Temporal Logic To Query Semantic Graphs Using The Model Checking Method

Mahdi Gueffaz, Sylvain Rampacek, Christophe Nicolle
LE2I, UMR CNRS 5158
University of Bourgogne
BP 47870, 21078 Dijon Cedex, France
{Mahdi.Gueffaz, Sylvain.Rampacek, CNicolle}@u-bourgogne.fr

***Abstract*— Semantic interoperability problems have found their solutions due to the use of languages and techniques from the Semantic Web. The proliferations of ontologies and meta-information have improved the understanding of information and the relevance of search engine responses. However, the construction of semantic graphs is a source of numerous errors of interpretation or modeling, and scalability remains a major problem. The processing of large semantic graphs is a limit to the use of semantics in current information systems. The work presented in this paper is part of a new research at the border of two areas: the semantic web and the model checking. This line of research concerns the adaptation of model checking techniques to semantic graphs. We present a first method of converting RDF (Resource Description Framework) graphs into NμSMV and PROMELA (Process Meta Language) languages in order to be checked with the temporal logic property and queried by the temporal logic query. SPARQL (Simple Protocol and RDF query Language) query language is the standard for querying the Semantic Web, but it has a lot of limitations. Our primary goal with the temporal logic query is to overcome this limitation of the SPARQL query language. To reach this goal, three tools have been developed. The first two tools "RDF2SPIN" and "RDF2NμSMV" are used to transform the Semantic graph into a model written in PROMELA and respectively in NμSMV languages – in order to be understood by the SPIN and respectively the NμSMV model checkers. The STL Resolver tool is used to find solutions to the temporal logic query. It is based on the model checking algorithms.**

***Index Terms*— Semantic graph, model checking, temporal logic, temporal logic query, SPARQL.**

I. INTRODUCTION

W3C (World Wide Web Consortium) aims to standardize the representation and the exchange of information on the WEB. This objective should be able to make the information understandable for both automated processes and users. The homogenization of computer exchanges took place due to the introduction of the XML (eXtensible Markup Language) [1] standard. This standard has enabled the program to manipulate information through languages with hierarchical structure mark-up defined by grammars that are derived from the XML standard. However, this effort has not been able to improve the user's understanding of information. Thus, new standards have been developed to enable the semantic representation of information in the form of XML-derived languages. This base is called Semantic Web standards and is usually represented as a stack of languages ranging from automatic processes oriented languages to languages representing more abstract concepts of formal semantics [2]. These languages are used to represent the semantics associated with information, whatever their form or structure. To allow the construction of a semantic graph, many tools have been developed, such as Annotea [3], which is a project of the W3C that specifies the infrastructure for the annotation of Web documents. RDF represents the main format used in the annotation and the types of documents that can be annotated are HTML (Hypertext Markup Language) or XML based documents. However, none provides the functionality to verify the consistency of semantics or to reduce error annotations.

This paper proposes a new way to check these semantic graphs by using the model checking technique in order to reduce errors in annotation, for example, and make the data more relevant. The model checking is an automatic verification technique which has been applied to many cases in industry, in the Netherlands for instance; [4] has revealed several serious flaws in the design of the control system of a barrier protection in the main port of Rotterdam against floods. The large "Intel" manufacturing company processor has used the model checking to detect the bug in its Pentium II processor that caused a loss of 475 million dollars damage to the reputation of Intel. Finally, the model checking succeeded in finding an error in the handling baggage system at the Denver airport (USA), which delayed opening its doors for nine months and caused a loss of 1.1 million dollars per day.

The model checking is a powerful tool for the system verification because it can reveal errors that were not discovered by other formal methods, such as testing or simulation. The model checking uses the temporal logic to describe the properties checking the system model. As we have seen in the examples above, the model checking can handle complex problems with large amounts of information, stored as a graph, in order to verify critical systems. In comparison, in the semantic web, the use of graphs is pervasive and serious problems of scalability appear [5]. Thus, it is appropriate to use the algorithms developed for the model checking in the field of the Semantic Web.

Pnueli pioneered the use of temporal logic as formal language for reasoning about reactive systems [6]. The temporal logic allows the model checking to represent the property that needs to be checked. One limitation of the

doi:10.4304/jsw.7.7.1462-1472

model checking is that it gives simple "yes or no" answers: either the system under study satisfies the temporal formula, or it does not. [7] presents the first research work on the use of the temporal logic queries as an extension of the model checking, in order to help the user understand the system behaviors, because of the fact that the use of the model checking has not been emphasized enough in the literature yet. A temporal query is a temporal formula where the special symbol "?" occurs as a placeholder. In [7], a temporal logic query can have at most one placeholder, but in our research work, a query can have at least one placeholder.

In this paper, we introduce a new method to qualify and query a semantic graph by using the formal method and especially the model checking. We developed tools to convert semantic graphs into a model understood by the model checker. We introduce a new language based on the operators of the temporal logic to query this model by using the model checker algorithms. We have implemented a prototype STL Resolver query engine.

A survey of popular RDF query languages conducted by the W3C identified more than 20 languages that are either under development or have been implemented [8]. Some in the lines of traditional database query languages (e.g. SQL (Structured Query Language), OQL (Object Query Language)), others based on logic and rule languages. Some of them are: RQL (RDF Query Language) [9] is a typed language for querying RDF repositories; SquishQL (Simple RDF Query Language) is a SQL-style query language that permits simple graph navigation in RDF sources; RDQL (RDF Data Query Language) [10] is an implementation of SquishQL; RDFQL (RDF Query Language) is a statement-based query language with a SQL-style to perform queries, inference operations, and construction of views on RDF structured data; TRIPLE [11] is a language that allows rule definition, inference and transformation of RDF models; Notation 3 (N3) [12] provides a text-based syntax for RDF; VERSA is a graph-based language with some support for rules; SeRQL (Sesame RDF Query Language) combines characteristics of languages like RQL, RDQL, N-Triple, N3 plus some new features; RXPath (Regular XPath) is a query language based on XPath (XML Path Language) [13, 14]. The W3C SPARQL [15] is an RDF query language designed to meet such requirements and design objectives mentioned previously. It defines a query language with a SQL-like style, where a simple query is based on query patterns, and query processing consists of binding of variables to generate pattern solutions (graph pattern matching). SPARQL is still a work in progress.

Our research primary goal is to define a powerful and expressive query language for semantic graphs. The other rather competing goal is to keep the query language simple enough so that it could be easily built and understood.

The rest of this paper is organized as follows. In Section 2 we present an overview of the semantic graphs, especially the structure of the RDF graphs and the SPARQL query language, the model checking, the temporal logic and the temporal logic queries. Then, Section 3 presents the related work of our approach. Section 4 refers to the mapping of the semantic graphs into models, and the ScaleSem approach. Section 5 presents our tool that solves the query checking using a semantic graph model. Finally, we end with a benchmark followed by a conclusion.

II. BACKGROUND

A. Semantic web

The semantic Web aims at organizing and structuring the huge quantity of information present on the Net. It consists of a semi-structured language based on XML. Figure 1 shows one of the versions of the organization in layers suggested by the W3C. Each layer is built upon the layers below it. Thus, the whole set of layers uses the XML syntax. This allows taking advantage of all technologies developed around XML: XML Schema, tools for exploiting XML resources (JAVA libraries, etc.), XML databases. XML stems from the SGML (Standard Generalized Markup Language) language, but contrary to HTML, the structure and the presentation of XML documents are conceptually separated. XML is a language which uses tags as a universal representation format of the data. An XML document contains at the same time the data and the indications about the role that these data play. As a result, the same contents can be presented in various forms according to the role of the user in the AEC (Architecture Engineering Construction) project. XML is the keystone of information exchanges on the Web. Unfortunately, XML is insufficient to describe all the semantics required in the Web.

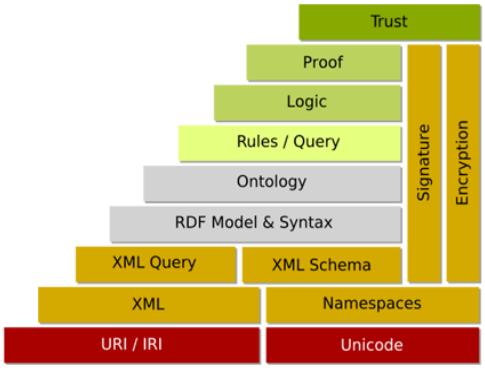

Figure 1. Stack of languages of the Semantic Web. [2]

This paper is based on the RDF layer of the Semantic Web. RDF is a language developed by the W3C to bring a semantic layer to the Web [16]. It allows the connection of Web resources using directed labeled edges. The structure of RDF documents is a complex labeled directed graph. An RDF document is a set of triples <subject, predicate, object>. In addition, the predicate (also called property) connects the subject (resource) to the object (value). Thus, the subject and the object are nodes of the graph connected by an edge directed from the subject towards the object. The nodes and the edges belong to "resource" types. A resource is identified by a Uniform Resource Identifier [17].

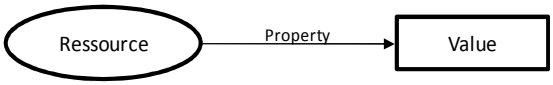

Figure 2. RDF triple.

The declarations can also be represented as a graph, the nodes as resources and values, and the arcs as properties. The resources are represented in the graph by circles; the properties are represented by directed arcs and the values by a box (a rectangle), see Figure 2. Values can become resources if they are described by additional properties. For example, when a value is a resource in another triplet, the value is represented by a circle [18].

The RDF graphs considered here are represented as XML verbose files, in which the information is not stored hierarchically (so-called graph point of view). These RDF graphs are not necessarily connected, meaning they may have no root vertex from which all the other vertices are reachable. To handle the RDF graphs, several designs and implementations of RDF query languages have been proposed. In 2004, the RDF Data Access Working Group, part of the W3C Semantic Web Activity, released a first public working draft of a query language for RDF, called SPARQL [15]. Since then, SPARQL has been rapidly adopted as the standard for querying the Semantic Web data. In January 2008, SPARQL became a W3C Recommendation. SPARQL queries [19] are pattern matching queries on triples that constitute an RDF data graph. The official SPARQL query introduces four different query forms:

- **SELECT** query, which returns the value of the variable, which may be bound by a matching query pattern;
- **ASK** query, which returns true if a given query matches and false if not;
- **CONSTRUCT** query, which returns an RDF graph by substituting the values in given templates;
- **DESCRIBE** query, which returns an RDF graph that defines the matching resource.

B. Model checking and temporal logic Overview

Formal methods [4] offer great potential for an early inclusion of verification in the design process, providing technical audit more efficiently and reduce the verification time. Formal methods are highly recommended techniques for the software development. They have led to the development of some very promising verification techniques that facilitate early detection of defects. Two types of formal verification methods can be distinguished: methods based on the proof of the theorem and the methods based on models.

Methods based on the proof of the theorem verify the correctness of systems by properties in a mathematical theory. These properties are proven with the highest possible precision using tools such as theorem provers and proof checkers. Theorems proofs are also called proof assistants.

Methods based on models describe the possible system behavior in a mathematical precise and unambiguous manner. The system models are accompanied by algorithms that systematically explore all states of the system model. This provides the basis for a whole range of verification techniques ranging from an exhaustive exploration "Model checking" to experiments with a restrictive set of scenario in the model "Simulation." Simulation allows the user to study the system behavior. It is less suited to detect errors because it is difficult to generate all possible scenarios of the system and to simulate them all. Model checker is a verification technique that explores all possible system states. In this way, it can be shown that a given system model truly satisfies a certain property.

The model checker examines all relevant system states in order to check whether they satisfy the desired property. The model checker gives a counter example that indicates how the model can violate the property. With the help of a simulator, the user can locate the error and adapt the model or the property to prevent the violation of property, as shown in Figure 3.

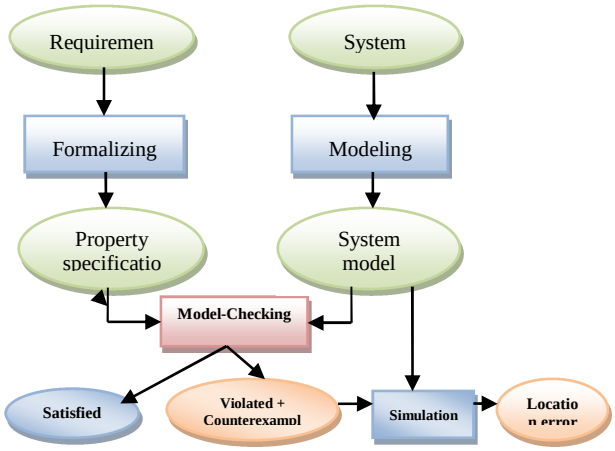

Figure 3. The model checking approach.

For our approach, we will use the model checking to analyze semantic networks. We use both linear time logic "LTL" and computation tree logic "CTL" for describing the specifications of the properties to be verified with the model checking.

Algorithm: Model-checking
Begin
While stack ≠ nil ***do***
P := top (stack);
while ¬ satisfied (p) ***then***
Refine the model, or property;

Else if satisfied (p) ***then***
P := top (stack);

Else *// out of memory*
Try to reduce the model;
End
End

The concepts of temporal logic were used for the first time by Pnueli [6] in the specification of formal properties that are fairly easy to use. The operators are

very close in terms of natural language. The formalization in temporal logic is simple enough although this apparent simplicity therefore requires significant expertise. In [20], the temporal logic allows representing and reasoning about certain properties of the system, so it is well-suited for the systems verification. There are two main temporal logics, that is linear time and branching time. In linear time temporal logic, each execution of the system is independently analyzed. In this case, a system satisfies a formula f, if f holds along every execution. The branching time combines all possible executions of the system into a single tree. Each path in the tree is a possible representation of the system execution.

- **Linear Temporal Logic** or **LTL** allows representing the behavior of reactive systems using properties that describe the system in which time proceeds linearly. Clearly, we specify the expected behavior of a system, by specifying the only possible future as a sequence of actions that follow; LTL uses for that temporal operators: X (Next), F (Finally or Eventually), G (Always), U (Until).
- **Computation Tree Logic** or **CTL** suggests several possible futures from a system state rather than having a linear view of the considered system. The operators of CTL are obtained by adding A (for any execution) or E (there is an execution) before the operators of linear temporal logic that are: AX φ (all successor states immediately satisfy φ), EX φ (there is an execution whose next state satisfies φ), AF φ (for any execution, there is a state where φ is true), EF φ (there is an execution, leading to a true state φ), AG φ (for any execution, φ is always true), EG φ (there is an execution, where φ is always true), AφUψ (for any execution φ is true until ψ is true), EφUψ (there is an execution in which φ is true until ψ is true).

The model checking uses a kripke [21] structure to represent the behavior of the system. Kripke structures are an abstract representation of the behavior of an algorithm whose certain properties must be tested. A Kripke structure is used to represent the relation between states of the system to be checked. It is a directed graph where nodes, called states, are labeled by states of the system, as seen in Figure 3.

C. Temporal logic queries

Temporal logic queries are a generalization of the model checking [22, 23, 24], which allows system properties not only to be verified, but also to be computed in a systematic manner. A temporal logic query is an incomplete temporal logic specification containing a special placeholder symbol "?". Intuitively, the query asks for those system properties which yield a correct specification when inserted into the query. [25, 26] define two types of queries, queries with one placeholder and queries with multiple placeholders.

Definition 1. A temporal logic query with a single placeholder $?_1$, denoted $\varphi[?_1]$, is an expression containing a symbol $?_1$, where replacing $?_1$ by a propositional formula yields a CTL or LTL formula.

Definition 2. A temporal logic query with multiple placeholders $?_1,\ldots,?_n$, denoted $\varphi[?_1,\ldots?_n]$is an expression containing symbols $?_1,\ldots?_n$ where replacing $?_1,\ldots?_n$ by a propositional formula yields a CTL or LTL formula.

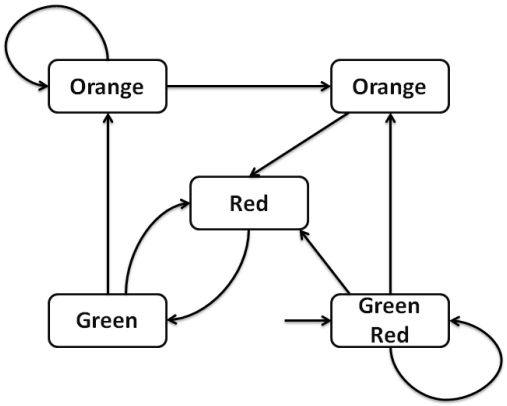

Figure 4. An example of Kripke structure: a traffic light.

Note that the second definition allows multiple occurrences of the same placeholder symbol in the query. To understand better, let us illustrate this with an example. Assume that we are currently designing a traffic light system, as in Figure 4 (The traffic light alternates in the right order and without blocking), and rely on the model checking to verify if it satisfies a typical temporal specification such as:

$$G(\text{orange} \rightarrow F\ \text{red}) \qquad (S)$$

stating that "all orange lights are eventually followed by red lights". The model checking will provide a yes-or-no answer: either the system satisfies (S) or it does not.
Temporal queries lead to a finer analysis of the system. The query

$$G(? \rightarrow F\ \text{red}) \quad (Q)$$

asks for conditions that always lead to red. Computing solutions for (Q), in our system, will tell us, among other things, whether the system satisfies its specification: (S) is satisfied iff the orange light is a solution for (Q). However, it will tell us more. For example, if (S) is not satisfied, answering (Q) can lead to the discovery that, in reality, the property that our system satisfies is G(green →F orange).

III. RELATED WORK

In this section, we briefly discuss some of the researches related to the verification and the query of the Semantic graphs using the model checking. There are very few researches about the use of the model checking method to qualify a Semantic graph. On the contrary, there are many more researches on the verification of the Web application. The work in [27] proposes a new way of

converting an RDF graph into the BCG (Binary Coded Graph) format that was used in the CADP (Construction and Analysis of Distributed Processes) toolbox. The latter represents a verification toolbox for asynchronous concurrent systems. The toolbox accepts as input several languages and all of them are compiled into LTS (Labeled Transition System), which is a state/transition graph representing the behavior of concurrent systems. CADP provides several representations for LTS; one of these representations is the BCG format.

There are few research works on the study of the use of the temporal logic to query a system. We are the first to use the temporal logic to query the semantic graph models. The temporal logic query was introduced by William Chan [7] in order to speed up design understanding by discovering properties unknown yet. [25, 26] show that query checking is applicable to a variety of model exploration tasks, ranging from invariant computation to test case generation. They illustrate it by using a CCS (Cruise Control System), which is responsible for keeping an automobile traveling at a certain speed. In their study, the tool they created searches all the propositional formula to hold the temporal logic query, while in our research work, our tool searches only the state that holds the temporal logic query, as explained in Figure 4. The difference between the two works consists in the states content. [28] studies the problem of computing all minimal solutions to arbitrary temporal queries over arbitrary Kripke structures and [29] presents a tool that finds the solutions to any CTL query.

In the work of Chan, a temporal logic query can have only one placeholder while in [25, 26, 29, 29], it can have multiple placeholders. Chan's temporal logic queries have a unique representation, using computation tree logic operators, while the research in [30] tries to extend Chan's work to other temporal logic formulas, such as CTL*, which includes the linear temporal logic. Our resolution tool can have multiple placeholders in the temporal logic query.

IV. The ScaleSem Approach

This section details our approach which consists in transforming semantic graphs into models in order to be verified by the model checker. For this, we have developed two tools called "RDF2SPIN" and "RDF2NμSMV", that transform semantic graphs into PROMELA and respectively into NμSMV [31] languages.

We use SPIN [32] and NμSMV as model checkers to check the model of semantic graphs. We want to compare them in terms of capabilities. SPIN is a software tool for verifying system models. The system is described in a language model called PROMELA. NμSMV is the amelioration of SMV model checker, working on the same simple principles as SMV. SPIN verifies the correctness of properties expressed in linear time logic; on the other side, NμSMV verifies the properties in both linear time logic and computation tree logic.

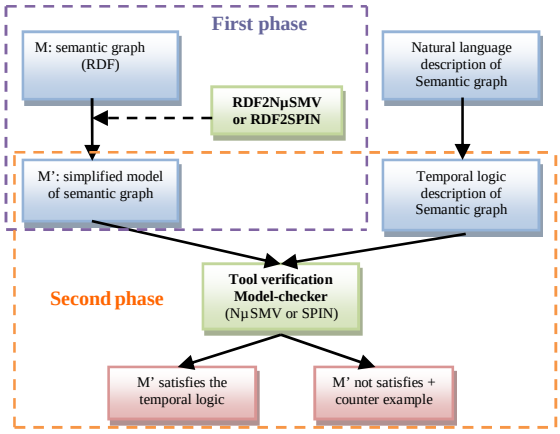

Figure 5. The ScaleSem architecture.

In Figure 5, we present the architecture of our approach. We can get the semantic graph (RDF) and its description in temporal logic from a natural language description, as shown in Figure 9 of section V. We divide this architecture in two phases. The first phase concerns the transformation of the semantic graph into a model using our tools RDF2SPIN and RDF2NμSMV. There are three steps in this transformation. The first step is to explore the entire RDF graph to obtain the triples table. The second step is to determine a root for the graph, and the last step is to write the model that represents the semantic graph in the PROMELA or NμSMV languages. The second phase concerns the verification of properties expressed in temporal logic on the model using the SPIN or the NμSMV model checkers. The choice of the model checker depends on the tool that one uses to convert the semantic graphs. For example, when using RDF2SPIN, one must use the model checker SPIN to check the model.

A. Introducing RDF

The RDF graphs [18] considered here are represented as XML verbose files, in which the information is not stored hierarchically (the so-called graph point of view). These RDF graphs, that represent, in fact, the semantic model of an RDF file, are not necessarily connected, meaning they may have no root vertex from which all the other vertices are reachable. The RDF graph transformation into a model is articulated in three steps: exploring the RDF graph, holding election of the root vertex and generating the model of the semantic graph.

B. Exploring RDF graph

In order to exploit the RDF graphs by using SPIN or NμSMV, we therefore have to determine whether they have a root vertex, by analyzing RDF triples, and if this is not the case, we must create a new root vertex by taking care to keep the size of the resulting graph as small as possible.

We achieve this by appropriate explorations of the RDF graphs, as explained below. Let us consider that an RDF graph is represented as a couple (V, E), where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. For

a vertex x, we note $E(x) = \{ y \in V \mid (x, y) \in E \}$ the set of its successor vertices. This corresponds to the classical data structure for representing graphs in memory, consisting of an array indexed by the vertices and containing in each entry the list of successor vertices of the corresponding vertex. There are several algorithms to traverse a large graph, of these basic algorithms include the best known, depth-first search (DFS) and breadth-first search (BFS). We use the depth-first search algorithm, illustrated below, to explore the graph, knowing that the breadth-first algorithm also works in this context. We considered here an iterative variant of DFS, which makes use of an explicit stack, rather than the recursive variant given in [33]; this is required in practice to avoid overflows of the system call stack when the algorithm is invoked for exploring large graphs.

```
Algorithm: procedure Dfs (x):
begin
  visited(x) :=  true;
   // vertex x becomes visited
  p(x) := 0; // start exploring its successors
  stack := push(x, nil);
 while stack ≠ nil do
y := top(stack);
if p(y) < |E (y)| then
 // y has some unexplored successors
 z := E (y) p(y);
    p(y) := p(y)+1;
    // take the next successor of y
    if ¬ visited (z) then
    visited(z) := true;  // visit it
    p(z) := 0;//start exploring its successors
    stack := push(z, stack)
    endif
   else //all successors of y were explored
   stack := pop(stack)
  endif
 end
end
```

C. Determining a Root Vertex

If the RDF graph has no principal vertex root but multiple roots, we must create a new root whose successors are all the other roots already existing in the graph, but this will increase the number of edges. We look forward to doing this by adding as few edges as possible. A vertex x of a directed graph is a partial root if it cannot be reached from any other vertex of the graph. If the graph contains only one partial root, all other vertices of the graph can be reached from the root, otherwise there would be other roots in the partial graph. If the graph has multiple partial roots, the most economical way to provide a root is to create a new record with all the roots as a partial successor: this will add to the graph a minimum number of edges. We compute the set of partial roots in two phases, each one consisting in successive explorations of the graph. The first phase identifies a set of candidate partial roots, and the second one refines this set in order to determine the partial roots of the graph.

Remark: a property must always have a resource and a value; the resource should never be a value with the same predicate, i.e. a loop in the graph.

```
Algorithm: procedure RootElection():  //
precondition: ∀ x ∈ V.visited(x) = false
Begin // first phase
 root_list := nil;
 forall  x ∈ V do
  if ¬ visited(x) then
   Dfs(x);
   root_list := cons(x, root_list)
  endif
 endfor;
//second phase
 if |root_list|= 1 then
  root := head(root_list)
   // the single partial root is the global root
  else
  forall x ∈ V do  visited(x):= false;
  endfor;
  forall x ∈ root_list do
   // reexplore partial roots in reverse order
   if ¬ visited(x) then Dfs(x)
   else
    root_list := root_list \ {x}
     // partial root is not a real one
   endif
  endfor;
  if |root_list| = 1 then
   root := head(root_list)
   // a single partial root is the global root
   else
   root := new_node();
   // new root predecessor of the partial roots
   E(root) := root_list
  endif
 endif
```

The first phase explores all the vertices of the graph, and inserts in root_list all vertices that have no predecessor. If root_list contains a single vertex, it means that it represents the global root of the graph since all the other vertices are accessible from it, and it is useless to go to the second phase. Otherwise, any vertex contained in root_list could also be a root of the graph: the goal of the second phase is to determine the root of the global graph among the partial roots.

The second phase performs a new wave of exploration of the roots contained in the partial root_list in the reverse order they were inserted in the list. If a root in the root_list is to be visited by a partial root, it is removed from the list because it is not a partial root. At the end of this phase, all partial roots of the graph are present in the root_list. Indeed, each vertex is unreachable from the partial roots which were explored during the second phase. A new root is created, as in Figure 6, having as successor all the partial roots of the root_list, which ensures that all vertices of the graph are accessible from the new root. Therefore, such a summit is inaccessible from other nodes of the graph.

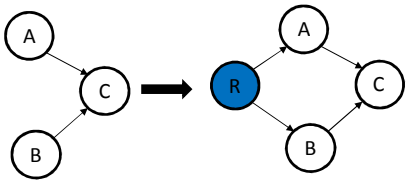

Figure 6. A root is a single node that has no predecessor. In this graph, we have a node A and a node B, two roots, and then we will create a new virtual root (blue circle "R") that points to the two roots.

The algorithm for determining a root has a complexity O(|V|+|E|), linear in the size of the graph (number of vertices and edges), since each phase visits every state and traverses every edge of the graph only once. Given the fact that the graph must be traversed entirely in order to determine whether it has a root or not, this complexity is optimal.

D. Generating the model

The third step is divided into three sub-steps. The first one consists in creating the table of all triples by exploring the entire graph; the second one consists in generating the table of resources and values for RDF2SPIN but for RDF2NμSMV, it generates the table of association. The last one consists in producing the model representing the semantic graph written in PROMELA or in NμSMV languages.

Table of triples - We will create a table consisting in resources, properties and values, by exploring the RDF graph. In our RDF graph, the resource and the value are represented by nodes, and the property is an edge directed from the resource towards the value. The table of triples of the RDF graph is useful for the next sub-step.

In this second sub-step, RDF2SPIN generates a table of resources and values, while RDF2NμSMV generates a table of association.

- *Table of resources and values* - Browsing the table triples seen in the previous step, we attribute a unique function for each resource and for each value. These functions are of proctype type. We combine all these functions in a table called table of resources and values.
- *Table of association* – This table contains an identifier for each resource, property and value.

The model - In this last sub-step, we will write the model in PROMELA language for the RDF2SPIN tool or in NμSMV language for the RDF2NμSMV tool, corresponding to the RDF graph that we want to check.

V. QUERYING THE MODELS OF SEMANTIC GRAPH

Before showing our approach that solves the temporal logic queries on the semantic graph models, we will present the limitations and the complexity of the SPARQL query language, which represents the Semantic Web standard.

The limitation of SPARQL. Given that SPARQL is a comparably young technology, the current W3C specification [34] still has a couple of limitations, which become obvious when comparing SPARQL to established query languages such as SQL or XQuery (XML Query Language). The following list surveys important features and constructs that are (to date) missing in SPARQL.

- **Aggregation**: The current specification does not support aggregation functions, such as summing up numeric values, counting, or average computation.
- **Updates**: While the SPARQL standard supports data extraction from RDF graphs, constructs for inserting new triples into RDF graphs and manipulating existing graphs (in the style of SQL Insert and Update clauses) are missing.
- **Paths expression**: SPARQL does not support the specification of (constrained) path expressions, e.g. using a single SPARQL query it is impossible to compute the transitive closure of a graph or to extract all nodes that are reachable from a fixed node.
- **Views**: In traditional query languages such as SQL, logical views over the data play an important role. They are crucial to both database design and access management. SPARQL does not currently support the specification of logical views over the data; however, that materialized views over the data can be extracted from the input graph using the CONSTRUCT query form.
- **Support for constraints**: Mechanisms to assert and check for integrity constraints in the RDF database are not covered in the current SPARQL specification. In SQL, such as integrity constraints are implicitly derived from primary and foreign key specifications established in the schema design phase. Beyond that, it is possible to enforce user-defined constraints using the Create Assertion statement.

The complexity of SPARQL. The analysis of SPARQL complexity is not new: the preliminary investigation of the combined complexity of SPARQL in [35] shows that the evaluation problem for full SPARQL expressions is PSpace-complete. As a consequent enhancement of this initial analysis, [34] systematically explored the complexity of all expression and query fragments, where a fragment means a class of expressions or queries that can be built using a fixed subset of the SPARQL operators. One central result is that the Evaluation problem for the operator Optional alone is already PSpace-hard. The author further shows that this high complexity is caused by an unlimited nesting of Optional expressions. Still, as a key insight, the operator Optional is by far the most complicated construct in SPARQL. This observation suggests that special care in query optimization should be taken in queries containing the operator Optional and will serve as a guideline for the SPARQL optimization.

In [36] the query optimizer can choose optimal join orders even for complex queries, with a cost model that

includes statistical synopses for entire join paths. The absence of a global schema and the diversity of predicate names pose major problems for the physical database design. In principle, one could rely on an auto-tuning to materialize frequent join paths; however, in practice, the evolving structure of the data and the variance and dynamics of the workload turn this problem into a complex task. The fine-grained modeling of RDF data queries with a large number of joins will inherently form a large part of the workload, but the join attributes are much less predictable than in a relational setting. This calls for specific choices of query processing algorithms, and for careful optimization of complex join queries.

Extensions of SPARQL. The work in [37] shows that SPARQL – with some minor extensions – can be used to express a large class of constraints and to extract constraints from RDF graphs when these are specified using a predefined vocabulary for encoding constraints.

Aggregation functions for SPARQL were proposed in [38]. The latter work defines an extension of SPARQL, called SPARQL++, which embeds standard aggregate functions in Construct and Filter clauses. The motivation for this extension was to express schema mappings through SPARQL Construct queries. It is also worth mentioning that some existing SPARQL engines, e.g. ARQ [39] and Virtuoso [40], have already implemented their own strategies to aggregation.

Path expressions for SPARQL have been identified as an important feature in several research contributions [41; 42; 43]. The common idea to all these approaches is to extend SPARQL by constructs that allow expressing relations between nodes that go beyond what can be expressed by simple basic graph patterns, e.g. transitively connected nodes. It is natural to assume that querying for (constrained) paths is an important feature in the context of a graph structured data model like RDF. The approach in [41] uses so-called regular path patterns, akin to regular expressions, to express complex path relations between nodes in RDF graphs. These regular path patterns are used to extend SPARQL to a dialect called SPARQLeR (SPARQL extended with Regular paths). In [42] a SPARQL extension called nSPARQL (Navigational SPARQL) is proposed, driven by the idea of navigating through the RDF graph using a set of predefined axes, very much in the style of the XPath axes for navigating through XML documents. Another reasonable approach is the PSPARQL (Path SPARQL) [43] query language. It relies on an extended version of RDF, called PRDF (Path RDF), where graph edges (i.e. predicates in RDF triples) may carry regular expression patterns as labels. The PSPARQL query language is then defined over such PRDF patterns.

In [7], a temporal logic query is presented as a string in which the placeholder appears exactly once. In our research work, a temporal logic query for querying a semantic graph can have multiple placeholders. The placeholder is represented with the special symbol "?".

Definition 3 (solution). Consider a CTL (or LTL) query, K a Kripke structure, and φ a propositional formula. We write ϒ[φ] to denote the result of substituting φ for the placeholder in the query. If K |= ϒ[φ], then we say that φ is a solution to ϒ in K. We denote the set of all solutions to a query ϒ in a Kripke structure K by sol(K, ϒ) = { φ | K |= ϒ[φ]}.

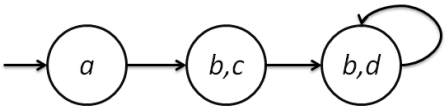

Figure 7. Example of a Kripke structure.

For example, consider the Kripke structure K shown in Figure 7 and the CTL query ϒ= A((a ∨ c) U AG?). It is easy to see that b, d, and b ∧d are solutions to ϒ in K.

This query has one placeholder. [25] introduces the query with multiple placeholders. If a query contains multiple placeholders, it is transformed into a CTL formula by substituting a propositional formula for each placeholder. Given a query with n placeholders, with L_i being the lattice of propositional formulas for the $_i$th placeholder, the set of all possible substitutions is given by the cross product $L = L_1 \times \ldots \times L_n$.

In our approach, we developed a new tool named STL Resolver. After transforming the semantic graph into a model (section IV), we query this model by using the temporal logic query. Figure 8 represents the architecture of the STL Resolver tool.

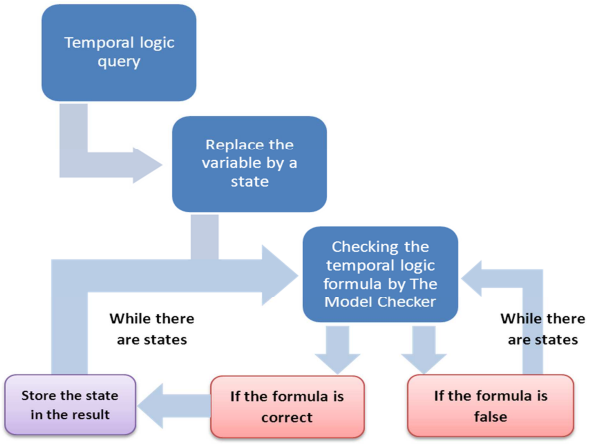

Figure 8. The STL Resolver tool.

First, we transform the temporal logic query into temporal logic formula (i.e. there is no placeholder in the temporal logic) by replacing the placeholder of the query with a state of the semantic graph model. The query placeholder can be replaced by a state of the graph and not by a propositional formula, as seen above. Furthermore, if the temporal logic formula is verified by the model checker, we store this state in the set of results, else we replace the query placeholder with another state; afterwards, we repeat the verification with the model checker. When we have no state to replace in the

temporal logic query, we return the set of all the solutions.

The Simplicity and the expressivity of the query checking. Previously, we saw that the SPARQL query language is the Semantic Web standard but is characterized by complexity and limitations. We use the temporal logic query to simplify the SPARQL queries and to reach expressivity with the temporal logic operators. Our STL resolver is based on the model checking techniques. In [36] the problem under discussion is to choose optimal joins for complex SPARQL queries. For example, note the SPARQL query below:

```
SELECT ?x
WHERE {
        ?x hasName ?y.
        ?y hasAdress ?z.
        ?z hasAge "26"
}
```

This query selects the subject ?x that has the age of 26. In temporal logic query, the SPARQL query becomes as follows:

Eventually (?x → next next next 26)

Note that the query becomes simpler than in the SPARQL query language, due to the use of the temporal logic operators. We use three Next operators (Section 2) in the query because there are three nodes to access the node "26" representing the age.

We noticed above the simplicity of the temporal logic query, and we will see below the expressivity due to the temporal logic operators. We saw above that the SPARQL query language had a lot of limitations. For instance, the path expression gives the path between two resources, which means that the first resource crosses all the nodes in order to reach the second resource. To express this characteristic with the temporal logic query, we use only the negation of the two resources and at least one **Next** operator which represents the number of nodes that separate the resources. For example in the temporal logic below, we want to know the path of length three that separates the two resources (resource1 and resource2):

! **Eventually** (ressource1 → **Next Next Next** ressource2)

The model checking algorithms return a counterexample that contains the path between the two resources, as a true temporal logic formula has been negated.

Example of Resolution. The example below helps to better understand how the STL Resolver works. The graph in Figure 9 describes the further affirmations [44]:

"Ninety-three is a novel by Victor Hugo published in 1874, whose theme is the French Revolution. Victor Hugo was born in February 26, 1802 in Besançon".

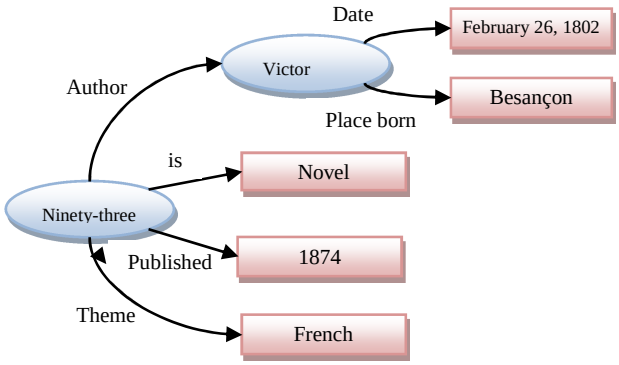

Figure 9. Example of an RDF graph.

Due to our tools "RDF2SPIN" and "RDF2NμSMV", the graph in the Figure 9 can be transformed into a model, written in PROMELA and respectively in NμSMV languages (see section IV) in order to be checked with the temporal logic or to be queried by the temporal logic queries. The query below searches all the states that follow the state "Ninty-three":

SPARQL:

SELECT ?x **WHERE** {"Ninety-Three" ?y ?x}

Temporal logic query**:**

Eventually (Ninety-three → **Next** ?x)

To answer this query, we use the STL Resolver which is based on the model checking algorithms. We replace the placeholder of the temporal logic query with all the states of the graph one by one, and we check all the properties with the model checker. The temporal logic is expressed by an automaton that invalidates the property; the model of the RDF graph represented in Figure 9 is also expressed by an automaton. A synchronized product is built; it is an automaton recognizing the intersection of two languages. Then, if the language is empty, i.e. there is no execution sequence of the model that invalidates the property, the STL Resolver stocks this state as a result. Otherwise, the model checker returns a counterexample representing an execution on the model that is not allowed by the property. In both cases, if there rest other unreplaced states of the graph, we replace them in the temporal logic query, and we repeat the process described above; else we return the set of all the solutions, as follows:

{Victor_hugo, Novel, 1874, French}

If these states are replaced one by one in the temporal logic query, we notice that the model checker will always return true.

Another example is represented by the query below that searches the person (RDF resource) who was born in "Besançon" on the "February 26, 1802".

SPARQL:

```
SELECT ?x WHERE
        {?x Place Born  "Besançon" .
         ?x Date "February 26, 1802"}
```

Temporal logic query**:**

Eventually (?x → **Next** Besançon ^ ?x → **Next** February 26,1802)

We notice that in this query there are two placeholders which are identical. The query result is the resource “Victor Hugo”.

The following example cannot be expressed in a SPARQL query. This query gives the path between the resource “Ninety-three” and “Besançon.”

! **Eventually** (Ninety-three → **Next Next** Besançon)

We use two Next operators because between the resource “Besançon” and the resource “Ninety-three” there is a length of two resources. The result of this query is illustrated by the following path:

{Ninety-three, Victor Hugo, February 26 1802}

VI. BENCHMARK

Now, we will be able to transform the RDF graph with our tools "RDF2SPIN" and RDF2NµSMV" into a model in order to check each temporal logic formula and see if it is verified or not in the model with the SPIN and NµSMV model checkers. In this way, we can verify the semantic graphs.

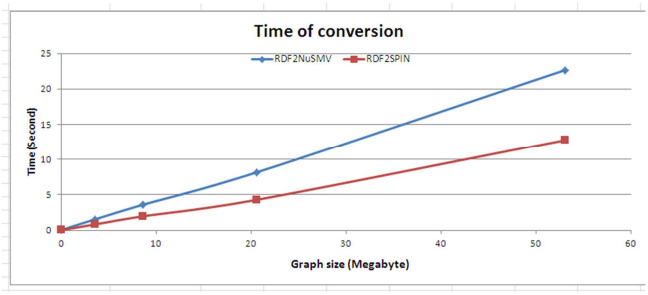

Figure 10. Time of conversion of Semantic graphs.

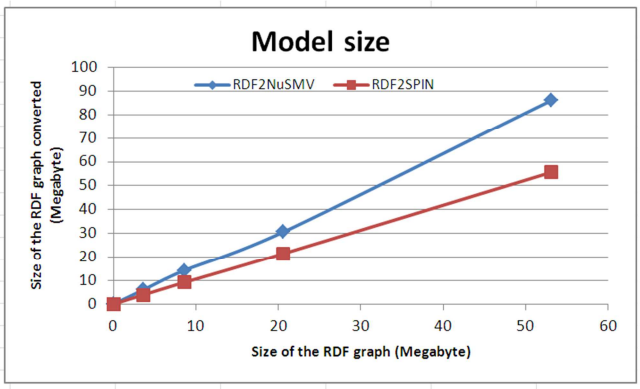

Figure 11. Size of the models.

We tested our tools on several RDF graphs, and we calculated the time of conversion as shown in Figure 9. Note that the RDF2SPIN tool is faster in converting semantic graphs than the RDF2NµSMV tool. Both tools are quick in converting semantic graphs; we obtain less than 15 seconds for a graph of 53 MB size using the RDF2SPIN tool and almost 21 seconds using the RDF2NµSMV tool. Both transformation tools follow a polynomial curve. In Figure 10, we see the size of converted semantic graphs from RDF to PROMELA language with RDF2SPIN and NµSMV language with RDF2NµSMV. We notice that the sizes of the PROMELA model are smaller than the NµSMV model.

VII. CONCLUSION

This paper presents a new technique for the semantic graphs verification by using a model checker. Knowing that the model checker does not understand the semantic graphs, we developed two tools RDF2SPIN and RDF2NµSMV to convert them into PROMELA and NµSMV languages in order to be verified with the temporal logic formulas. There are formulas that can be presented in LTL and not in CTL and vice versa. The advantage of the NµSMV model checker is that the verification can be made with both linear time logic and computation tree logic formulas. We also use the temporal logic query to query the semantic graph model. We have implemented a query checker to resolve the query on the semantic graphs model.

In our future work, we aim to convert the SPARQL query language for RDF graphs into queries using the operator of the temporal logic, in order to have a better verification of RDF graphs representing, for example, the building industry. The SPARQL queries have a lot of limitations, but due to the model checking technique, we continue to find solutions for this gap.

REFERENCE

[1] T. Bray, J. Paoli, C. Sperberg-McQueen. M., Maler, E., Yergeau, F., Cowan, J.: Extensible Markup Language (XML) 1.1 (second edition) W3C recommendation. (2006)

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. Scientific American. pp. 34–43. (2001)

[3] J. Kahan, M. Koivunen, E. Prud'Hommeaux, R. R. Swick. Annotea: An Open RDF Infrastructure for Shared Web Annotations, in Proc. of the WWW 10th International Conference, Hong Kong. (2001)

[4] J. P. Katoen: The principl of Model Checking. University of Twente. (2002)

[5] K. Homma, K. Takahashi, A. Togashi. Modeling and Verification of Web Applications Using Formal Approach. IEICE Tech. Rep., vol. 109, no. 40, SS2009-8, pp. 43-48. (2009)

[6] A. Pnueli. The temporal logic of programs. In proc. 18th IEEE Symp. Foundations of Computer Science (FOCS’77), Providence, RI, USA. pages 46-57. (1977)

[7] W. Chan. “Temporal-Logic Queries,” Proc. 12th Conf. Computer Aided Verification (CAV ’00), pp. 450-463, July 2000.

[8] R. Angles and C. Gutierrez. Querying RDF Data from a Graph Database Perspective. 2nd. European Semantic Web Conference (ESWC2005), May 2005, Heraklion, Greece. Lecture Notes in Computer Science, Volume 3532, pp. 346-360. 2005

[9] G. Karvounarakis, S., Alexaki, V. Christophides, D. Plexousakis, M. Scholl. RQL: A Declarative Query Language for RDF. In: Proc. of the 11th WWW conference, ACM Press. Pages 592–603. 2002.

[10] A. Seaborne. RDQL - A Query Language for RDF, W3C Member Submission 9 January 2004.

http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/

[11] M. Sintek, S. Decker. TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. Proc. of the 1th ISWC (2002)

[12] T. Berners-Lee. Notation 3 - An RDF Language for the Semantic Web. http://www.w3.org/DesignIssues/Notation3 (2001)

[13] A. Magkanaraki, G. Karvounarakis, T.T. Anh, V. Christophides, D. Plexousakis. Ontology Storage and Querying. Tech. Report 308, ICS-FORTH - Hellas (2002)

[14] P. Haase, J. Broekstra, A. Eberhart, R. Volz. A Comparison of RDF Query Languages. In: Proc. of the 3th ISWC conference. Number 3298 in LNCS, Springer-Verlag 502 (2004)

[15] E. Prudhommeaux, A. Seaborne. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/ (2005)

[16] D. Becket, B. McBride: RDF/ XML Syntax Specification (Revised). W3C recommendation. (2004)

[17] T. Berners-Lee. W3C recommandation. (2007)

[18] V. Bönström, A. Hinze, H. Schweppe: Storing RDF as a graph. Latin American WWW conference, Santiago, Chile. (2003)

[19] A. Chebotko, S. Lu, H. M. Jamil, and F. Fotouhi. Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns. Technical report, Wayne State University, TR-DB-052006-CLJF, 2006.

[20] M. Mukund. Model Checking: Automated Verification of Computational Systems. Pages 667-681. (2009)

[21] S. Bardin. Introduction to the Model Checking. CEA,LIST, Safety Software laboratory. 2008

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

[23] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Proceedings of the Workshop on Logics of Programs, volume 131 of Lecture Notes in Computer Science, pages 52–71. Springer-Verlag, 1981.

[24] JP. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Proceedings of the 5th International Symposium on Programming, volume 137 of Lecture Notes in Computer Science, pages 337–350. Springer-Verlag, 1982.

[25] A. Gurfinkel, B. Devereux and M. Chechik. Model exploration with temporal logic query checking. SIGSOFT 2002/ FSE-10, 2002.

[26] A. Gurfinkel, M. Chechik and B. Devereux. Temporal logic query checking: a tool for model exploration. IEEE computer society. (2003)

[27] R. Mateescu, S. Meriot, S. Rampacek. Extending SPARQL with Temporal logic. Technical report. (2009)

[28] S. Hornus and Ph. Schnoebelen. On solving temporal logic queries. Algebraic Methodology and Software Technology. Lecture Notes in Computer Science, Volume 2422/2002, pages 73-89. 2002.

[29] M. Gheorghiu and A. Gurfinkel. TLQ: A query solver for states. 2006.

[30] S. Hornus and Ph. Schnoebelen. Queries in temporal logic. Technical report. 2009.

[31] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri. NµSMV: a new symbolic model checker. (2000)

[32] M. Ben-Ari. Principles of the SPIN Model Checker. Springer. ISBN: 978-1-84628-769-5. (2008)

[33] R. E. Tarjan: Depth-First search and linear graph algorithm. SIAM Journal of Computing 1, 2, 146-160. (1972).

[34] M. Schmidt. Fondations of SPARQL query optimization. *PhD Thesis, Albert-Ludwigs-Universität Freiburg (Germany)* 2009.

[35] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *Best Paper Award* , 5th International Semantic Web Conference, ISWC 2006.

[36] T. Neumann and G. Weikum. Scalable Join Processing on very large RDF graphs. SIGMOD'09. (2009)

[37] G. Lausen, M. Meier and M. Schmidt. SPARQLing Constraints for RDF. In EDBT, pages 499–509, 2008.

[38] A. Polleres, F. Scharffe, and R. Schindlauer. SPARQL++ for Mapping Between RDF Vocabularies. On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, pages 878–896, 2007.

[39] ARQ SPARQL Processor for Jena. http://jena.sourceforge.net/ARQ/.

[40] C. Blakeley. Mapping Relational Data to RDF with Virtuoso's RDF Views, 2007. OpenLink Software. (2007)

[41] K. Kochut and M. Janik. SPARQLeR: Extended SPARQL for Semantic Association Discovery. In ESWC, pages 145–159, 2007.

[42] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A Navigational Language for RDF. In ISWC, pages 66–81, 2008.

[43] F. Alkhateeb, JF. Baget, and J. Euzenate. Extending SPARQL with regular expression patterns (for querying RDF). Web Semantics, 7(2):57–73, 2009.

[44] B. Vatant. Metadata to describe resources (Semantic Web Languages). In Proceedings of the INRA Seminar : Metadata: changes and prospects. (2008)

Mahdi Gueffaz obtained his engineering in computer science at the University of Algiers in 2008 and a Master degree in image and computer science at the University of Burgundy in 2009. He has been working toward his PhD degree at the LE2I laboratory (electronics, Image and computer Science) of the University of Burgundy since September 2009. He is also teaching at the University of Burgundy. His research interests lie on formal methods, software engineering and applying the model checking method to improve the Semantic Web quality.

Sylvain Rampacek is associate professor at the University of Burgundy. He obtained a PhD in Computer Science at the University of Reims in 2006. His researches are focused on formal methods and software engineering, applied to semantic web.

Christophe Nicolle is full professor at the University of Burgundy. In 1996, he obtained a PhD in Computer Science. His research area is the semantic interoperability of information systems. From his research, he founded the company ACTIVe3D in 2003. This company develops a facility management web platform based a combination of 3D Digital Mockup and semantics.