Technische Universität München

Department of Informatics

and

Chair of Electronic Design Automation

Interdisciplinary Project

Author:	Gurusiddesha Chandrasekhara
Supervisor:	Prof. Dr.-Ing. Ulf Schlichtmann
Advisor:	Yong Hu
Submission Date:	31.10.15

Contents

1 Introduction

Network on Chip (NoC) is a communication subsystem on an integrated circuit typically between intellectual property(IP) cores in a System on Chip(SoC). In the modern NoCs there are numerous possible topologies and similarly in the integration of IPs for SoCs there can a lot of possible architectures.

In order to find the optimal solution, some algorithms model the system as a graph and then optimize it iteratively. Graph algorithms play an important role in the these optimizations in particular graph transformation or graph rewriting techniques.

Graph transformation concerns the technique of creating a new graph out of an original graph algorithmically. The basic idea is that the, design is represented as graph and the transformations are carried out based on transformation rules. Such rules consist of an original graph, a subgraph which is to be matched on to the original graph and a replacing graph, which will replace the matched subgraph. Formally, a graph rewriting system usually consists of a set of graph rewrite rules of the form $L- > R$, with L being called the pattern graph (or left-hand side, LHS) and R being called the replacement graph (or right hand-side of the rule, RHS). A graph rewrite rule is applied on the host graph by searching for an occurrence of the pattern graph and by replacing the found occurrence by an instance of the replacement graph.

As we apply these series of rules (often interchangeably), the number of resultant graphs increase significantly, which means more designs. Sometimes there can be lot duplicates in the resultant graphs as we apply rules interchangeably. Therefore, we need to identify the potential duplicates in order to avoid unnecessary designs. Usually the rule applying process goes in layers and represents the resultant graphs as nodes of a tree. There should be a logic to identify repeated node(graph in a tree) and merge this resultant node with the existing node. This can be termed as `node merging` problem.

1.1 Tasks

As explained before, to do node merging, so we need to identify duplicates, we need to compare graphs. Now this problem became a exact graph matching or graph isomorphism problem. This calls for a fast, efficient and reliable graph isomorphism algorithm since, each time we apply a rule we get a new graph, which has to be compared with all the previously obtained graphs. For example, we apply a rule and get graph n, which has to be compared with $n-1$ times. For n rules the number of comparisons given by 1.1,

$$n(n-1)/2 \tag{1.1}$$

In addition to node merging, a little work was carried out on creating a Graphical editor for changing and viewing NoCs. The NoC can be modeled using Eclipse Modeling Framework(EMF). Graphical Modeling Framework (GMF) is an Eclipse Modeling Project, project that aims to provide a generative bridge between the Eclipse Modeling Framework and Graphical Editing Framework.

Chapter 2 discusses the details of node-merging such as logic, graph isomorphism problem and algorithm, results. Chapter 3 discusses the introduction and possibility of using GMF in the current project. Chapter 4 gives conclusion and possible future work.

2 Node Merging

2.1 Node Merging Implementation

General process of node merging follows like this. Application of rule gives a new graph and it needs to be compared with the existing tree nodes(graphs). If we find that this node exists already, we just add a edge to that node otherwise create a new node. The corresponding code listing can be seen below.

```
for(E rule: rules){
V newNode = ruleExecutor.execute(node, rule)
if(newNode != null) {
/* For all the older nodes in the tree check if it has any match*/
for(V oldNode:tree.getVertices()){
      if(nodeComparator.compare(oldNode, newNode)){
                 /* If yes then just add an edge to the existing node */
          Integer edgeId = new Integer(tree.getEdgeCount());
           edgeIdToRuleMap.put(edgeId, rule);
           tree.addEdge(edgeId, node, oldNode, EdgeType.DIRECTED);
          flag = true;
          break;
      }
   }
/* If this is a unique graph, create a new node and add it to the tree*/
         if(!flag){
                 Integer edgeId = new Integer(tree.getEdgeCount());
              edgeIdToRuleMap.put(edgeId, rule);
             tree.addVertex(newNode);
                 tree.addEdge(edgeId, node, newNode, EdgeType.DIRECTED);
      }
   }
```

2.2 Graph Matching

Two graphs are said to be isomorphic if they have the vertices connected in the same way. Formally, two graphs G and H with graph vertices $V_n = \{1, 2, ..., n\}$ are said to be isomorphic if there is a permutation p of V_n such that $\{u, v\}$ is in the set of graph edges $E(G)$ iff $\{p(u), p(v)\}$ is in the set of graph edges $E(H)$.

Exact graph matching or graph isomorphism problem is the computational problem of determining whether two finite graphs are isomorphic.graph isomorphism algorithms importance is very high in multiple fields such as electronic design automation, Chemistry and molecular biology etc.

2.2.1 Complexity

Besides its practical importance, the graph isomorphism problem is a curiosity in computational complexity theory as it one of a very small number of problems belonging to NP, neither known to be solvable in polynomial time nor NP-complete. At the same time, isomorphism for many special classes of graphs can be solved in polynomial time and in practice, graph isomorphism can often be solved efficiently. This is a special case of the subgraph isomorphism problem, which is known to be NP-complete. However Laszlo Babai has claimed that the Graph Isomorphism can be solved in quasipolynomial time. Quantities which are exponential in some power of a logarithm are called "quasipolynomial."

2.2.2 Negative checks

Before going into the algorithm, so called negative checks can be performed to avoid the process of going into the matching algorithm.

1. `Vertex number equivalence:` The number of vertices should be equal
2. `Edge Count equivalence:` The sum of the edges should be equal
3. `Edge order check:` This is one of the important checks since, when the graphs don't change much, the edge and vertices numbers may not change. Therefore, sorting the edge arrays and comparing them will give a good candidates for matching algorithm.

2.3 VF2 Algorithm

Lot of algorithms exists to solve graph isomorphism problem, such as Ullman, nauty, VF2 etc. VF2 algorithm written by Italian scientists was chosen. VFLib developed by

```
PROCEDURE Match(s)
    INPUT:  an intermediate state s; the initial state $s_0$ has $M(s_0)=\emptyset$
    OUTPUT: the mappings between the two graphs

    IF $M(s)$ covers all the nodes of $G_2$ THEN
      OUTPUT $M(s)$
    ELSE
      Compute the set $P(s)$ of the pairs candidate for inclusion in $M(s)$
      FOREACH p in $P(s)$
        IF the feasibility rules succeed for the inclusion of p in $M(s)$ THEN
          Compute the state s´ obtained by adding p to $M(s)$
          CALL Match(s')
        END IF
      END FOREACH
      Restore data structures
    END IF
END PROCEDURE Match
```

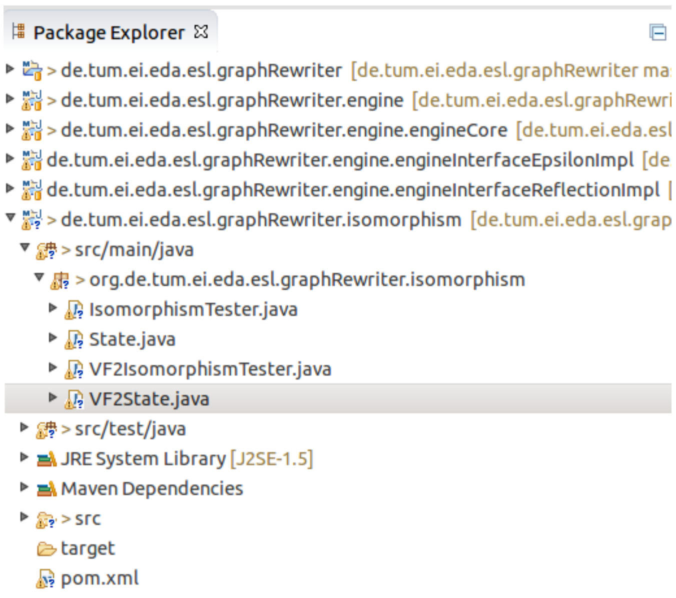
Figure 2.1: VF2 algorithm

same people with the VF2 paper are good point to start writing the custom implementation.

VFLib defines a single interface (State) that a variety of subgraph isomorphism matchers can implement in order for it to work interchangeably. Understand the VF2 algorithm itself was a big challenge. One of the ways to understand any graph matching algorithm is to understand the commonalities among all of the approaches currently used. Below are the some of the listings of some possibilities.

- `Recursion:` Any implementation typically has a method that calls itself
- `State accumulation:` The recursive method gradually building up a map of nodes from source graph to target graph, one pair of nodes at a time. Sometime it fails, so it needs to go back to last successful match (backtrack). When it succeeds and needs to report success.
- `Mapping:` The implementation typically uses an internal map to keep track of what it's done. So getting getting mapped nodes is easy.

The basic idea behind VF2 is that, it tries to find the mapping of vertices between two graphs. This process of finding the mapping function can be suitably described by State Space Representation (SSR). Each state s of the matching process can be associated to a partial mapping solution M(s), which contains only subset of the complete mapping. A transition from a generic states to a successor state s′, represents the addition of a pair



```
Package Explorer
> de.tum.ei.eda.esl.graphRewriter [de.tum.ei.eda.esl.graphRewriter ma
> de.tum.ei.eda.esl.graphRewriter.engine [de.tum.ei.eda.esl.graphRewri
> de.tum.ei.eda.esl.graphRewriter.engine.engineCore [de.tum.ei.eda.esl
de.tum.ei.eda.esl.graphRewriter.engine.engineInterfaceEpsilonImpl [de
de.tum.ei.eda.esl.graphRewriter.engine.engineInterfaceReflectionImpl [
> de.tum.ei.eda.esl.graphRewriter.isomorphism [de.tum.ei.eda.esl.grap
  > src/main/java
    > org.de.tum.ei.eda.esl.graphRewriter.isomorphism
      IsomorphismTester.java
      State.java
      VF2IsomorphismTester.java
      VF2State.java
  > src/test/java
  JRE System Library [J2SE-1.5]
  Maven Dependencies
  > src
  target
  pom.xml
```

Figure 2.2: folder structure

of matched nodes to partial mapping associated to s in the SSR. The algorithm explores the search graph in the SSR according to a depth-first search strategy.

In figure 2.1, the Match(s) procedure plays the role of recursive function, while s and s′ play the dual role of state accumulators and feature comparators. P(s) represents the set of candidates to be added to state s. There are set of feasibility rules can be defined to check the consistency condition. These rules can be both syntactic(depend only on the structure of the graph) and semantic (depends on the attributes of nodes and edges).

2.3.1 Implementation

In the present work basic idea of VF2 is preserved, while lot of modifications were made for the practical purpose. Hereinafter graph1 is mentioned as source graph and graph2 as target graph.

Figure 2.2 shows the VF2 implementation in the current project. As seen in the figure 2.2 IsomorphismTester.java is an interface which defines a function which will be implemented in VF2IsomorphismTester.java.

```
public interface IsomorphismTester {
/*
 * Return true if the graphs are Isomorphic
 */
boolean areIsomorphic(Graph<EObject, EClass> g1, Graph<EObject, EClass> g2);
}
```

State.java is an interface, which corresponds to the state s of the VF2 algorithm. VF2State.java implements State interface.

Before explaining the functions, some of the data structures used are,

```
/*possible candidates to add it to state */
private ArrayList<Pair<EObject>> candidates;

private ArrayList<EObject> sourcePath; // Holds matched vertices of source
private ArrayList<EObject> targetPath; // Holds matched vertices of target

/* holds the matched source nodes for each node in the target graph */
public Map<EObject, Set<EObject>> matchedNodeCandidates;
```

VF2IsomorphismTester.java defines match procedure along with implementing areIsomorphic() function.

```
boolean areIsomorphic(Graph<EObject,EClass> g1, Graph<EObject, EClass> g2)
{
       State state = new VF2State(g1,g2);

       if(!state.checkIndividualMatch()){
              System.out.println("VF2:␣Individual␣nodes␣cann't␣be␣matched");
              return false;
       }
       maps.clear();
       match(state);

       if(!maps.isEmpty()){
              if(maps.get(0).size() == g1.getVertexCount())
              return true;
       }
```

```
        return false;
}
```

An empty state is created initially, in which the initial candidate is selected as opposed to loading every possible vertex match. A little optimization is carried out to identify a uniques start node(vertex should have only one match in source and target). A further optimization would be to check to see if it has unique neighbors.

`state.checkIndividualMatch()`, checks if every vertex can be matched between source and target, this is an interesting check since, for each vertex in source we need to have at least one matching vertex in the target.

After this, match function is called. The result is returned based on the number of mapped vertices.

```
private boolean match(State s) {
                if(s.isGoal()){
                        maps.add(s.getVetexMapping());
                        return true;
                }
                if(s.isDead())
                        return false;

                boolean found = false;
                while(!found && s.hasNextCandidate()){
                        Pair<EObject> candidate = s.nextCandidate();

                        if(s.isFeasiblePair(candidate)) {
                                State nextState = s.nextState(candidate);

                                found = match(nextState);

                                if(nextState.backtrack()){
                                }
                                else{
                                        return false;
                                }
                        }
        }
                return found;
}
```

s.isGoal() returns true if all vertices are mapped.
s.isDead() returns false when vertex number mismatch happens.

After the necessary checks, it choses the initial candidate as our start node pair and checks for feasibility of the candidate. And if it is a feasible candidate a next state is created, with the current candidate will be added to map. And match procedure will be called on the next state. Once we run out of candidates match function returns and checks for backtracking, if it is a successful backtrack matching continues else match will terminate by returning false.

`Feasibility Function Listing`

```
public boolean isFeasiblePair(Pair<EObject> match) {

              if(map.containsKey(match.getFirst())
              || map.containsValue(match.getSecond()))
              {
                     return false;
              }

       Set<EObject> matchedNodes = matchedNodeCandidates.get(match.getSecond());

              if(!(matchedNodes.contains(match.getFirst())))
                            return false;

              if(!immNbrMatch(match)){
                                   return false;
              }
              return true;
              }
```

In original VF2 algorithm there are lot of feasibility rules (semantic and syntactic) are applied before deciding the feasibility. The current graph structure does not lot of checking since we are concerned with node property matching as well. A check is carried out to make sure it is not already mapped and secondly if these candidate data can be matched and lastly if we can map its neighbors. This neighbor checking eliminated lot of potential duplicate candidates and avoid unnecessary backtracking.

`s.nextState()`: `s.nextState` returns a new VF2State, by copying all the previous state values and add the current candidate to the state. Additionally it loads the new candidates for matching, which are the combinations of neighbors of previous candidate.

Backtrack() Function Listing

```
public boolean backtrack() {
                if(sourcePath.isEmpty() || isGoal()){
                        map.clear();
                        return true;
                }
                if(isHeadMapped()){
                        return true;
                }

                EObject last = targetPath.get(targetPath.size()-1);

                if(matchedNodeCandidates.get(last).size() == 1){
                        return false;
                }
                map.clear();

                for(int i = 0; i<sourcePath.size() -1 ; i++)
                {
                        map.put(sourcePath.get(i), targetPath.get(i));
                }
                return true;
        }
```

Our program spends a lot of time backtracking since, this is essential part to find the appropriate match, but if the necessary checks are not carried out this can increase the time significantly. First we check for conditions if its goal or if the mapping hasn't started at all. If the head is mapped, which is when we have mapped the candidate and all of its neighbors successfully, then we safely go back to previous step keeping the current mappings. If these conditions fail, then we remove the last mapped candidate and continue the process.

There is one more important check needs to be carried out to avoid backtracking to a great deal. Since we have matchedNodeCandidates(map of all vertices matches between source and target) which can be used effectively to determine if removal of the last matched candidate and continuing would yield the result. To remove the last successful match, there should be different vertices to match for these candidate vertices. If we don't have any then we need to stop backtracking, since there can be no vertex which can be mapped to candidate vertices.

2.4 Results

The most of the comparisons were eliminated by negative checks and individual node matching step. For graph which has 315 nodes the time taken for successful match is 17 seconds and for unsuccessful match is 12.5 seconds. This is generally good result considering the setup time.

Figure 2.3 shows the resulting graph obtained by node merge.

2.5 conclusion

The implemented algorithm does a good job for exact graph matching, however it can also be used for subgraph isomorphism. The main time consuming part was to understand the algorithm and how it works so that exploiting this for the practical purpose becomes easier.

The main time consuming part is where setting up the array of individual vertex map. Overall it is efficient for practical usage purpose where it exploits the graphs characteristics, however further improvements can be carried out, like making the interfaces to work on generic data structures which will be part of future work and with little changes it can be extended for subgraph isomorphism as well.

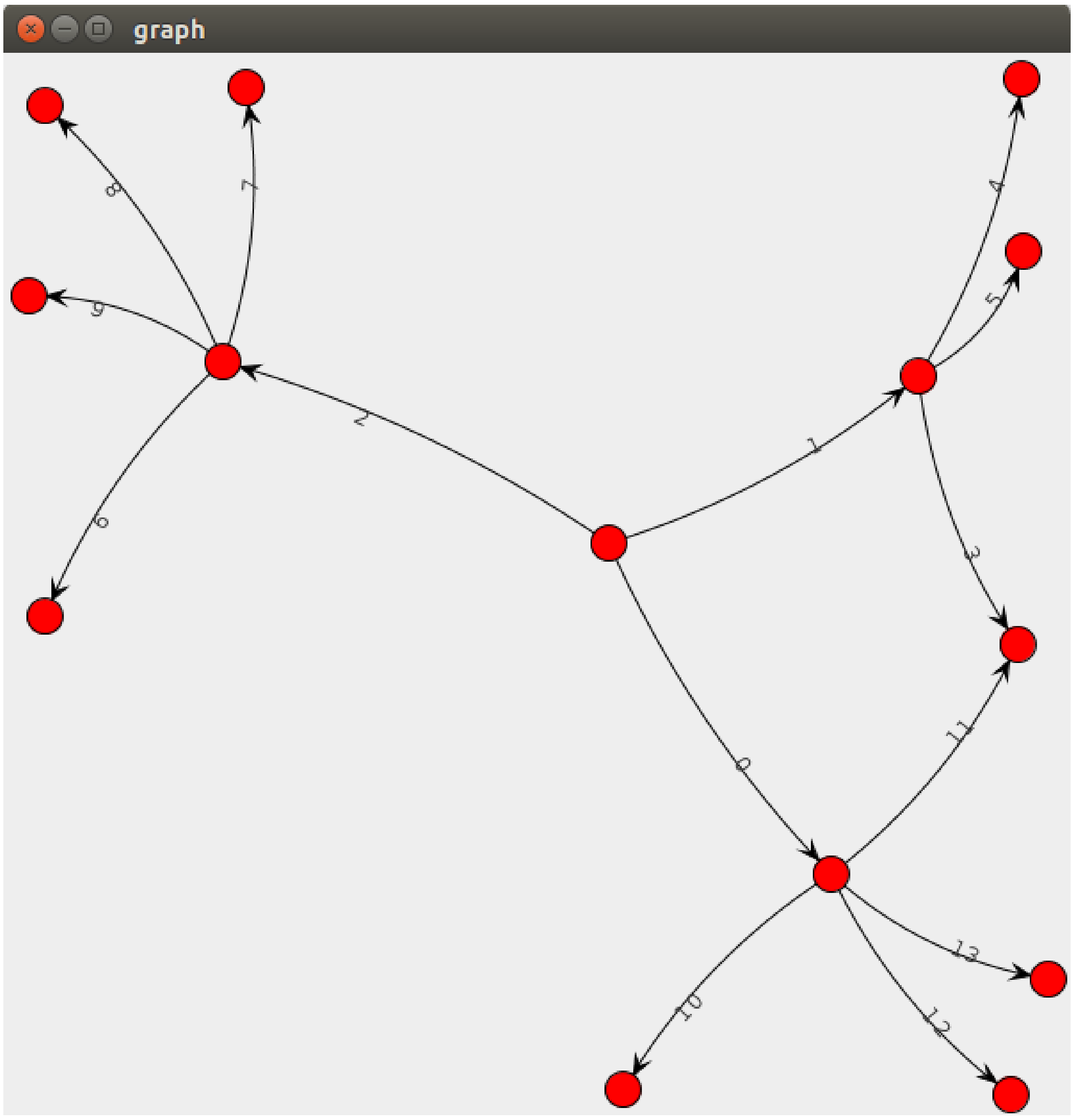


Figure 2.3: Resultant graph after node merge

3 Graphical Modelling Framework

4 Conclusion

List of Figures

List of Tables