# Collaborative Knowledge Acquisition under Control of a Non-Regression Test System

Gérôme Canals [123], Amélie Cordier[45], Emmanuel Desmontils[6], Laura Infante-Blanco[123], and Emmanuel Nauer[123]

[1] Université de Lorraine, LORIA, UMR 7503, 54506 Vandœuvre-lès-Nancy, France,
[2] CNRS – 54506 Vandœuvre-lès-Nancy, France,
[3] Inria — Villers-lès-Nancy, F-54602, France
[4] Université de Lyon, CNRS
[5] Université Lyon 1, LIRIS, UMR 5205, F-69622, Lyon, France
[6] LINA Université de Nantes, 2 rue de la Houssinière BP92208, F-44300 Nantes Cedex 3, France
{Gerome.Canals,Emmanuel.Nauer,Laura.InfanteBlanco}@loria.fr
Amelie.Cordier@liris.cnrs.fr,Emmanuel.Desmontils@univ-nantes.fr

**Abstract.** This paper introduces BeGoood, a generic system for managing non-regression tests on knowledge-bases. BeGoood is a system allowing to define test plans in order to monitor the evolution of knowledge-bases. Any system answering queries by providing results in the form of set of strings can be tested with BeGoood. BeGoood has been developed following a REST architecture and is independent of any application domain. This paper describes the architecture of the system and gives a use case to illustrate how BeGoood is able to manage a collaborative knowledge evolution in the framework of a case-based reasoning system.

## 1 Introduction

In knowledge-based systems, making evolve knowledge sources is a difficult task, for many reasons. One of the difficulties is to make sure that, when adding knowledge, or modifying existing knowledge, one must not jeopardize the results produced by the system. Therefore, one way to assess the impact of a change in a knowledge-base is to evaluate the impact of this change in the results of the system.

In this paper, we describe BeGoood, a generic system for managing non-regression tests on knowledge-bases. The system provides us with all the features usually found in test systems, such as assertions, test plans, test reports, etc. Although generic, BeGoood has been initially developed to test ontologies in distributed systems.

The paper is organized as follows. Section 2 presents the motivation for this work. Section 3 reviews related work on ontology management and test systems. Section 4 describes the BeGoood system, its architecture and its data model. To illustrate how the system works and is able to manage a collaborative knowledge evolution, section 5 details an example of use case. Lastly, section 6 concludes this paper and discusses future work.

## 2 Context and motivations

The motivation to develop this system came from the KOLFLOW project[1]. The aim of this project is to develop distributed web environments enabling humans and artificial agents to collaborate in order to build knowledge used by both humans and artificial agents. Therefore, distributed knowledge bases are at the heart of KOLFLOW. These knowledge bases are continuously modified and, as a consequence, conflicts arise quickly. Therefore, procedures have to be defined in order to monitor and handle conflicts. In a previous work [13], we have proposed an architecture called K-CIP (Knowledge Continuous Integration Process) for managing knowledge bases integration. The corner stone of K-CIP is the notion of test. Indeed, the whole process is controlled by tests, and the transition from one state to the next one is determined by the results of the tests. This paper describes the test framework required for implementing K-CIP.

One of the use case of the KOLFLOW project was to manage cooking knowledge bases used by a Case-Based Reasoning (CBR) system called TAAABLE [5]. With this system, we had to face ontology evolution problems. For example, TAAABLE used return good results when queried for tomato salad recipes because tomatoes were defined in the ontology as vegetables. But if someone changes the ontology to state that tomatoes are fruits, instead of vegetables, TAAABLE then provides strange results when asked for tomato salad recipes again (e.g. fruit salad recipes). We had to find a way to handle this type of problems. This example is detailed in section 5.

Goals and needs of the KOLFLOW project have guided design choices for BEGOOOD, but we have decided to implement it as a generic framework for facilitating its reuse in other contexts. BEGOOOD is generic in two ways: it is domain independent, and it has a REST architecture. The system can test any application that, in response to a query, returns a set of results in the form of strings. Besides, the architecture of BEGOOOD complies with the requirements of REST, which makes the system easy to connect with any web-based application. Hence, BEGOOOD is a suitable tool for ontology engineering and ontology management in distributed environments, such as in KOLFLOW.

## 3 State of the art

Social semantic web [12, 9, 10] opens interesting perspectives for man-machine collaboration [11, 16]. It creates interactive spaces where users and smart agents can collaborate to improve and maintain knowledge exploitable by humans and machines. Humans can build resources assisted with smart agents that gather information from linked data [4]. The engineering of ontologies is a central concern in knowledge engineering in social semantic web domain. Many tools have been proposed to improve development of ontologies. Most of these tools (such

[1] `http://kolflow.univ-nantes.fr`

as Protégé[2]) are designed for centralized usage without social collaboration process. However, some collaborative approaches have been proposed: Co$_4$ [6], DILIGENT [15], OntoWiki [1], etc. These expert team based tools, often centralized, are not convenient in the social web context, with an unstructured network and without an identified expert team. In this context, quality issues turn out to be of critical importance [3]. Beyond the "logical quality", [3] highlights the importance of usefulness of knowledge. For Baumeister and Nalepa, this criteria is difficult to measure and social methods can help to cope this problem. To achieve this aim, collecting user satisfaction by collecting its feedback seems to be a good solution. The goal of the K-CIP approach [13] is to provide such features focusing on ontology engineering in a social and distributed context. BeGoood, presented in this paper, aim to implement the knowledge unit tests part of K-CIP.

For that, we draw inspiration from continuous integration and agile programming, using software engineering methods. Continuous Integration process [8] is an agile software engineering process aiming at shortening the integration of software. The principle is to allow the developer to make many small updates (and this is even recommended). Each update is subject to a set of unit tests then this update is integrated into the whole software. Next, one or more phase(s) of integration tests (according to their time cost) is/are processed to ensure the consistent development of the software. A notification system allows the developer to know the state of integration. In case of failure, he/she is quickly notified so that corrections can be quickly made.

Such a process can be applied to knowledge building in collaborative and distributive contexts. In ontology engineering, unit tests can be applied at different levels [14] :

1. *affirming derived knowledge* defining a positive test ontology containing axioms that must be derivable by the ontology and a negative test ontology with axioms that must not be derivable,
2. *formalized competency questions* always formalized in a query language and associated with a set of expected answers that the system uses to check if the ontology meets the requirements stated,
3. *expressive consistency checks* with a secondary ontology which, included to the usable one, allows knowledge managers to perform consistency checks,
4. *auto-epistemic operators to guarantee the completeness* useful to check an ontology not only with regards to its consistent usage, but also with regards to some explicitly defined understanding of completeness,
5. *constraints to control the usage of resources* with the use of domain and range not to classify the subject or the object but to implement these constraints.

We propose BeGoood as a generic Web service solution to the second level useful not for the maintenance of the system, but rather for its initial build [14]. Note that our web service can also be used to implement the first level. In fact,

[2] The Protégé Ontology Editor and Knowledge Acquisition System: `http://protege.stanford.edu/`

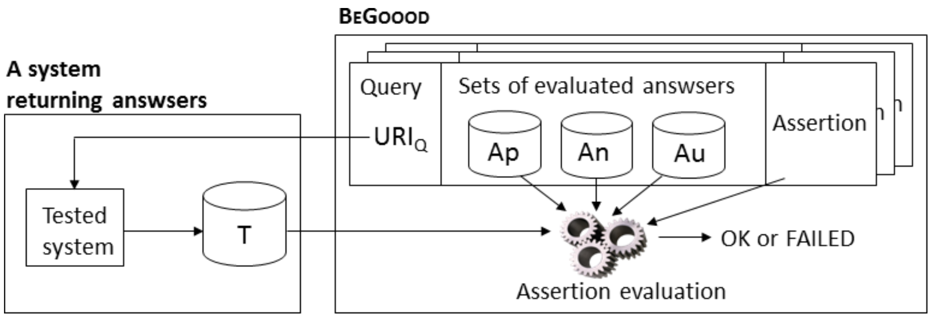

**Fig. 1.** Link between BeGoood and the system that has to be tested.

some positive and negative axioms can be expressed by queries (like with the "ask" sentence in SPARQL) or by positive and negative answers of a test. Protégé Ontology Editor Plug-In called OWL Unit Test Framework[3] is also a generic software for knowledge based unit tests, but is dedicated to the first level. From a technical point of view, our approach is largely inspired by xUnit test frameworks for software development, like JUnit.[4]. In particular, BeGoood assertions and the BeGoood assertion evaluation engine are very close to assertions that can be expressed and evaluated with the DBUnit framework.[5] The main difference with these frameworks is that we do not have test fixtures, since our purpose is to test the behavior of a system *and* its current state. Setting up a well defined state of the system for testing it is thus just not useful.

In the next section, we presents the BeGoood generic web service.

## 4 BeGoood: a generic system for managing non-regression tests

BeGoood addresses the evaluation of various kind of systems, including case-based reasoning systems (e.g. Taaable), information retrieval systems (like it is done in the TREC[6] evaluations), etc. The idea is to evaluate the *quality* of a system, according to the set of answers $A$ it returns for a query $Q$, and the evaluation consists in comparing $A$ with the set of answers that are known to be relevant for $Q$, denoted by $Ap$ (for positive answers), the set of answers that are known to be non relevant for $Q$, denoted by $An$ (for negative answers), and the set of answers that have already be returned for $Q$, without knowing if they are relevant or not, denoted by $Au$ (for undefined answers).

[3] `http://www.co-ode.org/downloads/owlunittest/`
[4] `http://junit.org/`
[5] `http://www.dbunit.org/`
[6] `http://trec.nist.gov`

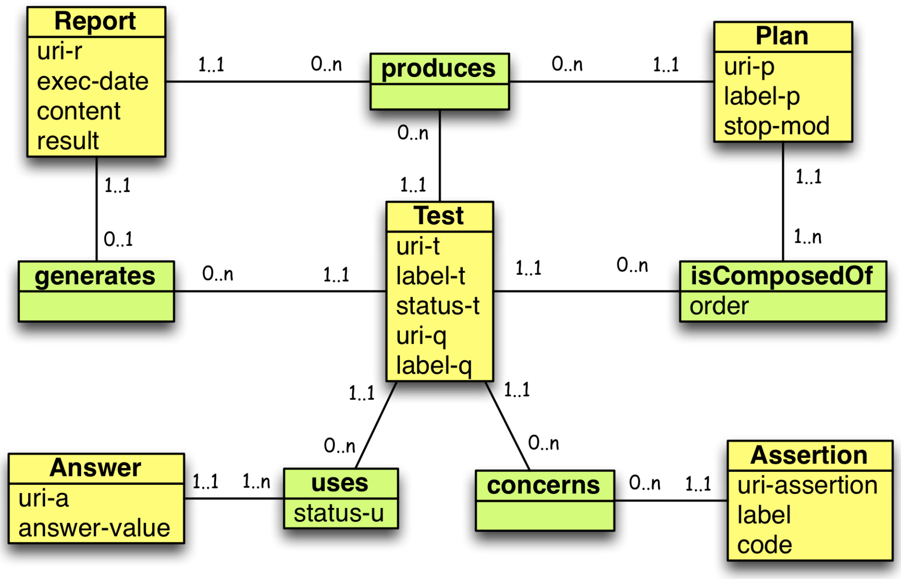

**Fig. 2.** Data Model.

Fig. 1 presents how BeGoood can be plugged to a system that has to be tested. BeGoood stores a set of assertions. An assertion is linked to a query $Q$, than can be run again thanks to its URI $uri{-}q$ and to the $Ap$, $An$, and $Au$ sets. When the system must be tested, assertions are evaluated thanks to $Ap$, $An$, $Au$, and $T$, the answers resulting from a new execution of $Q$.

## 4.1 Non-regression test model

The objective of a test is to prove that the answers to a query satisfy a set of assertions. In order to evaluate the behavior of a system, a test plan, composed of a sets of tests, is executed and a report is produced. Fig. 2 introduces the data model used in BeGoood for managing such a testing system.

*Test.* A test is identified by an URI ($uri{-}t$). A label describes its goal and its status indicates if it is active or not. A test is associated to a query, identified by an URI ($uri - q$), which allows to run the query again, and a label gives a short textual explanation about the semantic of the query.

*Answer.* An answer is identified by an URI ($uri{-}a$). As BeGoood wants to be a generic testing system, an answer ($answer{-}value$) is considered to be a string. Answers and tests are linked in order to get the $Ap$, $An$, and $Au$ sets. An answer whose status ($status{-}u$) is $p$ (resp. $n$ and $u$) belongs to $Ap$ (resp. $An$ and $Au$).

*Assertions.* Assertions are logical expressions, and we choose to encode them in XML. An assertion is identified by an URI ($uri - assertion$) which allows to use it into different tests, a label which is a short textual explanation about the semantic of the assertion and an XML code. Figure 3 describes the DTD schema of assertions in XML. An assertion handles two types of sets:

– the sets of answers resulting from the construction of the test: $Ap$, $An$, $Au$, and $A = Ap \cup An \cup Au$, and
– the sets of answers obtained during the application of the test: $T$ is the set of all returned answers, $Tp$ denotes the subset of positive answers of $T$ ($Tp = T \cap Ap$), $Tn$ denotes the subset of negative answers of $T$ ($Tn = T \cap An$), $Tu$ denotes the subset of undefined answers of $T$ ($Tu = T \setminus Tp \cup Tn$).

An assertion is a logical expression which handles sets ($Ap$, $An$, $Au$, $A$, $T$, $Tp$, $Tn$, $Tu$ and $\emptyset$), set operators (union, intersect, equal, card, etc.), numbers (for cardinality constraints) and standard number operators.

Involving only the sets mentioned above, assertions are totally independent of the application and of the type of answers. Assertions are used both to determine a property of a test and serves to evaluate this property. In the first case, only the sets $Ap$, $An$, $Au$ and $A$ will be used. To evaluate a test, an assertion have to be composed of sets from the test ($Ap$, $An$, $Au$ and $A$) and from the query results ($T$, $Tp$, $Tn$ and $Tu$). For instance, a test called "a majority of positive answers" can be defined by $|Tp| > 2.|Tn|$ (see Fig. 4). Similarly, a successful test like "all positive known answers" can be defined by the statement $Ap \cap T = Ap$ (see Fig. 5).

*Plan.* A plan is identified by an URI ($uri{-}p$). A label describes its objective. A plan is composed of a set of tests which can be ordered. A stop condition ($stop{-}mod$) can be defined for a plan. With $stop{-}mod{=}true$, the plan will be stopped once the first test fails, without evaluating the other tests belonging to the plan.

*Report.* A report is generated after a test was run. The report is identified by an URI ($uri{-}r$), the date and time of its execution, and the results of the test(s). A report can be generated by a single test execution or by a plan execution. In this case, the result is built by concatenation of the reports of the tests composing the plan. Section 5.2 shows examples of report.

### 4.2 BeGoood advantages

One of the strengths of BeGoood is its independence from the application domain, the knowledge representation method and even the method used to manage knowledge. Indeed, a test is nothing more than a string submitted to the standard REST[7] client application. In addition, this one returns a set of answers as strings.

The only constraint to the response format is that it has to be canonical, i.e. there must be correspondence between the string that represents the answer

```
<!ENTITY % logic-assert "(and | or | not)" >
<!ENTITY % num-comp "(eq | lt | gt | ge | le | neq)" >
<!ENTITY % set "(Ap | An | Au | A | Tp | Tn | Tu | T | Empty-set)" >
<!ENTITY % set-comp "(eq-set | neq-set " >
<!ENTITY % set-cons "(union | intersect | minus | %set;)" >
<!ENTITY % num-op "(card | plus | dif | times | divide | number)" >
<!-- The root  -->
<!ELEMENT assertion (%logic-assert;|%num-comp;|%set-comp;) >
<!-- Logic operators -->
<!ELEMENT and ((%logic-assert;|%num-comp;|%set-comp;),
              (%logic-assert;|%num-comp;|%set-comp;)) >
<!ELEMENT or ((%logic-assert;|%num-comp;|%set-comp;),
             (%logic-assert;|%num-comp;|%set-comp;)) >
<!ELEMENT not (%logic-assert;|%num-comp;|%set-comp;) >
<!-- Set operators -->
<!ELEMENT union (%set-cons;, %set-cons;) >
<!ELEMENT intersect (%set-cons;, %set-cons;) >
<!ELEMENT minus (%set-cons;, %set-cons;) >
<!ELEMENT card (%set-cons;) >
<!-- Set comparison -->
<!ELEMENT eq-set (%set-cons;, %set-cons;) >
<!ELEMENT neq-set (%set-cons;, %set-cons;) >
<!-- Numeric comparison -->
<!ELEMENT eq (%num-op;, %num-op;) >
<!ELEMENT neq (%num-op;, %num-op;) >
<!ELEMENT lt (%num-op;, %num-op;) >
<!ELEMENT gt (%num-op;, %num-op;) >
<!ELEMENT ge (%num-op;, %num-op;) >
<!ELEMENT le (%num-op;, %num-op;) >
<!-- Numeric operators -->
<!ELEMENT plus (%num-op;, %num-op;) >
<!ELEMENT dif (%num-op;, %num-op;) >
<!ELEMENT times (%num-op;, %num-op;) >
<!ELEMENT divide (%num-op;, %num-op;) >
<!-- Numeric value -->
<!ELEMENT number (#PCDATA) >
  <!-- Set values -->
<!ELEMENT Ap EMPTY >
<!ELEMENT An EMPTY >
<!ELEMENT Au EMPTY >
<!ELEMENT A EMPTY >
<!ELEMENT Tp EMPTY >
<!ELEMENT Tn EMPTY >
<!ELEMENT Tu EMPTY >
<!ELEMENT T EMPTY >
<!ELEMENT Empty-set EMPTY >
```

**Fig. 3.** assertion-en.dtd, the DTD schema to encode assertions in XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE assertion SYSTEM "assertion-en.dtd">
<assertion>
    <gt><card><Tp/></card>
        <times><number>2<number><card><Tn/></card></times>
    </gt>
</assertion>
```

**Fig. 4.** XML encoding of the assertion "a majority of positive answers", i.e. $|Tp| > 2.|Tn|$.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE assertion SYSTEM "assertion-en.dtd">
<assertion>
    <eq><inter><Ap/><T/></inter>
        <T/>
    </eq>
</assertion>
```

**Fig. 5.** XML encoding of the assertion "all positive known answers", i.e. $Ap \cap T = Ap$.

and the answer itself and that, whatever the context, this string is unique. This ensures the ability to compare answers using a single string comparison.

Another genericity factor of our system is the generic construction of assertions. Indeed, they are described with an XML structure whose elements do not depend on the application (Fig. 3).

To use BeGoood, the tested system must only provide an URI access for being requested. All tests, assertions and answers are identified by an URI. Therefore, the same test can be used by several parallel applications. BeGoood is ready to manage ontology's engineering in a distributed environment.

### 4.3 Implementation

The BeGoood test application is implemented as a web service and is available under a AGPL license on github[7]. The different functions are available through a RESTful API. The general architecture of BeGoood is depicted in Fig. 6. The main components of the application are:

**the test database** that stores every test artifacts : tests and test plans, associated queries, assertions, expected result sets and test reports. Every database artifact is a REST resource that can be managed through the API using the classical CRUD (Create, Retrieve, Update, Delete) operations.

[7] `https://github.com/kolflow/begoood`

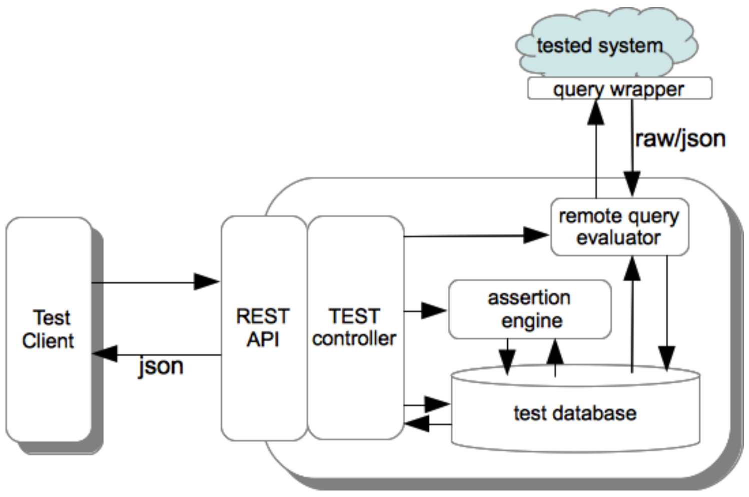

**Fig. 6.** Architecture of BeGoood.

**the remote query evaluator** is in charge of evaluating test queries on the tested system. Queries are seen by the test application as simple URIs. Thus the evaluation of a remote query simply consists in submitting a GET request on the URI. This means that the tested system is assumed to provide a web interface for query evaluation. If it is not the case, a query wrapper must be implemented. The remote query evaluator then collects the query result returned by the tested system and makes it available to the assertion engine.

**the assertion engine** is in charge of evaluating assertions over the expected and effective query result sets. It does so by querying the local database and evaluating basic logical set operators.

**the REST API** offers the test functionalities as web services. Every resource in the database can be created/retrieved/updated and delete through this API. A particular resource, associated to every test and test plan, is used to create and execute tests runs. Runs are orchestrated by the **TEST controller**. Each test in a plan is executed according to the plan definition. A test plan report is generated and made available as a resource through the API.

**the test client** is a simple programming interface to build test clients. It provides a more convenient way of interacting with the BeGoood web service rather than directly writing REST request and decoding JSON data in the REST response.

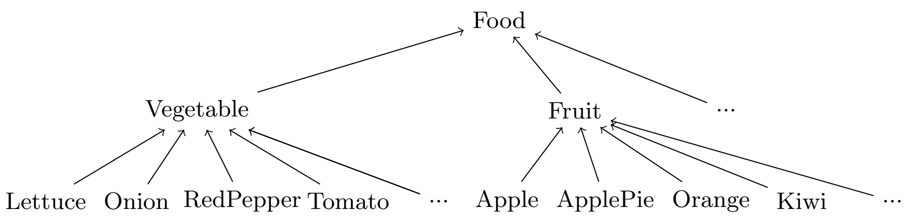

**Fig. 7.** An excerpt of the domain knowledge.

| *Id* | *Title* | $idx(R_i)$ |
|---|---|---|
| $R_1$ | Rice salad | $SaladDish \wedge RedPepper \wedge Shrimp \wedge Rice \wedge Mayo$ |
| $R_2$ | Summer salad | $SaladDish \wedge Lettuce \wedge Onion \wedge Ham \wedge OliveOil \wedge Vinegar$ |
| $R_3$ | Special one | $SaladDish \wedge Onion \wedge RedPepper \wedge SaladSauce$ |
| $R_4$ | Fruit salad | $SaladDish \wedge ApplePie \wedge Kiwi \wedge Orange$ |
| $R_5$ | Fresh salad | $SaladDish \wedge ApplePie \wedge Apple$ |

**Table 1.** Five examples of recipes and their indexes.

## 5 Use case: adapting cooking recipes

This section presents a use case, in the framework of the TAAABLE[8] application, a CBR cooking system. The TAAABLE principles are first presented. Then, an illustrative example is given for showing how BEGOOOD is able to manage a knowledge modification to guarantee the correct behaviour of TAAABLE.

### 5.1 TAAABLE principles

TAAABLE is a CBR system which retrieves and creates cooking recipes by adaptation [2]; it was developed to participate in the Computer Cooking Contest.[9] TAAABLE uses a set of knowledge which is encoded in a Semantic Wiki, called WIKITAAABLE [5]. WIKITAAABLE contains an ontology of the cooking domain which is used to search the source cases that are the most similar to a target case (i.e. the query). This ontology includes several hierarchies (about food, dish types, etc.). An excerpt of the food hierarchy is illustrated in Fig. 7.

The case base is composed of recipes. Each recipe $R$ is represented by $idx(R)$, the index of the recipe $R$ which is a conjunction of literals from the domain ontology. Five recipes are given in the example in Table 1. $idx(R_1)$ is an abstract formal representation of the recipe $R_1$, a salad dish whose ingredients are red pepper, shrimp, rice and mayo (and nothing else).

[8] `http://taaable.fr/`

[9] `http://computercookingcontest.net`

***Retrieval and adaptation process.*** According to a query entered in the system by a user, and also represented by a conjunction of concepts, the system searches in the case base for some cases (recipes) satisfying the query. For example, $Q = SaladDish \wedge Tomato$ represents the query "I would like a recipe of salad dish with tomato." Recipes matching exactly the query, if they exist, are returned to the user; if not, the system searches for similar recipes, using the hierarchies to generalize the user query. Adaptation consists in substituting some ingredients of the source cases by the ones required by the user, and is encoded by a substitution `R|A->B`, meaning that "in recipe $R$, $A$ has to be substituted by $B$". For this example, the results provided by TAAABLE for $Q$ are:

```
R1|RedPepper->Tomato
R2|Lettuce->Tomato or Onion->Tomato
R3|Onion->Tomato or RedPepper->Tomato
```

## 5.2 Using BeGoood to test the Taaable system

Suppose that a user modifies the food hierarchy by removing that tomato is a vegetable and by adding that a tomato is a fruit. In this case, the answers returned by TAAABLE, will be:

```
R4|ApplePie->Tomato or Kiwi->Tomato or Orange->Tomato
R5|ApplePie->Tomato or Apple->Tomato
```

***Test base.*** For $Q$, let the sets of known answers be:

- $Ap = \{$ `A1 = "R1|RedPepper->Tomato"`,
  `A3 = "R3|Onion->Tomato or RedPepper->Tomato"`$\}$
- $An = \emptyset$
- $Au = \{$ `A2 = "R2|Lettuce->Tomato or Onion->Tomato"`$\}$

***Assertions.*** Let the assertions be those previously presented:

- "a majority of positive answers" (see Fig. 4);
- "all positive known answers" (see Fig. 5).

***Examples of results.*** Given the results collected when $Tomato$ is a subclass of $Vegetable$, both assertions succeed. Assertion evaluation returns the following information:

- id-p: `/plans/1` *(tests plan for tomato correctness)*
- id-t: `/tests/1` *(query all salads with tomato)*
- exec-date: `11-03-2013`
- content:

```
{"Ap":["/answers/A1",  "/answers/A3"],
 "An":[]
 "Au":["/answers/A2],
 "T":["R1|RedPepper->Tomato",
      "R2|Lettuce->Tomato or Onion->Tomato",
      "R3|Onion->Tomato or RedPepper->Tomato"],
 "assertions":["/assertions/1","/assertions/2"]
 "results":[true,true]}
```

– global-result: `true` *(all assertions are proved true)*

When *Tomato* becomes a subclass of *Fruit*, both assertions fail. Assertion evaluation returns the following information:

– id-p: `/plans/1` *(tests plan for tomato correctness)*
– id-t: `/tests/1` *(query all salads with tomato)*
– exec-date: `13-03-2013`
– content:

```
{"Ap":["/answers/A1",  "/answers/A3"],
 "An":[]
 "Au":["/answers/A2],
 "T":["R4|ApplePie->Tomato or Kiwi->Tomato or Orange->Tomato",
      "R5|ApplePie->Tomato or Apple->Tomato"],
 "assertions":["/assertions/1","/assertions/2"]
 "results":[false,false]}
```

– global-result: `false` *(almost one assertion is proved false)*

In conclusion, whether a critical modification in the knowledge base like moving *Tomato* from *Vegetable* to *Fruit* produces a failure, the knowledge modification will be rejected.

## 6 Conclusion and ongoing work

In this article, we introduced BeGoood, a generic system for the management of non-regression tests. Although generic BeGoood was originally designed to facilitate modifications of distributed ontologies BeGoood has two valuable properties. Firstly, it is domain independent. Secondly, because of its RESTFul architecture, it is easy to connect BeGoood with any web application providing canonical results in response to a query The system is fully functional.

Future work will focus on the deployment of the system in the continuous knowledge integration process (K-CIP), implemented within Kolflow. Among future work, we also plan to build complete sets of tests to evaluate Taaable and, more generally, to build a gold standard with which we could evaluate and compare culinary applications such as the ones entering the Computer Cooking Contest.

## 7 Acknowledgements

This work is supported by the French National Research agency (ANR) through the KOLFLOW project (code: ANR-10-CONTINT-025), part of the CONTINT research program. We would also like to thank Juliette Dompé, a student from Université de Lorraine, who helped us to develop the first prototype.

## References

1. S. Auer, S. Dietzold, J. Lehmann, and T. Riechert. OntoWiki: A tool for social, semantic collaboration. In N. F. Noy, H. Alani, G. Stumme, P. Mika, Y. Sure, and D. Vrandecic, editors, *Proceedings of the Workshop on Social and Collaborative Construction of Structured Knowledge (CKC 2007) at the 16th International World Wide Web Conference (WWW2007) Banff, Canada, May 8, 2007*, volume 273 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.
2. F. Badra, R. Bendaoud, R. Bentebitel, P.-A. Champin, J. Cojan, A. Cordier, S. Després, S. Jean-Daubias, J. Lieber, T. Meilender, A. Mille, E. Nauer, A. Napoli, and Y. Toussaint. Taaable: Text Mining, Ontology Engineering, and Hierarchical Classification for Textual Case-Based Cooking. In *ECCBR Workshops, Workshop of the First Computer Cooking Contest*, pages 219–228, 2008.
3. J. Baumeister and G. J. Nalepa. Verification of distributed knowledge in semantic knowledge wikis. In *FLAIRS'09: Proceedings of the 22th International Florida Artificial Intelligence Research Society Conference*, pages 384–389. AAAI Press, 2009.
4. T. Berners-Lee. Linked data-the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.
5. A. Cordier, J. Lieber, P. Molli, E. Nauer, H. Skaf-Molli, and Y. Toussaint. Wikitaaable: A semantic wiki as a blackboard for a textual case-based reasoning system. In *SemWiki 2009 – 4th Semantic Wiki Workshop*, Heraklion, Greece, May 2009.
6. J. Euzenat. Corporate memory through cooperative creation of knowledge bases and hyper-documents. In *Proc. 10th workshop on knowledge acquisition (KAW), Banff (CA)*, pages (36)1–18, 1996.
7. R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Phd dissertation, University of California at Irvine, 2000.
8. M. Fowler and M. Foemmel. Continuous integration, http://martinfowler.com/articles/continuousIntegration.html, 2005.
9. T. Gruber. Collective knowledge systems: Where the social web meets the semantic web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(1):4 – 13, 2008. Semantic Web and Web 2.0.
10. J. Hendler and T. Berners-Lee. From the semantic web to social machines: A research challenge for ai on the world wide web. *Artificial Intelligence*, 174(2):156 – 161, 2010.
11. R. Hoffmann, S. Amershi, K. Patel, F. Wu, J. Fogarty, and D. S. Weld. Amplifying community content creation with mixed initiative information extraction. In *Proceedings of the 27th international conference on Human factors in computing systems*, CHI '09, pages 1849–1858, New York, NY, USA, 2009. ACM.
12. P. Mika. Ontologies are us: A unified model of social networks and semantics. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(1):5 – 15, 2007.

13. H. Skaf-Molli, E. Desmontils, E. Nauer, G. Canals, A. Cordier, and M. Lefevre. Knowledge Continuous Integration Process (K-CIP). In *WWW 2012 - SWCS'12 Workshop - 21st World Wide Web Conference - Semantic Web Collaborative Spaces workshop*, pages 1075–1082, Lyon, France, Apr. 2012.
14. D. Vrandečić and A. Gangemi. Unit tests for ontologies. In *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, pages 1012–1020. Springer, 2006.
15. D. Vrandečić, S. Pinto, C. Tempich, and Y. Sure. The diligent knowledge processes. *Journal of Knowledge Management*, 9(5):85–96, 2005.
16. F. Wu and D. S. Weld. Autonomously semantifying wikipedia. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, pages 41–50, New York, NY, USA, 2007. ACM.