FELADATKIÍRÁS

A feladatkiírást a **tanszék saját előírása szerint** vagy a tanszéki adminisztrációban lehet átvenni, és a tanszéki pecséttel ellátott, a tanszékvezető által aláírt lapot kell belefűzni a leadott munkába, vagy a tanszékvezető által elektronikusan jóváhagyott feladatkiírást kell a Diplomaterv Portálról letölteni és a leadott munkába belefűzni (ezen oldal HELYETT, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell megismételni a feladatkiírást.

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Michelberger Tamás

MODELLTRANSZFORMÁCIÓ ELOSZTOTT FUTTATÁSA SZÁMÍTÁSI FELHŐBEN

KONZULENS

Dr. Mezei Gergely

BUDAPEST, 2014

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Michelberger Tamás**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2014. 05. 12.

.....

Michelberger Tamás

Összefoglaló

A diplomámban az elosztott mintaillesztés és modell transzformáció lehetőségét vizsgáltam meg, amely alkalmazható felhő-alapú rendszerekben éppúgy, mint bármely más elosztott környezetben. A modell transzformáció és a modellvezérelt szoftverfejlesztés lassan elér a mindennapi szoftverfejlesztés világába is, ugyanakkor a modellek egyre nagyobbakká válnak, ami egyre nehezebbé teszi a kezelésüket. Hasonlóképpen, a nagy számítási kapacitást igénylő komplex modelltranszformációk is egyre gyakoribbá válnak. Például a modell alapú refaktorálás egy terjedelmes kódbázison vagy egy közösségi oldalakon végzett, arcfelismerést is tartalmazó keresés már könnyen túllépheti egyetlen számítógép számítási teljesítményét. Az elosztott megközelítés lehetőséget ad a korlátok leküzdésére.

Az általam kidolgozott elosztott módszer két lépésből áll: particionálás és mintaillesztés. A particionálás felosztja a modellt kisebb részekre, ezzel könnyítve a feldolgozásukat. Az egyes partíciókat elküldjük a feldolgozó egységekhez, ahol a mintaillesztés történik.

A diplomamunkában áttekintést adok a modellezés, mintaillesztés és modell transzformációk alapvető koncepcióiról. Bemutatok néhány meglévő gráfparticionáló algoritmust, különös tekintettel egy több szintű algoritmusra, ami három lépésből áll és több különálló algoritmust ötvöz. Ezek után rátérek a mintaillesztésre. Az elosztott mintaillesztéshez három különböző stratégiát dolgoztam ki: Az első csak lokálisan keres, a második proxy jelleggel éri el a távoli partíciók tartalmát használ, míg a harmadik részleges találatokat próbál keresni az egyes partíciókban és utána ezek összefűzésével állítja elő az eredményt.

Az elméleti megoldásokat megvalósító rendszer tervezése és létrehozása szintén a diploma részét képezi. A diplomámban részletesen kitérek a megvalósított alkalmazás architektúrájára: hogyan és miért jutottam el bizonyos döntésekig és ezek hogyan befolyásolták a rendszer implementációját.

Munkám során néhány pontban nem várt eredményekre is jutottam, ill. sok tapasztalattal gazdagodtam. Dolgozatomat az elért eredmények ismertetésével, és a jövőbeli tervek bemutatásával zárom.

Abstract

In my thesis, I explored issues and solutions of creating a distributed approach to pattern matching and model transformations applicable in cloud systems, or any other distributed system. Model transformations and model-driven engineering are slowly making their way to mainstream software development. However at the same time models are getting larger and larger, making it hard to reason about them and to work with them. Similarly, model transformations with high computation requirements are getting more and more common. For example, model based refactoring of a huge codebase or facial recognition in a social network can easily exceed the capabilities of a single machine. A distributed can set us free form some of the scalability constraints that the computation power of a single machine imposes.

My distributed approach has two main steps: partitioning and pattern matching. Partitioning divides up the model into smaller ones, making it easier to process. Smaller chunks of the model are then sent out to worker nodes for pattern matching.

As part of my thesis, I explain the basic concepts of modelling, pattern matching and model transformations. I describe some of the existing algorithms for graph partitioning, particularly focusing on a multilevel approach that combines three existing algorithms. I have created three different strategies for pattern matching: a local-only matcher, a proxy-based matcher and a distributed pattern matcher. The distributed matcher can find partial matches in different partitions and combine the results.

Designing and building a *proof of concept* implementation is also part of my thesis. I elaborate the architecture of the application in detail. I explain the most important architectural decisions and how these affect the implementation of the system.

During implementation I encountered some unexpected results and revelations. Finally, I close my thesis by summarizing the results, giving a summary of my work and mentioning some of the main directions for future work.

1 Introduction

Model transformation is becoming more and more prominent as model-driven engineering (MDE) slowly makes its way to mainstream software development. MDE is a discipline in software engineering where models are first-class citizens. MDE aims at developing and maintaining software through model transformations.

In the thesis, I explore the possibility of implementing model transformations in a distributed way. Before going into details about how to create such a system, basics of model transformations and distributed systems are detailed.

1.1 Models and modelling

A model is a simplification of reality. Models are created to better understand the world surrounding us. The real world is too diverse, there are too many unknowns and variables to use the world as it is, when we try to tackle a problem. A model uses the right level of abstraction to hide unimportant details and highlight the ones that really matter.

Modelling is the act when we try to define a segment of reality. For example, if I am building a system to manage car-pooling, car would be an atomic entity. There is no need for us to care about the exact details of a car. However, when I am building a program for car mechanics I would have to use a more detailed car model including the engine and tyres, etc.

Modelling is about trade-offs. When we create a model to gain a better understanding of a problem, we have to decide how detailed the model should be. Just as the previous example shows, we can model the same information on different levels. Deciding which parts of the model should be more accurate and which can stay on rudimentary level is the most difficult part of modelling.

We can model anything, which means we can also build a model for our models. These models are called *meta-models*. One possible realization of a meta-model is the instance model. Usually when we are talking about a model we mean the *instance* model, not the meta-model. Maybe the best-known example for meta-models is the UML class diagram.

We could create a model for creating meta-models and continue it as long as we like, but usually there is no practical benefit for this. Usually, two levels of modelling provide enough expressiveness.

1.2 What is model transformation?

When we are working with models, they change inevitably and model transformation is an automatable way to keep these changes under control and keep the model in a consistent state. Model consistency means that the instance model conforms to the rules set by the meta-model.

In the narrowest terms, model transformation only means the modification of the model. On the other hand, if we look at a broader interpretation, pattern matching becomes part of the transformation.

When we try to find a predefined pattern in a model, the process is called *pattern matching*. The pattern refers to an instance model and it usually has the same meta-model as our *regular* model. After the pattern is defined, we can search our model and identify which parts of our model are matching the provided pattern. This search can be an exhausting search to find all the matching sub-models or it can stop after a certain number of matches, most likely after the first one.

After finding matches the next step is the transformation phase. Along with the pattern, we have to provide the modification to be applied by the transformation. These modifications can range from simple ones such as changing a property in an object to complex model rewriting algorithms. A transformation can add new objects or delete old ones, it also can create or remove connections between objects.

1.3 Distributed systems

Definition of distributed systems as Leslie Lamport states in *Time, Clocks, and the Ordering of Events in a Distributed System*: “A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages.” [1] He also says: “A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.” [1]

These conditions are easily satisfied by a system which has multiple components located on networked computers. The components communicate over the network using any kind of messages (usually something over the IP protocol). In this case, passing messages around is usually a few orders of magnitude more expensive than any operation on a local instance.

Communication costs could seem like a high price to pay, but the benefits of a distributed system can outweighs this. The benefits mainly are parallelization and scalability.

When multiple components work on the same problem toward a common goal and they do it at the same time, it is called parallel computing. For example, a simple linear search can be significantly speeded up if we cut up the search space into multiple sub-spaces and search those at the same time. Distributed systems exhibit this property by default.

Distributed systems can naturally scale very well. A single processing unit can only be so fast, but applying multiple units theoretically have no limits. Using multiple processing units at the same time is called horizontal scaling.

A distributed system has autonomous computational entities. These are independent of each other and the failure of one does not necessarily mean that the whole system fails. Distributed systems can be very robust, if they are well written. Usually, the structure (topology) of the system is not known in advance. The system can consist of different kinds of computers and network links as long as they all “speak” the same *language*.

The memory model is also distinctive. Each component has only a limited view of the whole system. Also, each component has its own private memory where it stores its knowledge about the system. This is why components have to communicate using messages instead of just sharing their memory.

Examples of distributed systems vary from SOA-based systems to massively multiplayer online games to peer-to-peer applications. Maybe the best known example is the Internet itself. That is actually a shining example of it: it proves every day that distributed systems worth the extra effort.

1.4 Motivation

Distributed systems are complex, hard to develop, hard to understand and even harder to maintain. On top of this, there are capable and performant implementations for pattern matching and model transformation. The obvious question presents itself: why do we want to create a system for distributed model transformations?

Applying a model transformation in a distributed fashion makes things more complicated, but as models are getting bigger, the computational power of a single machine might not be sufficient. When we utilize model transformations they quickly becomes an integral part of our daily work, so having a quick feedback-cycle is essential.

Refactoring old source code is something that is very familiar for every software developer: we practice it every day. Some of the codebases grow so large that refactoring becomes mind-bogglingly difficult. Creating a model for the source code and applying refactoring algorithms as model transformation, can have a huge impact. This is where we could use some of the benefits of a distributed system.

Sometimes the model is so big, it cannot fit into a single memory of computer. Imagine we run some kind of image processing (e.g. face recognition or image manipulation) as part of the transformation. Images can be quite big and when we have a few million model objects we may run out of memory.

A distributed implementation could possibly overcome these issues. Using multiple machines has the promise of better performance and scalability.

1.5 Covered topics

In my thesis, I am going explore the possibility of implementing pattern matching in a distributed way, but not the transformation itself. Building an end-to-end system which is not only capable of finding a pattern in a model, but also to transform it is not a one-man effort. As I have worked alone on this project, I am focusing on building an independent, but working component instead of a half-complete, but whole system.

Throughout the next chapters, I am going to explain how I am approaching the problem. How I destructured this complex problem, what were the challenges when

trying to create a flexible enough architecture and how I created a distributed pattern matching system.

2 Plan overview

In this chapter, I am going to cover how I was approaching the problem of distributed pattern matching and model transformation.

As previously mentioned, the main advantage of distributed systems lies in parallel execution. We would not get better results, if we would simply make several machines to work on the same problem - we would get the same result multiple times. To utilize the full power of distributed systems, we need to cut up the problem space and run only a smaller part of the calculation on a single machine. Usually cutting up a problem space does not automatically result in a new set of completely independent smaller problems, therefore the newly created smaller parts needs to be aware of each other.

2.1 Model representation

Every model representation can be distilled into two parts: model entities and their connections. A model entity has a set of properties and can take basically any form or shape, but they can always be connected to other model entities. Models can be translated naturally into graphs, where entities are represented by vertices and their connections are by edges. A vertex in a graph can be anything, thus the representation does not impose any restriction on our model.

Using graphs has a great advantage: most of the mechanisms and structure we want to describe with a model, can be translated into a graph "operation". For example, model transformation becomes a graph transformation and pattern matching becomes isomorphic sub-graph matching.

2.2 Pattern matching in a distributed fashion

In most of the graph transformation frameworks, we have the advantage of having access to the whole graph at any time. This is achieved easily, since transformations are applied usually on a single computer. In a distributed environment, a worker process has access only to a small piece of information. The worker can acquire more by asking other workers, but that is a costly operation compared to the (local) calculation.

In my thesis, I am assuming that workers are located on separate machines located in the cloud and can only be accessed via a network connection. It is essential to minimize the network communication between the workers, because it is at least an order of magnitude slower than accessing information from the memory of the instance.

Before we could start any actual work on distributed worker nodes, we need to cut up the model and send it to the worker nodes. This process is called *partitioning*. It is not enough to just divide up the model into multiple partitions. Partitioning has to create *good* partitions to effectively benefit from parallel execution of transformations. Good partitions are highly cohesive and have the least amount of outgoing edges.

Partitioning needs to access to the whole model. This feature could be problematic since our premise is that the model cannot fit into the memory of a single machine. Fortunately, partitioning needs to know only the structure of the model, so we can reduce every entity into a unique identifier.

When we have our partitions, we can start working on the actual task. In our case this is pattern matching. At first glance, applying pattern matching is quite easy. We have a pattern which is also a graph and we also have a matcher implementation which tries to find a matching sub-graph in the model. We start a few machines and load the partitions but things get complicated when a match crosses the partition borders.

I am going to explore three different strategies for distributed pattern matching:

1. Local only: we ignore all matches that cross partition boundaries.
2. Proxy-based: remote nodes are accessed transparently.
3. Partial matching: we search locally for partial matches and delegate remote parts to other instances.

When we consider both part of the problem (partitioning and pattern matching), the whole process can be summarized in four steps:

1. Partition the model.
2. Send out partitions to a set of machines for processing.
3. Try to find a match for a predefined pattern.
4. Report back with the result.

It is important to note that the indented behaviour is to partition the model once (since it is an expensive operation) and use it multiple times for finding matches for different patterns.

2.3 Proof of concept

A proof of concept implementation is a realization of an idea, it demonstrates its feasibility for real world usage. Usually it is small and incomplete. A proof of concept is a "cheap" way to explore ideas before real production begins. It can scale from a single algorithm to a whole system. In software development it usually means a short-lived piece of software to explore a new feature or a product's market-fit.

In my thesis, I am only going to provide a proof of concept implementation for distributed pattern matching. My goal is to determine whether it is a viable alternative of the current methods. Moreover the purpose behind is to explore advantages for what it is worth pursuing a production ready implementation in the future.

As a proof of concept implementation is not going to cover all possible cases, it focuses on the "happy path". There will not be any safety measures which are a must for any production system: no authentication, authorization, fault tolerance or failover support. However, the approach can be extended later by these features by request.

3 Architecture

This chapter discusses the architecture of the system. The evolution of the architecture is detailed from an idea to the actual implementation. It is also discussed what the driving force was behind my decisions and how the parts connect together.

3.1 Goals

The most important goal in creating the architecture was to keep it as flexible as possible. Extensibility was emphasized from the beginning, since this system is designed to work with different models and data sources. Every model has its own special needs that have to be taken into account when trying to create an efficient pattern matcher. For example, certain properties of the model can affect the order in which the nodes should be visited during pattern matching. With these in mind, the architecture follows the semantics of a plug-in system: those parts of the system which deal directly with user supported data, can be redefined and tuned for the exact use-case. This is important because the shape of the graph representing the model can determine which partitioning algorithm should be used or what the best order is in which the nodes should be matched to the pattern.

To achieve the required flexibility, the system has to be more of a framework rather than an actual application. The framework has to provide a rich set of APIs and all the management tasks that come with a distributed system. Also it has to provide an easy way to swap out the default implementation with an alternative one which fits better for a particular use-case.

3.2 Overview

To create a plug-in architecture, the system has to be highly modular and decoupled. One way to achieve this is through the heavy use of interfaces. Most parts of the framework do not know how others work: each part of the system is hidden behind an interface. Here, an interface can refer to an actual C# interface or a REST API of a service.

Early on, I decided to use interfaces where possible, but I kept making a mistake: I tried to design an interface before knowing what I will use it for. During

development I had to revisit several interfaces and make major adjustments to accommodate the actual needs. I tried to create a framework on its own, instead of extracting it from an existing system. Now I can see, why this was a particularly bad direction: I succeeded to overcome most of the limitations of my preliminary design, but if I had to start over, I would choose another way to enforce the separation of interfaces and implementations. I would not create so many separate assemblies and I would focus on extensibility instead of replaceability.

The system can be divided into three major parts:

- a partitioning module
- a pattern matching module
- and the glue that holds these together

Partitioning and pattern matching can be completely separated from other parts of the system and it is important to do so because these parts are dependent on the model and they have to be replaceable when necessary. This is especially true for the matching component because every pattern needs its own matcher implementation.

To distribute the workload I used a hybrid master-slave peer-to-peer approach. The master is the coordinator, but it does not have complete control over the system. It only initiates the process: it serves as an entry point. The slave modules are the actual workers. They form a peer-to-peer network: when a slave module needs some data, it turns to the other slaves directly instead of the master.

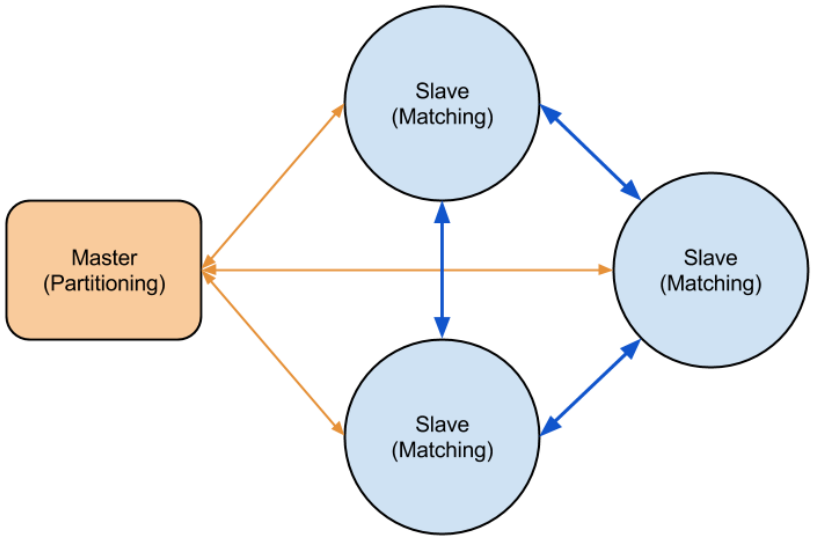

Figure 3-1: system overview

Human users of the system can only interact with the master module. The slaves are hidden, and cannot be reached directly. Every command comes from the master and every report gets back to it. Also the master is the only entity that knows all the slave modules. When a slave needs to contact another module, first it has to go through the master to obtain the address of the target module. The centralization simplifies the system itself and the protocol used between the module instances, but introduces a single point of failure. The following figure shows how the master translates command to messages known by the system and distributes it among the slave modules.

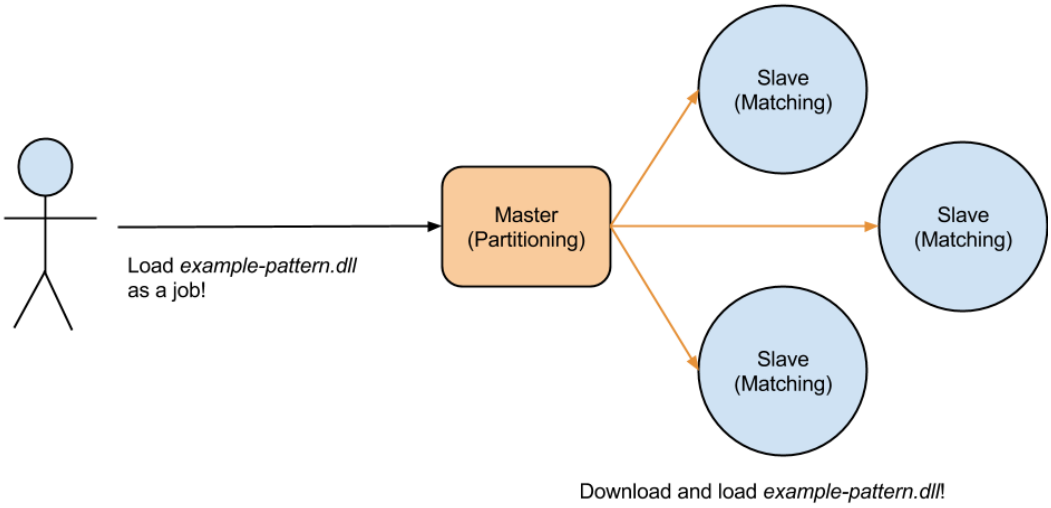

Figure 3-2: human interaction

The master is responsible for partitioning the model, assigning created partitions to slaves, starting and stopping the matching process. The only job of the slaves is to run the provided pattern matcher implementation on the assigned partition and to communicate with each other if necessary.

Earlier, the third part was referred to as the *glue*. This glue consists of components that are not directly related to the subject of my thesis. The components are responsible for the communication between the master and slave modules, loading and saving data and also for the handling the user interaction.

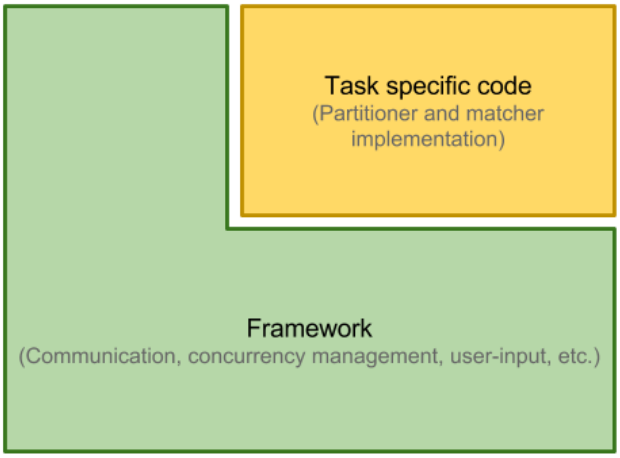

Figure 3-3: module high-level overview

The partitioning module and the master module greatly overlap each other and this is also true for the pattern matching and the slave modules. This is the reason why I used these concepts interchangeably before. From now on, I reference to the master module as the *partitioning module* and to the slave module as the *matching module*.

3.3 Project structure

The ultimate goals of the architecture are extensibility and modularity. These properties are reflected in the project structure, therefore I am going to present the architecture by explaining how each sub-project interact with others.

It was mentioned earlier that one way of decoupling code is to introduce interfaces. I followed this path when designing the architecture: interfaces and implementations are completely separated. In my case this means that there are interface assemblies and implementation assemblies. Interfaces are not mixed with implementation on any level. I maintained a clear boundary between them.

This separation gives us the opportunity to free ourselves from the implementation details when using a particular component. For example, when using the graph partitioning algorithm, we do not have to know how it works on the inside, it can be treated as a black-box. This also means that the implementation cannot be referenced directly. We have to use *inversion of control* to satisfy dependencies without knowing anything about their actual implementations.

The ideal model of this is the following:

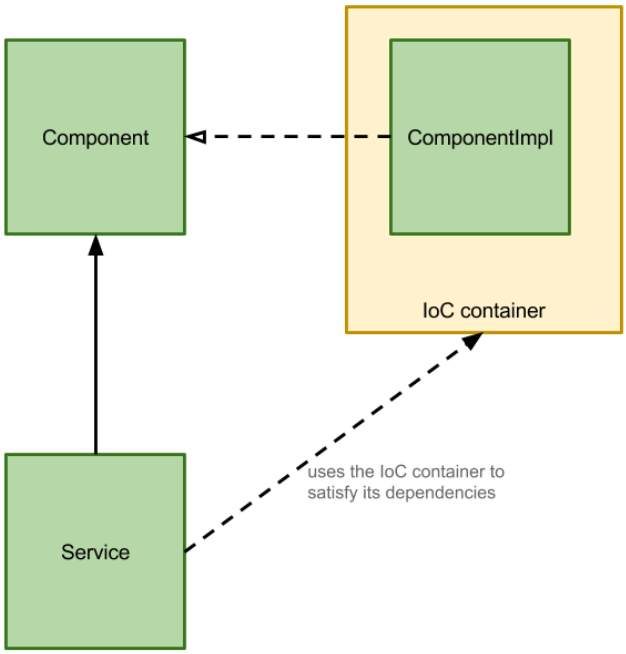

Figure 3-4: inversion of control

Figure 3-4 is too idealistic to be used in real life. It assumes that every component is self-contained and almost completely independent. For the first version of the architecture I used a slightly modified version of this model. Figure 3-5 includes the domain specific projects only. None of the management or application projects are listed.

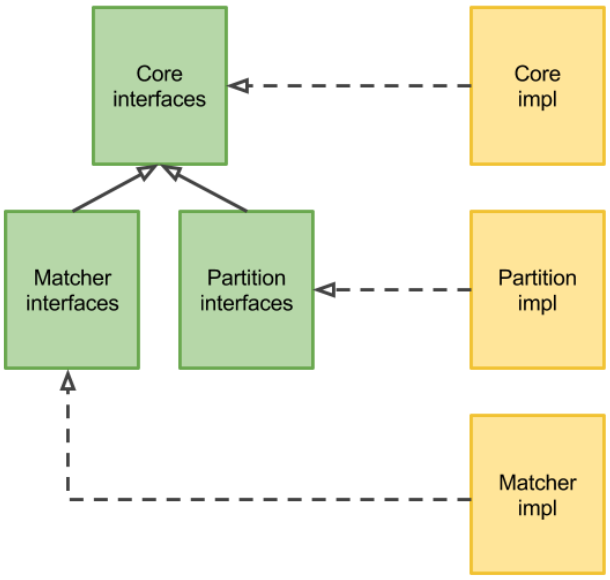

Figure 3-5: first version of the architecture

The boxes on Figure 3-5 represent assemblies, also known as projects (as Visual Studio refers to them). I am going to use the later naming convention from now on.

The *Core interfaces* project contains the common and domain model interfaces. The *Partition interfaces* project is for the partitioning algorithm and finally the *Matcher interfaces* contains the pattern matcher's framework facing interfaces. Further explanation of inner mechanisms of these projects can be found in the next chapter.

As Figure 3-5 shows, the implementation components are not aware of each other, they have access only to the interfaces. This helps in achieving the desired extensibility and flexibility: every implementation component or even a single class of the implementation can be replaced with a new one without affecting any other components. This mechanism works well for some parts, such as the matcher or partition components because these are "end" implementations - no one depends on them.

On the other hand, Core interfaces and implementations are building block we use in the "end" implementations and thus supposed to be reused. Core mostly contains domain classes (e.g. Node and Edge) which implement common functionality. These functions are the same in every parts of the system. For example a Node has multiple edges and can connect to other nodes. The implementation of this feature does not change between the partitioning and a matcher modules.

The first version of the architecture did not allow for this kind of reusability. An interface in *Partition* could extend an interface from *Core*, but they could not share the implementation. This goes against the very basic of object-oriented design. Internally I would have liked to reuse the implementation of *Core* in *Partition*, thus I modified the architecture to accommodate this. The new, improved version can be seen on Figure 3-6

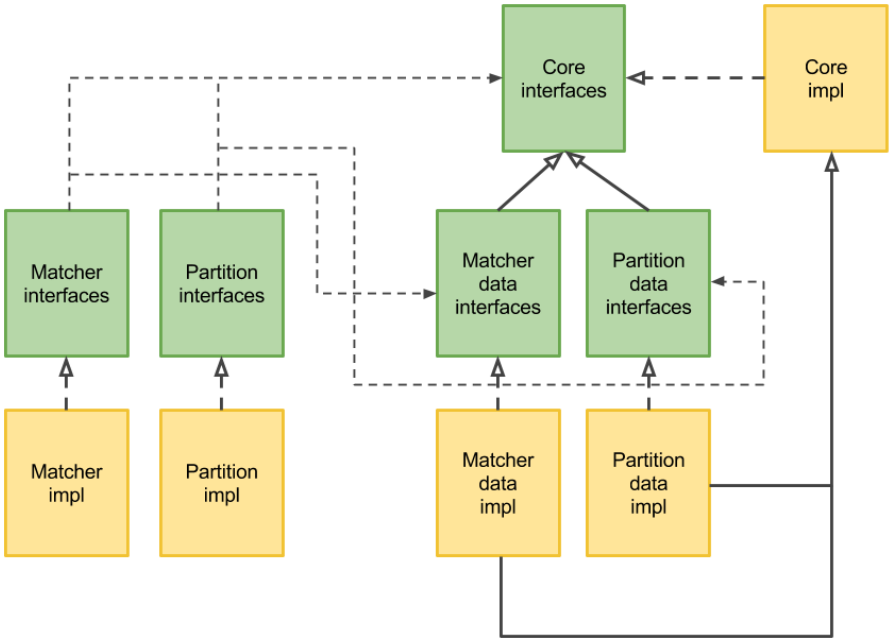

Figure 3-6: improved architecture

As Core mostly contains data classes, it was fairly straightforward to separate the data and service interfaces not only on the implementation but on the interface level as well. This separation allows us to reuse implementation classes from Core without sacrificing flexibility. In the *Partition Data* project we can implement data interfaces by reusing the *Core* classes and at the same time we can replace even the single (service) classes in the *Partition* project with our implementation.

As next, it is explained how partitioning and pattern matching are used by the application itself: The master and slave module have their own executable. They are actually self-contained class libraries with a single entry point and an executable facade. This has the advantage of easily changing the host environment around them. Currently, the system can only be run as a console application, but there are plans to run it in an Azure [2] worker role in the cloud.

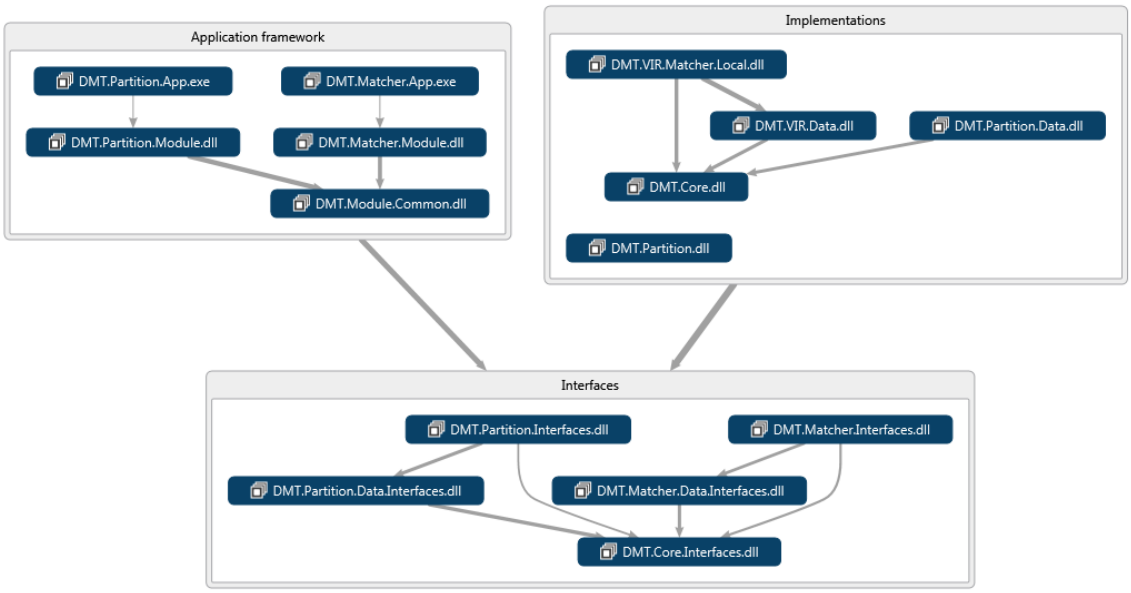

Figure 3-7: dependency of assemblies

3.4 Domain model

Pattern matching and model transformation requires a model to operate on. This model has to be represented somehow in our system and a graph representation comes very naturally. Every model object can be translated into a node and the connection between the objects can be represented with edges.

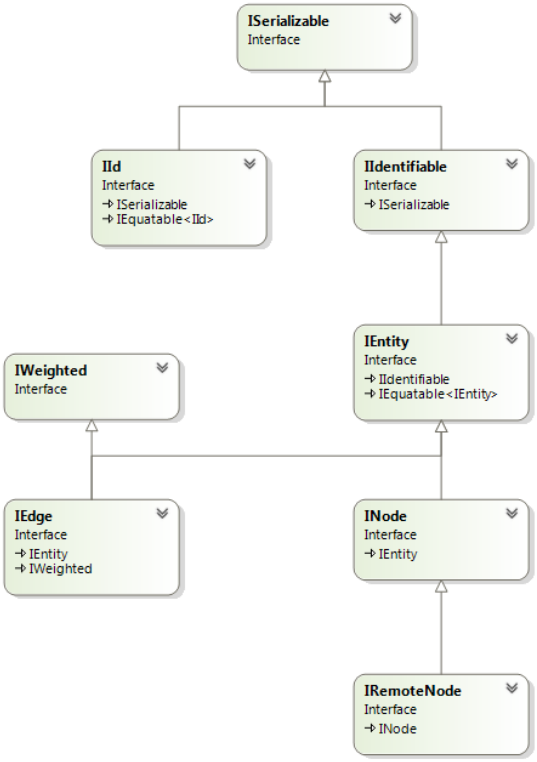

Figure 3-8: data interface hiearchy

Figure 3-8 shows all the data interfaces. Everything must be serializable because we need to distribute the model between the matcher module instances. The two main interfaces are `INode` and `IEdge` which represents nodes and edges respectively. `IRemoteNode` represents a node when it is not available in a given partition.

An edge has always exactly two endpoints; the endpoints are the nodes that the edge connects. Edges also have a direction property. This is necessary because the graph can be directed and the node stores the edges in a single collection.

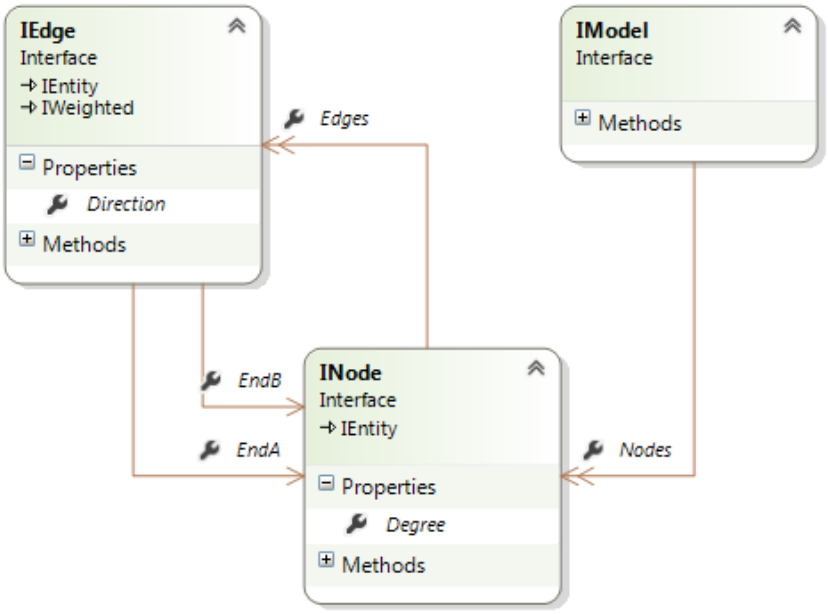

Figure 3-9: interfaces for graph representation

Nodes and edges are not enough to represent a graph. There are several graph representations. I have chosen the incidence list to represent the graph. This can be seen on Figure 3-9. Each node stores its incident edges, and each edge stores its incident nodes. Finally, a list of nodes is stored in the model. I have chosen this representation over a matrix because this allows me to store extra information in node and edge objects. This extra information can be any kind of data relevant to the model, e.g. a name of person or price of a car.

3.5 Where are the limits of the framework?

This is one of the questions that we have to ask ourselves continuously and at the same time one of the hardest to answer. A framework that wants to solve every corner case is on its way to become bloatware. This is especially true in this project: every model and pattern has different requirements.

In my case, the framework provides only the glue between the pattern matcher and the partitioner. The framework coordinates the system, but almost every domain specific task is delegated to user-provided code. This is especially true for the pattern matcher. The matcher is completely transparent from the system's point of view. The framework only needs a binary that contains implementation for certain interfaces. Further details are presented in the next chapter.

On the other hand, partitioning is a more general problem. When partitioning, a model is handled as a graph and it loses all of its domain specific semantics. The partitioning algorithm only needs to know about nodes and edges and these elements can be found in every model. This lets the framework to have finer grained control over the partitioning process. In practice this means that the partitioner has several interfaces and the framework has greater control over what happens.

3.6 Inversion of control

Inversion of Control [3] (IoC) describes a design where custom-written components of the system receive the flow of control from a generic, reusable library. In most of the cases inversion of control is seen as a defining characteristic of a framework. As the name suggests, with this design we can invert the control of a program. In traditional procedural programming our custom-written code calls a reusable library or framework to take of a generic task, but with inversion of control the reusable library or framework calls the custom code that we wrote.

The following example illustrates how inversion of control works on a micro-scale.

```
class RemoteOrderService : IOrderService {
     public void Send(Order order) {
          // ...
     }
}
class Order {
     private IOrderService service;
     public Order() {
          this.service = new RemoteOrderService();
     }
     public void Send() {
          // preprocess order ...
          this.service.Send(this);
     }
}
new Order().Send();
```

In this code-snippet, we can see that the Order class depends on the OrderService but the implementation is hard coded into the Order class. This way, we have no control over what kind of order service we use. With a slight modification this class can be made more flexible.

```
class RemoteOrderService : IOrderService {
     public void Send(Order order) {
               // ...
     }
}
class LocalOrderService : IOrderService {
     public void Send(Order order) {
          // ...
     }
}
class Order {
     private IOrderService service;
     public Order(IOrderService service) {
          this.service = service;
     }
     public void Send() {
          // pre-process order ...
          this.service.Send(this);
     }
}
new Order(new RemoteOrderService()).Send();
new Order(new LocalOrderService()).Send();
```

In this second example, the Order class takes a constructor argument and stores it for later usage. This solution decouples the implementation of IOrderService from the Order class. As the example shows, we can use another service without touching the Order class.

Inversion of control is a powerful principle to decouple interfaces from implementations. One of the main benefits of this design is that it helps to enforce the single responsibility principal. The principle states that every class should have a single responsibility. It also helps to modularize code and promotes the use of contracts. This means that only the interface of a component is known from the outside, the inner workings of a component stay hidden.

Inversion of control allows for better (unit) testing. As it promotes modular design, the dependencies of classes can be easily mocked or stubbed out, so the class can be tested in total isolation.

3.6.1 Dependency injection

There are several techniques to implement inversion of control from quite simple cases such as the factory pattern to complex solutions such as a container-based dependency injection.

Dependency injection in its simplest form is when dependencies of an object are satisfied from outside and the dependency is made part of the object's state. This means that an object must not (and sometime is not even able to) instantiate the service objects that it uses. These services have to be passed to the client object.

Dependency injection is not a concrete implementation. It has several types. The code example above shows how we can implement constructor injection manually. There are other types as well: parameter injection and setter injection. There are frameworks for almost every programming language to facilitate some kind of dependency injection. In the .NET world, the easiest way is to use the Managed Extensibility Framework (MEF) [4] which ships with the standard library.

MEF makes heavy use of attributes. The two main attributes are `ImportAttribute` and `ExportAttribte`. MEF uses a container in the background which loads the exported types and instantiates them. When an object needs a dependency, it marks a property or a field to be imported and the MEF container takes care of everything else.

Assuming we have set up the container correctly, the following code example shows how to use MEF:

```
public interface INameService {
     string GetName();
}
[Export(typeof(IService))]
class NameServiceImpl : INameService {
     public string GetName() {
          // do what needs to be done
          return "Joe";
     }
}
class Greeter {
     [Import]
     public INameService Service { get; set; }
     public void SayHello() {
          Console.WriteLine("Hello, " + Service.GetName());
     }
}
// ...
CompositionContainer container = ...;
```

```
var greeter = new Greeter();
container.ComposeParts(greeter);
```

3.7 Use of dependency injection

Dependency injection is an easy way to achieve the flexibility that I aimed for. Separating interfaces from implementations is the first step in preparation for the use of dependency injection. I wanted to design a system where the reference implementation can be safely and completely replaced with another one. Therefore, none of the implementation assemblies are directly referenced by the application. On Figure 3-10 I grouped the assemblies for better understanding, but it clearly shows that almost every project references the interfaces and none of them references the implementations.

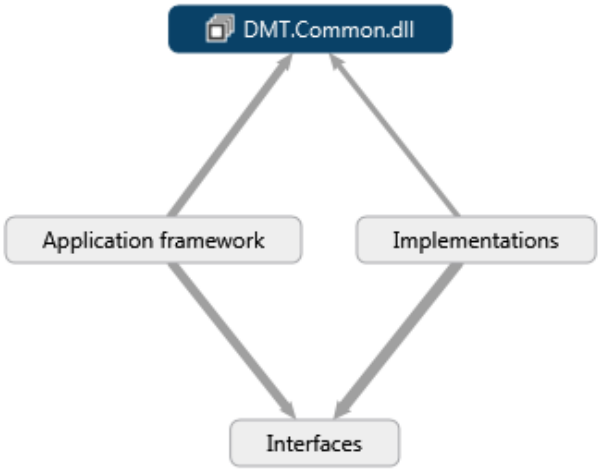

Figure 3-10: dependency graph of project assemblies

If there is no direct reference to assemblies, how can we use them? This is where MEF helps us out. In .NET dynamically linking libraries is fairly easy and MEF makes it even easier. I set up a `CompositionService` class which is responsible for populating the composition container and injecting imports into newly created objects.

I created two levels for plug-ins. The first one is for the default implementation. This was written by me as part of my thesis. The default implementations can be easily overridden by plug-ins. This is the second level. When we want to provide an alternate implementation, all we have to do is to implement the desired interfaces and copy the resulting assembly (and its dependencies) to the applications `Plugins` folder. On the next start up the application will pick up the new assemblies and the imports will be satisfied from the newly provided implementation.

3.8 Problems faces during planning and implementation

The first problem I hit during implementation was that vertical (inheritance) and horizontal (plug-ins) extensibility seemed to be incompatible with each other. Implementations were locked in their own assemblies and the classes were hidden from the outside world. There was no way to reuse them as base classes in other assemblies. This forced me to rethink the architecture of the application. This led to separating data and service interfaces as shown on Figure 3-6.

The new architecture solved the reusability problem, but introduced new ones instead. The most obvious was that it made the system more complex and harder to understand. It also has a hidden danger that loading and importing classes from different sources can cause inconsistencies in the system. The following code is an example for this:

```
// In Core interfaces
interface ICoreService {
     void CreateNode();
}
// In Extension interfaces
interface IExtensionService : ICoreService {}
// In Core implementation
[Export(typeof(ICoreService))]
class CoreImpl : ICoreService { ... }
// In Extension implementation
[Export(typeof(IExtensionService))]
class ExtensionImpl : CoreImpl { ... }
// In Plug-in implementation
// third party re-implements and re-exports ICoreService
[Export(typeof(ICoreService))]
class PluginCore : ICoreService { ... }
// example usage introduces the inconsistency
class SomeClass {
     [Import]
     ICoreService core;
     [Import]
     IExtensionService extension;
     void SomeMethod() {
          core.CreateNode();
          extension.CreateNode();
     }
}
```

The solution represented by the code can introduce very subtle and hard to recognize bugs in the system. We would expect the `CreateNode` method to return the same type of node for `core.CreateNode()` and `extension.CreateNode()`. However to our surprise, these two methods would return two different types of nodes because

`extension` inherited its `CreateNode` method from the core implementation and that was unchanged when the third-party plug-in library re-exported the `ICoreService` interface.

Unfortunately, I did not find a satisfactory solution which would solve this issue. The only way to avoid the issue is to restrain ourselves from extending service interfaces and we should prefer composition over inheritance.

The improved version can be seen below:

```
// In Core interfaces
interface ICoreService {
     void CreateNode();
}
// In Extension interfaces
interface IExtensionService {}
// In Core implementation
[Export(typeof(ICoreService))]
class CoreImpl : ICoreService { ... }
// In Extension implementation
[Export(typeof(ICoreService))]
[Export(typeof(IExtensionService))]
class ExtensionImpl : CoreImpl, ICoreService, IExtensionService { ... }
// In Plug-in implementation
// third party re-implements and re-exports ICoreService
[Export(typeof(ICoreService))]
class PluginCore : ICoreService { ... }
// example usage introduces the inconsistency
class SomeClass {
     // Will be imported from
     [Import]
     ICoreService core;
     [Import]
     IExtensionService extension;
     void SomeMethod() {
          core.CreateNode();
          // This does not exists, there is no way to introduce the
inconsistency
          // extension.CreateNode();
     }
}
```

3.9 Actual project structure

At the end of this chapter I am going to provide a list of the projects that are in the Visual Studio solution, along with a brief description of their responsibilities.

The projects are divided into five sections:

- Applications
- Common
- Interfaces
- Implementation
- Jobs (pattern matcher implementation)

3.9.1 Applications

- *DMT.Module.Common*: common classes which are shared between the matcher and partitioner modules.
- *DMT.Matcher.Module*: a self-contained module for the pattern matcher application. It mostly takes care of administrative task such as communication.
- *DMT.Matcher.App*: host application for the DMT.Matcher.Module.
- *DMT.Partition.Module*: a self-contained module for the partitioner application. It also acts as the master module and entry point for the whole system.
- *DMT.Partition.App*: host application for DMT.Partition.Module. Basically the same as DMT.Matcher.App.

3.9.2 Interfaces

- *DMT.Core.Interfaces*: shared interfaces between the partitioner and matcher. It mostly contains data interface and factories.
- *DMT.Matcher.Data.Interfaces*: data interfaces for pattern matching such as extended nodes and edges.
- *DMT.Matcher.Interfaces*: service interfaces that are needed by the DMT.Matcher.Module to run the search.
- *DMT.Partition.Data.Interfaces*: data interfaces for partitioning such as extended nodes and edges.
- *DMT.Partition.Interfaces*: service interfaces that are needed by the DMT.Partition.Module to partition the model.

3.9.3 Implementation

- *DMT.Core*: basic implementation for core domain classes and their serialization
- *DMT.Partition.Data*: extends DMT.Core with classes specific to partitioning.
- *DMT.Partition*: default implementation of a graph partitioner.

3.9.4 Jobs

- *DMT.VIR.Data*: it extends DMT.Core with model specific data classes and factories.
- *DMT.VIR.Matcher.Local*: a concrete pattern matcher implementation with three alternative modes to run.

4 Module details

In this chapter, the details of the partitioner and matcher modules are presented. It is also explained how they work together, how a concrete pattern matching looks like from the point where the model gets loaded into memory to the very end where the system informs the user whether there was a match or not.

4.1 Partitioner module

The partitioner or master module is the coordinator in the system. Only one instance is running at once. Its main responsibility is to partition a given model, spin up matcher (or slave) modules and send the partitions out to the matchers for processing.

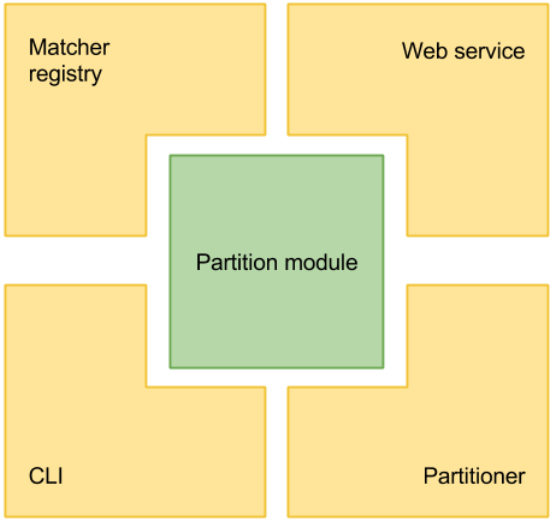

Figure 4-1: partitioner module overview

The `PartitionModule` class is the entry point of the application. Every component around `PartitionModule` has a distinct and well defined responsibility. The matcher registry is responsible for keeping track of the running matcher modules. It knows their state and address. It can instruct all matchers to start or stop. This is the main hub of communication.

The web service is a simple RESTful service through which the partition module can be reached and queried. Through this, the matcher modules can contact the partitioner module and acquire the information needed for the modules to function. For

example, the matcher modules can ask for a partition or a new job to run. I am going to explain all the endpoints later in this chapter.

The CLI or command-line interface is responsible for processing the user input. There are some critical points during the lifetime of the application when it asks the user to provide some input. All of these inputs can be provided ahead of time as command line arguments when starting the application. In that case, the application runs through a whole cycle before asking the user.

Possibly the most interesting part is the partitioner component. The partitioner takes care of the actual partitioning of the model. It is an independent component which is loaded dynamically on start up by the Managed Extensibility Framework. As an input, it gets a graph and returns with a k-way partitioning of it.

From the application's point of view, this is a black box. It only knows the `IPartitionManager` interface which is the entry point for any compatible partitioning algorithm. It has a single method that returns a list of partitions. There is an alternative interface called `IThreeStepPartitionManager` which allows a finer grained control over the partitioning process. I am going to go into details about the partitioner in the next chapter.

On application start up the partitioner module does the following:

1. Parses command line arguments and takes actions if necessary. E.g. asks the user for the model file's path.
2. Loads the model into memory.
3. Partitions the model.
4. Starts the web service.
5. Starts the matcher modules.

After step 5, the partitioner goes to a dormant state and waits for matcher modules to contact with.

4.2 Matcher module

The matcher (or slave) module does the heavy lifting in the system. Its job is to take a partition and try to find a matching sub-graph for a given pattern, optionally it could reach out to other matchers if a node is not present in the partition. The number of

running matcher modules equals to the number of partitions. Every one of the matchers is assigned to exactly one partition.

The architecture of the matcher module is very similar to the module of partitioner.

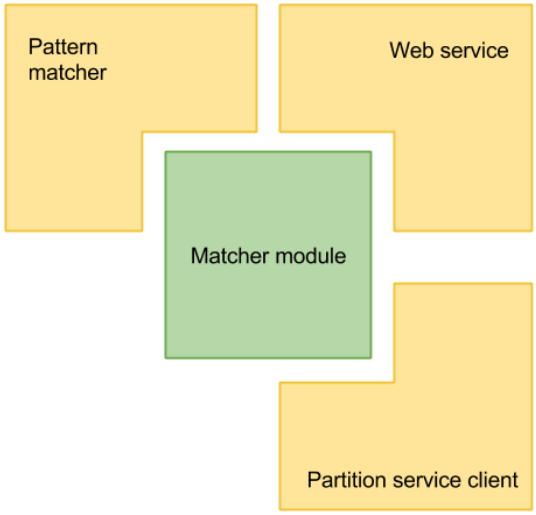

Figure 4-2: matcher module overview

The entry point of the application is the `MatcherModule` class. It serves the same purpose as the `PartitionModule`. It holds the state of application and controls the main flow.

The web service publishes endpoints for the partitioner module as well as for other matcher modules. A matcher can be started or cancelled via a specific endpoint. It also can be asked to try to find a partial pattern match. Similarly to the web service of the partitioner it is a RESTful service.

The service client is a simple wrapper around some REST endpoints, but it has a few extra responsibilities. For example, it has to deserialize the incoming partition.

The most interesting part is the pattern matcher. Again, this is very similar to the partitioner module in a sense that it is also a black box. The application does not know anything about the pattern matcher implementation. The framework only knows its interface: `IMatcherJob`. An instance of a pattern matcher is called a *job*. Upon initialization it gets an `IMatcherFramework` instance through which it can communicate

with the outside world. For example it can ask another matcher to find a partial match for a pattern.

4.2.1 State machine

The matcher module acts as a simple state machine. The start-up sequence maps cleanly into the states. The start-up sequence is the following:

1. Start matcher web service.
2. Register matcher at the partition module.
3. Download a partition.
4. Download the first job to run.
5. Send ready signal to the partition module.

The following figure shows the inner states of the matcher.

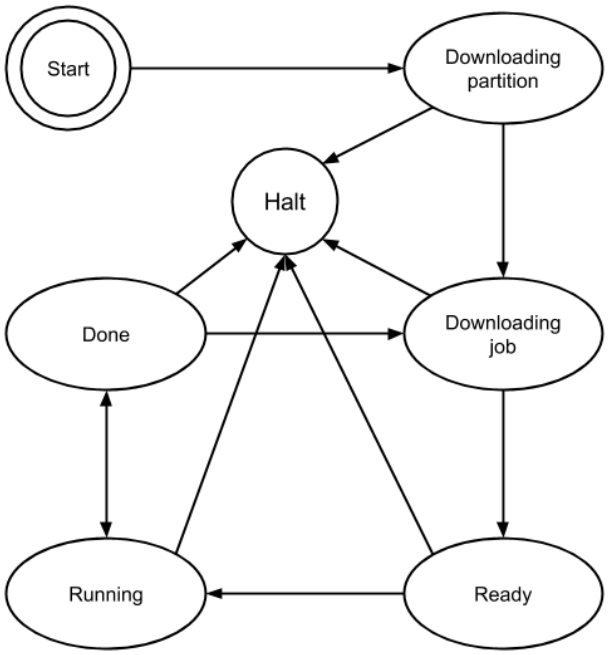

Figure 4-3: inner states of the matcher module

When the matcher is first started, it asks for a partition. This happens only once, because partitioning is really expensive and the partition itself can be quite big.

In the framework, a job is just a binary that gets loaded and the implementation of `IMatcherJob` is instantiated. After this the matcher is ready to start looking for a

match. When the matcher is in *ready* state the master module has to initiate the start of the actual search.

Things get interesting when the search finishes. This is a point where the system has to ask the user what to do next. The user could restart the last job or provide a new one. In the latter case, the system restarts itself and goes back to download a new job.

A quit message from the master module can stop the matcher at any point of its life cycle. If a quit message is sent, all the matcher modules get interrupted and halt, however this is not a problem, because a matcher has no critical state which could cause inconsistency when interrupted. The cancel interrupt has to be taken into account when a matcher job is developed.

4.3 How do modules work together?

We saw how each module works on its own, but they need to cooperate for the whole system to work. As it is a distributed system, the original plan was to run each module instance on its own machine. Therefore, the communication between the modules has to go through the network, presumably over the internet.

This poses the question what protocol to use. There is no need to reinvent the wheel and develop a new protocol above the IP protocol. This leaves us with a choice between UDP and TCP. UDP could be faster in some cases, but its unreliability makes it hard to apply. TCP seemed like a better choice because it is more important to deliver messages reliably than deliver them as fast as possible.

It was clear from the very beginning that contacting another module over the network will be the biggest bottleneck of the system. This is why partitioning algorithms aim to create partitions that have the least amount of edges leading to other partitions. Assuming the partitioning is fairly good, we do not have to over optimize the communication.

4.3.1 First attempt: WCF

Windows Communication Foundation (WCF) [5] is a framework for building service-oriented applications. It is part of the .NET Framework and supports a variety of protocols and formats. Typically WCF services publish a WSDL (Web Service Description Language) document which describes the service itself. By default it uses

SOAP (Simple Object Access Protocol) to define messages. It is quite an extensive protocol, but has the big disadvantage of having to know the concrete types beforehand.

The architecture of the system is built upon public interfaces and hidden implementations. There is no way for the framework to know about the concrete types, moreover concrete types can and do change with models - every model has its own set of concrete `INode` and `IEdge` implementations. The SOAP protocol is not suitable for this and serialization becomes impossible.

WCF supports protocols other than SOAP. We could use JSON encoding for the messages with a REST interfaces managed by WCF, but serialization would be a problem again. It is possible to customize the serializer, but that does not solve the problem of interfaces. WCF was designed to work with classes.

It quickly became clear that this approach would require a lot more effort than just rolling our own URL router and using simple HTTP messages.

4.3.2 Representational state transfer

Representational state transfer or REST for short is an architectural style. Basically it is a set of loose guidelines without an actual framework or official standard. It usually uses HTTP as its transfer protocol. One of its core principles is that every URI (Uniform Resource Identifier) addresses a single resource in the system. It uses HTTP verbs as methods to define what we would like to do with the given resource. The most commonly used HTTP verbs are: GET, POST, PUT, and DELETE.

For example, if we store users and every user has a numeric identifier we could address one by:

```
GET http://example.com/users/1
```

If we initiate an HTTP GET request to that address it would return some details about the user. When we would like to edit some details of that user, we could simply send a PUT request with correct body to the following address:

```
PUT http://example.com/users/1/edit
```

There is no one correct answer how the body of requests and responses should look like. Usually resources represented as XML, JSON or HTML. It is also important to mention that REST is not tied to any programming language or environment, it is stateless and cacheable. These properties make it a good choice for building APIs.

4.3.3 Using REST as a communication layer

I abandoned WCF for a simple REST-like API. Its flexibility outweighs the inconvenience of having to use raw HTTP requests. My solution is only REST-like because I did not follow every guideline, and pattern matching is very far from the usual application of REST.

Both the partitioner and the matcher module have its own web service. The implementation of the REST service is simple and mostly contained in the `RestServiceHost` class. It listens on a given port and has a URL router which directs every request to a registered handler.

Registered routes can have named parameters. This helps to create *reusable* routes. For example the route where the matcher modules can obtain a partition looks like this:

```
RegisterRoute(HttpMethod.Get, "/matchers/{id}/partition", new
GetPartitionHandler());
```

The implementation of the service contains only the bare minimum to be able to function. There is no built in authentication, authorization or even fault tolerance. This is enough for the proof-of-concept implementation but it is not production ready.

4.4 Module cooperation

Now we are familiar with how the modules look from the inside and how they communicate with each other. The only thing left is to show how they cooperate. I am going try to explain the pattern matching process from the beginning to very end where the system informs the user whether there was a match or not. The whole process is the following:

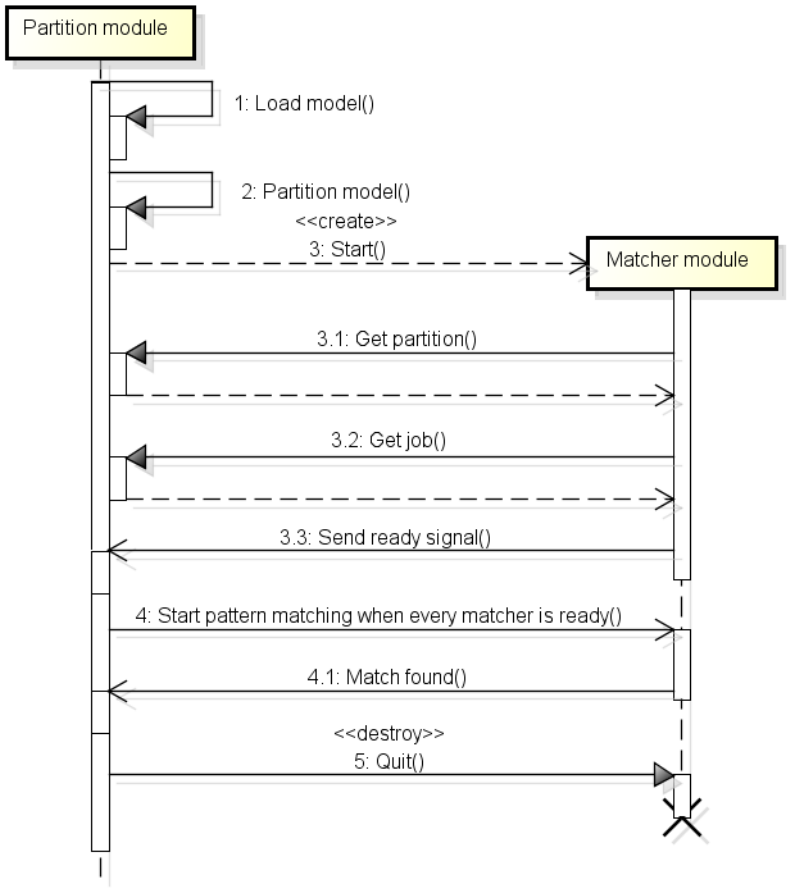

Figure 4-4: module cooperation (partitioner and matcher)

The sequence diagram above shows only one matcher but usually there is more than one. Every matcher instance acts independently. There is only one synchronization point: in the beginning every matcher has to send a ready signal. Only after every matcher is ready, can the actual pattern matching be started. When a matcher finds a match, all other matchers get cancelled.

As the above sequence diagram shows there is initialization phase for loading and partitioning the model. When we have all the partitions, we can start up the matcher modules; every partition gets its own matcher instance. The distribution of the partitions is initiated by the matchers and this is also true for the matcher jobs. We can run multiple matcher jobs during the lifetime of the application.

5 Graph partitioning

In Chapter 2, it was mentioned that the first step of distributed pattern matching is partitioning the model. Without partitioning, there is no point talking about parallelization. Partitioning allows parallelization but it also allows us to work on models which would not fit into the memory of a single machine. In theory this gives us a few advantages: the same computation runs faster on partitions (since they contain less elements). By aggregating the results received on partitions, we can decrease the overall time needed to produce the same solution on a single machine. However it really does matter which kind of partitioning we use, as we will see.

Partitioning a graph introduces new challenges:

1. How to apply partitioning?
2. What should we optimize for?
3. What are the trade-offs that we have to take?

I am going to answer these questions in this chapter.

5.1 Goals

In a distributed system, usually one of the most expensive operation is the communication. In our case this means that the more edges cross the boundaries of partitions, the more communication will be needed when they are getting processed. This sets the first and the most important goal for the partitioning algorithm: it must minimize the number of edges crossing the boundaries of the partitions. We also have to keep in mind that the nodes should be evenly distributed among the partitions. It would not be useful if the partitioning algorithm would produce one huge and a lot of small partitions even though this would certainly minimize the number of crossing edges.

There are two more constraints that we have to take into account: the number of partitions cannot go infinitely large. We assign every partition to a different machine; therefore we need as many machines as many partitions we have. Even in a cloud infrastructure, having a large amount of machines is cumbersome. To make things even more complicated, we should be able to set the approximate amount of nodes in a

partition, or at least an upper bound for them. It is really probable that our machines can only store a certain amount of nodes.

These goals go against each other. We cannot limit the the number of nodes in a partition and limit the number of partitions at the same time. Moreover, we cannot have partitions with approximately the same size while trying to minimize the number of crossing edges. We have to make trade-offs in order to meet all the criteria at once.

5.2 What is graph partitioning?

Partitioning a graph is simply cutting it into smaller components. For example a k-way partitioning divides the nodes into *k* smaller components. A good partitioning is in which the number of edges between partitions is small.

There is a special type of partitioning where the created components are about the same size. This is called *uniform graph partitioning*. For us this property is very important because we aim to produce somewhat uniform partitions.

Typically, graph partitioning is considered to be a NP-hard problem, but there are solutions using heuristics and approximations. Unfortunately, uniform graph partitioning is an NP-complete problem, which means it has no polynomial time approximation algorithm with finite approximation factor.

A naive solution is to do an exhaustive search: produce all possible partitions and choose those, which have the least crossing edges. The number of possible partitions is 2^n, where n is the number of nodes in the graph. This alone makes this approach unusable for graphs that have a few hundred nodes or more or more.

5.3 Researching existing algorithms

Since graph partitioning is a hard problem but encountered quite often in real-life scenarios, there have been extensive research in the area. Most practical solutions use some kind of heuristic to produce a *good enough* solution. In my research, I found two approaches.

5.3.1 Streaming approach

The first is from the paper *Graph Partitioning Algorithm for Model Transformation Frameworks* [6]. Its preposition is that the graph is not available at the

beginning of the partitioning. The nodes are arriving one by one in a stream. In the paper, three phases are introduced:

1. Growing partitions
2. Separating partitions into sub-partitions
3. Refining partitions using a modified KL algorithm

When a new node arrives it is put into the partition that has the most connections to it. When a partition grows too large, it is split into two. After every node has been put into a partition, the graph is coarsened and refined using the Kernighan–Lin algorithm (which I am going to explain later in this chapter). The final step is to uncoarsen the graph to restore its original form.

Figure 5-1 shows how the algorithm chooses the initial partition for a node.

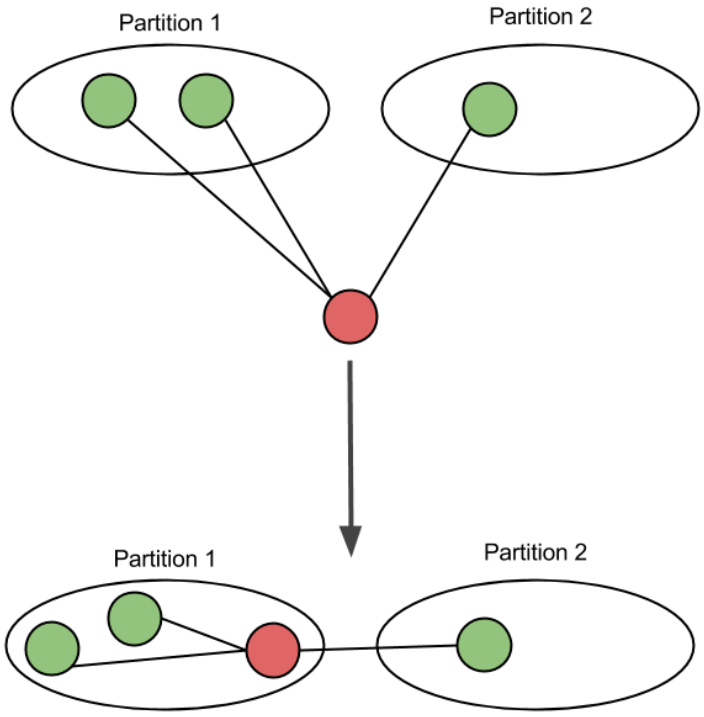

Figure 5-1: graph partitioning, choosing initial position for node

The second step is really similar to the first one. We choose two nodes, which will be the centres of the new partitions. As next, we execute the same method using every node of the old partition.

In third phase, the algorithm tries to further refine the partitions. This is applied by moving nodes to a partition, where it seems appropriate. Refinement is an expensive

operation; therefore the model is coarsened first to decrease the number of nodes. Coarsening is basically node contraction - it creates multinodes.

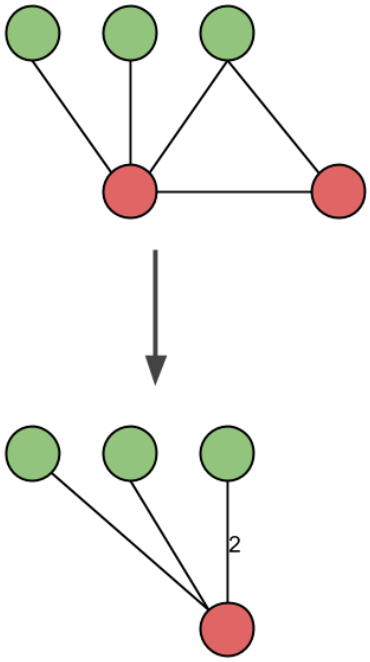

Figure 5-2: coarsening

The streaming nature of this approach makes impossible to form a global view of the model. This approach was tailored for distributed model transformations and it could be used as a distributed algorithm but it does not take network latency into account. Partitioning would require too many messages between machines and network communication being overly expensive makes this option inefficient for real-life application.

5.3.2 Multilevel approach

Another popular approach is to use multilevel processing. It is best described in the paper by Bruce Hendrickson and Robert Leland [9]. A high-level overview of their approach can be summarized in three steps:

1. Coarsen the graph until it is small enough.
2. Partition the graph.
3. Uncoarsen the graph and partitions until the original graph is restored. Partitions can be locally refined on every pass if desired.

The greatest advantage of the approach is its flexibility. Every step can be implemented using a different algorithm because every step is independent from the others.

In the original paper, coarsening is done through finding a maximal matching in the graph and contracting the nodes marked by the edges in the maximal matching. This is repeated several times, creating multiple levels and decreasing the number of node in the graph. This can be seen on the following figure.

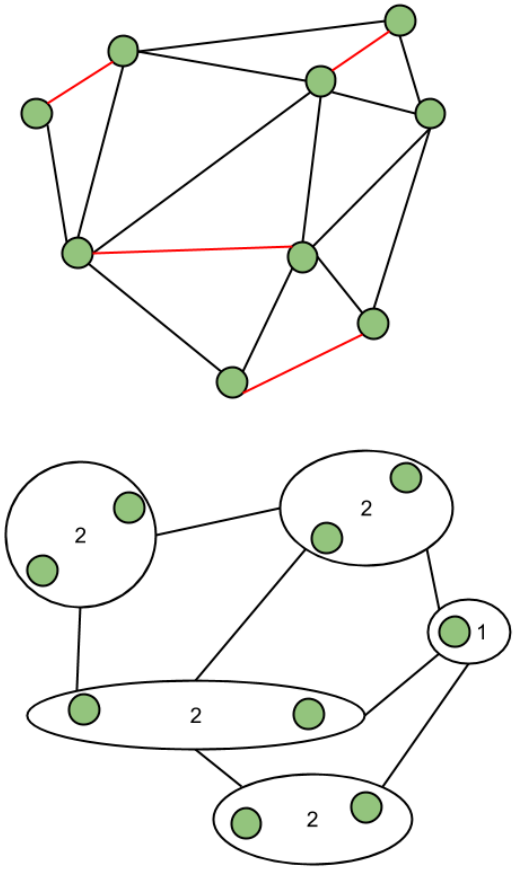

Figure 5-3: maximal matching

A matching in a graph is a set of pairwise non-adjacent edges which means that no two edges share a common node.

The maximal matching can be defined as “a matching M of a graph G with the property that if any edge not in M is added to M, it is no longer a matching, that is, M is maximal if it is not a proper subset of any other matching in graph G. In other words, a matching M of a graph G is maximal if every edge in G has a non-empty intersection with at least one edge in M.” [8]

We need coarsening because graph partitioning is usually an expensive operation and the fewer nodes we have the less time it takes to partition them.

In the paper spectral partitioning (or spectral bisection) is used. This is an expensive but effective partitioning algorithm. It creates a 2-way partitioning of the graph. It can be easily extended to create k-way partitioning by applying the algorithm multiple times. In this case, k = power of 2 must hold.

The final step of the algorithm is uncoarsening and refinement. During uncoarsening optionally we can refine the partitions as uncoarsening does not affect which node is in which partition. One of the best-known algorithm is Kernighan–Lin (KL) algorithm [10].

The KL algorithm is based on a really simple idea: swap nodes between two partitions which reduces the inter-partition cost.

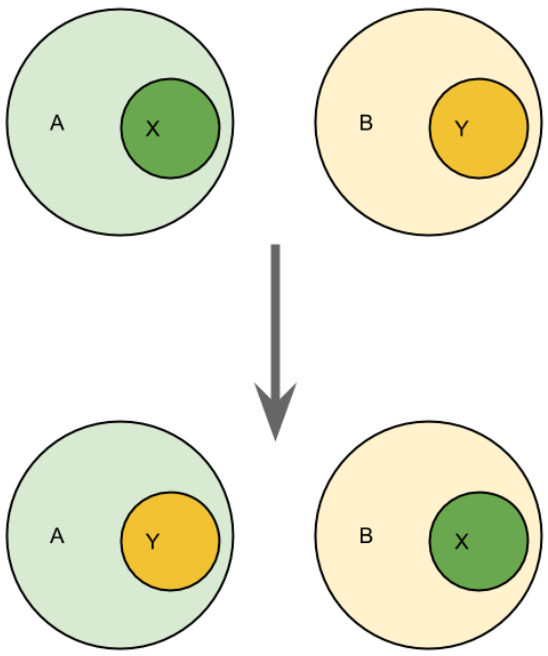

Figure 5-4: KL-algorithm, swapping of nodes

In Figure 5-4 *A* and *B* are partitions, *X* and *Y* are subset of nodes in A and B respectively. Swapping nodes in X with node from Y decreases the cost of edges crossing between A and B partitions. KL algorithm defines a heuristic to choose which nodes are part of X or Y.

Let us define the following costs for partition A. External cost:

$$E_a = \sum_{y \in B} c_{ay} \; for \; each \; a \in A$$

Internal cost:

$$I_a = \sum_{x \in A} c_{ax} \; for \; each \; a \in A$$

Similarly, define $E_b, I_b \; for \; each \; b \in B$. Let $D_z = E_z - I_z \; for \; all \; z \in A \cup B$.

Let D_z express the difference between external and internal costs. When interchanging two node (a from A and b from B) the gain is exactly

$$gain = D_a + D_b - 2c_{ab}$$

Now that we have defined all the necessary values, we can take a look at the actual algorithm. In each pass of the algorithm, we optimize for maximum gain. The algorithm continues until there is a gain greater than zero. The result is the local minimum cost of the two partitions. [7]

```
Compute T = cost of partition N = A U B
Repeat
     Compute costs D(n) for all n in N
     Unmark all nodes in G
     While there are unmarked nodes
          Find an unmarked pair (a,b) maximizing gain(a,b)
          Mark a and b (but do not swap them)
          Update D(n) for all unmarked n, as though a and b had been
swapped
     End while

     Pick j maximizing Gain = sum i=1...j gain(i)
     If Gain > 0 then
          Update A = A - {a1,...,aj} U {b1,...,bj}
          Update B = B - {b1,...,bj} U {a1,...,aj}
          Update T = T - Gain
     End if
Until Gain <= 0
```

Although the KL algorithm is created for exactly two partitions, it is easy to extend in order to use (refine) *k* partitions. In our case, this is a very important feature because we are aiming for a k-way partitioning.

5.4 Implementation

I based my implementation on the multilevel approach. The main reason behind the decision is the flexibility of the algorithm. Any step (coarsening, partitioning or refinement) can easily be replaced without affecting the other two . This behavior

resonates very well with one of the architectural goals, namely with the high level of extensibility. As a result that my generic partitioning implementation can be easily fine-tuned or even replaced, if a model has some specific properties that can aid the partitioning process.

In my version of the algorithm, I decided to disregard weights (defined by the model) on the edges. The weight can have different semantics for different models, but during partitioning this semantic meaning is lost. During partitioning the weight of an edge is the cost of communication between the machines that hold the partitions. I assume that the cost of every message is the same, so every edge should have the same cost which is 1. This simplifies the goal of the partitioning. In this case minimizing the cost is simply minimizing the number of crossing edges between partitions.

A high-level overview of the implementation is the following:

1. Coarsen graph until it shrinks to the 1/n of its original size.
2. Partition the graph.
3. Uncoarsen and refine on every level.

Some of the decisions that I made during early development were mistakes. I am going to highlight these in this chapter.

5.4.1 Coarsening

The very basic idea of coarsening is to decrease the number of nodes in the graph. Partitioning is expensive whether we are talking about time or space complexity. Fewer nodes mean less time and memory spent on partitioning.

Besides decreasing the number of nodes this algorithm tries to build clusters of nodes with high connectivity. This is important because partitioning will operate on the clusters produced by the coarsening. Keeping highly connected nodes in the same cluster will help to form partitions with smaller initial inter-partition cost.

The first version of my coarsening was a greedy algorithm. The overview of the algorithm:

```
passes := determine the number of passes needed
while passes > 0
     determine the number of nodes to contract
     while there are more nodes to contract
          node := choose the node with the highest degree that has not
            been marked
```

```
            create a supernode from the selected node and add neighbours of
               the node to the supernode
            mark the node and its neighbours
        end while
        wrap every not yet marked node in a separete supernode
        connect supernodes with edges based on the previous level
        passes = passes - 1
end while
```

This algorithm wraps every node in a super node after each pass. A super node is a node itself, but it can contain several other nodes. This containment property of a super node is transparent, in the next passes super nodes act like simple nodes. In the figure below, the orange nodes are super nodes and the blues are the ones that represent the result of the previous pass. Wrapping every node ensures that every node on every level has the same type. We can also notice that the connections from the previous level are mapped after each pass. The orange edges represent the new connections built after the super nodes were in place.

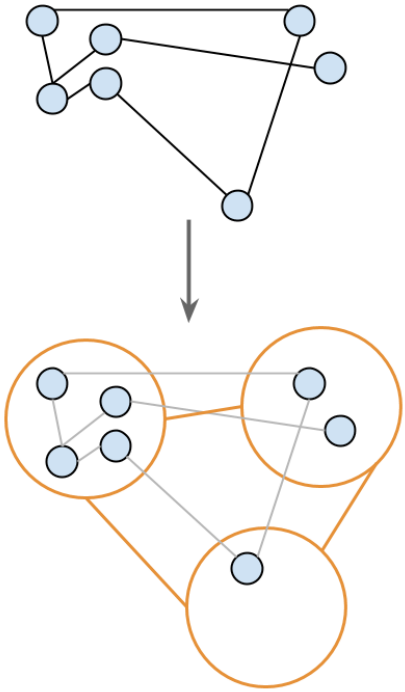

Figure 5-5: wrapping nodes during coarsening

Unfortunately, the algorithm fails to prevent the formation of super nodes that act like black holes. Running the coarsening algorithm on an evenly distributed graph (every node has the same number of outbound edges) produces one big super node and a few really small ones containing only a few normal nodes. The algorithm is a greedy algorithm and by definition it creates only a few big nodes and a lot of small ones.

I attempted to mitigate this issue by changing the node selection process. Instead of simply choosing the node with the highest degree, I gave a weight to each node and took that into account during the selection. One component of this weight was the number of nodes contained in a super node; the other was time dependent. The selection tried to prefer those nodes that had not been selected for a while.

Although the improved version of the algorithm fixed some of the issues, further testing showed that the greedy algorithm is far from optimal and overly complex compared to other options. Therefore I decided to abandon the original greedy coarsener algorithm for a simpler, less effective but working *maximal matching* algorithm.

A maximal matching can be implemented with a simple greedy algorithm:

```
M := empty set of edges
While (no more edges can be added)
    Select an edge (e) which does not have any node common with edges in M
    M = M U e
End while
return M
```

After we have found a set of edges, we take the endpoints of the edges and wrap them in a super node.

The whole coarsening process can be described with the following code snippet:

```
N := calculate the number of nodes to reach
While (node count > N)
    M := find maximal matching
    contract edges of M
End while
```

The actual implementation can be found in the `MaximalMatchingCoarsener` class.

The method proved to be fast but it has the disadvantage of not being effective enough. It can only halve the number of nodes in one pass. This means that the algorithm needs about a thousand iterations to reduce one million nodes to one thousand. In a bigger model, this could easily lead to overabundance of super nodes, which could cripple the whole system.

5.4.2 Partitioning

The second phase of the algorithm is where the actual partitioning takes place. My implementation is overly simple. There are more complex and more accurate ways to do this, but as my implementation of the system is only a proof of concept, I wanted to create something that works and iterate on it later if there is time left.

The partitioning algorithm I used calculates *k-way* partitioning. It can control the number of nodes in a partition, but not the number of partitions. This problem has to be addressed in the future. My algorithm acts like a bucket sort, but all the buckets are equal. There is no distinguishing factor.

As an input, the algorithm gets a list of nodes. It iterates through the list, and puts every node in a bucket that has not been filled yet. At the end, the buckets created will be the partitions.

```
current := create bucket
while there are nodes the list
    if current bucket can hold one more node
        put the node in the bucket
    else
        current := create bucket
        put node in the bucket
    end if
end while
```

5.4.3 Refinement

The last phase is refinement and uncoarsening. They are applied in one step. Uncoarsening is a bit like popping a balloon - it discards one layer of super nodes. It is the reverse operation of coarsening but it is a lot simpler because each level of super nodes is independent from any other. Uncoarsening removes the super node wrappers and the edges between them.

For refinement, I used the KL algorithm described earlier in this chapter. The final phase looks like the following:

```
while G is not the original graph
    G := uncoarsen graph
    refine G using KL
end while
```

5.5 Evaluation

In order to validate the correctness of the implemented algorithm, I used it on a randomly generated evenly distributed graph. With a few hundred nodes, it showed promising results but when I increased the number of nodes to 10 000 it took too much time to be useful in a real-life scenario. This was before I tried to use it on the dataset I prepared for the case study. The dataset consisted of more than 100 000 nodes. The results were even worse. After a few attempts to fix some critical errors, it was clear that this algorithm and implementation was not ready for usage at this scale.

On a real-life example, coarsening proved to be a much less effective than what I first anticipated. This led to thousands of coarsening passes which meant that there were thousands of refinement steps. With each step the number of nodes grew (due to uncoarsening) and this made refinement slower on every step.

At this point, I had no time left to spend on the partitioning algorithm; I had to make a compromise to continue with the implementation of the pattern matcher. I created a very simple implementation with one goal in mind: it had to be fast.

The new implementation gets a list of nodes, shuffles them and divides them up into n partitions evenly. For reproducible results the shuffling can be seeded.

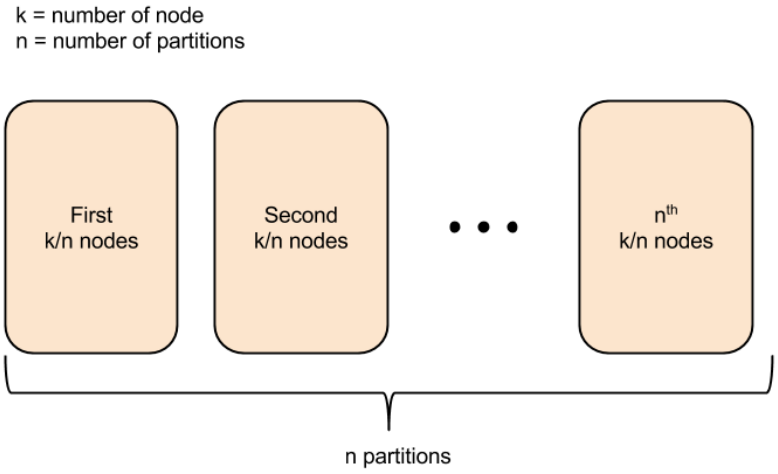

Figure 5-6: evenly distributed partitions

As a conclusion, I must say partitioning is far more difficult than I would have thought. It is hard to customize existing methods and algorithms because there are a lot more constraints when it comes to a distributed system. If I had to start over, I would try to use an existing library instead of creating my own.

6 Pattern matching

Earlier I listed three strategies that I was going to explore. In this chapter, I am going to explain these strategies in detail. Also I am going present a case study through which the viability of the distributed approach is demonstrated. The case study also helps in illustrating the strategies. Before I go into any further details, I am going to present the problem of partition matching and introduce a few existing solutions.

6.1 Overview

Pattern matching is a really broad problem and has several meanings. When I refer to pattern matching, I refer to isomorphic subgraph matching, where the model and the pattern are represented as graphs. When we try to find a match for a particular pattern, what we really want is to know whether there is a subgraph in the model (graph) that fits the pattern.

Subgraph isomorphism problem can be defined as: “Given *G* and *H* graphs as input, one must determine whether there is any subgraph in G that is isomorphic to H. This problem is a generalization of the problem for testing if a graph contains a Hamilton cycle and therefore it is NP-complete.” [11]

This generally means that there is no "good" solution to this problem. Surprisingly in my experience this was not an issue. The dataset in my case study is moderate-sized (containing more than 120000 nodes) but never had any problem related to the computational complexity.

In my research, I found two different approaches for implementing pattern matching. The first one is to use a general algorithm which can work with any pattern. The other way is to manually try to match each node in the pattern by using tree-search enumeration. The latter approach might need prior knowledge about the pattern.

One of the first general algorithms was described by J. R. Ullmann in his paper *An Algorithm for Subgraph Isomorphism [12]*. That is a recursive backtracking algorithm which takes polynomial time to match fixed patterns. A more recent algorithm is VF2 which was presented in a paper titled *A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs [13]*.

The manual method showed to be faster due to non-algorithmic optimizations (e.g. compiler optimization) in my research. Moreover, this approach is usually combined with code generation which provides flexibility to work with any kind of pattern.

I chose the latter approach mainly for its simplicity. It is important to keep things as simple as possible in the pattern matcher component. There are multiple reasons for this. Firstly, the distributed nature of the system carries enough complexity and hidden difficulties. We would not want to introduce anything that is not absolutely necessary. Secondly, when I was designing the framework for the pattern matchers, I wanted to minimize the role of the framework in every sense. Adding more functionality (such as a general pattern matcher algorithm) to the core framework would go against this design principle. The flexibility of the system allows us to easily replace the pattern matcher, even more it requires us to provide a new implementation for every pattern.

6.1.1 Three different approaches for a distributed solution

The biggest challenge for a distributed pattern matcher is that a significant portion of the nodes are not available locally. Nodes can be easily obtained but their connections are not manageable that easily. How should we access nodes from another partition? What should we include in the message when downloading a node from another partition? Should it contain the node only? Or maybe all of its neighbours? These questions are hard to answers. There a lot of factors that has to be taken into account. If we send less information about the nodes it will not be usable, on the other hand, if we send everything, the messages could become bloated and slow down the transfer over the network and thus the whole process.

My first approach was a local-only implementation. This is basically a crippled version of a non-distributed pattern matcher because it does not have access to all the nodes. This version does not contact any other partition for additional nodes and information. The second approach is a natural extension of the original matcher "algorithm" by making remote node access completely transparent. Finally, the third approach is the truly distributed one. In this case, I search for partial matches locally and delegate any remote search to the right partitions.

6.2 Case study overview

It is important to describe the dataset used in the case study before we go into any further details about the implementation of pattern matchers.

Choosing a good case study is very important. A contrived example could yield unusable results. I explored multiple options for the case study:

- Generating the input model to mimic a social graph.
- Using data from IMDb (International Movie Database, [14]).
- Using data from the VIR database (Villanykari Információs Rendszer, [15]).

I wanted to use a real world example, therefore I abandoned the possibility to use a *generated model* early on during evaluation.

IMDb has a huge database of movies, actors, directors and other types of movie related data. At first glance, it seemed to be a really good choice but their data is not readily available. It would have taken me a significant amount of time to figure out how their normalized data can be reassembled again. Even if I had done that, the amount of data would have been too big to process. They have more than two million movies alone.

The VIR database has enough records for it to make sense to use it as a case study in a distributed setting. Undoubtedly its main attraction is that I am very familiar with its internal structure, therefore it was relatively easy to convert it to a format which is understood by the system.

I have chosen the last option because it fitted my needs and it was the easiest to apply.

6.2.1 Metamodel

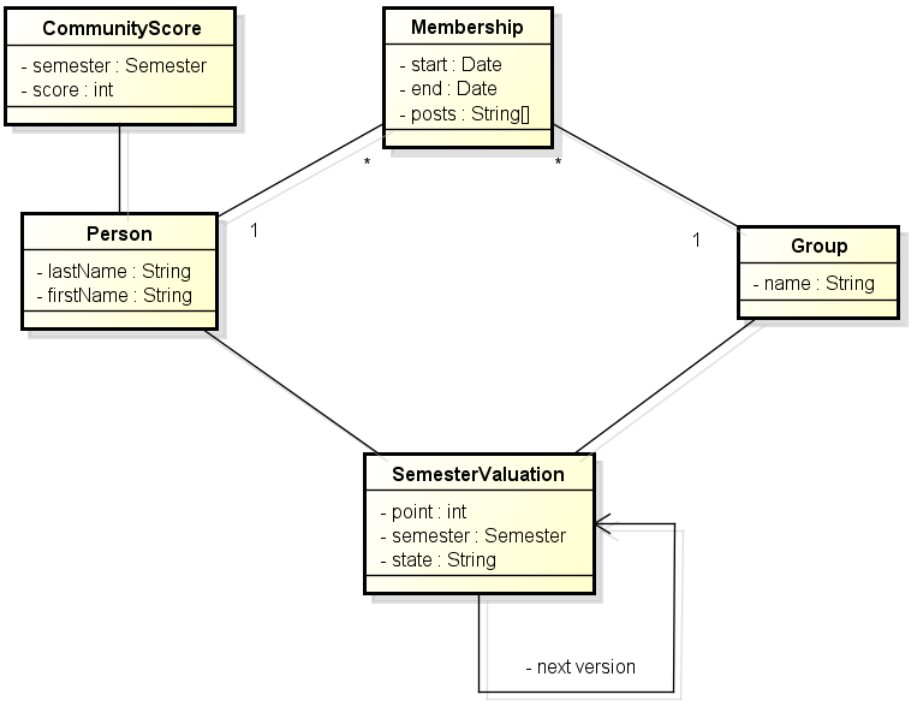

Figure 6-1: metamodel for the case study

The figure above shows the metamodel of the dataset. The framework cannot provide any tools or methods for ensuring the integrity of the model. This means that we have to manually ensure that the model corresponds to the meta-model before feeding it into the system.

The VIR database stores student information and historical data about their community contributions. In the last 10-12 years, most of the students at our faculty (VIK) made it into the VIR database and a lot of them gathered "community points" over the years. Students can join different groups and as a member, they can help the community with their work. This help can be anything from providing food delivery in the Schönherz Zoltán Kollégium or by maintaining crucial web services such the VIK Wiki. For their work they get "community points" every semester. The amount is based on their contributions.

There is a lot of personal information in the database. During pre-processing, I stripped these out to protect the privacy of students.

Description of entities in the meta-model:

- Person: represents a student.
- Group: represents a group of students.
- Membership: it represents the relationship between a person and a group. It has a start and end date. Also it can have posts which are the roles of a given persons in the group.
- CommunityScore: it is the sum of community points for a semester.
- SemesterValuation: this is a proposal for community points for a group at a given semester. It has a state field which indicates that it has been accepted or not.

The raw data was not usable by itself; it had to be transformed into a format that is understood by the system (the partitioning module to be exact). I created a small application that creates a model file from the input data. This can be found under the *DMT.VIR.Data.Parser* project.

6.2.2 Pattern

The pattern is the core of the case study. The model (data) is very important but a simple pattern can render the case study useless. Without a complex pattern, the case study cannot be used to demonstrate the viability of the distributed approach.

I came up with a pattern that is complex enough and it could successfully demonstrate that distributed pattern matching can be a viable option, but not too complex so that we will not lose sight of our original goals.

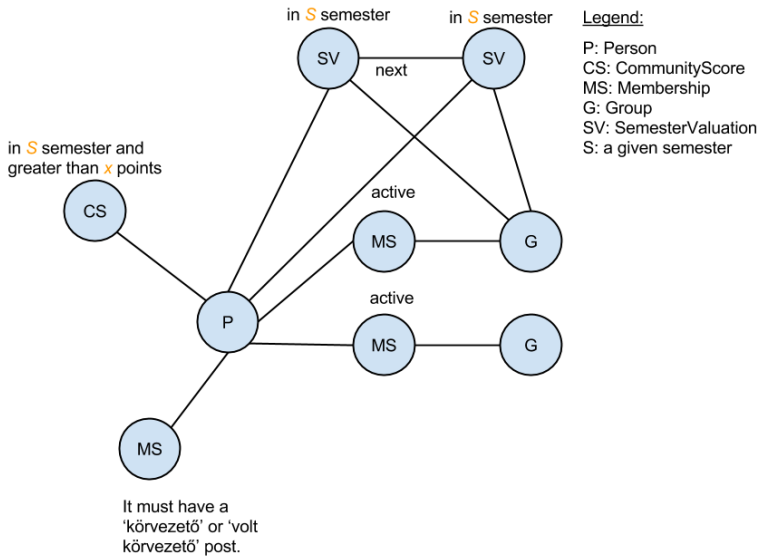

Figure 6-2: pattern for the case study

In the figure above, we can see the shape of the pattern and the conditions that have to be met for the nodes to be matched in the pattern.

Given a semester *S* and community point *x*, we are looking for a person who (i) has *x* community points in *S* semester, (ii) is or was a group leader, (iii) has an active membership in a group and also (iv) has an active membership in another group with at least two versions of semester valuations for the given *S* semester. In my concrete case I used $x = 60$ and $S = spring\ of\ 2012/2013$.

It is important to note that to find a match, the pattern cannot contain duplicates nodes. For example a membership that matches the *group leader* condition cannot be reused as an active membership.

6.3 Matcher algorithm

All three strategies are based on the same algorithm but every one of them has a slight difference. The core concept is to try to match nodes from the model to nodes in pattern. Basically it is an extensive search through the whole model but when a node is proven to be not fit for the pattern, the node and its subgraph is skipped.

Firstly we try to match the person in the pattern then we look at the neighbours of the matched person node. We take a neighbour and try to match it to the next pattern node (which is a neighbouring node in the pattern). If it does not match, we try another neighbour of the matched person. If we find a matching node, we continue with the next pattern node. The method highly resembles the depth-first search algorithm.

The following figure shows how each model node gets matched to the pattern. To simplify the matcher implementation, the algorithm tries find matches in a predefined order. It continues to match model nodes in a subgraph (in the case below subgraph of MS node) until it hits a leaf edge in the pattern or there is mismatch between the model and the pattern.

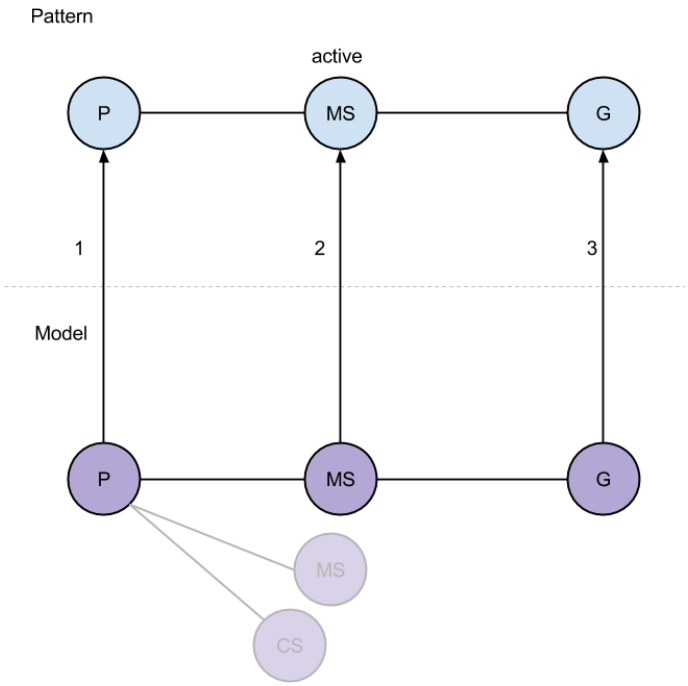

Figure 6-3: model nodes are being matched to pattern nodes

The next figure shows a situation where the first neighbour of the P node cannot match the corresponding pattern node. In this case, the algorithm skips the whole subgraph and continues with the next neighbour.

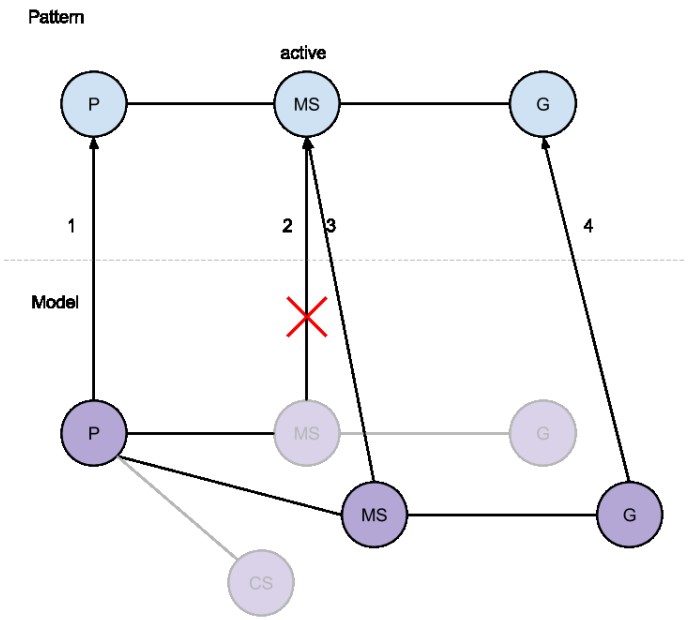

Figure 6-4: skipping a subgraph when a match is not possible

The algorithm does not have to be manually developed every time we want to create a matcher for a new pattern. There are some general features that make it possible to generate the necessary code instead of writing it by hand.

Generating code also has the advantage of the being more secure and robust. We only have to verify the correctness of the generator once and all the following generated code will work correctly. Unfortunately, code generation was not implemented due to time constraints.

6.4 Local-only matcher

The first and simplest strategy is the local-only matcher. When partitions are created, inter-partition edges are marked so that we know which edge leads to which partition. This is essential because every strategy treats remote nodes differently.

Local-only strategy was created with testing purposes in mind. It is not really distributed since the autonomous matcher instances are not communicating with each other at all. They only use the locally available nodes and skip every node that is not present. The main goal of creating the algorithm was to test if the system is properly working: all the matcher processes are started up and all the partitions are sent out correctly.

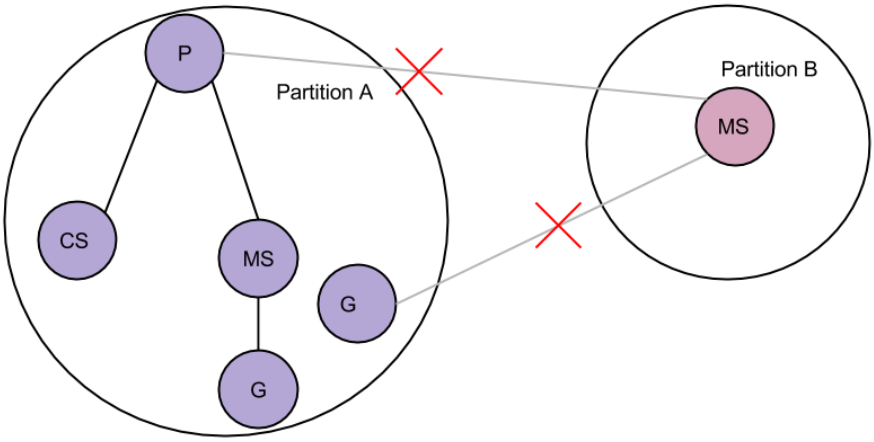

Figure 6-5: unreachable remote nodes

On the figure above the crossed out edges cannot be traversed because they lead to or come from another partition. These edges are basically non-existent from the matcher's point of view.

The implementation is as simple as the idea itself. Every time we traverse from one node to another, we check if the edge is leading to another partition or not. If it does, we skip it and we also skip the subgraph where it leads.

The pseudo code for the implementation:

```
loop through every person node
     reset matched nodes in the pattern
     match person to the pattern
     loop through every edge of the person node
          if edge leads to another partition
               continue with next edge
          end if
          node := other end of the edge
          if found a match for the node
               // found a match
               break outer loop
          end if
     end loop
end loop
```

The actual implementation can be found in the `VirLocalMatcherJob` class and its base class `VirMatcherJobBase`. They are placed in the `DMT.VIR.Matcher.Local` project.

6.5 Proxy matcher

The second approach is a simple extension of a non-distributed algorithm. When we hit a remote node, we simply ask another matcher instance for it and download the node itself. A question arises here: do the neighbours of the remote node should be transferred too? If we just download the node itself and nothing else, we lose the option to continue the search, because there would be no edges to continue on.

It is clear that we need some *extra* information about the node but if we download all neighbours of the node, we might end up downloading the whole partition. Besides, we do not need the neighbouring *nodes* to continue the search, we only need the *edges*.

On Figure 6-6 the small green circles represent the remote node stubs. When a matcher hits a node that is not available in the partition, it downloads the required node and temporarily adds it to the partition. This means that the newly downloaded node is discarded if there is no match on that branch. It is also important to note that when trying to traverse to the next node (in this case G in Partition C), the whole process starts over.

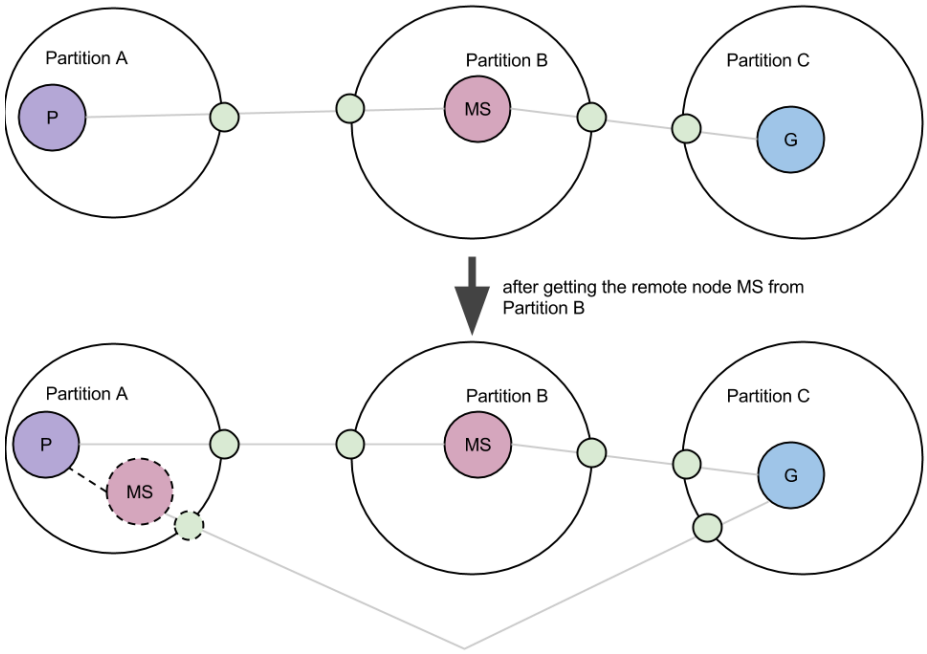

Figure 6-6: proxy nodes

This implementation is really simple but has a few drawbacks:

- It blocks the search while retrieving the remote node
- Sending complete nodes over the network can be really expensive. For example if a node contains a large image or a lot of text.
- In a worst case scenario, traversing the edges of a remote node can lead to a lot of network communication.

The last disadvantage presented itself quite quickly during the implementation. In the first version, the remote node was completely separated from the host partition and every node lookup went through the network even if it was in the same partition. This can be seen in the figure below.

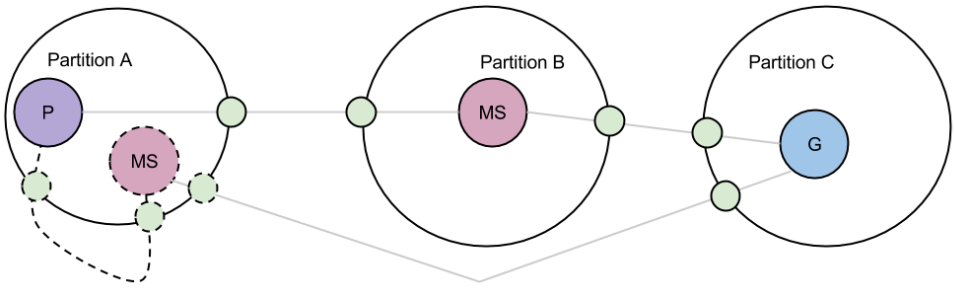

Figure 6-7: proxy nodes without optimization

This little detail rendered the implementation completely useless, it was so slow that matching even the simplest pattern was impossible. To fix this, on every node

lookup I try to find it in the local partition first and downloading remote nodes are started only if there is no local match found. It is illustrated in the following code snippet:

```
public INode GetNode(IId partitionId, IId nodeId)
{
     // try to lookup the node in this module (this should be O(1))
     var node = MatcherModule.Instance.GetNode(nodeId);
     if (node != null)
     {
          return node;
     }
     MatcherInfo matcher = FindMatcher(partitionId);
     MatcherServiceClient client = new MatcherServiceClient(matcher.Url);
     return client.GetNode(nodeId);
}
```

6.6 Distributed matcher

The last approach is the most complex one, but it also has the most potential. The main idea is that every matcher instance tries to match some part of the pattern, but not necessarily the whole one. Basically, when we hit a remote node, we offload a small part of the search to another matcher instance. There are only a few requirements: we have to be able to continue the search from any pattern node and we need a way to send the pattern over the network.

This approach is surprisingly straightforward, but there a few problems that we have to address:

1. How to serialize the pattern and what to include?
2. What happens if we hit another remote node during a partial match in the second matcher instance?
3. How to handle matches that are spread through 3 or more partitions?

When a matcher processes a partial match request, it only looks at the local nodes, it never steps out of the partition. It also does not try to match the whole pattern, but only a subgraph. This can be seen on the figure below. Figure 6-8 shows a partially matched pattern where the greenish node is currently being matched in a remote matcher. In this case, only the subgraph starting from the greenish *MS* node will be matched (G and two SV nodes). The bluish *CS* node will not be matched in the remote matcher because that is not part of the subgraph including MS.

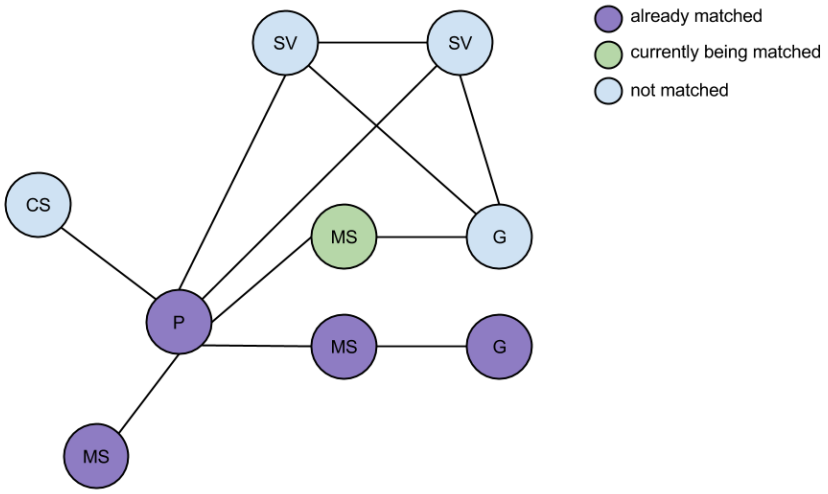

Figure 6-8: partially matched pattern

The remote matcher tries to match as many nodes as it can. When it finishes it send back the pattern with the newly matched nodes, thus the original matcher can continue its job.

To make it more efficient the remote search happens asynchronously. The match always starts from a person node and every person node get its own thread. When the search hits a remote node, the thread blocks but releases the main thread so the search can continue with a new person. This can be seen on the next figure.

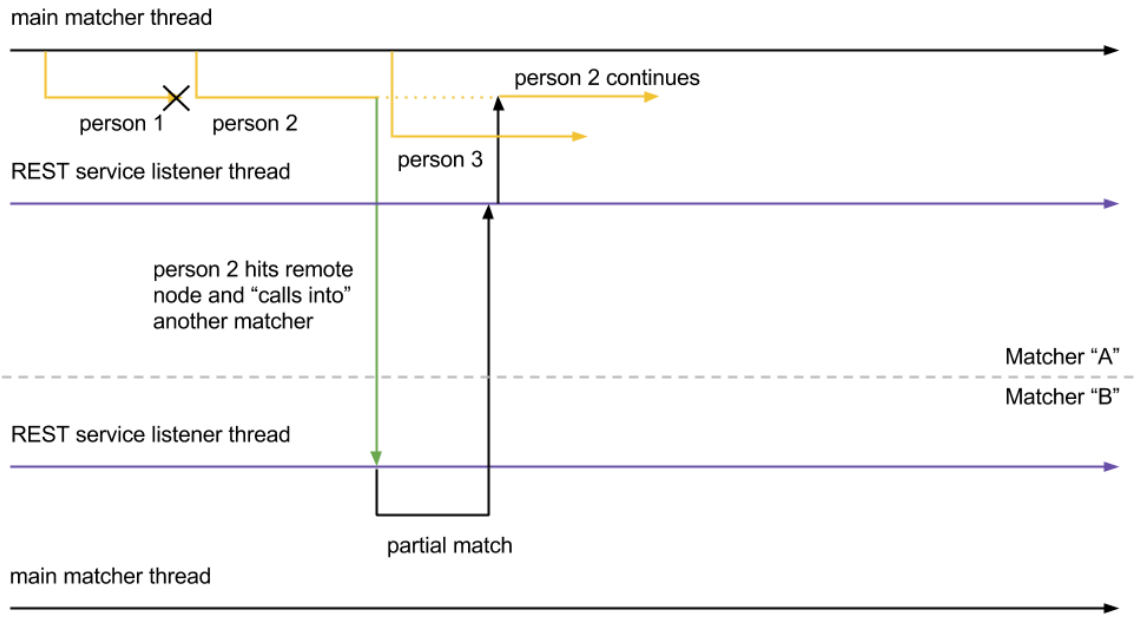

Figure 6-9: active threads in a partial matcher

Every matcher has (at least) two threads: one for the REST service listener and one for the actual pattern matcher algorithm (main thread on Figure 6-9). Although every Person node has its own thread, these threads block the main thread and only release it when one of the following three conditions has been met:

1. the match fails completely
2. a match is found
3. hits a remote node and offloads the work to another matcher instance

The most interesting is the third one. On Figure 6-9 "person 2" hits a remote node and that is where the partial matching actually happens. When a remote node is found the pattern is serialized into an XML document and sent to the matcher that will run the partial search. When the partial search finishes, the pattern is sent back, but this time it contains the newly matched nodes and the remote edges of those newly matched nodes. This ensures that when we continue the search in the original matcher, we will know where to go next if we hit another remote node. This is important because we do not allow the matcher to pass around a partial match. A partial match always involves *exactly* two matcher instances: the initiator and the processor.

All the implementation classes can be found in the *DMT.VIR.Matcher.Local.Partial* namespace.

6.6.1 Test results

Tests have shown that this approach is working: it can find a match in all cases regardless of the total number of partitions. The results are a bit disappointing though, when we look at the run times. The test machine has two cores so we can assume almost full parallelization when running only two matcher instances.

Results for a non-distributed run (using only one partition):

```
Sum wait time for remote matches: 0,00 s
Elapsed time for the whole matching: 6,92 s
0 % of the time was spent on waiting for remote matches to finish.
```

Using 2 partitions:

```
Sum wait time for remote matches: 182,42 s
Elapsed time for the whole matching: 246,37 s
74,04 % of the time was spent on waiting for remote matches to finish.
```

This shows that the communication overhead can be so big that it is not even worth to run the search in a distributed way. This is because our node matching is too

fast. To match a node, we only need to compare some properties which is very far from a CPU intensive calculation. Also the partitioning is really inefficient because it does not minimize the number of inter-partition edges. Unfortunately, to improve this we would have to rewrite the partition algorithm.

Simulating a CPU intensive calculation is a lot easier than designing a new graph partitioning algorithm. All we have to do is to add a `Thread.Sleep()` at the right place. I slowed the pattern matchers when they try to match the `CommunityScore` pattern node and re-run the program. The results are convincing:

Non-distributed run (using only one partition) with 10 ms sleep:

```
Sum wait time for remote matches: 0,00 s
Elapsed time for the whole matching: 618,50 s
0 % of the time was spent on waiting for remote matches to finish.
```

Using 2 partitions with 10 ms sleep:

```
Sum wait time for remote matches: 329,80 s
Elapsed time for the whole matching: 530,33 s
62,18 % of the time was spent on waiting for remote matches to finish.
```

This reinforces the theory that a long computation is worth running in a distributed way. Even 10 ms delay was enough, because it is on par with the communication overhead.

6.7 Conclusion

Implementing and testing the pattern matcher without a proper implementation of a partitioning algorithm proved how important it is to create good partitions. With too many inter-partition edges, the communication overhead seriously cripples the performance potential of the whole system. In the future, it is essential to develop a partitioning algorithm which creates really good partitions.

Implementing the matchers also made it clear what should be in the framework. For the matcher, it is better to have a framework that is really thin and only provides a few interfaces despite all the generalizable parts I encountered. This is especially true for the partial matcher strategy: most of the implementation details could be reused when implementing a different pattern. Instead of making everything part of the framework I think a utility library would serve better.

7 Summary

In my thesis, I have created a distributed pattern matcher. Through a proof of concept implementation I showed that the task is possible. In this document, I presented my findings and a reference implementation. The thesis also documents some of the most important issues I met and how I handled them including the bypasses that turned out wrong.

Although this is only a proof of concept implementation, I extensively covered the architecture of the whole system. Designing a flexible and extensible architecture is part of the problem due to the nature of pattern matching. I could not assume anything about the model when designing the framework, it had to be model agnostic. Highly flexible and extensible architecture has to make a lot of compromises. The most obvious one is complexity. The design of the system allows for a high degree of extensibility (e.g. the partitioning algorithm can be completely replaced) but manages to stay relatively simple. If I had to start over I probably would do a lot of things differently. For example I would not make heavy use of (container based) dependency injection (MEF). It overshadows and de-emphasizes a few key properties of the framework.

The problem of distributed pattern matching could be divided into two smaller ones. The first one is graph partitioning. During development it turned out that this is an equally big or even bigger problem than the actual pattern matching. Although graph partitioning is a heavily researched area and lot of work has been done in the last 50 years, some unique properties of the system made it very hard to apply the readily available solutions and algorithms. I tried to modify a multilevel algorithm with small success. For testing I had to fall back to a very simple but ineffective solution. This clearly pointed out how important partitioning really is. Without a good partitioning the distributed approach has no benefits.

I implemented three different strategies for the actual pattern matching, however only the last one has the potential to be used in real-life scenarios. For a complex pattern it has the promise of better parallelization by only trying to find a partial match in every partition. Surprisingly making the pattern matching distributed was very straightforward. There were some technical challenges but the applied principles are pretty ease to grasp.

The pattern matcher is a black box from the framework's point of view. At the moment this means that every implementation has to start from scratch. During implementation I saw a lot of opportunity for generalization, there are lot of parts of the code that are reusable.

In my thesis I created something that can be built upon in the future. It is not ready but it lays down the foundations. The implementation my not be complete but the ideas and concepts are readily usable. In my opinion applying pattern matching in a distributed system is a viable option.

8 Future improvements

On several occasions throughout my thesis I suggested. I am going to summarize these in this chapter and I am going to touch upon what is needed to be done for the system to be production ready.

The most important issue to solve would be to create a new and effective partitioning algorithm. I have already mentioned this but partitioning is a critical feature of the system. Tests proved that without good partitions the system is practically useless.

In the pattern matcher, I observed a few specific tasks that are very common and would be needed in basically every implementation. Currently there is no standard way to share code between implementations; the framework does not provide any support either. It would ease future development of new pattern matchers if there would be a common library with readily usable components. For example the only model and pattern specific parts of the partial matcher implementation are the matcher functions. Almost everything else could be reused in another implementation.

Another feature that would help implementing future pattern matchers is code generation. The brute-force pattern matcher algorithm is easy to generate since it is basically a series of embedded *for loops*. Code generation might not be necessary if there is a comprehensive toolkit library for building pattern matchers.

The current implementation is only a proof of concept: there is no error handling, no protection against misuse or malicious user. To be production ready there a few features that has to be added to the system:

- Fault tolerance: when a message cannot be sent to another module (matcher or partition) it must be handled and possibly retried.
- Authentication and authorization: currently there is no authentication when two modules communicate with each other. This way anybody can send messages to modules which could halt the system.
- Multiple data sources: currently the only allowed source of the model is an XML file. Models can come from a variety of places, e.g. directly from the internet.

If everything in this chapter is I think the system would be close to be production ready.

References

[1] Leslie Lamport: Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM, Volume 21 Issue 7, July 1978, pp. 558-565*

[2] Azure, http://www.azure.microsoft.com/en-us/ (19 April 2014)

[3] Wikipedia: *Inversion of Control*, http://en.wikipedia.org/wiki/Inversion_of_control (04:53, 17 April 2014)

[4] Managed Extensibility Framework, http://msdn.microsoft.com/en-us/library/dd460648(v=vs.110).aspx (19 April 2014)

[5] What Is Windows Communication Foundation, http://msdn.microsoft.com/en-us/library/ms731082(v=vs.110).aspx (20 April 2014)

[6] László Deák, Gergely Mezei, Tamás Vajk, Krisztián Fekete: *Graph Partitioning Algorithm for Model Transformation Frameworks, EUROCON, 2013 IEEE, Zagreb, 2013, pp. 475 - 481.*

[7] CS267: Lectures 20 and 21, Mar 21, 1996, Apr 2, 1996: *Graph Partitioning, Part 1*, http://www.cs.berkeley.edu/~demmel/cs267/lecture18/lecture18.html#link_4.2 (21 April 2014)

[8] Wikipedia: *Matching (graph theory)* http://en.wikipedia.org/wiki/Matching_(graph_theory) (revision 03:03, 15 April 2014)

[9] Bruce Hendrickson, Robert Leland: *A multilevel algorithm for partitioning graphs, Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference, Article #28*

[10] B.W. Kernighan, S. Lin: *An Efficient Heuristic Procedure for Graph Partitioning, Bell System Technical Journal, Volume 49, Issue 2, Feb. 1970, pp. 291 – 307.*

[11] Wikipedia: *Subgraph isomorphism problem* http://en.wikipedia.org/wiki/Subgraph_isomorphism_problem (revision 03:45, 7 November 2013)

[12] J. R. Ullmann: *An Algorithm for Subgraph Isomorphism*, *Journal of the ACM, Volume 23 Issue 1, Jan. 1976, pp. 31-42.*

[13] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento: A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs, IEEE Transactions on Pattern Analysis and Machine Intelligence, *Volume 26 Issue 10, October 2004, pp. 1367-1372.*

[14] IMDb http://www.imdb.com/ (23 April 2014)

[15] Villanykari Információs Rendszer, https://korok.sch.bme.hu (24 April 2014)