Technische Universität München

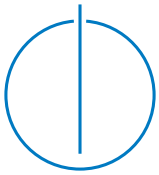
Department of Informatics

and

Chair of Electronic Design Automation

Interdisciplinary Project

Author:	Gurusiddesha Chandrasekhara
Supervisor:	Prof. Dr.-Ing. Ulf Schlichtmann
Advisor:	Yong Hu
Submission Date:	31.12.15



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 [**noc**]. 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, particularly in 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 a 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, called the pattern graph (or left-hand side, LHS) and R, called the replacement graph (or right hand-side of the rule, RHS) [1]. 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 increases significantly, which means more designs. Sometimes there can be 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 a 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

For node merging, we need to compare graphs. Thus, this problem now becomes an exact *graph matching* or *graph isomorphism* problem. This calls for a fast, efficient and reliable graph isomorphism algorithm because a new graph is obtained every time a

[1] `https://en.wikipedia.org/wiki/Graph_rewriting`

new rule is applied, and 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 is given by 1.1,

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

In addition to node merging, options for creating a Graphical editor for changing and viewing NoCs were explored. The NoC can be modeled using the Eclipse Modeling Framework(EMF). The *Graphical Modeling Framework* (GMF) is an Eclipse Modeling Project, and aims to provide a generative bridge between the Eclipse Modeling Framework and Graphical Editing Framework.

This IDP report is structured into four chapters. Chapter 1 provides an overview of the NoC and motivation for using the VF2 algorithm in node merging. Chapter 2 discusses the details of node merging. This includes the graph isomorphism problem, the VF2 algorithm including tracing, the implementation and the results. Chapter 3 explains the GMF, including the possibility of using it in further work, and the GrGen.NET systems. Chapter 4 summarizes the conclusions drawn from this project and the limitations of this project.

2 Node Merging

This chapter explains the process of node merging and the VF2 algorithm in detail.

2.1 Node Merging Implementation

In graph transformation, rules are applied on the host graph and the resulting graphs are represented as nodes of a tree. When the application of a rule gives a new graph (node), it needs to be compared with the existing tree nodes. If that node exists already, an edge is added to that node, otherwise, a new node is created. 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 $E(H)$ [**iso1**].

Exact graph matching or graph isomorphism problem is the computational problem of determining whether two finite graphs are isomorphic. The importance of Graph isomorphism algorithms 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 [**iso2**]. 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" [**iso3**].

2.2.2 Negative checks

Before going into the algorithm, negative checks can be performed to avoid the graph matching algorithm and yield faster results.

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 because 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 good candidates for the matching algorithm.

2.3 VF2 Algorithm

Algorithms such as Ullman, NAUTY and VF2 can be used to solve the graph isomorphism problem. The VF2 algorithm is one of the fastest algorithms for graphs

```
PROCEDURE Match(s)
   INPUT:  an intermediate state s; the initial state s0 has M(s0)=∅
   OUTPUT: the mappings between the two graphs

   IF M(s) covers all the nodes of G2 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

containing many vertices. The software VFLib and the VF2 algorithm are good starting points for writing the custom implementation.

The VFLib defines a single interface (state) that a variety of subgraph isomorphism matchers can implement in order for it to work interchangeably. The challenges faced during this implementation were firstly, to understand the intricate VF2 algorithm and secondly, customizing the algorithm to the current problem. One of the ways to understand any graph matching algorithm is to understand the commonalities among all of the approaches currently used .

A note on the terminology: In graph comparison one graph will be compared to another, can be denoted as graph1 and graph2 or source graph and target graph.

The commonalities among all graph matching algorithms are:[**ric**]

- `Recursion:` Any implementation typically has a method that calls itself
- `State accumulation:` The recursive method gradually builds 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 mapped nodes is easy.

The basic idea behind the VF2 algorithm 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)* **[vf2]**. 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 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. A 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 (depend on the attributes of nodes and edges).

2.3.1 Tracing of VF2 algorithm

This section explains the tracing process of the VF2 algorithm. Consider two simple graphs $g1$ and $g2$ (Figure 2.2) and assume that node 1 is equivalent to a, 2 is equivalent to b and so on, both semantically and syntactically (color property). We have a candidate list (all possible mappings) and a *map*, which has the partial mappings from g1 and g2. Whenever the *map* size is equal to number of vertices in the graph, we stop the algorithm and return success.

```
Successful match:
```

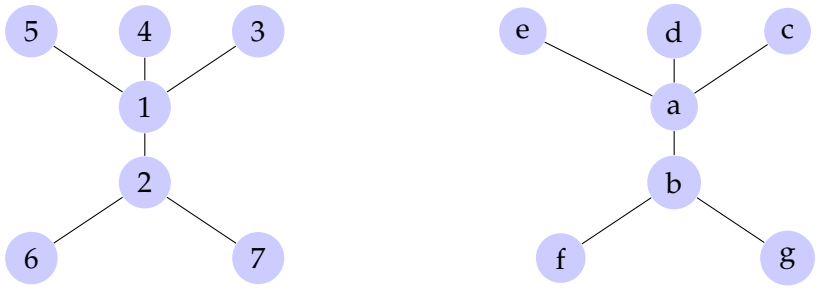


Figure 2.2: Graph g1 (left) and Graph g2 (right) for comparison

Before the start of the algorithm, *map* is *null*. The algorithm consists of the following steps:

- `Step 1:` Create initial state that has all the possible vertex combinations as a candidate list.
  Candidate list for initial match is: $\{(1,a)(1,b)(1,c)...(7,g)\}$

We remove a candidate from the candidate list and check for feasibility. The feasibility rules checks if the current candidate can be added to the *map*.

- `Step 2:` We get a match for (1,a). We add this to *map* and create a new state with a new candidate list (combinations of neighbors of 1 and a).
  map = {(1,a)}
  candidate list = {(5,e) (5,d)(5,b)..... (2,b)}
  Then, the match procedure is called on this new state.
- `Step 3:` A match is found between (5,e). This is added to *map* and a new candidate list is generated and the match procedure is called. *map* = {(1,a) (5,e)}
  candidate list = { }.
- `Step 4:` Now the candidate list is empty, so now it backtracks. In backtracking, it checks if the head(5) and all of its neighbors are mapped (in this case, it is true) and returns to the previous step. Now, the state is denoted as:
  map = {(1,a) (5,e)}
  candidate list = {(4,d)(4,b)..... (2,b)} –> Notice that (5,e) (5,d) etc have been removed
- `Step 5:` Check the feasibility of next candidate and continue until the map size is equal to number of vertices.
  .
  .
  .
- `Step n:` The size of *map* is equal to the number of vertices. Return true.

`Unsuccessful match:`

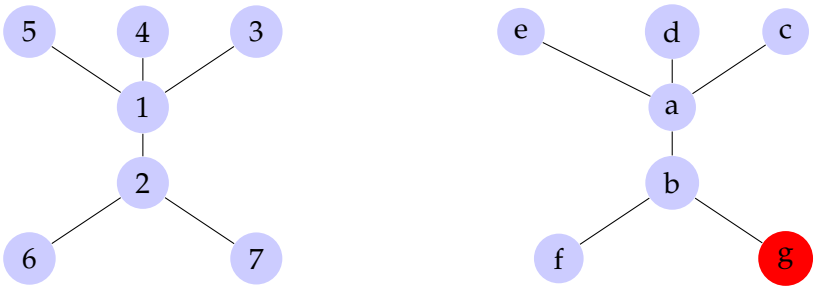


Figure 2.3: Graph g1 (on the left) and Graph g2(on the right) for comparison

Now, we have two graphs of the same structure but differing in their node properties. The algorithm runs like this. Consider we are in a step where the state looks like this,
map = { (1,a) (2,b) (6,f)}
candidate list = {(7,g)}

At this stage, it removes (7,g) from the candidate list to check for feasibility, but it does not match. It then goes back to the previous step (2) and checks if all of its neighbors are mapped. Since nodes 7 and *g* are not mapped, the algorithm removes 2 and 6 from the mapping and continues with a different part of the graph where it hopes to find the mapping for *b*. When all possible mappings are exhausted, the algorithm returns false.

2.3.2 Improvements to original algorithm

A few improvements or changes were done to the original algorithm in order to suit for the current graphs. In the current graphs, we have strict node properties, and node properties should match for the graphs to be equal.

- `Change 1: Using matchedNodeCandidates data structure`

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

  Before we start the match procedure, we check if a node from the target graph has any matching node in the source graph. If we do not find at least one match, then the matching process can be stopped.

- `Change 2: Selecting the start node`
  The traditional implementation takes the combination of all possible vertices as the initial candidate list. An improvement made to select a unique node candidate as the start node, which is a node having only one matching between source and target graph.

- `Change 3: Backtracking`
  Our program spends a lot of time backtracking, as this is an essential part to find the appropriate match. But, if the necessary checks are not carried out, this can increase the time significantly. If the head is mapped, that is when we have mapped the candidate and all of its neighbors successfully, then we safely go back to the 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 that needs to be carried out to avoid backtracking to a great extent. Since we have *matchedNodeCandidates*, we can effectively

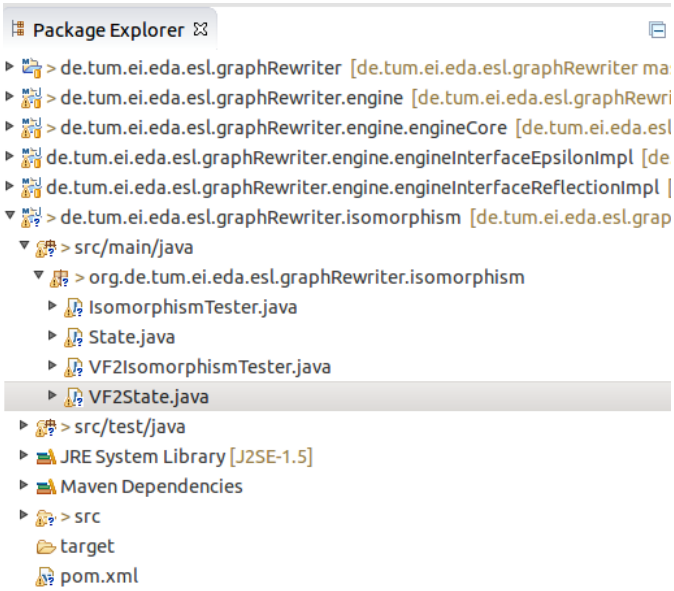


Figure 2.4: folder structure

determine if the removal of the last matched candidate and continuing the algorithm would yield the result. To remove the last successful match, there should be different vertices to match these candidate vertices. If we don't have any, then backtracking should be stopped, since there can be no vertices which can be mapped to the candidate vertices.

2.3.3 Implementation

In the present work, the basic idea of the VF2 algorithm is preserved, while a lot of modifications were made for the practical purpose.
Figure 2.4 shows the VF2 implementation in the current project. As seen in the figure 2.4 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.

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
```

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

```
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 added to *map*. And the match procedure is called on the next state. Once we run out of candidates, the match function returns and checks for backtracking. If it is a successful backtrack, the matching continues. Otherwise, match terminates by returning false.

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

2.4 Results

Most of the comparisons were eliminated by negative checks and the individual node matching step. For a graph having 315 nodes, the time taken for a successful match is 17 seconds and for an unsuccessful match is 12.5 seconds. This is a good result considering the setup time.
Figure 2.5 shows the resulting graph obtained by node merge.

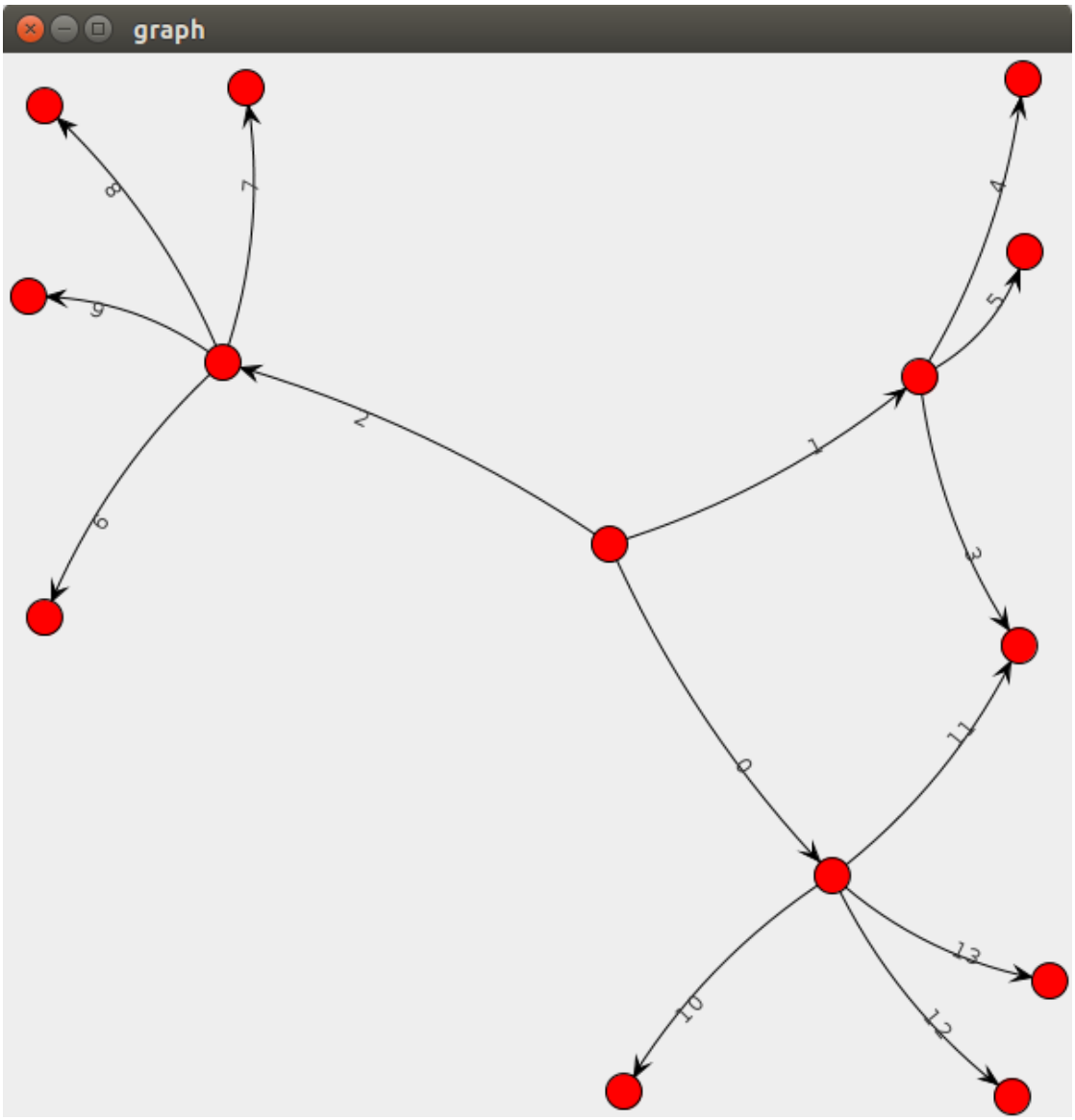


Figure 2.5: Resultant graph after node merge [**vf2**]

3 Other Work

This chapter explains the work carried out on grahical modeling framework and GrGen and the possibility of using them with current graph rewriting project.

3.1 Graphical Modeling Project

The Eclipse Graphical Modeling Project (GMP) provides a set of generative components and runtime infrastructures for developing graphical editors based on EMF and GEF. Since the current graph is built on EMF, it is easy to extend the project to have a graphical editor based on GEF (Graphical Editing Framework). The Graphical Modeling Framework (GMF) Tooling project provides a model-driven approach to generating graphical editors in Eclipse [**gmf**].
The GMF runtime is an industry proven application framework for creating graphical editors using EMF and GEF. With these things in picture a graphical editor can be easily created

3.1.1 GMF-Tooling workflow

The diagram 3.1 illustrates the main components and the work flows of the GMF.
Once creating the initial project the first task is to create a domain model or use an existing domain model(.ecore). Core to GMF is the concept of a graphical definition model. This model contains information related to the graphical elements that will appear in a GEF-based runtime. An optional tooling definition model is used to design the palette and other periphery (menus, toolbars, etc.). Once the appropriate mappings are defined, GMF provides a generator model to allow implementation details to be defined for the generation phase. The production of an editor plug-in based on the generator model will target a final model. GMF provides this work-flow in eclipse making it easy to generate.

3.2 GrGen

GrGen (Graph Rewrite Generator) is a software development tool that offers programming languages optimized for graph structured data, with declarative pattern

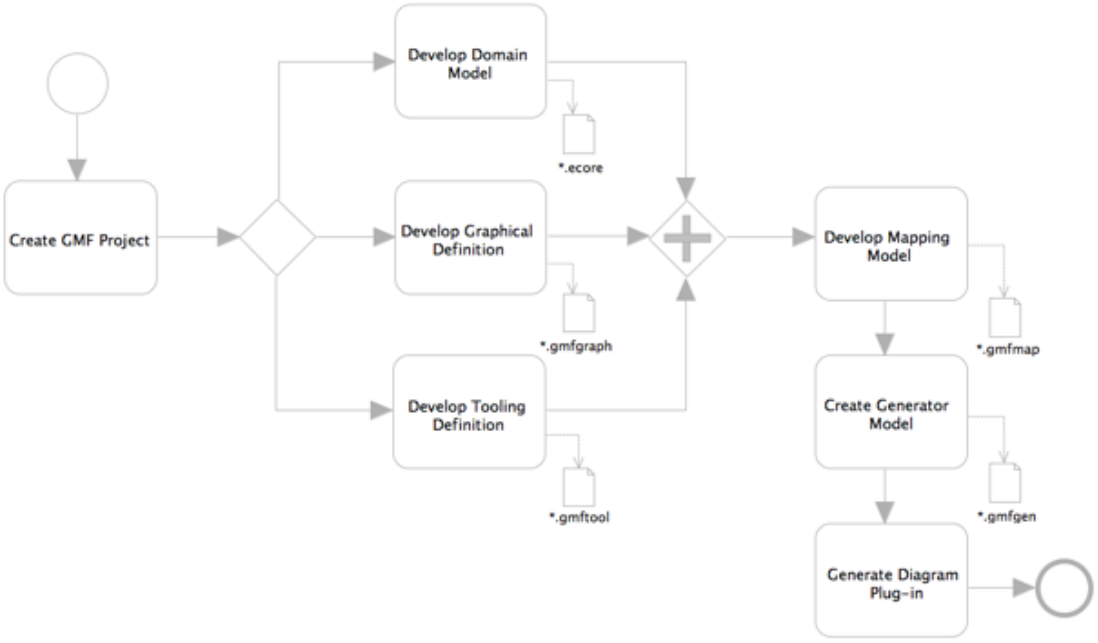


Figure 3.1: Overview of GMF

matching and rewriting at their core, with support for imperative and object-oriented programming, and a bit of database-like query-result processing[**grgen**].
The idea of was to use GrGen for graph transformation. For this requirement was to convert the current graph format(JUNG - Java Universal Network/Graph Framework) to the graph system (explained below) understood by GrGen.NET system and transform the graph and convert it back to JUNG.
GrGen.NET is fast and efficient but it needs a lot of effort to understand and use it.
Graph is represented using a language called graph modelling language(.gm) and rules are defined in *rulelanguage*(*.grg*) with embedded sequences. This rules on the Host graph can be controlled by a Shell script written in GrShell language or C-sharp application. These can be used for interactive debugging with the graph viewer.
The central object is the graph, it adheres to the specified graph model. (In general you have to distinguish between a graph model on the meta level, a host graph created as instance of the graph model, and a statically specified pattern graph of a rule that matches a portion of the host graph at runtime).

- Reasons for consideration of GrGen.NET
  - GrGen.NET offers processing of graph representations at their natural level

of abstraction. It is built on a rich and efficient metamodel implementing multi-graphs, with multiple inheritance on node and edge types.

- It reduces the graph transformation hassles(pointer modifications) which existed in low level languages.
- GrGen.NET is one of the fastest available engines for graph transformation.
- Graphs can be exported to .XMI format. XMI files as written by the Eclipse Modeling Framework (EMF) are a standard format in the model transformation community.

- This project was discontinued because
  - Converting the graphs in to generic language is very complicated process. Since the idea was to convert any given JUNG graph to GrGen format. It is difficult to write a software that writes a another language.
  - In the given time frame of the project it was not feasible to complete the work.
  - The ROI (Return on Investment) was very less.

4 Conclusion

The implemented algorithm does a good job for exact graph matching. However, it can also be used for subgraph isomorphism. Overall, the algorithm is efficient for practical usage, where it exploits the graph's characteristics. Future work can include making the interfaces to work on generic data structures, and extending it to subgraph isomorphism.
Working on this project was a very good insight to the design automation field and also gave a good knowledge of working with graphs and understanding the system as a whole. Node merging and exact graph matching were successfully achieved in this project. The Graphical Modeling Project can now be carried forward to have a very good graphical editor.

List of Figures