International Journal of Digital Information and Wireless Communications (IJDIWC) 1(1): 64-74
The Society of Digital Information and Wireless Communications, 2011(ISSN 2225-658X)

Verifying Semantic Graphs With the Model Checker SPIN

Mahdi Gueffaz[1], Sylvain Rampacek[1], Christophe Nicolle[1],
[1] LE2I, UMR CNRS 5158
University of Bourgogne,
21000 Dijon, France
{Mahdi.Gueffaz, Sylvain.Rampacek, CNicolle}@u-bourgogne.fr

ABSTRACT

The most frequently used language to represent the semantic graphs is the RDF (W3C standard for meta-modeling). The construction of semantic graphs is a source of numerous errors of interpretation. 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 between two areas: the semantic web and the model checking. For this, we developed a tool, RDF2SPIN, which converts RDF graphs into SPIN language. This conversion aims checking the semantic graphs with the model checker SPIN in order to verify the consistency of the data. To illustrate our proposal we used RDF graphs derived from IFC files. These files represent digital 3D building model. Our final goal is to check the consistency of the IFC files that are made from a cooperation of heterogeneous information sources.

KEYWORDS

Semantic graph, RDF, Model-Checking, Temporal logic, SPIN, IFC, BIM.

1 INTRODUCTION

The increasing development of networks and especially the internet has greatly developed the heterogeneous gap between information systems. In glancing over the studies about interoperability of heterogeneous information systems we discover that all works tend to the resolution of semantic heterogeneity problems. Now, the W3C[1] suggest norms to represent the semantic by ontology. Ontology is becoming an inescapable support for information systems interoperability and particularly in the Semantic. Literature now generally agrees on the Gruber's terms to define an ontology: explicit specification of a shared conceptualization of a domain [1]. The physical structure of ontology is a combination of concepts, properties and relationships. This combination is also called a semantic graph.

Several languages have been developed in the context of Semantic Web and most of these languages use XML[2] as syntax [2]. The OWL[3] [3] and RDF[4] [4] are the most important languages of the semantic web, they are based on XML. OWL allows representing the ontology, and it offers large capacity machines performing web content. RDF enhances the ease of automatic processing of Web resources. The RDF (Resource Description Framework) is the first W3C standard for enriching resources on the web with detailed descriptions. The descriptions may be characteristics of resources, such as author or content of a website. These descriptions are metadata. Enriching the Web with metadata allows the

[1] World Wide Web Consortium
[2] eXtensible Markup Language
[3] Web Ontology Language
[4] Resource Description Framework

development of so-called Semantic Web [5]. The RDF is also used to represent semantic graph corresponding to a specific knowledge modeling. For example in the AEC[5] projects, some papers used RDF to model knowledge from heterogeneous sources (electricians, plumbers, architects…). In this domain, some models are developed providing a common syntax to represent building objects. The most recent is the IFC[6] [6] model developed by the International Alliance of Interoperability. The IFC model is a new type of BIM[7] and requires tools to check the consistency of the heterogeneous data and the impact of the addition of new objects into the building.

As the IFC graphs have a large size, their checking, handling and inspections are a very delicate task. In [7] we have presented a conversion from IFC to RDF. In this paper, we propose a new way using formal verification, which consists in the transformation of semantic graphs into a model and verifying them with a model checker. We developed a tool called "RDF2SPIN" that transforms semantic graphs into a model represented in SPIN [8] language. After this transformation, SPIN verifies the correctness of the model written in PROMELA[8] language with temporal logic in order to verify the consistency of the data described in the model of the huge semantic graphs.

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 model checking. Then, in section 3, we describe the mapping of the semantic graphs into models and our approach is defined in section 4. Finally, we end with the conclusion.

2 AN OVERVIEW OF SEMANTIC GRAPH AND MODEL CHECKING

The RDF is also used to represent semantic graphs corresponding to a specific knowledge modeling. It is a language developed by the W3C to bring a semantic layer to the Web [9]. It allows the connection of the Web resources using directed labeled edges. The structure of the RDF documents is a complex directed labeled graph. An RDF document is a set of triples <subject, predicate, object> as shown in the Figure 1. 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 the "resource" types. A resource is identified by an URI[9] [10, 11].

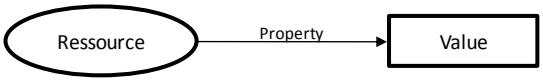


Figure 1. RDF triplet.

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). Values can be 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.

[5] Architecture Engineering Construction
[6] Industrial Foundation Classes
[7] Building Information Model
[8] Process Meta Language
[9] Uniform Resource Identifier

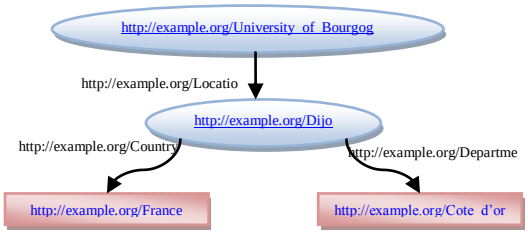


Figure 2. Example of partial RDF graph.

The RDF graph in the Figure 2 defines a node "University of Bourgogne" located at "Dijon", having as country "France" and as department "Cote d'Or". RDF documents can be written in various syntaxes, e.g., N3 [12], N-Triple [13], and RDF/XML. Below, we present the RDF\XML document corresponding to Figure 2.

```
<rdf:Description
rdf:about="http://example.org/univers
ity of Bourgogne">
  <ex:Location>
       <rdf:Description
rdf:about="http://example.org/Dijon">
       <ex:Country>
France</ex:Country>
       <ex:Department>Cote
d'or</ex:Department>
       </rdf:Description>
   </ex:Location>
</rdf:Description>
```

The model checking [14] described in Figure 3 is a verification technique that explores all possible system states in a brute-force manner. Similar to a computer chess program that checks all possible moves, a model checker, the software tool that performs the model checking, examines all possible system scenarios in a systematic manner. In this way, it can be shown that a given system model truly satisfies a certain property. Even the subtle errors that remain undiscovered using emulation, testing and simulation can potentially be revealed using model checking.

To make a rigorous verification possible, properties should be described in a precise unambiguous way. It is the temporal logic that is used in order to express these properties. The temporal logic is a form of modal logic that is appropriate to specify relevant properties of the systems. It is basically an extension of traditional propositional logic with operators that refer to the behavior of systems over time.

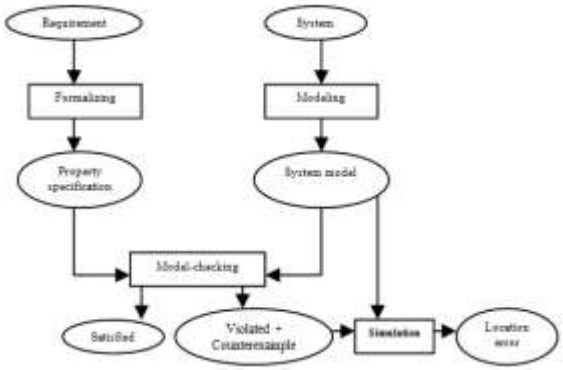


Figure 3. Model Checking approach.

The following algorithm explains the way that the model checking works. First we put in the stack all the properties expressed in the temporal logic. All of them are verified one by one in the model and if a property does not satisfy the model, it is whether the model or the property that we must refine. In case of a memory overflow, the model must be reduced. Whereas formal verification techniques such as simulation and model checking are based on model description from which all possible system states can be generated, the test, that is a type of verification technique, is even applicable in cases where it is hard or even impossible to obtain a system model.

```
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
```

3 THE MAPPING

This section speaks about our approach which consists in the transformation of semantic graphs into model in order to verify them with the model-checker. For this, we developed "RDF2SPIN" tool that transform semantic graph into PROMELA [8] language for the Model-checker SPIN.

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). On the one hand, these RDF graphs are not necessarily connected, meaning they may have no root vertex from which all the other vertices are reachable. On the other hand, the PROMELA language manipulated by the verification tools of SPIN always have a root vertex, which corresponds to the initial state of the system whose behavior is represented by the PROMELA language. The RDF graph transformation into PROMELA language is articulated in three steps: exploring the RDF graph, determining a root vertex and, final step, generating the Model of the RDF graph. [15]

3.1 Exploring RDF graph

In order to exploit the RDF graphs by using SPIN, we therefore have to determine whether they have a root vertex, 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, and we assume that these vertices are ordered from $E(x)_0$ to $E(x)|_{E(x)}|-1$. 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 depth-first search algorithm, illustrated below to explore graph, knowing that the breadth-first algorithm also work in this context.

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
```

We considered here an iterative variant of DFS, which makes use of an explicit stack, rather than the recursive variant given in [16]; this is required in practice to avoid overflows of the system call stack when the algorithm is invoked for exploring large graphs.

3.2 Determining a root vertex

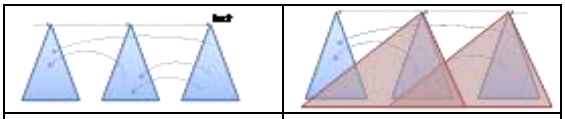
If the RDF graph has no vertex root, we must create a root as to be the successors of all vertices of the graph but it will increase the number of edges. We seek to do this by adding a few edges as possible. A vertex x of a directed graph is a partial root if it can not be reached from any other vertex 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 root partial, 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 RDF graph.

Algorithm: PROCEDURE ROOTELECTION(): // *precondition:* $\forall x \in V.visited(x) = false$

```
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
end
```



(a) First phase: identification of candidate partial roots.

(b) Second phase: selection of real partial roots.

Figure 4. Computation of the set of partial roots of an RDF graph by successive DFS explorations.

The first phase (see Figure 4(a)) explores the graph until it is fully explored, and inserts in root_list all vertices that have no predecessor. If root_list contains a single vertex, so overall it is the global root of the graph

since all the other vertex are accessible from him and it is useless to the second phase has passed. Otherwise, any vertex contained in root_list could also be a root of the graph: the role of the second phase is to determine which of the partial root is indeed the root of the global graph.

The second phase (see Figure 4(b)) performs a new wave of exploration of the roots contained in partial root_list in reverse order in which they were inserted in the list. If a root in 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 root_list. Indeed, each vertex is unreachable from the partial roots which were explored during the second phase. A new root is created (see Figure 5), having as successor all the partial roots of 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.

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 that the graph must be traversed entirely in order to determine whether it has a root or not, this complexity is optimal.

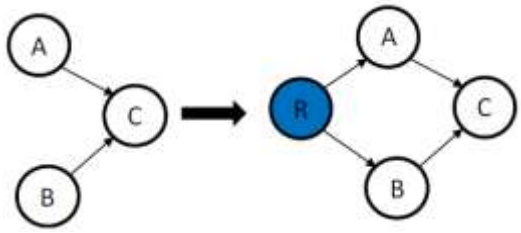


Figure 5. A root is a single node that has no predecessor. In this graph, we have node A and node B, two roots, and then we will create a new virtual root as shown in the figure (blue circle "R") that points to the two roots. The creation of the root can be useful in the field of construction, during the merger of two subset of a building, and don't have necessarily a common object that binds them. In this case the root is the object that will connect them.

3.3 PROMELA language generation

The third step is divided into three sub-steps. First and second one consists in generating two tables (triplets table and resources and values table). The last one consists in producing PROMELA language.

Table of triplets - Going through the RDF graph by graph traversal algorithms, we will create a table consisting of resources, properties and values. In our RDF graph, the resource is a vertex, the property represents the edge and the value is the successor vertex corresponding of the edge of the vertex. The table of triples of RDF graph is useful for the next step to create the table of resources and values.

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

PROMELA language - In this last step, we will write the PROMELA file corresponding to the RDF graph that we want to check. For this step, we will start by writing the function of the main root of the graph and for each property of the root, we call the function of the corresponding value. We will do the same for all "resource" functions defined in the table resources and values. In the other ones, all the function "value" we'll just display their contents.

3.4 Example of transformation

Consider an example of an RDF graph consisting of four resources, floor 1, Room 1, Floor 3 and Company. The Floor 1 resource his surface is 652, his construction date is 1943, and has a room which is the resource Room 1. The Floor 3 resource his surface is 607, his date of release is 2009 and his maintenance is the resource company. The Room 1 resource his type is Classroom and his surface is 60. The company resource his name is cleanQuick and his country is France. As you noted, Room 1 and company are both resources and values. As you can see in **Error! Reference source not found.**, the RDF graph of our example:

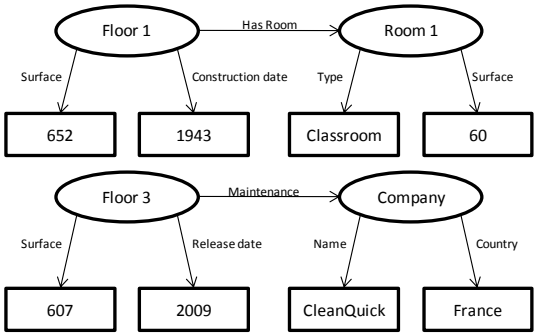


Figure 6. The RDF graph of our example.

After exploring the graph of the **Error! Reference source not found.**, the RDF graph has no single root. It is composed of two roots Floor 1 and Floor 3. For this, we will redefine another RDF graph that contains a single root. In **Error! Reference source not found.**, we create a root that is pointed on both roots floor 1 and floor 3. This root can be a building for example, to join the two floors, floor1 and floor 3.

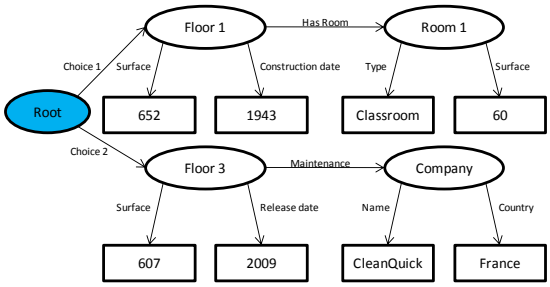


Figure 7. The RDF graph with a single root.

From the RDF graph in **Error! Reference source not found.** or the RDF representation in RDF/ XML format of the **Error! Reference source not found.**, you can generate a table of triplets. This table is composed of RDF triples i.e. "*resource - property - value*" of the RDF graph. The table of triplets of the RDF graph of our example is shown in the **Error! Reference source not found.**.

Table 1. Table of triplets.

Resource	Property	Value
Root	Choice 1	Floor 1
Root	Choice 2	Floor 3
Floor 1	Surface	652
Floor 1	Construction date	1943
Floor 1	Has room	Room 1
Room 1	Type	Classroom
Room 1	Surface	60
Floor 3	Surface	607
Floor 3	Release date	2009
Floor 3	Maintenance	Company
Company	Name	CleanQuick
Company	Country	France

The next step is to generate two tables, a table of resources and values. The table of resources provides a unique function *proctype* for each resource in the PROMELA language. It's the same for the table of values: it assigns for each value a function *proctype* in the PROMELA language. In our example, the table of resources and values are

presented in the **Error! Reference source not found.**.

Table 2. Table of resources and values.

Resource	Function PROMELA	Value	Function PROMELA
Root	Proctype root ()	652	Proctype v1 ()
Floor 1	Proctype Floor1 ()	1943	Proctype v2 ()
Room 1	Proctype Room1 ()	Classroom	Proctype v3 ()
Floor 3	Proctype Floor3 ()	60	Proctype v4 ()
Company	Proctype Company ()	607	Proctype v5 ()
		2009	Proctype v6 ()
		CleanQuick	Proctype v7 ()
		France	Proctype v8 ()

After election of a root, generation of a table of triplets of the RDF graph and the generation of the table of resource and values, the last step of processing is the generation of the PROMELA language. You can see below the RDF graph shown in figure 7 written in PROMELA language.

```
  /* Declaration of channels from
all the graph **/
  chan glob = [0] of { int } ;
  /* global variable that contains
  all paths of a resource function
  */
  int ch;

  /* the root function**/
  proctype Root() {
    printf(" Root \n ") ;
    if
            :: ch==0->printf("
            Choice 1 \n");run
            Floor1(); glob!1 ;
            :: ch==1->printf("
            Choice 2 \n");run
            Floor3(); glob!2 ;
    fi
  }

proctype Floor1() {
  printf(" Floor 1 \n") ;
  glob?1 ;
  if
          :: ch==0 ->
          printf("Surface \n") ;
          run v1(); glob!3 ;
          :: ch==1 ->
          printf("Construction_d
          ate \n"); run
          v2();glob!4 ;
          :: ch==2 -> printf("
          Has_room\n");
          run Room1(); glob!5 ;
  fi
}

proctype Room1() {
  printf(" Room 1 \n ") ;
  glob?5 ;
  if
          :: ch==0 ->
          printf("Type \n") ;
          run v3(); glob!6 ;
          :: ch==1 ->
          printf("Surface \n");
          run v4(); glob!7 ;
  fi
}

proctype Floor3() {
  printf(" Floor 3 \n") ;
  glob?2 ;
  if
          :: ch==0 ->
          printf("Surface \n
          ") ; run v5(); glob!8
          ;
          :: ch==1 ->
          printf("Release_date
          \n");      run v6() ;
          glob!9 ;
          :: ch==2 ->
          printf("Maintenance
          \n");        run
          Company() ; glob!10 ;
  fi
}

proctype Company() {
  printf(" Company \n") ;
  glob?10 ;
  if
```

```
          :: ch==0 ->
          printf("Name \n ") ;
          run v7 () ; glob!11 ;
          :: ch==1 ->
          printf("Country \n") ;
          run v8 () ; glob!12 ;
  fi
}

/* all functions of the values
of the graph from v1 to v8 */
proctype v1 () {
  glob?3 ;
  printf (" 652 ");
}

proctype v2 () {
  glob?4 ;
  printf (" 1943 ");
}

proctype v3 () {
  glob?6 ;
  printf (" Classroom ");
}

proctype v4 () {
  glob?7 ;
  printf (" 60 ");
}

proctype v5 () {
  glob?8 ;
  printf (" 607 ");
}

proctype v6 () {
  glob?9 ;
  printf (" 2009 ");
}

proctype v7 () {
  glob?11 ;
  printf (" CleanQuick ") ;
}

proctype v8 () {
  glob?12 ;
  printf (" France ") ;
}

init {
          atomic {
             run choice();
             run Root(); }
}

/* the function that initializes
the global variable from zero to
a number that is determined by
the value of maximum number of
properties that a node can
have*/
proctype choice(){
  do
          ::ch<3 -> ch++;
          ::ch==3 -> ch=0;
  od
}
```

In this example, we listed the main processing steps of the transformation of an RDF graph into PROMELA language.

4 THE VERIFICATION WITH THE MODEL CHECKER

As we saw in section 2, the model checker needs properties in order to check the model of semantic graphs. These properties are expressed in temporal logic. The concepts of temporal logic used for the first time by Pnueli [17] in the specification of formal properties 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. 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, which are 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.

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 a tool called "RDF2SPIN" that transforms semantic graphs into SPIN language.

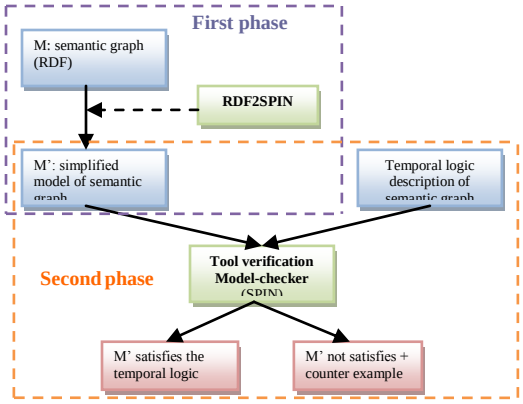


Figure 8. Our architecture.

The architecture of the Figure 4 is divided into two phases. The first phase concerns the transformation of the semantic graph into a model using our tool "RDF2SPIN", as described in section 3. The second phase concerns the verification of the properties expressed in temporal logic on the model using the model-checker SPIN.

To illustrate our approach, we take an RDF graph represented in the Figure 5 and a temporal logic expressed in the table 1 to verify if the BIM "b1" contains a floor.

Figure 9. Example of partial RDF graph.

Table 3. Temporal logic formula.

Temporal logic	Meaning	Result
Eventually (b1 → Next Next floor)	Is there a floor after two states starting from the state b1	True

We tested several RDF graphs on our tool "RDF2SPIN", graphs representing buildings as shown in Figure 6, using a machine that runs on a processor with a capacity of 2.4 GHz and 4 GB of RAM, calculating the time of conversion as shown in Figure 7. Note that the RDF2SPIN tool is faster in converting semantic graphs. We have almost 12 seconds for a graph of 53 MB size. The transformation tool follows a polynomial curve.

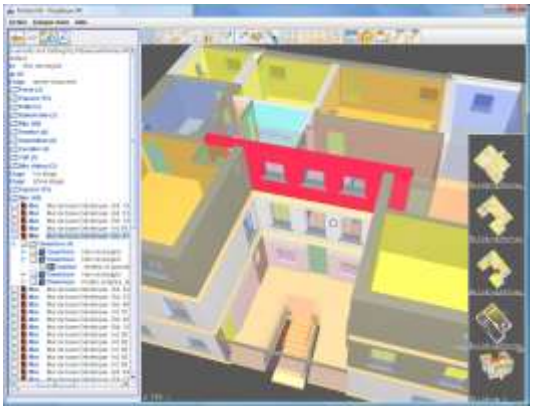


Figure 10. The 3D view of an IFC file.

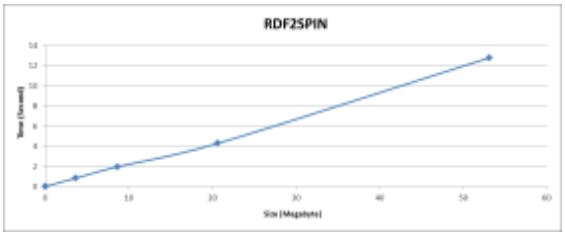


Figure 11. Time conversion of semantic graphs.

5 CONCLUSION

This paper presents how to transform a semantic graph into a model for verification by using a powerful formal method, that is the "model checking". Knowing that the model-checker does not understand the semantic graphs, we developed a tool RDF2SPIN to convert them into SPIN language in order to be verified with the temporal logics. This transformation is made for the purpose of classifying large semantic graphs in order to verify the consistency of IFC files representing 3D building.

6 REFERENCES

1. Gruber, T. R.: Toward principles for the design of ontologies used for knowledge sharing. Presented at the Padua workshop on Formal Ontology, later published in International Journal of Human-Computer Studies, Vol. 43, Issues 4-5, November 1995, pp. 907-928. March 1993.
2. Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., Cowan, J.: Extensible Markup Language (XML) 1.1 (second edition) W3C recommendation, http://www.w3.org/TR/2006/REC-xml11-20060816/. (2006)
3. Bechhofer, S., van Harmelen, F., Hendler J., Horrocks, I., McGuinness, D., Patel-Schneijder, P., Andrea Stein, L., OWL Web Ontology Language Reference, World Wide Web Consortium (W3C), http://www.w3.org/TR/owl-ref/, (2004).
4. Becket, D., McBride, B.: RDF/ XML Syntax Specification (Revised). W3C recommendation. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/. (2004)
5. Berners-Lee, T., Hendler, J., and Lassila, O. The Semantic Web. Scientific American. pp. 34–43. (2001)
6. IFC Model, Industrial Foundation classes, International Alliance for interoperability, http://www.buildingsmart.com/ (2008)
7. Vanland, R., Nicolle, C., Cruz, C.: IFC and Buildings Lifecycle Management, Journal of Automation in Construction, Elsevier, (2008).
8. Ben-Ari, M.: Principles of the SPIN Model Checker. Springer. ISBN: 978-1-84628-769-5. (2008)
9. Klyne, J. J. C. G.: Resource Description Framework (rdf): Concepts and abstract syntax. Tech. rep., W3C. (2004)
10. Bönström, V., Hinze, A., Schweppe, H.: Storing RDF as a graph. Latin American WWW conference, Santiago, Chile. (2003)
11. Berners-Lee, T. W3C recommandation. http://www.w3.org/DesignIssues/ HTTP-URI. (2007)
12. Berners-Lee, T., Connolly, D.: Notation3 (N3): A readable RDF syntax. W3C recommendation, http://www.w3.org/TeamSubmission/n3/. (2008)
13. Becket, D., McBride, B.: RDF test cases. W3C Working draft. http://www.w3.org/TR/rdf-testcases/ (2004)
14. Katoen, J. P. The principal of Model Checking. University of Twente. (2002)
15. Gueffaz, M., Rampacek, S., Nicolle, C. SCALESEM: Evaluation of Semantic graph based on Model Checking. The 7th International Conference on Web Information Systems and Technologies, WEBIST. Noordwijkerhout, Hollande, May 2011.
16. Tarjan, R. E.: Depth-First search and linear graph algorithm. SIAM Journal of Computing 1, 2, 146-160. (1972)
17. Pnueli, A. The temporal logic of programs. In proc. 18th IEEE Symp. Foundations of Computer Science (FOCS'77), Providence, RI, USA. Pages 46-57. (1977)