# Ontology Supported Selection of Versions for N-Version Programming in Semantic Web Services

Paweł L. Kaczmarek

Department of Computer Systems Architecture,
Faculty of Electronics, Telecommunications and Informatics,
Gdańsk University of Technology
pawel.kaczmarek@eti.pg.gda.pl

**Abstract.** Web Services environment provides capabilities for effective N-version programming as there exist different versions of software that provide the same functionality. N-version programming, however, faces the significant problem of co-relation of failures in different software versions. This paper presents a solution that attempts to reduce the risk of co-relation of failures by selecting for invocation services having relatively different non-functional features. We use an ontology-driven approach to identify and store information about software features related to differences in software versions, such as: software vendor, design technology or implementation language. We present an algorithm for selection of software versions using the designed ontology. The solution was verified in a prototypical implementation with the use of an existing OWL-S API library.

## 1 Introduction

N-version programming (NVP) is a resilience computing mechanism [1] used for decades to increase software dependability. The technique was initially used and researched in sequential systems, however, different research groups focus on NVP in distributed systems as described later in this paper. It seems that NVP can be successfully applied in Web Services and Semantic Web Services. The Web Services architecture assumes that services supplying the same functionality are advertised and available for clients. A client can either choose a service that supplies the best price and dependability or invoke different services in order to increase dependability.

The paper addresses the typical problem that NVP faces: there exists a co-relation between errors in different software versions [2]. The co-relation results from similar educational background, programming languages, the algorithms used and other factors. In Web Services, however, services differ in vendor and technology, which might lay foundations for creation of dependable N-version systems. In our solution, we attempt to design a technique for selection of services that are unlikely to fail for similar input or in similar conditions.

M. Bubak et al. (Eds.): ICCS 2008, Part I, LNCS 5101, pp. 317–326, 2008.

## 2 Semantic N-Version Invocation Module

The designed solution is aimed at increasing dependability of NVP without increasing the number of invoked versions of a service. The limited number of invoked services reduces the invocation costs. In this solution, we select those services that have relatively different non-functional features, which consequently limits the risk of repeating feature-specific errors during the execution of a selected set.

The first step is to identify service features, dependencies between features and their impact strength. An N-version features ontology is defined to describe the features related to differences in software versions. Examples of ontology concepts are: implementation language, software vendor, design process, runtime platform and the algorithms used (see Sect. 3.1). It is assumed that the already existing service registers know different services supplying the same functionality. It is also assumed that the existing servers already offer services of equal functionality. Relevant information about available services is stored in the ontology.

The next step is to design an algorithm for selection of services depending on service features. Generally, the algorithm calculates the number of common features for groups of services and selects a group in which services are relatively different(see Sect. 3.2).

An N-version invocation that uses our service selection mechanism consists of the following steps:

- A matching subsystem selects available services that match clients request.
- Services are selected from the initial set concerning service features.
  - The N-version features ontology is queried for service features
  - Binary service similarities are identified
  - Service similarities are calculated for potential groups
  - A group with the lowest service similarity is selected for invocation
- Selected services are invoked
- Result is voted and returned to the client.

Finally, the solution is implemented and validated.

### 2.1 Module Architecture

The architecture of the the N-version invocation module is shown in Fig. 1.

The system consists of the following submodules:

**Search and matching module** - performs typical tasks for service discovery and matching. It is assumed that an already existing matching module is used and the module is capable of delivering a set of services that match a client's request.

**Selection module** - selects services for an N-version invocation from available services supplied by the Search module. A service features knowledge base is used to create a configuration of possibly different services.

**Service knowledge base** - uses the N-version features ontology to store information about known services. It can be stored twofold:

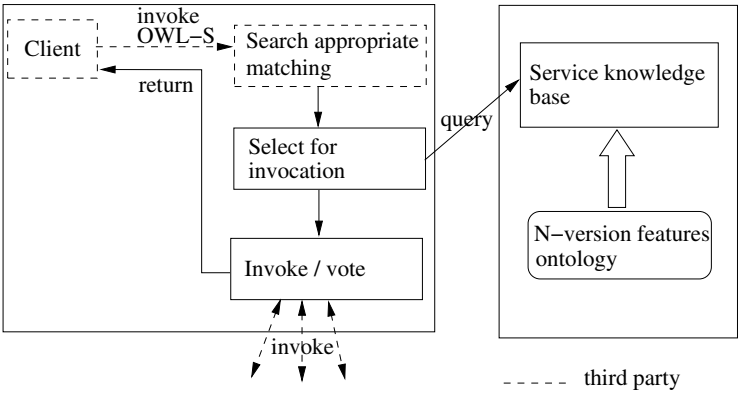


**Fig. 1.** Main parts of N-Version invocation modules with replicas selection

– locally - integrated in client's application
– remotely - accessible to different clients through Web Services

**Invocation module** - manages OWL-S [3] definitions, invokes N-version services, votes the result and returns it to a client.

## 3 Selection of Services

The identified service features and dependencies related to service differences in NVP are stored as an ontology. We use ontology-driven approach because of the following: (i) it is a systematic and organized way for entity description, (ii) there already exist ontologies and taxonomies that describe services and (iii) there are technological similarities between ontologies (OWL) and Semantic Web Services (OWL-S).

### 3.1 N-Version Features Ontology

We designed the N-version features ontology focusing on concepts concerning correlation of errors during N-version invocation. The designed ontology is based on the following existing ontologies and taxonomies: EvoOnt - A Software Evolution Ontology [4], Ontology and Taxonomy of Services [5], Core Software Ontology [6] and Service Ontology from Obelix [7].

Fig. 2 presents classes and relations defined in the N-version features ontology.

A *SemanticService* describes a service that contains ontological descriptions of the service bundle contents [7]. A service is a loosely coupled, reusable software component that semantically encapsulates discrete functionality and is distributed as well as programmatically accessible over standard Internet protocols. Concepts describing a *SemanticService* concern vendor, development and runtime information. A *Vendor* of a *SemanticService* is an organization or a person that supplies the service. A *SemanticService* is designed with the use

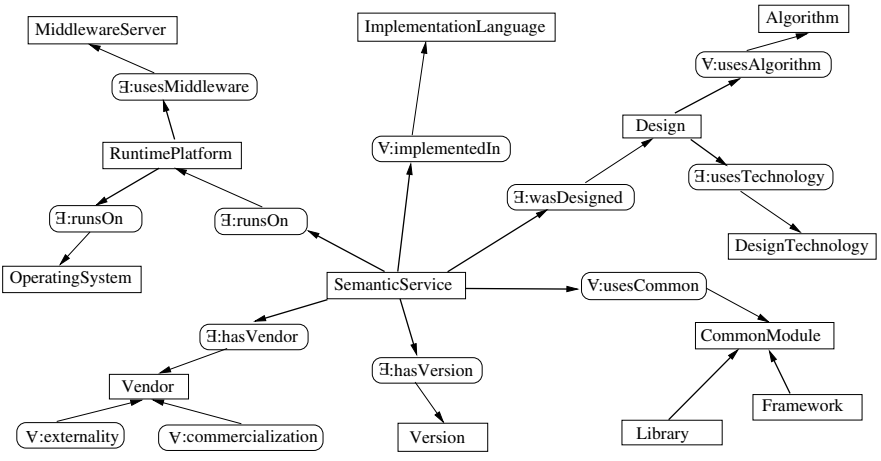


**Fig. 2.** Most important classes of the N-version features ontology

of a *DesignTechnology* such as the Waterfall model or the Spiral model and *Algorithms*. It is implemented in one or more *ImplementationLanguages*. A *CommonModule* and its subclasses describe third party modules that are included in service code as *Libraries* or *Frameworks*. Finally, a *SemanticService* runs on a *RuntimePlat form*.

### 3.2 Service Selection Algorithm

The ontological description is used by the selection algorithm to identify groups of services in which services have relatively different features. Services from one of the groups are selected for an N-version invocation. The algorithm selects services for N-version invocation from available services that match the client's request. The algorithm makes the following assumptions:

- A matching subsystem has already selected services that match the client's request.
- The N-version features ontology describes available services.

Ensuring that services in an N-version invocation differ reduces the risk of co-relating failures specific for the features. Generally, the algorithm is done in the following steps: (i) the ontology is queried for service features, (ii) common features for pairs of services are calculated, (iii) common features are added up for services from potential groups, (iv) services from a group with relatively small number of common features are returned.

The selection is done after basic client-server matching and before the actual invocation of different versions of a service. The input for the selection algorithm is the set of available services that supply a required functionality. The output is the set selected for an N-version invocation.

Algorithm 1 presents the most important selection steps.

**Algorithm 1.** Selection of service versions for N-version invocation.

**input:** *matchingServices* - all services that match the client's request
**input:** *groupSize* - number of versions that are invoked
**output:** services selected for invocation
query the N-version features ontology for information about *matchingServices*
**for all** *service* in *matchingServices* **do**
fetch *serviceFeatures*
**end for**
create $featureMatrix$ in which rows and columns correspond to services from *matchingServices*
**for all** $service_i$, $service_j$ in *matchingServices* **do**
$featureMatrix[i][j]$ = count common features for $service_i$ and $service_j$
**end for**
create *groupsList* containing all subsets of size *groupSize* from *matchingServices*
**for all** *group* in *groupsList* **do**
calculate *similarityMetric*: add up values from $featureMatrix$ for each pair of services from *group*
**end for**
identify *bestGroups*: select groups with the smallest *similarityMetric*
select one $finalGroup$ from *bestGroups*
return services from $finalGroup$

The presented listing simplifies the analysis treating all features as equally important. However, features from the N-version ontology may have different impact on correlation of failures in N-version programming. Although there is no research on such impact known to us, the designed algorithm should distinguish feature importance during service selection. We arbitrarily select features of primary and secondary impact strength. Concepts considered as of primary importance are: $ImplementaionLanguage$, $Vendor$, $MiddlewareServer$, $CommonModule$ and $DesignTechnology$. Other concepts are considered as of secondary importance.

### 3.3 Gathering Data for Service Descriptions

Although the structure of the N-version features ontology is statically defined, it is still necessary to fill in information about known services. The most desired approach is to fetch automatically data about service features from existing sources of information. This can be done in many cases as information is available for automatic processing in different places in Semantic Web infrastructure: WSDL, UDDI, OWL-S. Some features, however, need to be handled manually.

Information about service is available in WSDL definition, UDDI registry or OWL-S files on different abstraction level. Service vendor is described in UDDI definitions in "businessEntity" part of service description with optional detailed information. Service runtime can be fetched by querying service endpoint about middleware platform.

Existing sources do not provide information about service design and implementation. In particular, it will be necessary to handle manually information about development approach, design process and used algorithms. Information about implementation language and used frameworks is not normally available in service description. Additional information is either included directly in the N-version features ontology or in OWL-S descriptions of individual services.

## 4 Prototypical Implementation

The designed solution was verified by a prototypical implementation. The implementation covers a simplified N-version features ontology, service selection algorithm and invocation of semantic services. The simplified ontology contained the *SemanticService* class and the following classes that describe service features: $Vendor$, $RuntimePlatform$ and $ImplementationLanguage$. The implemented algorithm uses information stored in the ontology to calculate $similarityMetric$ for the potential groups of services. It is assumed that service features are of equal importance. Selected services are invoked using their OWL-S and WSLD definitions. Partial results from services are gathered and the final result is voted using simple majority voting. If consensus is achieved, it is returned to the invoker, otherwise an exception is thrown by the N-version invocation module.

We used the following third party libraries in the implementation:

- Protege-OWL - definition of the N-version features ontology.
- Mindswap OWL-S API - Java API for invocation of Semantic Web Services and transformation from WSLD to OWL-S.
- Jena SPARQL - execution of SPARQL queries on the N-version features ontology to fetch information. about services.

The implementation does not cover service matching between client's request and server's offering as it is not within the scope of this paper. The matching phase is realized by a mock matcher with fixed matching between services. Automated creation of the N-version features ontology is not yet implemented in the system.

Code snippets showing SPARQL query and service invocation are shown in Listings 1.1 1.2.

**Listing 1.1.** SPARQL query executed on the prototypical ontology

```
PREFIX ...
SELECT ?service ?runtimePlatform ?vendor ?implLanguage
WHERE {
    ?service rdf:type table:SemanticService.
    ?service table:hasRuntime ?runtimePlatform.
    ?service table:hasVendor ?vendor.
    ?service table:implementedIn ?implLanguage.
} ...
```

**Listing 1.2.** Code snippets for service selection and invocation

```
import com.hp.hpl.jena.query.*;
import org.mindswap.owls.process.*;
...
ProcessExecutionEngine exec;     //from org.mindswap.owls.io
OWLSReader reader;
...
public List selectServices (...) {
        ...
        Query query = QueryFactory.create(queryString);
        QueryExecution qe = QueryExecutionFactory.
                        create(query, model);
        ResultSet results = qe.execSelect();
... }
public String invokeNVariant (...) throws Exception {
        ...
        selectedServices = selectServices (...);
        for (int i = 0; i < selectedServices.size(); i++) {
                ...
                //invoke using Mindswap OWL-S API
                service = reader.read(owlsFile);
                process = service.getProcess();
                exec.execute(process, values);
        ... }
... }
```

### 4.1 Selection Example

As an example of service selection let us consider the following demo configuration. Six services supplying the same functionality differ in their implementation language, runtime platform and service vendor. NVP is configured to invoke triples of services. We select three services from six available ones in such a way that the selected services have possibly different features.

Table 1 shows features of demo services fetched by the SPARQL query.

Table 2 shows the number of common features for pairs of services. Let $ES$ abbreviate $ExemplaryService$. For example, services $ES1$ and $ES2$ have two common features: $RuntimePlatform$ and $Vendor$, while services $ES1$ and $ES3$ have no common features.

**Table 1.** Demo services

| Service id | ImplLanguage | RuntimePlatform | Vendor |
|---|---|---|---|
| ExemplaryService1 | CSharp | DotNet | SemanticDemoCorp |
| ExemplaryService2 | JSharp | DotNet | SemanticDemoCorp |
| ExemplaryService3 | J2EE | Axis2 | FreeSemanticProducts |
| ExemplaryService4 | J2EE | Axis2 | OntologyDemoUniv |
| ExemplaryService5 | J2EE | JBoss | SemanticDemoCorp |
| ExemplaryService6 | J2EE | JBoss | FreeSemanticProducts |

**Table 2.** Number of common features between services

| | $ES1$ | $ES2$ | $ES3$ | $ES4$ | $ES5$ | $ES6$ |
|---|---|---|---|---|---|---|
| $ES1$ | x | 2 | 0 | 0 | 1 | 0 |
| $ES2$ | - | x | 0 | 0 | 1 | 0 |
| $ES3$ | - | - | x | 2 | 1 | 2 |
| $ES4$ | - | - | - | x | 1 | 1 |
| $ES5$ | - | - | - | - | x | 2 |
| $ES6$ | - | - | - | - | - | x |

Then the algorithm calculates *similarityMetric* for groups of services. Assuming that triples are selected, there are 20 potential groups with *similarity Metrics* ranging from 1 to 5. Groups number 9 $\{ES1, ES4, ES6\}$ and 15 $\{ES2, ES4, ES6\}$ have the lowest value of the metric (one). Group 19 $\{ES3, ES5, ES6\}$, for example, has *similarityMetric* of 5. Either group number 9 or 15 is selected and passed on to the invocation and voting procedure.

In this scenario, the Java programming language is a common feature for the majority of invoked services for both 9 and 15 groups. It may happen that an error specific for Java programming will be repeated and will demonstrate in both implementations for some input. Other features differ in invoked services, which gives background to expect that a feature specific fault will not corrupt the N-version invocation. For example, an error in SemanticDemoCorp development process will probably not be repeated in other companies, therefore a fault specific for SemanticDemoCorp will be concealed by other versions. Analogously, if a failure demonstrate in the Axis2 middleware for some configuration, it will be concealed by services running on JBoss and .NET.

### 4.2 Dependability of N-Version Module Itself

The N-version invocation module may be a source of additional errors and a threat for computer system dependability. We propose to use two alternative invocation mechanisms that can be used in case the primary N-version invocation module fails: (i) a secondary, simplified N-version invocation module and (ii) a simple invocation of a service. The secondary module performs a standard N-version invocation, in which randomly chosen services are selected from the set of matching services and invoked. If both the primary and the secondary N-version modules fail, a simple invocation is performed on a service randomly selected from matching services. A possibly simple invocation stub that can detect failures or timeouts from N-version modules is created. The stub invokes either primary module, secondary module or a single service.

## 5 Related Work

Although dependability in distributed systems is a mature research discipline, the works related to N-version programming in service oriented architecture are

quite recent. Looker et al. [8] propose Axis stub for N-version invocations. The solution uses service location and majority-voting scheme. Our work differs in that we use semantic selection of different versions of a web service, additionally, we propose rules for service selection to achieve best dependability results. Santos et al. [9] propose a fault tolerant infrastructure that is based on FTCorba architecture. Similarly to the previous work, the solution does not concern semantic information and does not select services from available ones.

Cardoso [10] proposes semantic integration of Web Services with the use of WSDL-S and JXTA technology. The solution is based on creating Semantic Web Services proxies and peer groups that are used as N-version software. Our solution differs in that we concern service features for service selection. We use the N-version features ontology driven algorithm to select those service instances that should be included in N-Version invocations. Additionally, in our solution, versions of Web Services are unaware of each other as they are discovered and invoked by the invocation module.

Townend et al. [11] propose a replication based solution for grid environments. An "FT-Grid co-ordination service" is used to locate, receive, and vote upon jobs submitted by a client program. Our solution differs in that we use semantic information about services and select service versions depending on their features.

There is lot of work in dependability of SOA systems that is loosely related to the scope of this paper. [1] describes research in software diversity and off-the-shelf components. WS* standards were proposed for WS* dependability such as: WSReliableMessaging, WSReliability, WSSecurity, WSAtomicTransactions and others. The standards concern usually lower layers of software systems. Backward recovery and exception handling is addressed in [12] [13].

## 6 Conclusions and Future Work

The aim of this paper was to propose a technique for selection of service versions for N-version invocations. The solution is aimed at increasing software dependability without increasing the number of invoked services in order to reduce invocation cost. We presented an ontology of service features related to N-version programing and an algorithm for service selection. The designed ontology and algorithm show that services can be selected according to their non-functional features, which reduces the risk of repeating features-specific failures. A prototype implementation shows that this solution can be effectively applied in Semantic Web Services. The obtained results are promising, especially considering the fact that Web Services infrastructure supplies different infrastructures for service development and sharing.

The future work concerns further effort to fully implement the designed solution. Current implementation does not integrate matching module, feature gathering functionality and some classes from the N-version features ontology. It also needs to be updated to OWL-S 1.1 version. Additionally, experiments need to be performed to determine the impact of primary and secondary service

features on service execution. The distinction of impact strength was done heuristically and needs to be verified. Finally, the implemented system should be verified in a real-world application.

**Acknowledgments.** This work was supported by the Polish Ministry of Science and Higher Education under research project No. N519 022 32/2949.

## References

1. ReSIST: Resilience for Survivability in IST, A European Network of Excellence: Resilience-Building Technologies: State of Knowledge (2006)
2. Knight, J.C., Leveson, N.G.: An experimental evaluation of the assumption of independence in multiversion programming. IEEE Transactions on Software Engineering (1986)
3. W3C: OWL-S: Semantic Markup for Web Services (2004)
4. Kiefer, C., Bernstein, A., Tappolet, J.: Evoont - a software evolution ontology. Technical report, Dynamic and Distributed information Systems Group, University of Zrich (2007), `http://www.ifi.uzh.ch/ddis/msr/`
5. Cohen, S.: Ontology and taxonomy of services in a service-oriented architecture. The Architecture Journal, Microsoft Corporation (2007)
6. Gangemi, A., Mika, P., Sabou, M., Oberle, D.: An ontology of services and service descriptions. Technical report, OntoWare.org, Institute AIFB, University of Karlsruhe (2003), `http://cos.ontoware.org/`
7. Baida, Z., Gordijn, J., Akkermans, H.: Service ontology. Technical report, Ontology-Based ELectronic Integration of CompleX Products and Value Chains (2003)
8. Looker, N., Munro, M., Xu, J.: Increasing web service dependability through consensus voting. In: 29th IEEE Annual International Computer Software and Applications Conference (2005)
9. Santos, G., Lung, L.C., Montez, C.: Ftweb: A fault tolerant infrastructure for web services. In: Ninth IEEE International EDOC Enterprise Computing Conference (2005)
10. Cardoso, J.: Semantic integration of web services and peertopeer networks to achieve fault-tolerance. In: IEEE International Conference on Granular Computing (2006)
11. Townend, P., Xu, J.: Replication-based fault tolerance in a grid environment. In: U.K. e-Science 3rd All-Hands Meeting (2004)
12. Xu, J., Romanovsky, A., Randell, B.: Concurrent exception handling and resolution in distributed object systems. IEEE Transactions on Parallel and Distributed Systems (2000)
13. Kaczmarek, P.L., Krawczyk, H.: Remote exception handling for pvm processes. In: Dongarra, J., Laforenza, D., Orlando, S. (eds.) EuroPVM/MPI 2003. LNCS, vol. 2840. Springer, Heidelberg (2003)