2) about the services and providers, their status and contextual information, are dynamic and have to be obtained on demand at runtime of a service-oriented system utilising the framework. These facts describe actual configuration of the system's service-oriented architecture and status of its components/services.

The facts (2), such as existing services and providers, are constructed by the framework at runtime according to discovered migratable services and available service providers (the providers can emerge and/or leave the system during its runtime). Moreover, we extended Apache Jena with custom procedural build-in primitives, which will be used particularly by the Jena's general purpose rule engine in Jena rules (4). The implementations of the custom build-in primitives are based on class *com.hp.hpl.jena.reasoner.rulesys.builtins.BaseBuiltin*.

Namely, we implemented classes for the primitives: *checkServiceStatus(in URI, out status)*, *checkProviderStatus(in URI, out status)*, and *getStatusProperty(in status, in propertyName, out propertyValue)*. The first two build-in primitives access a Web service with a given URI which reports status of a migratable service or a service provider, respectively. The interface of the accessed Web service is standardised across the framework, which defines the abstract section of the Web service's description in WSDL, i.e., a list of its provided operations, their arguments, data types, and a message format. The last build-in primitive is able to extract a value of a property by its name from given status and it has to be used together with one of the first two build-in primitives (a service's or a provider's status has to be checked before its properties can be extracted).

Finally, additional facts, especially the facts representing migration decisions (e.g., the decision whether a migration of a particular service is needed), will be derived according to Jena rules (4). These rules consist of preference rules and migration decision rules. The preference rules describe preferences of migratable services and service providers, such as which providers suit a particular migrating service as its possible migration destination or which services are suitable to be provided by a particular provider (for example, a service can specify minimal requirements for potential providers to be able to provide the service). The framework can get these rules in a simplified format at runtime, by a standardised Web service interface which has to be implemented by each provided migratable service and each service provider. Then, the preference rules are used together with migration decision rules, which represent the framework's migration decision strategy, to reason towards migration decisions (see Section V).

A. Reasoning Migration Controller

The reasoning migration controller is the framework's core component. It monitors a system's service-oriented architecture and evaluates status and contextual information provided by individual services and service providers in order to make migration decisions (e.g., that migration of a particular service is needed and how it can be performed). The controller also

orchestrates services and providers participating in the migration (it migrates services to target service providers, guarantees delivery of incoming messages during and after the migration, etc.; for details of the orchestration, see [3]).

During its life-cycle, the migration controller can be in the three modes, which are described in this section, namely: under initialisation, monitoring, and performing migration. The initialisation is performed only once, early after the execution of a system utilising the framework. The next two stages are performed repeatedly, i.e., the controller is monitoring and evaluate information, and performing the migration if needed.

The *initialisation of the controller* includes the loading and preparing the appropriate resources. In our case, the reasoning migration controller loads a schema with the migration ontology (the ontology described in Section IV) into an object implementing *com.hp.hpl.jena.rdf.model.Model* interface. It also creates an instance of Jena's default OWL reasoner, with the pre-built standard configuration, specialised to the previously loaded schema by methods *ReasonerRegistry.getOWLReasoner-().bindSchema(schema)*. Finally, the controller starts a WS-Discovery server which will probe for predefined Web services required by the controller in the next stages and provided by emerging service providers and their service (the Web services reporting status and preference rules of migratable services and service providers). The server is utilised by the framework in the next stages to discover available service providers and their migratable services by means of Web Services Dynamic Discovery (WS-Discovery, [21]). For the server, we adopted Java libraries from the *java-ws-discovery* project[2].

In the second stage, *the controller monitors* available service providers and their migratable services by means of the WS-Discovery server. For each service provider and for each migratable service, the controller creates an RDF resource representing the provider or the service. It also sets RDF properties of the resource to their values or related objects (e.g., an URI of a service, or a relationship to its service provider) by *createResource*, *createProperty*, and *createTyped-Literal* methods of an instance implementing *com.hp.hpl.jena-.rdf.model.Model* interface. The resulting set of RDF resources, properties, and literals forms an RDF model describing facts about the current configuration and status of a system's service-oriented architecture. The model is kept up to date with new information provided continually by the WS-Discovery server.

Besides the continuous monitoring in the second stage, *the controller also periodically evaluates* status and contextual information provided by available service providers and their migratable services in order to make migration decisions. The evaluation is done by means of Jena reasoners. More specifically, the controller creates an information model as an instance implementing *com.hp.hpl.jena.rdf.model.InfModel* interface to generate additional entailments. The entailments are generated from the actual set of RDF data describing current configuration of the system's SOA by the Jena's OWL reasoner configured in the initialisation stage of the controller (the reasoner uses the schema representing the migration ontology). The information model is used to get all providers (represented by RDF resources) which currently provide (described by a RDF property) some migratable services by method *listStatements* of the information model instance. For each migratable service and each its provider, the controller invokes a particular Web service required by the framework (see the end of Section VI) to obtain simplified preference rules, i.e., the rules describing the service's or the provider's preferences (e.g., minimal requirements which has to meet a service provider for providing the service). These rules are translated into Jena rules and together with additional Jena rules defined by the framework's migration decision strategy (see Section III), they are used to configure an instance of Jena's generic rule reasoner working in hybrid mode by methods *GenericRuleReasonerFactory.theInstance().create-(configuration)*. Finally, the resulting generic rule reasoner is utilised in a new information model (another instance implementing *com.hp.hpl.jena.rdf.model.InfModel* interface) to check if there are providers or services which need migration (e.g., the cases when requirements stated by a migratable service are not meet by its current service provider).

If the migration is needed by a migratable service, the controller makes another migration decision with the same rule reasoner to get the best destination service provider for the migration. Then, the controller enters into the third stage and performs the migration as it has been described in [3].

VII. Discussion and Future Work

In the previous sections, we proposed the modification of the Web service migration framework replacing fully customisable migration decision strategies with semantic description of migratable service, service providers, and rules which control migration decisions. The proposed approach makes the framework easy to use. Potential users of the framework just have to equip the migratable services and service providers with Web service interfaces providing actual values of their custom properties (i.e., to read the status of the services and providers) and providing a set of simplified rules describing their preferences (i.e., minimal requirements for potential providers to be able to provide a service, or limitations of potential services which may be provided by a given provider; see the end of Section VI). However, the proposed approach does not reach the same flexibility as the original framework with custom migration decision strategies (e.g., it is very difficult to make a large-scale custom migration decision based on a combination of statuses of multiple migratable services of service providers because their preference rules are provider individually).

Performance of the proposed migration controller, especially its intensive memory and network usage during reasoning, is another problem which should be addressed. To reduce complexity of the OWL reasoning, we can use an OWLMicro implementation of Jena's default OWL reasoner, which achieves much higher performance then "full" OWL. Contrary to the full implementation of Jena's OWL reasoner, the OWLMicro implementation omits OWL cardinality restrictions and equality axioms, which enables it to achieve much higher performance [19]. In this case, the migration ontology needs to be described in the plain RDF Schema (RDFS) plus OWL constructs *intersectionOf*, *unionOf*, and *hasValue*. In this case, OWL *some-ValuesFrom* constraint, which is analogous to the existential quantifier of Predicate logic, can not be used [22].

To reduce network traffic overhead, we can improve the custom build-in primitives used in Jena rules. Build-in primitives are evaluated as needed by the Jena's general purpose

[2]https://code.google.com/p/java-ws-discovery/

rule engine. Custom build-in *getStatusProperty* is "monotonic" (according to Jena's terminology), i.e., it will always return the same output for the same inputs. In terms of logic, the predicate will be always true or always false for the same valuation. Therefore, all occurrences of this build-in primitive with the same parameters can be evaluated at once. On the contrary, the status checks in *checkServiceStatus* and *checkProviderStatus* make these build-in primitives "non-monotonic"; their outputs will be changing over time with the corresponding status changes of checked services or providers. To reduce frequency of Web service calls for the status check, an observer approach and caching can be considered. Calls of *checkServiceStatus* and *checkProviderStatus* will be cached for each accessed Web service and a corresponding checked service or provider will announce changes of their statuses which will clear the cache.

Another weak point resulting into network traffic overhead is monitoring of services' and providers' status changes. Currently, the migration controller periodically asks available service providers and their migratable services for status updates. Corresponding Web service invocations generate appreciable network traffic. This problem can be solved by switching the migration controller from the active role (i.e., the controller asks services or providers for updates) to passive role (i.e., the controller is notified on the status updates by updated services or providers). This feature can be implemented by means of *Web Services Eventing* (WS-Eventing, [23]), e.g., by adopting and adapting the results of the *Open WS-Eventing* project[3].

Besides working on the above-mentioned limitations and improvements, our future work will focus mainly on a prototype implementation and an evaluation of the proposed modification of the framework to measure its usability and performance in real-world case studies.

VIII. Conclusion

In this paper, we proposed an extension of the Web service migration framework by ontology-based context modelling in OWL/RDF and reasoning by means of Jena reasoners and rules to make migration decisions. The ontology reasoning is based on a static ontology and migration decision rules defined by the framework, and on a dynamic set of facts and preference rules obtained by the framework from migratable service and their providers at its runtime. The proposed approach makes the framework easy to use for potential users, who do not need to implement individual migration strategies, while it preserves customisability of the migration decision process with the strength of ontological reasoning.

Acknowledgement

This work was supported by the research programme MSM 0021630528 "Security-Oriented Research in Information Technology" and by the BUT FIT grant FIT-S-11-2.

References

[1] B. N. Schilit and M. M. Theimer, "Disseminating active map information to mobile hosts," *IEEE Network*, vol. 8, no. 5, pp. 22–32, Sep. 1994.

[2] A. K. Dey and G. D. Abowd, "Towards a better understanding of context and context-awareness," Georgia Institute of Technology, College of Computing, Tech. Rep. GIT-GVU-99-22, Jun. 1999.

[3] M. M. Kazzaz and M. Rychlý, "A web service migration framework," in *ICIW'13, The Eighth International Conference on Internet and Web Applications and Services*. IARIA, Jun. 2013, pp. 58–62.

[4] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa, "JADE: A white paper," *EXP in search of innovation*, vol. 3, no. 3, pp. 6–19, 2003.

[5] B. Chen, H. H. Cheng, and J. Palen, "Mobile-C: a mobile agent platform for mobile c-c++ agents," *Software: Practice and Experience*, vol. 36, no. 15, pp. 1711–1733, Dec. 2006.

[6] N. J. E. Wijngaards, B. J. Overeinder, M. van Steen, and F. M. T. Brazier, "Supporting internet-scale multi-agent systems," *Data and Knowledge Engineering*, vol. 41, no. 2-3, pp. 229–245, Jun. 2002.

[7] X. Song and U. Ramachandran, "Mobigo: A middleware for seamless mobility," in *RTCSA'07, 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2007, pp. 249–256.

[8] X. H. Wang, D. Q. Zhang, T. Gu, and H. K. Pung, "Ontology based context modeling and reasoning using owl," in *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*. IEEE, 2004, pp. 18–22.

[9] K.-C. Lee, J.-H. Kim, J.-H. Lee, and K.-M. Lee, "Implementation of ontology based context-awareness framework for ubiquitous environment," in *MUE'07, International Conference on Multimedia and Ubiquitous Engineering*. IEEE, 2007, pp. 278–282.

[10] D. L. McGuinness and F. Van Harmelen, "Owl web ontology language overview," *W3C recommendation*, vol. 10, no. 2004-03, p. 10, 2004.

[11] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "Swrl: A semantic web rule language combining owl and ruleml," *W3C Member submission*, vol. 21, p. 79, 2004.

[12] D. Ejigu, M. Scuturici, and L. Brunie, "An ontology-based approach to context modeling and reasoning in pervasive computing," in *PerCom Workshops'07, Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops*. IEEE, 2007, pp. 14–19.

[13] A. Bandara, T. R. Payne, D. de Roure, and G. Clemo, "An ontological framework for semantic description of devices," in *ISWC'04, International Semantic Web Conference*, 2004.

[14] D. Bianchini, V. D. Antonellis, M. Melchiori, and D. Salvi, "Lightweight ontology-based service discovery in mobile environments," in *DEXA'06, 17th International Workshop in Database and Expert Systems Applications*, 2006, pp. 359–364.

[15] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustdar, "Towards recovering the broken SOA triangle: a software engineering perspective," in *Proceedings of the 2nd International Workshop on Service Oriented Software Engineering*. New York, NY, USA: ACM, 2007, pp. 22–28.

[16] "Resource description framework (RDF): Concepts and abstract syntax," W3C, W3C Recommendation, Feb. 2004.

[17] A. W. Crapo, *Semantic Application Design Language (SADL) (Version 2): A Detailed Look*, General Electric Company, Sep. 2011.

[18] J. Carroll, I. Herman, and P. F. Patel-Schneider, "OWL 2 web ontology language: RDF-based semantics, second edition," W3C, W3C Recommendation, Dec. 2012.

[19] *Apache Jena – Reasoners and rule engines: Jena inference support*, The Apache Software Foundation, 2013.

[20] "OWL web ontology language: Overview," W3C, W3C Recommendation, Feb. 2004.

[21] T. Nixon and A. Regnier, "Web services dynamic discovery (ws-discovery) version 1.1," OASIS Open, OASIS Standard, Jul. 2009.

[22] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, "OWL web ontology language reference," W3C, W3C Recommendation, Feb. 2004.

[23] D. Box, L. F. Cabrera, C. Critchley, F. Curbera, D. Ferguson, S. Graham, D. Hull, G. Kakivaya, A. Lewis, B. Lovering, P. Niblett, D. Orchard, S. Samdarshi, J. Schlimmer, I. Sedukhin, J. Shewchuk, S. Weerawarana, and D. Wortendyke, "Web services eventing (ws-eventing)," W3C, W3C Member Submission, Mar. 2006.

[3] https://sourceforge.net/projects/wse/