Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

A general purpose local search-based pattern matching framework

MASTER'S THESIS

Author
Márton Búr

Advisor
Dr. Ákos Horváth
Zoltán Ujhelyi

December 11, 2015

Contents

Kivonat **i**

Abstract **ii**

1 Introduction **1**

1.1 Model-driven engineering 1

1.2 Main challenges 2

1.3 Summary of contributions and structure of the thesis 2

2 Preliminaries **4**

2.1 A motivating example: code smell detection in program code 4

2.2 Foundations of model-driven engineering 5

2.2.1 Metamodeling 5

2.2.2 Eclipse Modeling Framework 5

2.3 Implementing model queries using graph patterns 8

2.3.1 EMF-IncQuery pattern language 8

2.3.2 Incremental pattern matching 10

3 Overview of local search-based pattern matching **12**

3.1 Related work 12

3.2 Algorithm adaptation 13

3.3 Pattern matching workflow 15

4 Integration to EMF-IncQuery **18**

4.1 Details of the adapted algorithm 18

4.1.1 Search plan 18

4.1.2 Search plan calculation 20

4.2 Implementation details 24
4.2.1 Common matcher API 24
4.2.2 Flattening 24
4.2.3 Debugger tooling 26
4.3 Details of search execution 28
4.3.1 Sequential search 28
4.3.2 Multi-threaded execution 30
4.3.3 Advantages and weaknesses 31

5 Evaluation **32**
5.1 Measurement workflow 32
5.2 Measurement environment 33
5.3 Models and patterns used for assessment 33
5.4 Performance evaluation 34
5.5 Running the Train Benchmark 37
5.6 Evaluation summary 39

6 Conclusion and future work **41**
6.1 Conclusions 41
6.2 Summary of contributions 41
6.3 Future plans 42
6.3.1 Information about type cardinality 42
6.3.2 Adaptive cost calculation 42
6.3.3 Advanced parallel execution 42
6.3.4 Hybrid pattern matching 43

List of Figures **45**

List of Tables **46**

Bibliography **49**

Appendix **50**
A.1 A more detailed metamodel for program ASGs 50
A.2 Detailed measurement results for code anti-patterns 51

A.3 Detailed measurement results for Train Benchmark 53

HALLGATÓI NYILATKOZAT

Alulírott *Búr Márton*, 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 autentikált 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.

Budapest, 2015. december 11.

Búr Márton
hallgató

Kivonat

Szoftvermodellezési feladatok során a modellek, amiket jellemzően gráfként reprezentálnak, tartalmazzák a tervezési információkat. Ezek feldolgozásával használjuk ki ennek a megjelenítési formának az előnyeit, és hasznosítjuk az ismereteket. A modelleken végzett műveletek egyik célja hagyományosan a végrehajtható programkód előállítása.

Modelleken végzett tipikus műveletek egyike a keresés, mely során a cél bizonyos feltételnek megfelelő elemek, azaz egy almodell felderítése. Ez részfeladata a transzformációnak, ami vagy egy köztes modellt, vagy forráskódot állít elő a bemeneti modell alapján. Szintén a keresésen alapul a jólformáltság ellenőrzése, ami vizsgálja, hogy a modell felépítése követi-e a modellezési nyelv szabályait.

Dolgozatomban az almodellek keresését gráfmintaillesztést segítségével végeztem. A gráfmintaillesztés általánosan egy komplex problémakör, azonban különböző megközelítések már ismertek a megoldására. Egyik gyakori módszer a lokális keresésen alapuló mintaillesztési technika, mely egy kezdeti pontból kiindulva keresi meg az illeszkedéseket. A módszer hatékonyan működik optimális keresési terv mellett, azonban az optimális keresési tervet csakis az éppen vizsgált modell struktúrája határozza meg.

Munkám során adaptáltam és implementáltam egy lokális keresésen alapuló gráfmintaillesztési algoritmuscsaládot. A megvalósított módszer lényege, hogy a modell számossági jellemzői alapján számítja ki a keresési tervet. Emellett elkészítettem a keresés egy egyszálú, valamint egy párhuzamos verzióját is. A dolgozatban bemutatom az algoritmus főbb gondolatait és lépéseit, illetve a megvalósítás sajátosságait. A megoldást a nyílt forráskódú EMF-IncQuery keretrendszerbe is integráltam, törekedve a komponensek újrafelhasználására.

Mindkét típusú megvalósítás teljesítményét és skálázhatóságát mérésekkel is alátámasztom, és egymással, illetve a korábban már meglévő inkrementális algoritmus végrehajtási idejével is összehasonlítom. Emellett elemzem, hogy mely gráfmintaillesztési forgatókönyvek mellett melyik módszer bizonyul kedvezőbbnek. Az erdemények szerint az alkalmankénti futtatást igénylő, illetve a memóriaszegény környezetben végzett feladatok során is a lokális keresésen alapuló algoritmus részesítendő előnyben.

Abstract

Model-driven software development tasks involve creation of graph-like models, which store facts and parameters about the system under design. Developers make use of this representation when processing the models, and apply the contained information. The ultimate goal of the development process it to produce executable code, which may be preceded by several intermediate steps.

A frequent operation executed on a models can be search, which means an exploration of elements, in other words a submodel, that comply with a set of constraints. Other typical operations include search as a subtask, such as transformation, which derives a new representation of the model, or generates executable source code. Well-formedness validation also based on search, for it checks whether the model is valid in terms of the rules of the modeling language.

In this thesis I applied graph pattern matching as the underlying technique to find submodels. Generally, graph pattern matching is a complex problem, however, different approaches are commonly used to solve it. One of them is the local search-based pattern matching that finds all matches by starting the search from specified model elements. If the search plan is optimal, this technique is highly efficient. However, this is determined only by the structure of the model on which the search is performed.

During my work I adapted and implemented a local search-based pattern matching algorithm, which uses the statistics of the underlying model to calculate the search plan. I created two versions of search execution runtimes: a single-threaded, and a parallel. In this thesis I introduce the main ideas and key steps of the algorithm, as well as the peculiarity of the implementation. I also integrated the completed software components to the open-source EMF-INCQUERY framework.

I provide assessment results in order to show the scalability of the proposed solution. For this purpose I carried out performance comparison for both pattern matching runtimes. Additionally I evaluated the solution with respect to the already existing, incremental pattern matcher algorithm of EMF-INCQUERY. Finally, the applicability of the algorithm is discussed in case of different pattern matching scenarios. According to the results, the local search-based pattern matching technique is preferable in cases when tasks require only a single run, or the environment is memory constrained.

Chapter 1

Introduction

1.1 Model-driven engineering

The *model-driven engineering* (MDE) approach is becoming widespread in many areas of software and system engineering, such as designing safety-critical systems, where faults can cause severe injuries or serious environmental harm, as it delivers higher-quality products in a shorter development lifecycle (see e.g., [17]). MDE aims to focus on creating and analyzing models at different levels of abstraction during the engineering process, which are used to synthesize executable program code in the end.

Modeling may start as early as the *requirements* against the system under design are collected. It is followed by creating high-level abstract models, then producing lower-level design artifacts with design decisions and implementation details after a series of refining steps. The models can be continuously verified during the development process in order to identify design faults as soon as possible.

There are some extensible formalisms intended as a general purpose way of representing models (such as UML [22]), the industrial practice seems to prefer *domain-specific languages* (DSL) for describing models instead, which can be designed and modified to the needs of application domains and actual design processes.

On the other hand, developing such a DSL (and providing tool support) is a costly task requiring special skills. For this reason *domain-specific modeling* (DSM) technologies have emerged to support these purposes. The *Eclipse Modeling Framework* (EMF, [10]), which is built on the Eclipse platform, is a leading technology in this sense, which is considered a de facto industrial standard. A DSL development process with EMF starts with the definition of a metamodel, from which several components of the modeling tool can be automatically derived. It is defined using the *ECore* formalism, which is defined as part of the EMF. Numerous generative and generic technologies assist the creation of tool support (such as textual and graphical editors) for EMF-based DSLs. The user can define textual or graphical concrete syntax, while code generators can be created by specifying source code templates for the modeling language.

1.2 Main challenges

Model processing can be categorized into the following type of operations: *modification*, *transformation*, *filtering of elements*, and *search within the model*. All of these require a definition of an application condition. When this condition is fulfilled, corresponding actions should be executed. As software models have graph like presentation, a way of formulating the conditions for model elements is using graph patterns, where the task is to find all matches of a pattern in the underlying model, and execute the required operations on these matches.

However, the problem of graph pattern matching is strongly related to graph isomorphism, which is a complex computational task, especially when carried out on large graphs. There are tools that rely on search-based approach to find all matches of a given pattern in a model, while others are using incremental techniques to collect and constantly maintain the set of pattern matches.

There is a *trade-off between the computation speed and the size of the used operative memory*. The search based algorithms usually execute faster when the pattern matching needs to run once. However, when a pattern needs to be matched multiple times against slightly changing models, incremental solutions can easily outperform the search-based algorithms.

The incremental algorithm achieves incremental behavior by maintaining predefined indexes based on the structure of the underlying model. When this index structure builds up, the set of matches are directly available. Additionally, model changes are propagated in the index structure efficiently, and the set of matches is updated immediately. However, this approach may be constrained by memory limits, because the supplementary data structure can take up huge amount of memory, based on the complexity of the pattern and the model.

On the other hand, local search-based algorithms collect the matches by *traversing the model*, starting from given points, and gathering each tuple of elements that satisfy all constraints of the pattern. The main challenge here is to select the starting points and to determine the *optimal search plan* that guides the traversal.

1.3 Summary of contributions and structure of the thesis

Throughout my thesis, I use the pronoun "we" to refer to my supervisors and me. There are some publications that I coauthored, and used while writing this thesis. In the following there is an explicit enumeration of my own contributions to the topic:

- I adapted and implemented a local search-based pattern matching algorithm,
- I provided both single-threaded and parallel execution runtimes for executing search operations,
- I created a debugger tooling to support pattern development and optimization,
- I integrated the solution to the EMF-IncQuery official Eclipse project, and

- I evaluated the implementation from the aspect of performance.

The rest of the thesis is structured as follows: the basic background information about model-driven engineering (MDE) and the connecting technologies is introduced in Chapter 2. We introduce a motivating example for the application of MDE that is used throughout this thesis, then the relevant tools and frameworks used for our work are introduced.

Chapter 3 provides an overview of the proposed pattern matching framework. First, available research results and papers are referenced, highlighting their main ideas and achievements. It is followed by the discussion of algorithm integration aspects, and the key steps of our solution for local search-based pattern matching.

Chapter 4 describes the integration and implementation tasks, which includes detailed description of the algorithm, execution methods, and the accompanying tooling.

The performance, and scalability of the implemented framework is evaluated in Chapter 5 on both industrial and artifical models.

Chapter 6 concludes the thesis with a short summary of the completed work, and an enumeration of possible future development directions

Chapter 2

Preliminaries

The current chapter introduces the basic concepts of model-driven engineering (MDE) via a motivating example from the domain of software modeling. It also summarizes the major features and capabilities of the EMF-IncQuery graph pattern matching framework.

2.1 A motivating example: code smell detection in program code

In order to ensure code quality, and maintainability, several programing rules should be followed when creating a computer program. However, there are several typical mistakes that developers make. These so-called code smells (or in other words anti-patterns) can efficiently be detected by using code queries. For such purposes many advanced techniques use the *abstract syntax graph* (ASG) representation of a program. With the ASG synthesized from the code, graph pattern matching can be the underlying method to execute a query on the software.

Considering this application as a motivating example, we use the *domain of program ASGs*, the *Catch problem* code smell from [32] to pattern matching related tasks in details throughout this work. Source code of Java programs serve as the subject of this problem, and its definition is the following: in a `catch` block there is an `instanceof` check for the type of the `catch` block parameter. Instead of the `instanceof` check a new `catch` block should be added for the checked type and the body of the conditional has to be moved there.

To demonstrate this code smell, we placed a short Java code fragment in Listing 2.1. In this snippet, a `try-catch` block surrounds the `FileInputStream` object creation. In line 2, a checked exception of type `FileNotFoundException` may be thrown, but the programmer should also prepare for other kinds of (unchecked) exceptions as well, e.g., `NullPointerException`. In this case, additional `catch` blocks accepting parameters of different types should be implemented instead of adding only one catch block, and using the `instanceof` operator on the thrown object.

```java
try {
    FileInputStream fis = new FileInputStream(file);
} catch (Exception e) {
    if(e instanceof FileNotFoundException){
        logger.severe("File not found");
    } else {
        logger.severe("An exception was thrown");
    }
}
```

Listing 2.1. Java code snippet containing code smell.

2.2 Foundations of model-driven engineering

The aim of model-driven engineering is to increase development efficiency by creating various models of the system under development. These models help focusing on the important design decisions, and (ideally) they omit unnecessary details. In terms of software development, several frequently used representations of a program have a graph-like structure, e.g., class diagram, component diagram, state machine, and the ASG, as in our running example. The models can be then used for different purposes including source code generation, test case derivation, and even simulation.

A huge advantage of graph-like representation is that there are many analysis techniques, which can be applied to detect errors in models [9, 6]. Additionally, the definition of graph patterns can be done in a very concise, and expressive way.

2.2.1 Metamodeling

As it is described in [26], metamodels define available concepts, that can be used to build up the instance model containing only elements typed by the metamodel. These concepts can also have attributes in order to enrich the expressiveness of the language. In a metamodel, relations define connections between concepts, and links are the instances of such a relation. Via the allowed relations, the metamodel also defines the structure of the models.

2.2.2 Eclipse Modeling Framework

This section about Eclipse Modeling Framework (EMF) is based on [16]. EMF is a Java framework and code generation facility for building tools and other applications based on a structured model. EMF provides a metamodel (called ECore) for describing structured models. Using these structured models EMF provides a toolkit and runtime support to produce a set of Java classes representing the model in the Java Virtual Machine (JVM), a set of adapter classes that enable viewing, and a basic editor. As EMF already supports a large set of modeling constructs, our discussion mainly focuses on the ECore meta- and instance models and follows [4].

A simplified ECore model is depicted in Figure 2.1 in order to demonstrate the most important elements of the metamodel.

- `EClass` models classes themselves. Classes are identified by their name and can contain a number of attributes and references. To support inheritance, a class can refer to a number of other classes as its supertypes.
- `EAttribute` models attributes, the components of an object's data. They are identified by their name, and they have a type.
- `EDataType` models the types of attributes, representing object data types that are defined in Java, but not in EMF. Data types are also identified by their name.
- `EReference` is used in modeling associations between classes; it models one end of such an association. Like attributes, references are identified by their name and have a type.

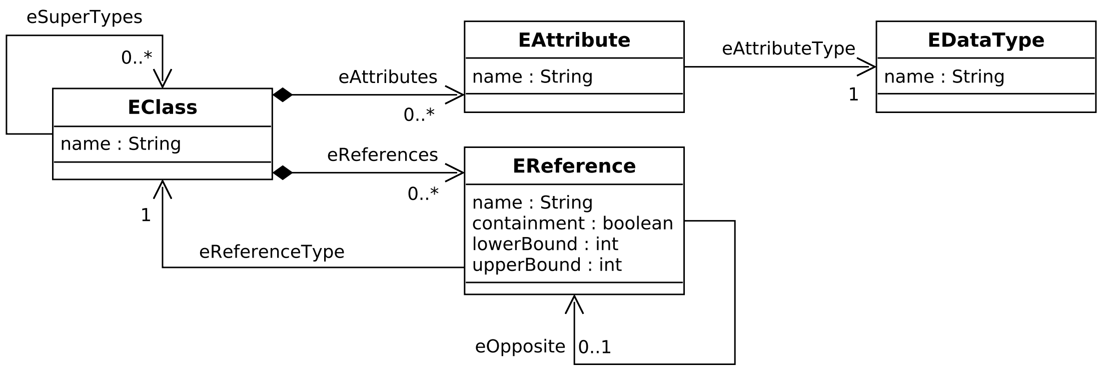


Figure 2.1. The ECore metamodel

Extension of the ECore

From an ECore model, the generator of EMF can create a corresponding set of Java implementation classes. Every generated EMF class extends from the framework base class, `EObject`, which enables the objects to be integrated and appear in the EMF runtime environment. `EObject` provides an efficient reflective API for accessing the properties of the object generically. In addition, change notification is an essential property of every `EObject` and an adapter framework can be used to support extension to the objects.

The reflective API [10] in EMF enables to manipulate all attributes and references attached to the `EObject` by using the `eSet` and `eGet` functions. This is conceptually equivalent to `java.lang.reflect.Method.invoke()` Java method, though it is much more efficient in the aspect of performance.

Notification observers (or listeners) in EMF are called adapters [5] because in addition to their observer status, they are often used to extend the behavior (that is, support additional interfaces without subclassing) of the object they are attached to. An Adapter, as a simple observer, can be attached to any `EObject` by simply adding the adapter to the `eAdapters` list of the `EObject`. This Adapter implements a function called `notifyChanged`, which is called any time when the `EObject`, which contains the Adapter, is manipulated. All information about the manipulation is held by a notification object which is the input parameter of the `notifyChanged` function. This adapter is responsible for sending the notifications to the `EMFInstanceAdapter` manager class.

Example EMF metamodel and model of a program ASG

As an running example, the an EMF metamodel is shown in Figure 2.2, as exported from *ECoreTools* [28]. The main parts of the metamodel are the following:

- `Expression` is the superclass for every expression type,
- instances of `Unary` represent expressions that only accept up to one expression as their operand,
- instances of `InstanceOf` stand for `instanceof` operators in the code,
- instances of `Identifier` represent valid Java identifiers in the ASG,
- `Named` is a superclass of any valid Java ASG element, that is supplied with a name (except for `Identifier` to avoid loops via the `refersTo` relation),
- instances of `Parameter` represent parameters in the program, and
- `Handler` marks an exception handler `catch` block for a `try-catch` construction in the ASG.

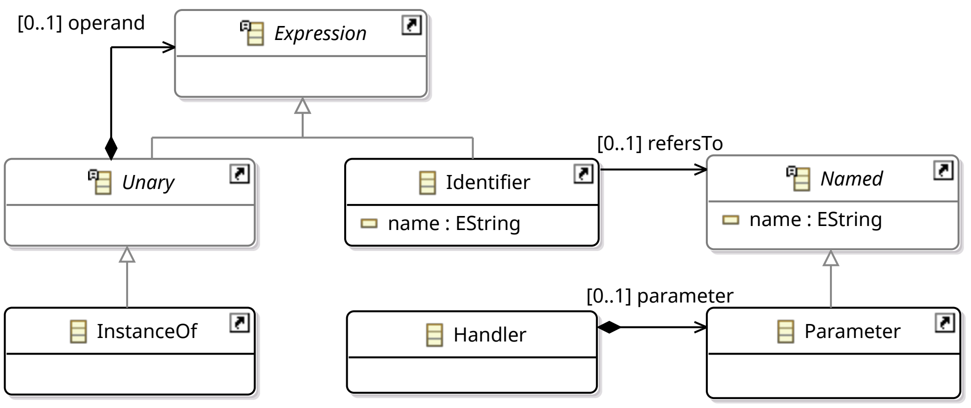


Figure 2.2. EMF metamodel fragment for ASGs

From the explanatory code snippet in Listing 2.1, using the same tool that was used to generate the ASGs of the programs included in [32], we obtained the EMF model. We include a hand drawn version of the instance model in Figure 2.3. It uses simplified object names and applies some trivial abbreviations of the type names. Each node in Figure 2.3 is adorned with its corresponding line number from Listing 2.1. We also used four different colors for the

Expression, Named, Statement and Handler types. The subtypes of these types are marked with the same color as the corresponding supertype. Some element types are not shown above in the metamodel fragment, for they would only complicate the structure, and would not help the comprehension of the example. However, we include a more detailed, but still incomplete metamodel in Figure A.1.1 in the Appendix.

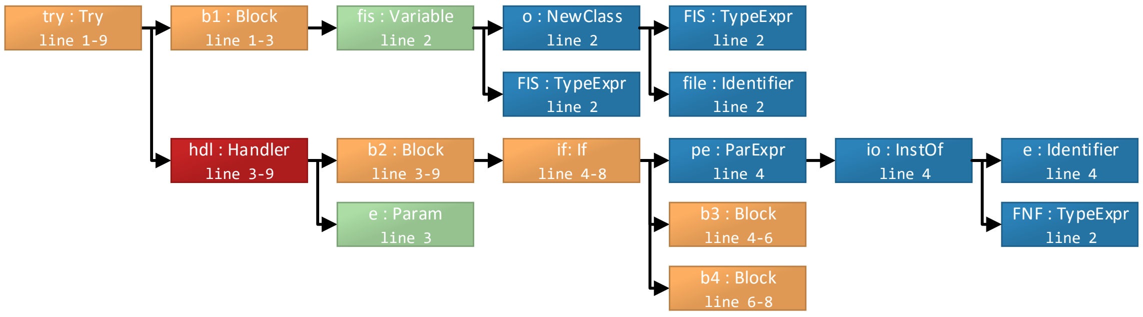


Figure 2.3. The ASG representation of the example code snippet

2.3 Implementing model queries using graph patterns

As described in [31], graph patterns are graph-like structures that encapsulate a set of constraints regarding the nodes, the attributes of the nodes, and edges. Model queries can be carried out by matching a graph pattern against instance model graphs. During the matching process the objective is to find suitable model elements that satisfy all constraints of the graph pattern. According to [3], EMF-INCQUERY is a framework with a *language for defining declarative queries over EMF models*, and a *runtime engine for executing them efficiently without manual coding*.

2.3.1 EMF-INCQUERY pattern language

The framework has its own highly declarative pattern language called IncQuery Pattern Language (IQPL). The language is similar to Datalog, which is a subset of Prolog. As stated in [3], the graph pattern based query language of EMF-INCQUERY references EClasses as node types, EReferences and EAttributes as edge types. Pattern variables will be mapped to EObjects of the instance model or attribute values. It is important to add that the language does not specify the order of constraint evaluation.

Based on [27], we provide a short summary of the constraints supported by the IQPL:

- **Classifier constraint:** checks if a variable is an instance of an EClass.
- **Path constraint:** requires a specific reference, an attribute, or a path of reference and attribute sequence between two variables.
- **Equality constraint:** specifies that two variables have to be mapped to the same model element.

- **Pattern call constraint:** enables the composition of multiple patterns. The *positive pattern call* refers to another pattern and specifies that the called pattern must be satisfied in the context of the actual parameters. Additionally, a pattern may define a *negative application condition* (`neg` keyword), which means that the target pattern is disallowed to have a valid match along the actual parameters. Using the `count` keyword, EMF-IncQuery can save the *number of matches* to a variable.

- **Binary transitive closure**: it is possible to describe the *transitive closure* of a two-parameter pattern by the + symbol.

- **Check constraint:** evaluates a specific attribute expression on the variables of the pattern and accept matches only if the result of attribute condition is true.

- **Eval constraint:** evaluates an expression defined inside the constraint. The return value of the evaluation will be stored in a variable.

In order to demonstrate the basic capabilities and structural constraints of the language, Listing 2.2 shows the example catch finder problem formulated in IQPL. It introduces two patterns, `catchProblemFinder` and `handlerVariable`, both with two symbolic parameters. There are type constraints applied to the parameters: the substitutions of `cBlock` need to be of type `Handler`, while `insOf` is expected to be an `InstanceOf`.

```
 1 pattern catchProblemFinder(cBlock : Handler, insOf : InstanceOf){
 2   Identifier(varRef);
 3   Unary.operand(insOf, varRef);
 4   find handlerVariable(cBlock, varRef);
 5 }
 6
 7 pattern handlerVariable(cBlock : Handler, variable : Identifier) {
 8   Handler.parameter(cBlock, param);
 9   Identifier.refersTo(variable, param);
10 }
```

Listing 2.2. Patterns to detect missing catch clauses.

The `catchProblemFinder` pattern specifies its match set as a set that holds tuples of type ⟨`Handler, InstanceOf`⟩ in line 1 using classifier constraints for parameters. In line 2 the pattern declares that the `varRef` variable should be subtituted with an instance of `Identifier`, which is also expressed using a classifier constraint. Line 3 prescribes by a path constraint that the `EObject` substituted to `insOf` should have `varRef` as its `operand`. In addition, the pattern also uses a pattern call constraint to reference the `handlerVariable` pattern, and the call will use the substitutions of `cBlock`, and `varRef` variables.

The latter pattern will have tuples of ⟨`Handler, Identifier`⟩ in its match set, according to line 7. Line 8 states that the variable `cBlock` shall have `param` as its `parameter`. In line 9 it is specified that the `EObject` substituted into `param` should be navigable from `variable` via an instance of the `refersTo` `EReference`.

2.3.2 Incremental pattern matching

The EMF-IncQuery framework initially was designed to provide pattern matching capability for its users, based on an incremental pattern matching algorithm (Rete [2]). This approach relies on the idea that along with the computation of the initial match set of each pattern, a data structure is built in the memory. This data structure is then maintained in a way that every change in the underlying model is propagated efficiently, so that the set of matches is updated. In case of EMF models this update mechanism implemented using the efficient EMF notification observers of the framework.

To demonstrate this in operation using our example, Figure 2.4 depicts a possible Rete network for the `catchProblemFinder` pattern. The Rete network is essentially a dataflow network, where every node contains tuples of arbitrary length. At the top of the diagram rectangles symbolize *input nodes*, which provide input for the network. The second node in the top-left corner is, in addition, a *production node* that means it already holds the match set of the `handlerVariable` pattern.

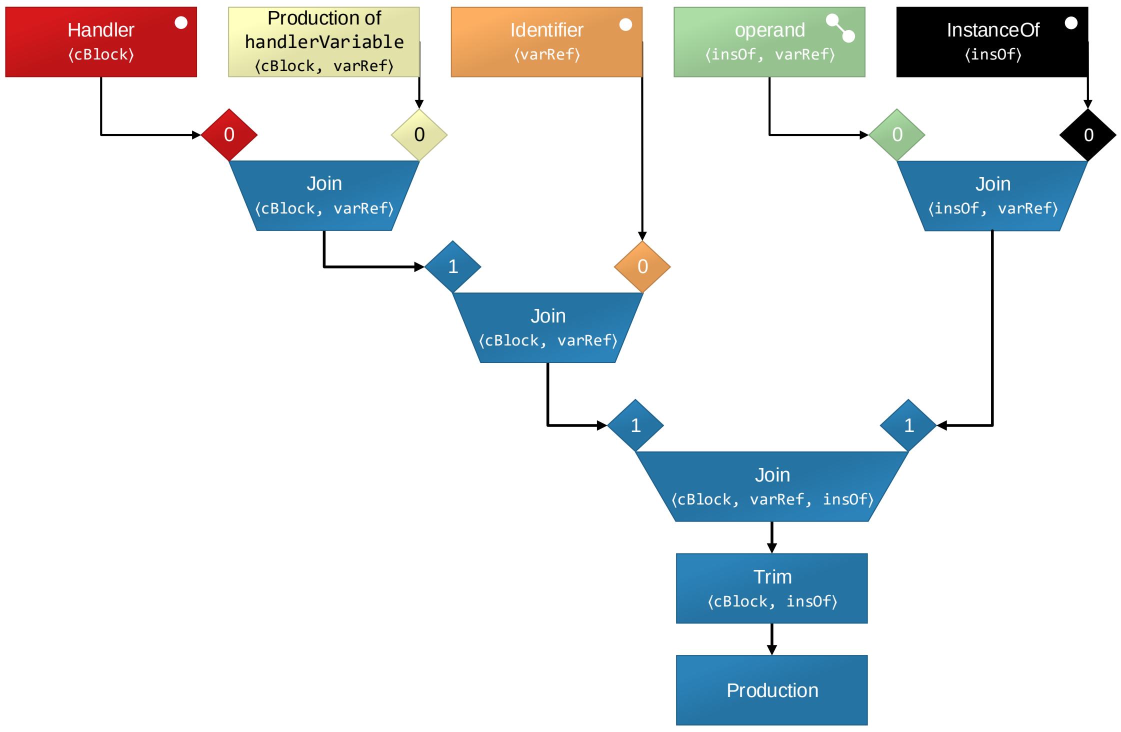


Figure 2.4. Rete network for the `catchProblemFinder` pattern

Nodes with *Join* annotation are a type of *worker nodes*. They carry out join operation on the input. The diamonds over the worker nodes indicate the operand indices from the previous node. The *Trim* node omits some values from the preceding worker node. In the example, ⟨`cBlock, varRef, insOf`⟩ is trimmed to ⟨`cBlock, insOf`⟩. Finally, *Production* will hold the match set for a pattern.

The main advantage of this solution is the ability to retrieve the match results in constant time after the first evaluation, if the model is unchanged. In case the model changes, the match result update time is proportional to the size of the change, and not the size of the complete model.

In the example, when a new object of type `Identifier` appears, the change is propagated in the Rete net on a directed path down to the production node. This way matches after small changes in the model are instantly updated.

However, a main drawback of this approach is the memory footprint of the internal data structure. Its size depends on the size of the model and the complexity of the pattern. In certain application scenarios, in which the models themselves are extremely large, this footprint means a bottleneck concerning the usability of the algorithm.

Chapter 3

Overview of local search-based pattern matching

Probably the biggest drawback of the EMF-IncQuery framework is the memory consumption of its incremental algorithm. In order to overcome this limitation, we adapted a model sensitive local search-based pattern matching algorithm described in [33]. This algorithm has a much smaller memory footprint, as well as the initial pattern matching execution takes also less time compared to the Rete incremental approaches.

3.1 Related work

There are several modeling tools that implement pattern matching in addition to EMF-IncQuery, such as ATL, Eclipse OCL, FUJABA, FunnyQT, and GrGen.NET.

ATL [19] defines a hybrid, textual language for defining graph transformations. It is called hybrid, for it has declarative language elements, but in order to ease formulation of complex rules, imperative instructions can also be used. ATL uses local search-based pattern matching, but applies different heuristics to determine the order of search operations. As it was described in [30], a parallel algorithm provides the matchings.

Eclipse OCL [11] supports the declarative definition, and evaluation of OCL [21] constraints over EMF models. In this case, despite the declarative description of the pattern, the OCL constraint encodes the execution order of search steps to find elements that conform all constraints of the description. Also, Eclipse OCL has incremental evaluation support [8].

FUJABA [20] provides a graphical language for specifying model transformation rules, and relies on local search-based pattern matching. It uses *story diagrams* for the execution [13], and applies a dynamic strategy concerning the traversal of the instance model. This means for every object, the following step is made in the direction of the reference with the lowest multiplicity. The execution times presented in [13] are very promising regarding this approach. We can say that *SDMLib* is the successor of FUJABA, which won the best performance award on the Transformation Tool Contest 2014 [12] by generating Java code for search execution.

FunnyQT is a Clojure library supplying a comprehensive set of model querying and transformation services to the user [15]. According to [14], it supports pattern matching using an internal DSL implemented with Clojure's metaprogramming facilities. The constraint evaluation order is defined by the user.

GrGen.NET [1] also implements pattern matching based on local search philosophy. It calculates search plans based on a cost model, that estimates a *backtracking* and an *execution time* for an operation. The used search plan has the lowest total cost of operations possible.

The algorithm we introduce in this work uses a search plan calculation algorithm that differs from the ones mentioned above. Also, the EMF-IncQuery framework is unique in a way that the declaratively defined patterns can be executed using any of the local search or the incremental algorithm, without specifying the concrete steps of the execution.

3.2 Algorithm adaptation

In order to efficiently integrate the local search-based algorithm, we were required to adapt many existing components of the EMF-IncQuery framework, more specifically parts of the *pattern matcher engine*. The architecture overview of the engine is depicted in Figure 3.1:

1. *EMF-IncQuery Base Index* collects the instances of `EClasses`, `EReferences` and `EDataTypes` contained in a model,
2. the *Pattern definition* holds the query that is to be evaluated over the instance model,
3. the *Pattern matcher* uses the pattern matcher algorithm to generate the result based on the EMF-IncQuery Base Index and the Pattern definition, and
4. the *Match set* contains the result tuples.

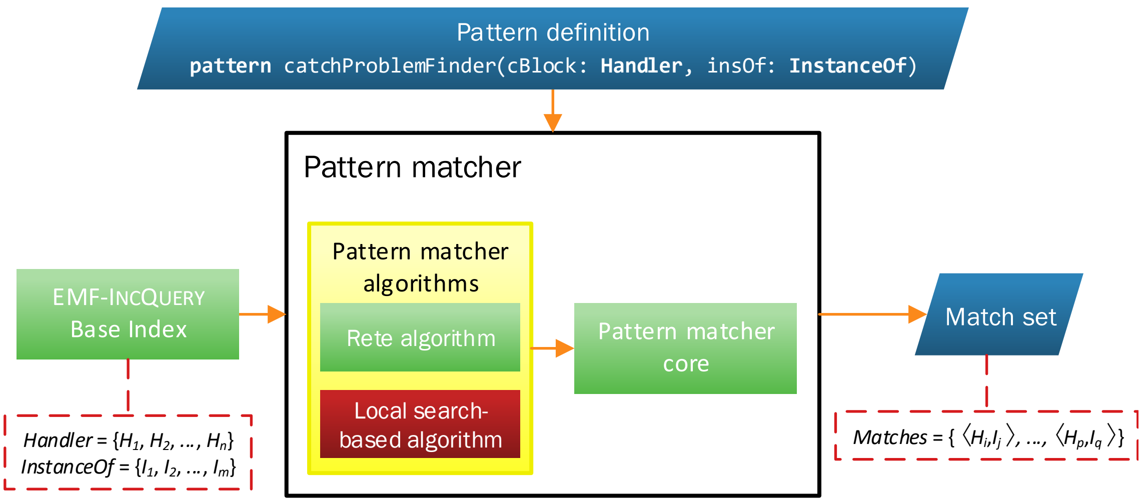

Pattern definition
pattern catchProblemFinder(cBlock: Handler, insOf: InstanceOf)
Pattern matcher
Pattern matcher algorithms
Rete algorithm
Local search-based algorithm
Pattern matcher core
EMF-IncQuery Base Index
Match set
$Handler = \{H_1, H_2, \dots, H_n\}$
$InstanceOf = \{I_1, I_2, \dots, I_m\}$
$Matches = \{ \langle H_i, I_j \rangle, \dots, \langle H_p, I_q \rangle \}$

Figure 3.1. The pattern matcher engine of EMF-IncQuery

In the introductory example, the base index would provide sets filled with the instances of the elements found in the metamodel, such as `Handler` and `InstanceOf`, as indicated in the lower left corner of Figure 3.1. The `catchProblemFinder` pattern definition states that the expected matches are supposed to be tuples with type ⟨`Handler`, `InstanceOf`⟩, as illustrated below the match set in the diagram.

Our contribution to the engine (emphasized with solid red background) is the *local search-based algorithm*, at least in the extent of this thesis. Prior to the implementation, we solved several adaptation issues. A typical task was to decide how to extend already existing interfaces in order to obtain all the information needed for executing the algorithm. One of the main design guidelines was generalization of the existing solution, so that several pattern matching algorithms may coexist in the EMF-INCQUERY framework.

In as a result of our work, we proposed a common interface for pattern matching algorithms. In addition, using the chosen solution, users can select the appropriate algorithm runtime, according to their needs. We also prepared the pattern matcher runtime for parallel search execution, for the computation of matches can be done by multiple threads simultaneously. The details about search execution is in Section 4.3.

While the *Graph pattern matcher engine* provides the basic functions of EMF-INCQUERY, there are many accompanying tools to ease the use of the provided features. To introduce the layered architecture of the toolkit based on [31], a diagram of its structure is shown in Figure 3.2.

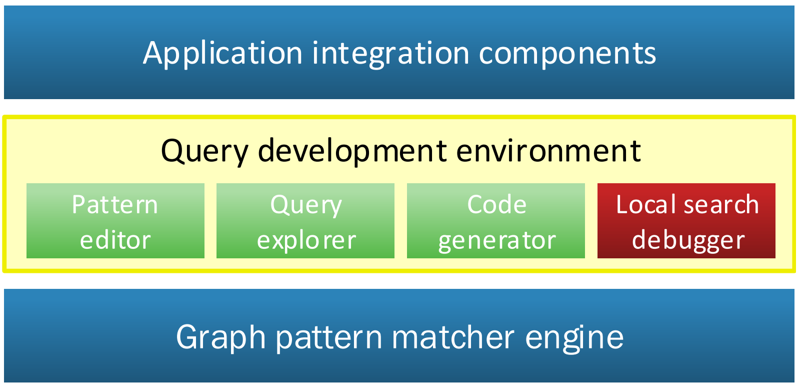

Application integration components
Query development environment
Pattern editor
Query explorer
Code generator
Local search debugger
Graph pattern matcher engine

Figure 3.2. The architecture of EMF-INCQUERY

The *Query development environment* (QDE) provides tooling related to defining and debugging queries. One of the major components is the (i) *Pattern editor*. This is an Xtext-based [29] editor for IQPL with syntax highlighting, auto-completion support and pattern well-formedness validation. The purpose of the (ii) *Query explorer* is to evaluate complex queries on selected EMF models, and to visualize the match set of each pattern. Another very important part of the QDE is the (iii) *Code generator*. It is tightly connected to the editor, as well as registered into the Eclipse builder framework. Thanks to this strong coupling with the IDE, code generation is executed after pattern definitions are modified, and saved. The output of the generation process is a pattern-specific Java code, that helps the integration of EMF-INCQUERY to Java applications by creating type-safe API for matchers.

To further extend the capabilities of the QDE, we developed a (iv) *Local search debugger* (highlighted with solid red background in the overview). It is designed to help the pattern developers understand and debug the local search-based pattern matching process by providing a visual representation of the calculated search plan, and to support step-by-step execution capability. Section 4.2.3 introduces the new debugger component and its capabilities in detail.

EMF-IncQuery provides several application integration components as well, as indicated in the top part of Figure 3.2. They are not discussed in this work in detail, but they provide an API for accessing the features of the EMF-IncQuery framework from Java programs.

3.3 Pattern matching workflow

We separate the local search-based pattern matching tasks into two categories. The first category is *design time* tasks, and the second is *execution time* tasks. Our pattern matching workflow is depicted in Figure 3.3.

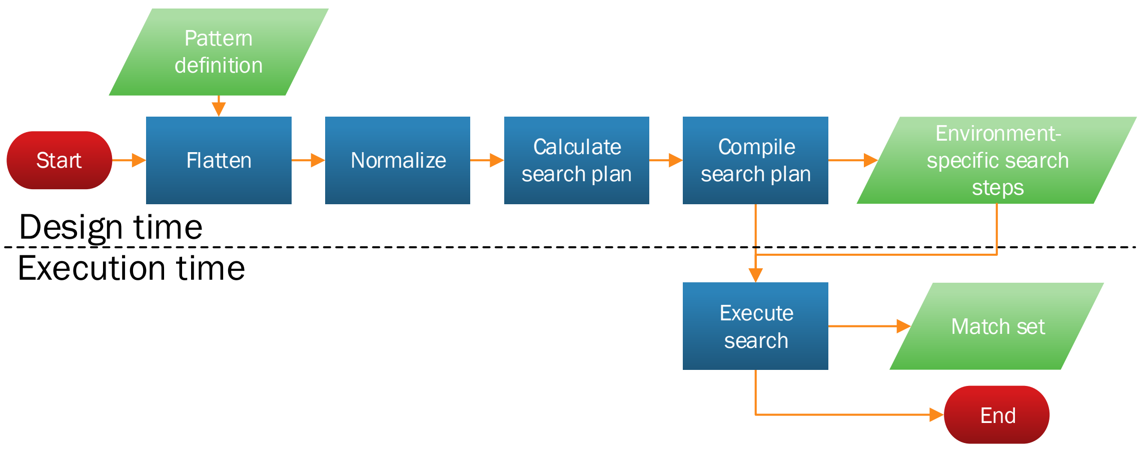


Figure 3.3. The local search-based pattern matching workflow

The majority of the steps of the workflow are considered to be *design time* tasks. The first task is to *flatten* the input *pattern description*. The pattern description may refer to other patterns, and flattening means the resolution of these references. As a result, a *flat pattern* is created that unifies all constraints and variables, both from the referrer and the referee patterns. This allows to optimize on a global scale, rather than locally for patterns. It is important to add that semantics of the pattern is preserved throughout the process. Details about the implemented flattener algorithm is discussed in Section 4.2.2.

In the next step the flattened pattern is *normalized*. This means an analysis in order to remove redundant, thus unnecessary constraints. For instance, type checking multiple times for the same variable, and with the same type is omitted. This step also unifies variables among equalities.

From the normalized pattern the *local search planner* calculates a *search plan* in the third step of the above workflow. In this case, this means an ordering of the constraints contained in the pattern. At this point all constraints and variables are directly contained in a normalized, flattened pattern that provides a global search space for the search plan calculation. As it was already mentioned in Section 2.3.1, the declarative definition of patterns using IQPL does not hold information about the order of constraint enforcement and computation of the matches of a pattern. This requires the local search planner component to create the search operation sequence that finds substitutions for variables. This list of search operation is derived from an ordered list of constraints, which is done in *compile search plan* step. The outcome of design time tasks is a list of *environment-specific search steps*. In case of EMF, these environment specific search operations heavily rely on the efficient EMF reflective API, which is introduced in Section 2.2.2, to obtain possible substitutions for the variables.

During *execution time*, this list of search steps determines the order of variable substitutions. In the *execute search* phase, the executor looks for substitutions that satisfy all constraints of the pattern. In the current application, it means that at some point, each variable is assigned a model element to check if it satisfies the description of the pattern. The details of search plan execution is detailed later in Section 4.3.

To demonstrate the effects of the tasks depicted in Figure 3.3, the flattened version of the `catchProblemFinder` pattern is presented in Listing 3.1 under the name `catchProblemFinder_flattened`.

```
pattern catchProblemFinder_flattened(cBlock : Handler, insOf : InstanceOf){
  Identifier(varRef);
  Unary.operand(insOf, varRef);
  cBlock = handlerVariable_cBlock;
  varRef = handlerVariable_variable;
  Handler(handlerVariable_cBlock);
  Indentifier(handlerVariable_variable);
  Handler.parameter(handlerVariable_cBlock, handlerVariable_param);
  Identifier.refersTo(handlerVariable_variable, handlerVariable_param);
}
```

Listing 3.1. Flattened pattern.

The body has all constraints from both `catchProblemFinder` and `handlerVariable`, as well as additional equalities that declare the variables used to serve as parameters for the pattern call constraint, and the corresponding variables from the flattened pattern are equal. New variables coming from the flattening process are prefixed with "`handlerVariable _`".

A normalized version of the `catchProblemFinder_flattened` pattern is `catchProblemFinder_flattened_normalized` (contained in Listing 3.2). To achieve better readability of the created pattern description, we unified the variables with longer names into variables with shorter names along equalities, and also simplified the name of `handlervar_param` to `param`.

```
pattern catchProblemFinder_flattened_normalized(cBlock : Handler, insOf : InstanceOf){
  Identifier(varRef);
  Unary.operand(insOf, varRef);
  Handler.parameter(cBlock, param);
  Identifier.refersTo(varRef, param);
}
```

Listing 3.2. Normalized pattern.

From the flattened and normalized pattern description, the search plan depicted in Table 3.1 can be obtained. The details of the search plan calculation is included in Section 4.1.1

	Constraint
1:	`varRef` is of type `Identifier`
2:	`varRef` is in `operand` relation with `insOf`
3:	`insOf` is of type `InstanceOf`
4:	`varRef` is in `param` relation with `refersTo`
5:	`param` is referenced by `cBlock` in `parameter`
6:	`cBlock` is of type `Handler`

Table 3.1. Search plan for the flattened pattern

If we compile the search plan for our introductory EMF-based example as the final step of the design time tasks, the list of environment specific search operations will be the following:

1. Collect all instances that are of type `Identifier`. Then, one-by-one substitute the values to variable `varRef` and advance to the next search operation.
2. For the current value of `varRef`, enumerate all values which can navigate to it on an `operand` reference. Substitute each enumerated element to `insOf`, then go on to the next operation.
3. Check if the type of the substituted element for variable `insOf` is `InstanceOf`.
4. Get all elements reachable from `varRef` by navigating on the `param` relation. Substitute each element reached this way in place of variable `refersTo`, then advance to the next operation.
5. Collect all elements that have the value of `param` in their `parameter` relation, then substitute each to `cBlock`.
6. Check if `cBlock` has a substituted element of type `Handler`.

This step is then followed by the *execution task* on the given instance model. During the execution in this current minimal example, there are only a few elements in every steps that should be considered. There is only one match in this case, a tuple that is $\langle \texttt{hdl}, \texttt{io} \rangle$ (using the notation of the hand drawn instance model in Figure 2.3).

Chapter 4

Integration to EMF-IncQuery

In this chapter the design and integration challenges are discussed in details. The implemented local search-based algorithm is from the paper [33]. The main advantage of this algorithm is its adaptive model sensitive approach for calculating search plans. It means that the properties of the instance model, on which the search plan calculation is executed, are considered. In this work we only introduce the main differences and key ideas that were required for a successful implementation and integration.

4.1 Details of the adapted algorithm

The published paper [33], which served as the basis of our implementation, is about pattern matching over EMF models. However, it did not include any programming source code, only pseudo code. For this reason, we adapted it to be executable on a JVM, and to also fit into the internal design of EMF-IncQuery. The current section introduces the concept of search plan, and the outline of the search plan calculation algorithm, along with an illustration of its execution on our running example. For a fully comprehensive description please refer to the article cited above.

4.1.1 Search plan

A search plan, in our case, means an ordered list of constraints of the pattern definition. As it was discussed in Section 3.3, the list of executable environment specific search steps are obtained by compiling the search plan. This compilation is simply a mapping between constrains and search steps, which only depends on the properties of the target modeling environment. For this reason, in the following we are only concerned about the search plan calculation. In the following we will refer to elements of the search plan as *search operations*, and we will represent them with the corresponding constraint from the query definition.

In order to clarify the purpose of the search plan, we define the concepts of *free variables* and *bound variables*. They can be interpreted as follows: if the search execution of the search plan was halted at a given point, which variables would already be assigned to a model element, and which variables would not have values yet. Variables without associated values are called *free variables* (F), and variables with assigned values are referred as *bound variables* (B).

The *pattern adornment* denotes the initial binding state of the parameter variables of a pattern at the beginning of the pattern matching, in other words describes witch parameter variables have initial values. The purpose of the search plan is to guide the search execution in a way that by reaching the end of the search plan each variable should already be bound.

The search operations can be categorized according to the binding state of the variables affected by the corresponding constraint. Two basic categories of search operations are *extend*, and *check*. A *check operation* verifies whether a variable substitution is in compliance with a constraint included in the pattern. This means that in case of a check, every variable of the operation is bound by the time of its execution. On the other hand, an *extend operation* has a list of substitution values for a variable, where these values are selected based on the corresponding constraint. During the search, all the elements in this list are to be substituted in order to find all matches in the model.

As a corollary of the definitions of check and extend operations, we can say that the position of the operation in the search plan decides to which category it belongs. However, there are some disallowed extend cases, which require a significant computational capacity during search execution, thus they are to be avoided in a search plan. For instance, negative pattern calls should always be check operations.

In the example, to find all occurrences in a given ASG of the code smell described by `catchProblemFinder`, the initial adornment should be FF, which means both `cBlock` and `insOf` parameters are unbound at the start of the pattern matching process. A possible search plan for the `catchProblemFinder` pattern is included in Table 4.1.

	Operation	Type	Bound variables
1:	`Indentifier(varRef)`	extend	{}
2:	`Unary.operand(insOf, varRef)`	extend	{`varRef`}
3:	`InstanceOf(insOf)`	check	{`varRef, insOf`}
4:	`handlerVariable(cBlock, varRef)`	extend	{`varRef, insOf`}
5:	`cBlock` is a `Handler(cBlock)`	check	{`varRef, insOf, cBlock`}

Table 4.1. Search plan for pattern `catchProblemFinder`

For there are no bound variables initially, the search plan starts with an extend operation. In this case it is an enumeration of instances of type `Identifier`. In the next step, by an *inverse navigation* from the bound `varRef` variable, along the `operand` relation possible elements for `insOf` are collected. When using the EMF-INCQUERY Base Index over a model, inverse navigation along edges are possible, even if the reference has no inverse in the metamodel. The third step is a check operation, which makes sure whether the value of `insOf` is of type `InstanceOf`. The fourth operation is an extend, for it binds the `cBlock` variable by calling the pattern `handlerVariable` with an adornment FB. As the result of the pattern call, each possible `Handler` block is collected for the value of `varRef`. In the final step of the search plan, the substituted value for `cBlock` is verified to be of type `Handler`. Note, that after the last search operation all variables are bound.

The search plan for the referred `handlerVariable` pattern with initial pattern adornment `FB` is shown in Table 4.2. However, we do not introduce the execution steps, because it is very similar to the steps of the search plan introduced for the `catchProblemFinder` pattern, except its initial pattern adornment is `BF`, which means that `cBlock` already has an assigned value.

	Operation	Type	Bound variables
1:	`Handler(cBlock)`	check	`{cBlock}`
2:	`Handler.parameter(cBlock, param)`	extend	`{cBlock}`
3:	`Identifier.refersTo(variable, param)`	extend	`{cBlock, param}`
4:	`Identifier(variable)`	check	`{cBlock, param, variable}`

Table 4.2. Search plan for pattern `handlerVariable`

To demonstrate the importance of flattening and normalization, we include a possible search plan for the pattern from Listing 3.2 in Table 4.3. As we previously discussed in Section 3.3, the flattened and normalized patterns have the same semantics as the original. However, the search plan for the normalized pattern is *simpler* than the two original search plans together, where by simple we mean it has less steps of the same operation type than the search plans calculated for the original descriptions.

	Operation	Type	Bound variables
1:	`Identifier(varRef)`	extend	`{}`
2:	`Unary.operand(insOf, varRef)`	extend	`{varRef}`
3:	`InstanceOf(insOf)`	check	`{insOf, varRef}`
4:	`Identifier.refersTo(varRef, param)`	extend	`{insOf, varRef}`
5:	`Handler.parameter(cBlock, param)`	extend	`{insOf, varRef, param}`
6:	`Handler(cBlock)`	check	`{insOf, varRef, param, cBlock}`

Table 4.3. Search plan for the flattened and normalized pattern

4.1.2 Search plan calculation

To find the ordering of the constraints that yields an efficient search plan is a non-trivial task. To estimate the time needed to execute the matching, we can assign *costs* to search operations, and from the individual costs of the operations we can derive the cost for the complete search plan. *Cheap* search plans are desired, for it means that the search can finish faster according to our cost estimation.

In addition to Table 4.3, the search plan listed in Table 4.4 can also be used to find matches for the `catchProblemFinder` pattern, they only differ in the order of operations. It is model-dependent, which provides faster execution.

	Operation	Type	Bound variables
1:	`Handler(cBlock)`	extend	{}
2:	`Handler.parameter(cBlock,param)`	extend	{`cBlock`}
3:	`Identifier.refersTo(varRef, param)`	extend	{`cBlock, param`}
4:	`Identifier(varRef)`	check	{`cBlock, param, varRef`}
5:	`Unary.operand(insOf, varRef)`	extend	{`cBlock, param, varRef`}
6:	`InstanceOf(insOf)`	check	{`insOf, varRef, param, cBlock`}

Table 4.4. An alternative search plan for `catchProblemFinder`

Calculating the cost of a search plan

The search plan calculation algorithm described in [33] proposes a method to calculate *operation weight* to encode the estimated costs of the operations and search plans. Our current implementation of the algorithm applies the proposed solution with minor modifications.

The main idea is to take every constraint form the pattern, and generate search operations with all possible and allowed variable binding combinations. This means, for every constraint it generates 2^a search operations, where a is the number of variables affected by the constraint. This could results in several search operation objects, however, typically constraints have 1 or 2 variables, and for pattern call constraint, we only allow `BB...B` adornments. We can say that in practical applications, the result search operation set will not reach extreme sizes typically. In the final search plan *each constraint will be represented by exactly one search operation*, and the unused ones will be discarded.

Along with the creation of search operations, an estimated number of potential substitutions values is calculated, which the executor has to check each time, when it is executed. This number is then used as its *weight* or *cost*. This means that check operations have a weight of 1, for they only verify whether the current substitutions satisfy a constraint.

For extend operations, we distinguish several cases. If the operation affects only one variable, which is in case of EMF-INCQUERY means a type constraint, then the cost of the operation is the cardinality of the type required by the constraint. In case of operations enforcing an IQPL path expression, which define that two model elements are connected by an `EReference` instance, there can be three cases according to the binding state of the of the affected variables:

- `FF`: this operation adornment is disallowed.
- `BF`: in this case the cost is estimated with the *average branching factor*, given by the formula $weight = \frac{cardinality\ of\ source\ class}{number\ of\ links}$.
- `FB`: similarly to the previous case, the average branching factor is calculated by $weight = \frac{cardinality\ of\ target\ class}{number\ of\ links}$.

Every other type of constraint, that is not flattened or normalized (`count find`, `neg find`, and inequality) are not allowed as extend operations, for they would typically require unmanageable search times. Positive pattern calls are at the moment always flattened.

The cited article distinguishes other constraint types as well, such as ternary constraints for ordered references, but in EMF-IncQuery there is no support to directly input such constraints yet.

From the w_{o_k} costs for each o_k operation, where k represents the position in the search plan, the total cost C_n of a search plan containing n search operations is calculated with the following recursive formula:

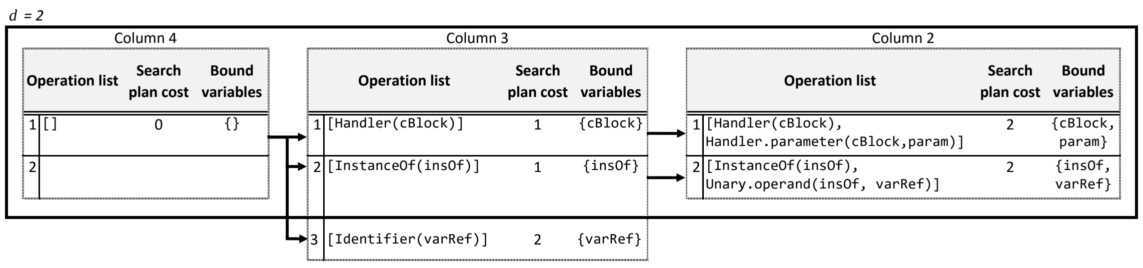
$$C_n = \sum_{i=1}^{n} \prod_{j=1}^{i} w_j = C_{n-1} + w_{o_1} \cdot w_{o_2} \cdot \cdots \cdot w_{o_n}.$$

Dynamic programming based search calculation

With the above definition of search plan cost, a dynamic programming based approach is used to create search plans that are cheap in this sense. The algorithm outline is as follows:

1. Initialize a table with dimensions $d \times (f + 1)$, where d is an input parameter of the algorithm. We call it *drop threshold*, because it influences which search plans are considered too expensive during the planning process, and thus ignored in later steps. The f symbolizes is the *number of free variables* in the initial adornment of the pattern. Each column symbolizes the number of free variables, and each cell represents a search plan. Column indices go from f to 0.
2. Starting from the fth column, we begin filling out the table using the rules below:
   - Based on a previous search plan, we pick an *applicable* extend operation, which means every variable affected by the operation and marked with `B` in its adornment is bound by the end of the previous search plan, and every variable marked with `F` in the operation adornment is free at the end of the selected, preceding search plan. When done, we append it to the search plan. In case we are filling out the fth column, and there is no preceding, then create a new one instead.
   - Append all applicable check operations to the search plan, then calculate a new cost using the recursive formula.
   - Insert search plans calculated this way in an increasing cost order to the table, and simply drop search plans that do not fit. The number of search plans in a column is determined by the parameter d.
3. Select the plan that is in the intersection of the first row and 0th column, because has the lowest cost, and yields `BB...B` variable binding after its final step.

Two steps of the running algorithm on the model depicted in Figure 2.3 is demonstrated in Figure 4.1 by filling up the columns 4, 3 and 2 of the table of search plans. The d parameter is set to 2.



Column 4

	Operation list	Search plan cost	Bound variables
1	[]	0	{}
2			

Column 3

	Operation list	Search plan cost	Bound variables
1	[Handler(cBlock)]	1	{cBlock}
2	[InstanceOf(insOf)]	1	{insOf}
3	[Identifier(varRef)]	2	{varRef}

Column 2

	Operation list	Search plan cost	Bound variables
1	[Handler(cBlock), Handler.parameter(cBlock,param)]	2	{cBlock, param}
2	[InstanceOf(insOf), Unary.operand(insOf, varRef)]	2	{insOf, varRef}

Figure 4.1. Table of search plans

Initially column 4 contains only one entry, which is a search plan with an empty list of operations, total cost of 0, and an empty set of bound variables. Based on the variable binding, the applicable constraints are `Handler(cBlock)`, `Identifier(varRef)`, and `InstanceOf(insOf)`, all with adornment `F`.

In the first step, the cost of the search plans containing only one of them is calculated. Using the recursive cost formula, it gives the cost of the operation plus the cost of the preceding search plan, which is in this case 0. Based on the instance model the costs are 1 for `Handler(cBlock)`, 2 for `Identifier(varRef)`, and 1 for `InstanceOf(insOf)`. The search plans are then inserted into column 3 in increasing cost order. We chose $d = 2$, so the third search plan is discarded, which means it is not stored in the table, thus not used as the basis of latter search plans.

In the second illustrated step, the same considerations are applied. First, the applicable search operations are collected for the search plans: `Handler.parameter(cBlock,param)` with adornment `BF` is for the plan in column 3, row 1, and `Unary.operand(insOf, varRef)` with adornment `BF` for the plan in column 3, row 2. Other operations are currently not available, neither extend operations, nor check operations, so only one new plan is derived for each of them. We can estimate the cost for `Handler.parameter(cBlock,param)` to $\frac{1}{1} = 1$, because our simple example model contains only 1 `Handler` element, that has only one `parameter` link. Similarly for `Unary.operand(insOf, varRef)` the cost is also 1. In both cases the costs of the new search plans will add up to $1 + 1 * 1 = 2$, using the recursive cost calculation formula again.

We do not introduce the complete run, but the paper [33], from which we adapted the algorithm, contains a complete running example on the process of calculating the search plan.

The complexity of the implemented algorithm is $\mathcal{O}(|V|^2 \cdot |O|^2)$, where V denotes the set variables, while O marks the set of constraints included in the pattern. The parameter d simplifies the calculation, because the plans that are more likely to have too high costs are discarded immediately, and not taken into account in later steps. However, if this value is chosen to be too small, the algorithm will execute similarly to a *greedy algorithm*. According to our experiences, choosing $d = 4$ is a reasonable setting for practical applications.

4.2 Implementation details

This section summarizes the main design decisions that were followed during the implementation. In Section 4.2.1 the common matcher interface for both the local search and the Rete pattern matcher introduced. Then in Section 4.2.2 the iterative flattener algorithm implemented for the local search planner is outlined. In Section 4.2.3 we describe the capabilities of the Local Search Debugger component, that can help developers to optimize the patterns.

4.2.1 Common matcher API

In terms of development of EMF-INCQUERY, one of the biggest recurring questions is to decide for which extent we are supposed to provide a generic solution to a problem, and to which extent we should give a specialized solution. Before answering this question, several aspects will be considered, such as modularity, reusability and backward compatibility.

When we decided to provide matchers that use the local search-based pattern matching algorithm for finding matches, there was already an existing Rete algorithm-based matcher implementation. To prevent breaking the API and causing backward incompatibility, the functions of the `RetePatternMatcher` were pulled up to an interface called `IQueryResultProvider`. For the new algorithm, a new class `LocalSearchResultProvider` was created that implements the (now common) `IQueryResultProvider` interface. This way users who already worked with a `RetePatternMatcher` can still do so, while users of the new API are advised to access a result provider via the `IQueryResultProvider` interface. Figure 4.2 shows the class diagram of the current solution.

Figure 4.2 illustrates the classes that are associated with the new `LocalSearchResultProvider`. The provider has a `LocalSearchPlanner` and a `LocalSearchMatcher`. The former is responsible for the search plan calculation, while the latter contains the search execution code. The planner has dedicated components for each process depicted in Figure 3.3: `PQueryFlattener`, `PQueryNormalizer`, `LocalSearchPlannerStrategy`, and `POperationCompiler`, respectively.

We reused the normalizer from the Rete algorithm. The other three components, which also realize steps of the search plan calculation workflow, are created with respect to modularization. Each provide a functionality that may be later replaced with a new implementation. This is especially true for the `LocalSearchPlannerStrategy`. It holds the business logic for the search plan calculation, so if an alternative version is to be created, only this component should be replaced or changed.

4.2.2 Flattening

The flattening is a novel feature in the EMF-INCQUERY framework. It is not needed by the Rete pattern matcher, but in case of local search-based pattern matching it supports cheaper

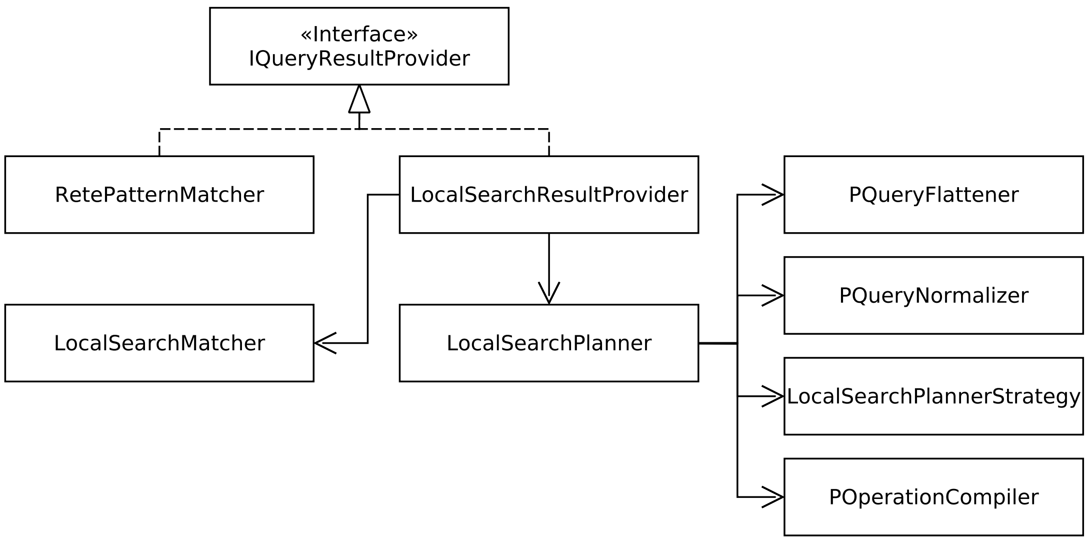


Figure 4.2. The internal structure of the new result provider

search plan calculation. Pseudo code of the core algorithm used for flattening is shown in Algorithm 1.

Algorithm 1: Flattening

```
input  : A pattern patternDescription
output: Semantically equivalent, flattened pattern
1  preStack ← ∅; postStack ← ∅;
2  preStack.push(patternDescription);
3  while preStack not empty do
4      item = preStack.pop();
5      postStack.push(item);                       // Schedule for flattening
6      for calledPattern ∈ set of patterns called by item do
7          if calledPattern needs flattening then
8              preStack.push(calledPattern);       // Fill up preStack
9          end
10     end
11 end
12 while postStack not empty do
13     item = postStack.pop();                     // Take out for flattening
14     item.copyCalledIntoCaller();
15 end
```

In line 1 two empty stacks are initialized: `preStack` holds the patterns that are encountered during the traversal of the call tree, while `postStack` holds patterns that need to be flattened, and all the referred patterns are flat. In line 2 the parameter pattern is scheduled for flattening. To flatten all calls, a *preorder depth-first traversal* is done on the pattern call tree between line 3 and line 11. This collects all calls that are necessary to be flattened at some point. This cycle leaves patterns untouched, only inserts them to the `preStack` by the time they are encountered during the traversal of the call tree.

From line 12 to line line 15 postorder actions are carried out. At this point we know, that the `item` in line 13 is a pattern that may refer only to flattened patterns, or to patterns that need no flattening. For this reason, the only remaining task is to copy the called patterns into the caller. The function `copyCalledIntoCaller` leaves the pattern untouched, in case it needs no flattening, in other cases it merges the called pattern constraints and variables to the caller. The reason for this can be the pattern contains no call, or some predicate tells that the pattern should not be flattened.

This algorithm can be used efficiently to flatten pattern calls. However, EMF-INCQUERY has language constructs, for which flattening cannot be applied, namely `neg find`, `count find`, and *binary transitive closure*. In these cases the matching should be done by calling matchers for the referenced queries.

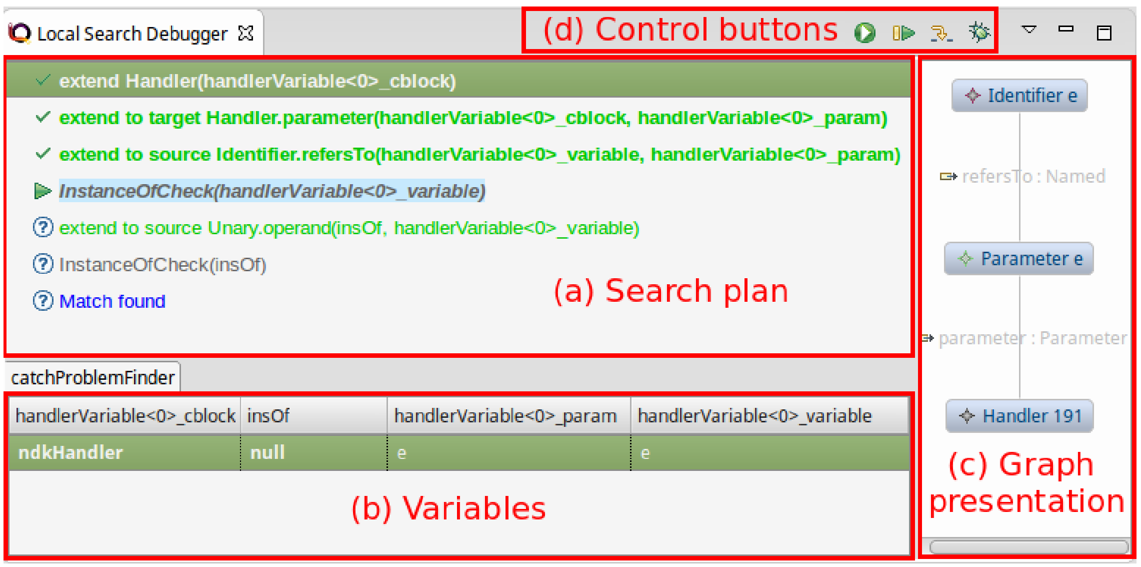
A limitation of the current implementation is the lack of support for recursive queries. This algorithm would end up in an infinite loop, if a recursive query was its input. For this reason, in the current implementation the planner first checks for recursion. If the pattern is not recursive, flattener can proceed, otherwise an exception is thrown.

4.2.3 Debugger tooling

This description of the Local Search Debugger tooling is from the paper [7]. The high-level, declarative nature of graph patterns sometimes results in hard to understand corner cases. In such cases simply looking at match results, as supported by the Query Explorer, does not provide enough details to locate the source of the problem. To support this use case, the development environment of EMF-INCQUERY has been extended with a *Local Search Debugger* view that follows through the execution of a search plan created for a pattern over a model.

As constraints of graph patterns are often not evaluated in the order of their definitions, it is hard to see which constraints are already evaluated during search execution. On the other hand, the ordered search operations visualize the status of pattern matching, and can be traced back to the source pattern. The view can also be used for pattern matching execution optimization, similar to explain plans [23] used for optimizing SQL queries.

As Figure 4.3 depicts a screenshot of the tool. Its view has four distinct parts to display information about as well as to control the execution. At the upper left corner (a) the search plan itself is shown, including the plans created for called patterns. Each line represents a search operation, and child nodes are operations of a called pattern. The current status of the execution is indicated with a set of icons: check marks are assigned to executed operations, question marks are assigned to operations not yet started, while the current operation is denoted with the 'Run' symbol. In this screenshot, the search plan contained in Table 4.4 is displayed for the `catchProblemFinder` pattern. It is also extended with a final virtual search step called *Match found*, which is only used to visualize the event of finding a match. The execution is halted after the execution of the first three extend operations, and the following check is ready for execution.



✓ extend Handler(handlerVariable<0>_cblock)
✓ extend to target Handler.parameter(handlerVariable<0>_cblock, handlerVariable<0>_param)
✓ extend to source Identifier.refersTo(handlerVariable<0>_variable, handlerVariable<0>_param)
▶ *InstanceOfCheck(handlerVariable<0>_variable)*
? extend to source Unary.operand(insOf, handlerVariable<0>_variable)
? InstanceOfCheck(insOf)
? Match found

catchProblemFinder

handlerVariable<0>_cblock	insOf	handlerVariable<0>_param	handlerVariable<0>_variable
ndkHandler	null	e	e

Figure 4.3. Local search debugger view

In the bottom left corner (b) a set of tables is presented on different tabs summarizing the found matches. The tabs have the same name as the corresponding pattern, and the tables include the found matches of all patterns in different tables on different tabs, including both parameters and local variables. In Figure 4.3, currently variables `cBlock`, `param`, and `variable` are assigned a value, and `insOf` is `null`, which indicates that it is not bound to a value.

In the right side (c) of the view, a graph representation is provided for the currently evaluated (partial) match, showing the current substitutions for the pattern variables along with the relationships between them. The example screenshot depicts that there is an object of type `Handler` (bottom), which is linked to a `Parameter` instance (middle). The `Parameter` instance is further linked to an object of type `Identifier` (top). The presentation also contains the names and types of objects, and the names of relations.

Finally, to control the execution, (d) standard debugging operations are available [25]. Breakpoints can be assigned to search operations either by selecting the operation and clicking on the bug icon, or by double clicking on an operation in the search plan. In addition, both step-by-step and continuous execution modes are available. The former is indicated with the *step into*, the latter is with the *continue halted execution* icon. To initiate the pattern matching process, the play button should be pressed after selecting the desired pattern in the Query Explorer.

This view complements the debugging capabilities of the Query Explorer, as the latter one is useful for identifying problematic cases by providing live feedback when the model changes, while the former visualizes the detailed execution of the search. The local search algorithm, in our experience, works similarly as a query developer reasons about a graph pattern, thus it eases the understanding of complex graph patterns.

4.3 Details of search execution

The search execution relies on the search plan to find suitable substitutions for the variables of the pattern. In the following sections a sequential, and a parallel execution approach is introduced in Section 4.3.1, and in Section 4.3.2, respectively.

4.3.1 Sequential search

A generic structure for a search plan is depicted in Figure 4.4. If an operation is scheduled for th first execution, is *initialized*, then *executed*. A search execution should always start with the initialization of the first operation.

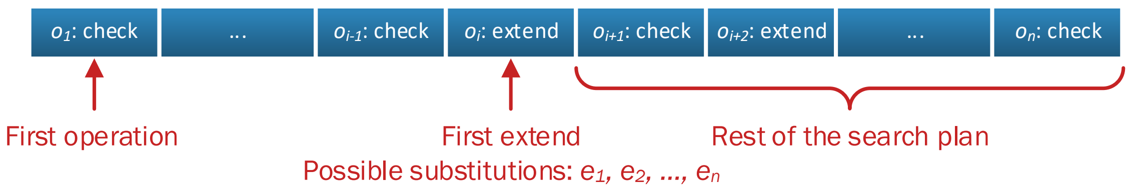


Figure 4.4. Search plan as list of search operations

For extend operations, the initialization means gathering all elements of a type, or enumerating model elements that are navigable from a reference of a given object. In general terms, it initializes a collection of potential substitutions for a variable. When executing the extend operation, there can be be two outcomes, which are *success* or *failure*. In case of success, the next value from the collection was successfully substituted into a variable. When execution fails, it indicates that there were no more potential substitutions available for the variable.

Initialization for check operations only means setting a boolean indicator `isExecuted` to `false`. When the check is executed, it sets `isExecuted` to true, and the substitutions of the affected variables are tested against the constraint that is assigned to the check operation. The result of the execution returns success, if the substituted values fulfill the requirements set by the constraint and the `isExecuted` flag was now set to `true`.

If the execution of an operation is successful, the executor may carry on to the next operation, otherwise it *backtracks*. Upon backtracking, the previous operation is not initialized again, only executed.

A match is found, when the last operation returns success. In this case, after storing the match, a backtrack is enforced on the last operation in order to find further matches, even though the operation returned success. Matching is finished, when the first operation returns failure.

A running example of the search is illustrated step-by-step in the followings. In this case the search is carried out on the example instance model captured in Figure 2.3, and uses the search plan included in Table 4.3.

Figure 4.5 displays the *search space tree* for the example. Its nodes are symbolizing different states from the aspect of variable bindings. Each of them is represented by a table containing a header with the names of all variables in the pattern, and a single row containing the names of the substituted model elements. The minus sign (—) symbolizes that a variable is unbound. The parameter variables are in the left part of the tables, while the right part, which is separated by double vertical lines, holds the rest of the pattern variables.

The directed edges are showing the possible transitions between the states by substituting a suitable value. There are no different states representing fully identical assignments to all variables.

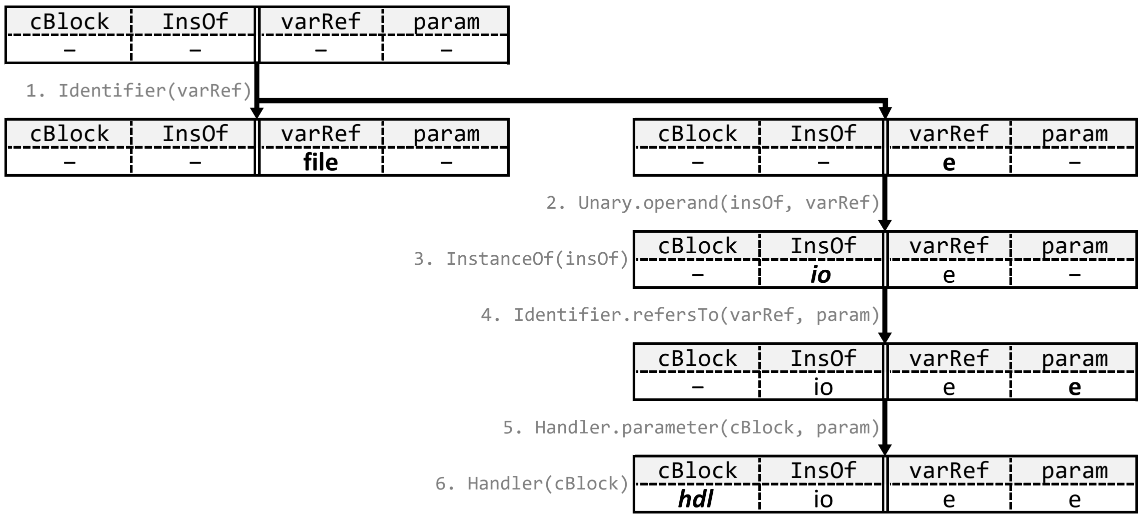

cBlock
InsOf
varRef
param
1. Identifier(varRef)
file
e
2. Unary.operand(insOf, varRef)
3. InstanceOf(insOf)
io
4. Identifier.refersTo(varRef, param)
5. Handler.parameter(cBlock, param)
6. Handler(cBlock)
hdl

Figure 4.5. Search space tree for search plan in Table 4.3

In our running example, the search starts with unbound variables. In the first step, which is the execution of an extend operation, the executor collects all possible substitutions for `varRef` in a list, which is in this case `[file, e]`. Then, it substitutes `file`, which is the first element, to `varRef`, and `proceeds` to the next operation. In Figure 4.5, the newly substituted values are indicated with bold letters. However, the model does not contain any element of type `Unary` from which the element `file` would be navigable via the `operation` reference, and for this reason the executor *backtracks*.

The next action is to substitute a new value to `varRef`, which is `e` with the type `Identifier`. Then again, the executor seeks for all `Unary` elements that has `e` as their `operand`. This case it gives a list of length 1, namely `[io]`. Then this only value is substituted to `insOf`. After substitution, it is made sure that its type is `InstanceOf` by the check operation, which is indicated by italic letters in the search space tree. Next, the elements of the `refersTo` relation is gathered, which is a list of containing the model element `e` of type `Parameter`, which is not to be confused with the `Identifier` already substituted to `varRef`. It is followed by collecting all objects from the model that can reach `e` via `parameter` reference. This case it is only the `hdl`, which is finally checked if it has the type `Handler`. This is also stands, so a match is found.

At this point the matching process is not over, for all other potential values gathered in extend operations should be tested, if they can form a match. For this reason, the execution backtracks until the last extend operation can pick a new value for substitution. However, this case every extend operations, including the first, have reached to the end of their list of substitution values. For this reason, after the backtracking is done, the pattern matching finishes.

4.3.2 Multi-threaded execution

If a search plan contains at least one extend operation, it means that the executor will test each possible substitutions one-by-one. This provides an opportunity, to distribute the substitution tasks among multiple executors. We proposed a simple solution in which the search plan is forked at the first extend operation. These forks are assigned to separate executors running on different threads in order to harness the additional resources available in multi-core/Hyper-threading computers. The basic idea of the solution is illustrated in Figure 4.6

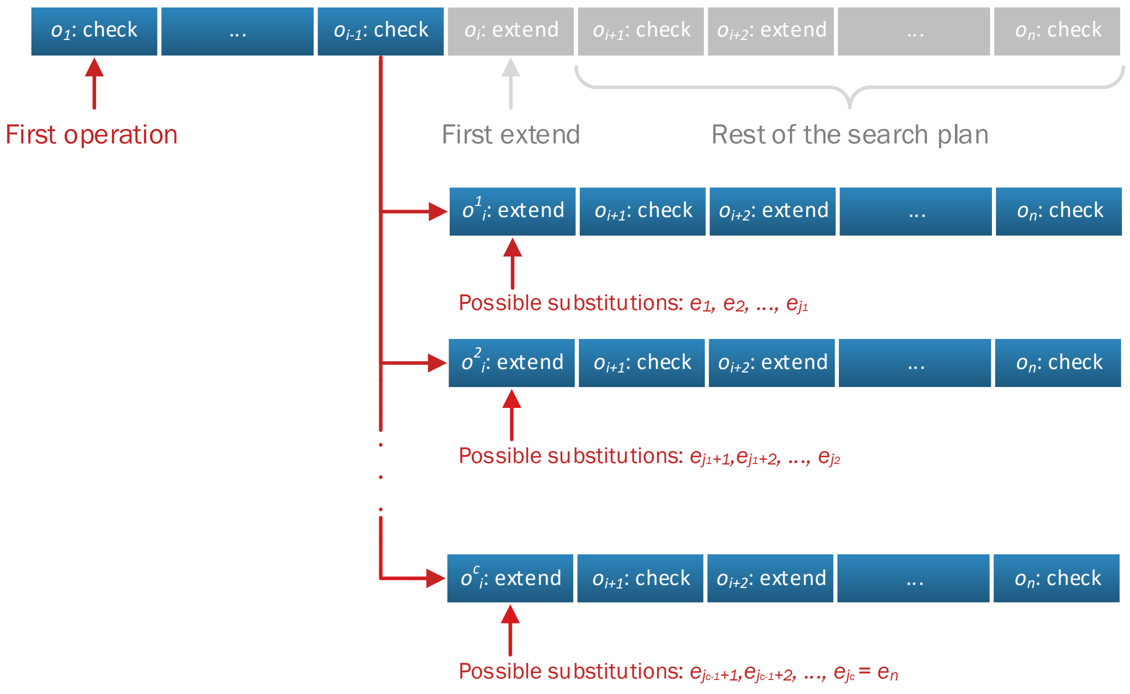


Figure 4.6. A possible parallelization of search plan execution

We can assume that the first extend operation in the search plan is o_i with the possible substitutions $e_1, e_2, \ldots, e_n$, and we also know that $e_p = e_q$ iff $p = q$, where $p, q \in [1, n]$. Then we can evenly distribute the values among c processors by creating c number of modified o^k_i extend operations. These modified extends should have substitution values $e_{j_k+1}, e_{j_k+2}, \ldots, e_{j_{k+1}}$, where $j_k = (k-1) \cdot \left\lfloor \frac{n}{c} \right\rfloor$ for every $k = 1, 2, \ldots, c-1$, and $e_{j_c} = n$

Creating a copy of the operations marked as *Rest of the search plan* in Figure 4.6 is necessary, because the search operations are represented by stateful objects. Preserving a state is required for extend operations to maintain the internal collection of potential substitutions, and also required for check operations to handle backtracks with the `isExecuted` flag.

We can summarize our solution for parallelization in three steps:

1. The task is to instantiate modified extend operations, marked with the o_i^k in Figure 4.6, and distribute the candidate elements of the first extend among them. This step also includes creating a copy from the rest of the search plan.

2. Carry out sequential execution for each newly created search plan. Each executor should maintain its own result set, instead putting the matches in a shared collection. The latter option was also considered, but in order to avoid synchronization and blocking, we decided to merge the results in a separate step.

3. Await all executors, then merge the results. The uniqueness of matches is automatically ensured by the fact that the modified extend operations work with disjoint lists of model elements, which is the result of the initial assumption of $e_p = e_q$ iff $p = q$, where $p, q \in [1, n]$.

4.3.3 Advantages and weaknesses

A huge advantage of the solution is the possibility to utilize all physical execution cores of the hardware. Based on *Amdahl's law*, using a CPU with c cores, the time t needed for the sequential *the search execution* could drop to $(1-p) \cdot t + \frac{p}{c} \cdot t$, where $p \in [0,1]$ expresses the portion of the instructions that can run in parallel. If there was $p \approx 1$, it would yield $\frac{t}{c}$ for the new execution time. This sounds promising, but the evaluation results in Section 5.4 and in Section 5.5 will show that the speedup in practice is not even close to this bound generally, which means a significant part of the instructions cannot run in parallel.

As a drawback, the solution has preparation overhead. This currently means the creation of the executor thread pool, the distribution of the model elements among executors, and the cloning of the rest of the search plan. Additionally, when creating sublist from the list of elements of the first extend operation, it is unknown how long will it take for the executor, until all matches that involve the element will be found. It may turn out soon that the value fails to fulfill even the constraint associated with the next check, but it is also possible that the substituted value is part of several matches, taking the executor long time to compute them all.

Chapter 5

Evaluation

In the current chapter the environment used for assessing the implementation is introduced in Section 5.1, Section 5.2 and in Section 5.3, then the performance of the implemented local search-based pattern matching algorithm is evaluated in Section 5.4. An additional measurement was done for scalability in an environment with limited memory in Section 5.5.

5.1 Measurement workflow

The measurement setup is composed of four phases: (i) *Read*, (ii) *Create engine*, (iii) *Calculate search plan*, and (iv) *Check*. The steps are depicted in Figure 5.1.

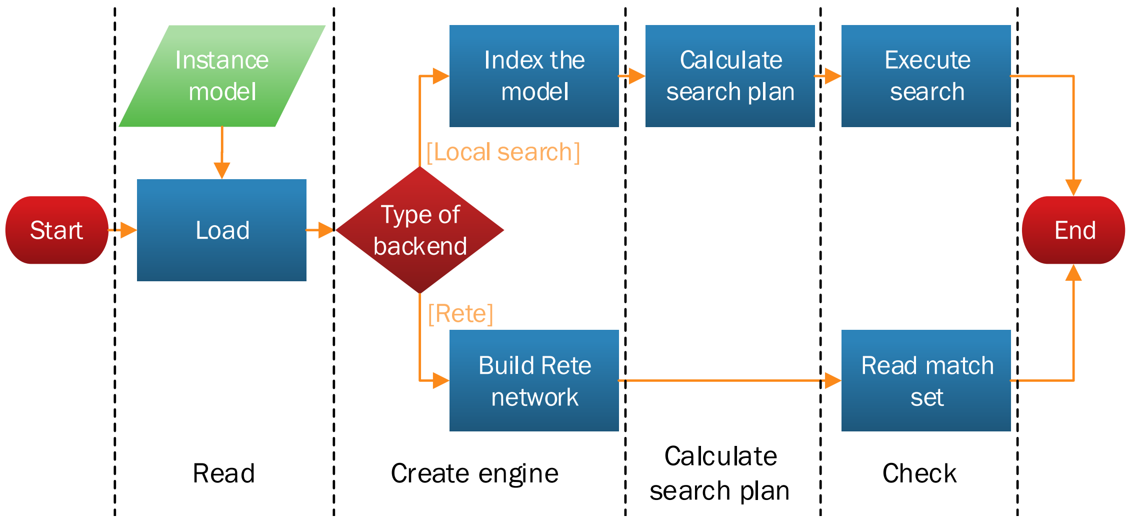


Figure 5.1. The workflow used for measurements

The Read step measures merely the time needed to load the EMF model into memory. The Create engine step is different for the incremental, and local search-based approaches. For a local search based-algorithm only a basic indexing of the elements is performed in order to provide cardinality information for the search plan calculation. In case of Rete, in addition to the indexing, the whole Rete network is created, which means most of the pattern matching is done here. Calculate search plan phase is only related to local search, for in this step the search plan calculation time is monitored. Finally the time needed for the retrieval of results is observed in the Check phase.

5.2 Measurement environment

The computer used for carrying out the measurements was a ThinkPad T440p laptop, running Linux 3.16.0-38-generic x86_64 operating system, and had the following hardware configuration:

- CPU: Intel® Core^{TM} i7-4700MQ CPU @ 2.40GHz
- Memory: 2 × 4096 MB DDR3 @ 1600 MHz
- SSD: Intel 320 Series SSDs, model: INTEL SSDSA2CW160G3

We used Java^{TM} SE Runtime Environment (build 1.8.0_66-b17). In order to successfully load every used model to memory, we supplied the JVM 6 gigabytes of heap size by using the `-Xms6G` and `-Xmx6G` parameters.

We tested the performance of the incremental Rete (referred as *Incremental*), the single-threaded local search-based (*Sequential*), and the parallel local search-based (*Parallel*) algorithms. We assessed the performance using the newest algorithm versions available on December 10, 2015 in EMF-INCQUERY.

5.3 Models and patterns used for assessment

In order to evaluate scalability of the algorithms, we carried out measurements on three different model sizes with four different queries. Both the models and queries are from [32].

The selected models for this work are the EMF representations of the ASGs of programs *Qwicap*, *Frinika*, and *Hibernate*. Qwicap is a library for Java web application development. Frinika is a music workstation software, which provides several features for editing and working with music. Hibernate is a tool that can be used to implement object-relation mapping for applications to persist data. We chose these softwares, because both are free and open-source, and have different sizes, thus they can help assessing scalability of the approaches. Table 5.1 summarizes the basic metrics of the source code and the generated ASGs.

Model name	Version	LOC	Node Count	Edge Count
Qwicap	1.4b24	443	7,903	21,222
Frinika	0.5.1	64,828	429,407	1,292,961
Hibernate	3.5.0	773,166	2,444,419	7,563,207

Table 5.1. Metrics of the analyzed ASGs

The four cases if code smell are named *Catch*, which is the same as the example `catchProblemFinder` pattern describes, *Constant compare*, *No default switch* and *Unused pa-*

rameter. The descriptions of the latter three anti-patterns are again taken from [32], and are as follows:

- *Constant compare*: When a String variable is compared to a String literal using the `equals()` method, it is unsafe to have the variable on the left hand side. Changing the order makes the code safe (by avoiding null pointer exception) even if the String variable to compare is `null`.
- *No default switch*: Missing default `case` has to be added to the `switch`.
- *Unused parameter*: When unused parameters remain in the parameter list they can be removed from the source code in most cases.

In Table 5.2 we summarized the total number of variables, constraints, and any type of pattern calls used to describe the problem using IQPL. These numbers does not provide precise description, however, they tell that these patterns are adequate for performing measurements to test scalability of the approaches from the aspect of pattern complexity. The *No default switch* is considered to be the simplest, while *Unused parameter* the most difficult to match.

Anti-pattern problem	Variables	Constraints	Pattern calls
Catch	6	9	1
Constant compare	9	10	4
No default switch	3	5	1
Unused parameter	19	29	9

Table 5.2. Main attributes of the patterns

5.4 Performance evaluation

The output of the assessment for the three different models are summarized in Figure 5.2. For detailed numeric results, please refer to Table A.1, Table A.2, and Table A.3 in the Appendix. Based on the times needed for each task, we obtained multiple conclusions.

In the Read phase, loading the model takes significant amount of time. In case of large models this is the longest task, but for smaller sizes it is still comparable to engine creation. This complies with the fact that engine creation involves indexing the model elements, and for the Rete algorithm, also building the Rete network based on the index. This also explains why it takes multiple times more, according to the measurements, to finish the engine creation phase for the incremental algorithm.

For the search plan calculation phase, however, unexpectedly large execution times were encountered. This is the step, when the constraints of the pattern are put into an ordered list that makes up the search plan. The current implementation of the algorithm uses cardinality information of types about the instance model in order to determine the order of the operations. After a more detailed analysis of the relevant module, it turned out, that the base indexer has a

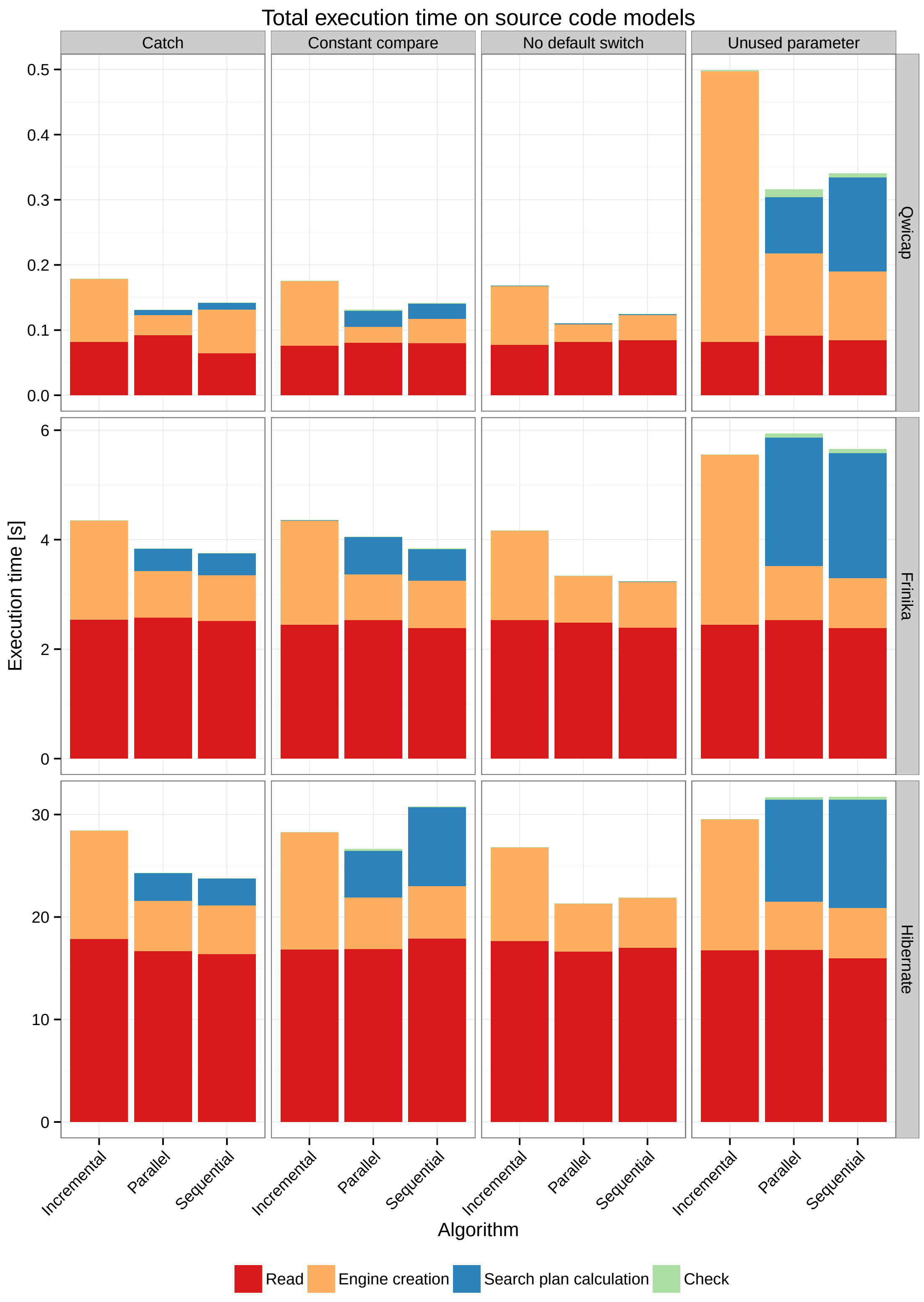


Figure 5.2. Measurement results for anti-pattern detection in ASGs

fairly suboptimal implementation for this part. The method `countTuples(IInputKey key, Tuple seed)` in `EMFQueryRuntimeContext` currently returns the result of the `size()` method called on the result of `baseIndex.getAllInstances(eClass)`. The already existing implementation of the `getAllInstances(EClass type)` method of the `NavigationHelperImpl` class is included in Listing 5.1.

```
 1 @Override
 2 public Set<EObject> getAllInstances(EClass type) {
 3     Set<EObject> retSet = new HashSet<EObject>();
 4     Object typeKey = toKey(type);
 5     Set<Object> subTypes = contentAdapter.getSubTypeMap().get(typeKey);
 6     if (subTypes != null) {
 7         for (Object subTypeKey : subTypes) {
 8             final Set<EObject> instances = contentAdapter.getInstanceSet(subTypeKey);
 9             if (instances != null) {
10                 retSet.addAll(instances);
11             }
12         }
13     }
14     final Set<EObject> instances = contentAdapter.getInstanceSet(typeKey);
15     if (instances != null) {
16         retSet.addAll(instances);
17     }
18     return retSet;
19 }
```

Listing 5.1. The implementation of the `getAllInstances` method.

This method, as its name advices, returns a collection of all objects of a given type. If we inspect the implementation, in line 3, a new `Set` is instantiated. All subtypes of the given type are collected in line 5. Between line 6 and line 13 all instances of all subtypes are added to `retSet`, if there were any. In line 16 all direct instances of the given type are added to the set containing the collected subtype instances. Finally, in 18 `retSet` is returned. According to our measurements, adding approximately 100,000 elements to an empty set, then calling `size()` takes about 20 ms on the computer, on which the performance evaluation was carried out. For these cardinality information are not cached, if it is needed multiple times, the same gathering of elements is carried out again. It also turned out, that the planner in many cases does ask for cardinality information several times for a type.

An important observation is that generally the execution of Check phase may require orders of magnitude less time than other steps. In case of the Rete algorithm this step consists of only returning a copy of a collection in which matches are stored, because it is already prepared by the time the Rete network is built. The local search-based algorithm, however, computes the matches in this final phase of measurement. Thanks to the efficient, model sensitive search plan, which is calculated in the preceding phase, the search for matches is done under a few milliseconds for most cases. We experienced slightly notable durations for the largest used model, Hibernate. In this case the matcher for *Constant compare* and *Unused parameter* patterns worked several hundred milliseconds to compute matches. It is also important to add, that these patterns are considered to be more complex than *Catch* and *No default switch*.

The results also shed light on the shortcomings of the current search parallelization. The implementation creates a thread pool with c threads, where c equals the number of currently available cores of the CPU. Then, the parallelization starts when the search execution reaches the first extend operation, where the potential substitutions are divided between the threads. The creation of a thread pool and distributing the work among threads seemingly imposes some overhead, as it was foreseen in Section 4.3.3.

To decide which algorithm should be used in a certain scenario depends on several circumstances. According to the results, we can estimate the number of runs, where the Rete algorithm outperforms the local search-based one in runtime. If we were to match the *Catch* problem against the model *Frinika* only once, it would take $2.535+1.813+0+0.0017 = 4.3497$ seconds with the incremental algorithm. The same scenario would took $2.512 + 0.842 + 0.400 + 0.0009 = 3.7549$ seconds for the sequential local search-based algorithm. However, if we increased the number of runs on a loaded model with an already created engine, each additional run after the first one would took only 0.0017 seconds with Rete, and 0.4009 seconds with the current implementation of the sequential local search-based version. From these data, we can obtain a run count threshold, for which incremental algorithm is more beneficial with respect to execution time. This run threshold r comes from the equation

$$2.535 + 1.813 + r \cdot 0.0017 = 2.512 + 0.842 + r \cdot (0.400 + 0.0009),$$

and it yields $r \approx 2.5$. So it means, if we run the same anti-pattern detection several times on the model only one or two times, it is worth using local search, in other cases Rete will finish faster. If we apply the above calculations for the measurement results of *No default switch*, we get a threshold $r \approx 1546.7$. However, we have to remark three important factors regarding the above calculations:

- Rete network update times are neglected in this calculation, for they are assumed to be minimal for small model changes.
- Currently the search plan calculation is highly ineffective due to a shortcoming in the implementation of the base indexer. If this problem were fixed, the time needed for plan calculation for local search would notably drop.
- In these cases the memory bounds were not a limitation.
- Search plans are not cached for later execution.

5.5 Running the Train Benchmark

Train Benchmark is a benchmarking framework aiming to test query evaluation performance of model-driven engineering tools [18]. This evaluation is carried out by matching the same patterns against several models with growing sizes. For this purpose Train Benchmark has a model generator component that creates synthetic models for a given size range. It also provides predefined patterns to match against the generated models, for which basic complexity information are included in Table 5.3. For detailed description of the models provided by the generator and the patterns, please refer to the cited technical report above.

Pattern	Variables	Constraints	Pattern calls
ConnectedSegments	7	7	0
RouteSensor	4	4	1
SemaphoreNeighbor	7	8	1
SwitchSet	4	7	0

Table 5.3. Train Benchmark pattern names and complexities

In our case we used this framework to test the scalability of the pattern matching algorithms on the benchmark scenario depicted in Figure 5.1. We ran these measurements on a virtual machine with the following hardware configuration:

- CPU: Intel® Xeon® CPU E5-2630L v2 @ 2.40GHz (12 cores)
- Memory: 32 GB DDR3 @ 1600 MHz
- SSD: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]

We used Java^{TM} SE Runtime Environment (build 1.8.0_66-b17), and ran the JVM without extra parameters, which means the heap size limit was 1 GB.

Benchmarking memory consumption is a non-trivial issue in case of managed environments like Java. For this reason we chose a limit for the maximum heap size. From this limit we will not know the exact amount of memory needed, an *out of memory error* can indicate that the program could not fit in the available size. This strategy can be used to decide which algorithm needs more memory than others.

The performance characteristics were similar to what we described previously in Section 5.4. The search plan generation times for the patterns of Train Benchmark, depicted in Figure 5.3, were increasing with model size (in the diagram both axes are logarithmic). This also leads to the conclusion that obtaining type cardinality information depends on model sizes. The diagram also shows that search plan calculation for different patterns for the same model size take time based on the complexity of the pattern.

On the synthesized models of Train Benchmark, we measured mostly linear characteristics for both the sequential and the parallel local search-based pattern matching. The results are depicted in Figure 5.4 (again, note the logarithmic axes in the diagram). In this cases, we can observe almost immediate retrieval of results in case of Rete for every pattern. However, for the local search based-pattern matching solutions both scale up in a linear way with the model sizes. We can see that the parallel version of the implementation has an overhead compared to the sequential solution, by comparing the check times needed for small model sizes. This overhead, however, may pay off, since in case of large models the parallel version completes faster.

The benchmark also showed that the incremental algorithm is more constrained by memory than local search-based approaches. While running the benchmark with *ConnectedSegments*

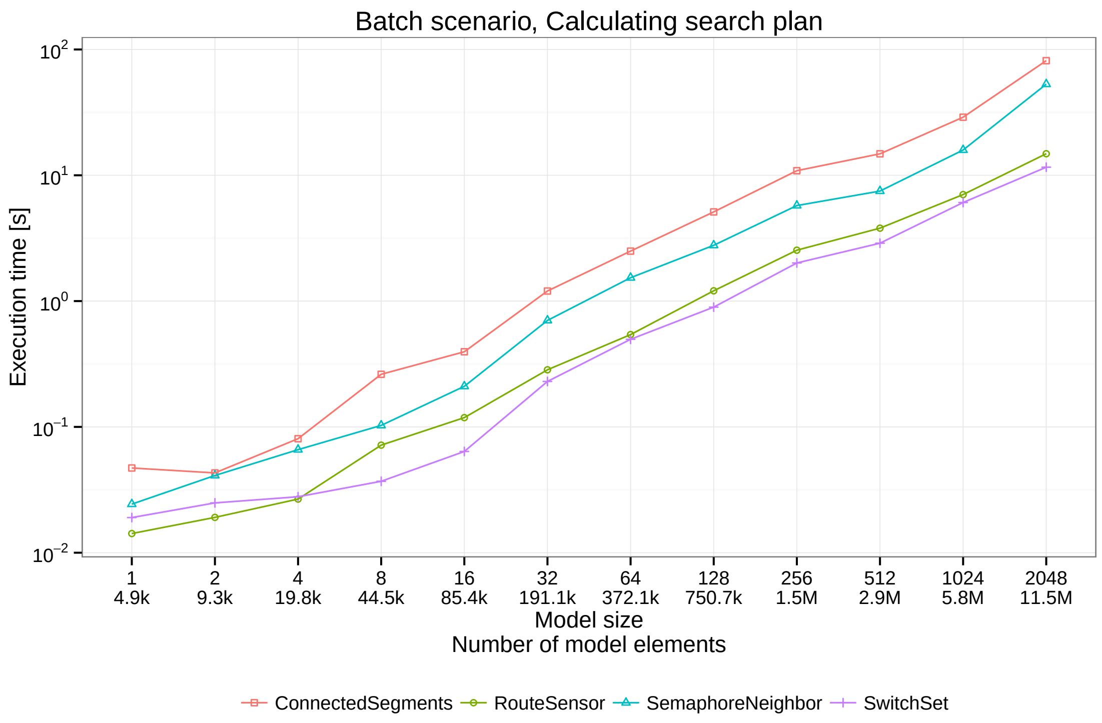


Figure 5.3. Search plan calculation times for different patterns

on model scale 2048, the measurement was terminated by an out of memory error, showing that the Rete network grew too large in this case.

The rest of the diagrams and tables, which contain the evaluation results of the benchmark, are included in Section A.3 in the Appendix.

5.6 Evaluation summary

Based on the performed measurements, we came to the conclusion that the *incremental Rete algorithm* runs significantly slower for the first time, compared to the implemented local search-based approach. It also requires more memory for execution. However, we recommend to use this algorithm despite these extra requirements if the same pattern should be matched against the model multiple times consecutively, and if sufficient amount of memory is available.

The sequential local search-based solution has lower memory requirement in many cases, compared to Rete. If the pattern matching is a one-time task, and the cardinality information about the model elements is available, then this is the recommended algorithm.

The parallel local search-based approach differs from the sequential version only in search execution. It has an initial overhead, but for large models it is more preferable in cases where local search should be used.

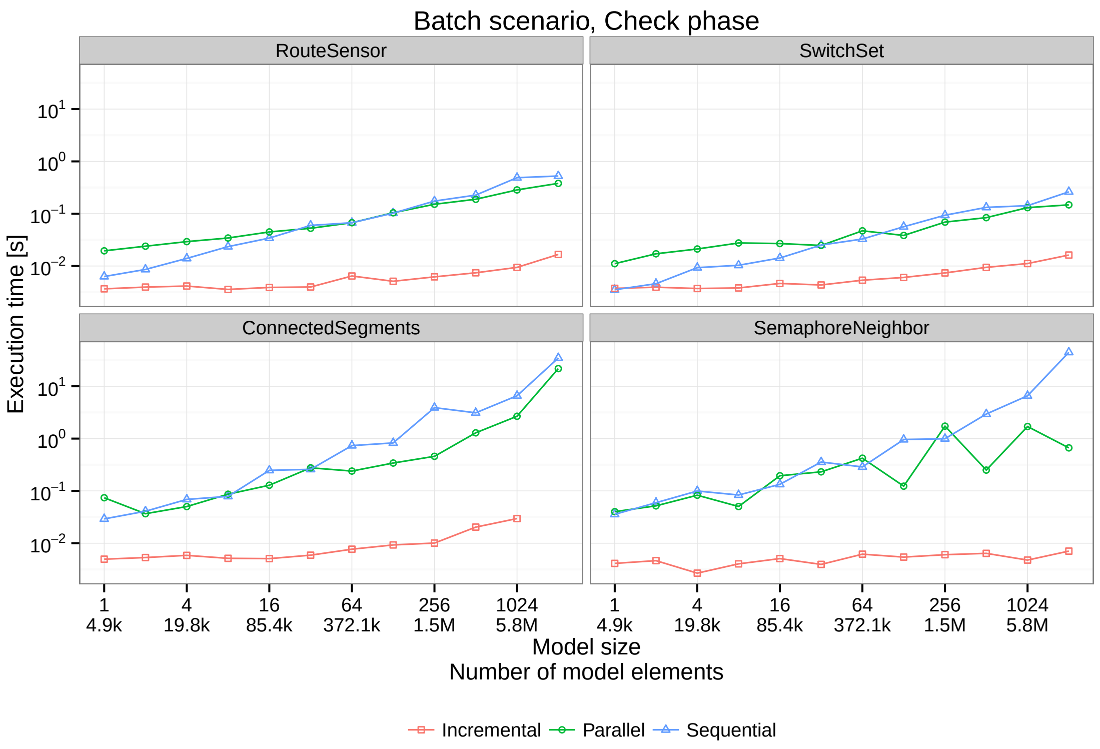


Figure 5.4. Times needed for the Check phase

Chapter 6

Conclusion and future work

6.1 Conclusions

We implemented and integrated to the EMF-IncQuery framework a model adaptive search plan calculation algorithm for local search-based pattern matching. We also provided two search execution runtimes, a single-threaded and a simple parallel runtime.

The implementation was tested for performance and scalability, and was compared to an already existing, incremental pattern matching algorithm from the aspect of execution time. This evaluation, and comparison was done for models coming from two different domains. First part of the measurements were done on source code models with patterns describing program code smells. Additionally, we used the Train Benchmark to assess the usability of the proposed solution. The results proved that in scenarios, in which patterns are only matched against a model.

We also provided tooling support for the local search algorithm with a Local Search Debugger component, which can help programmers find problems with the pattern definitions, and provide help in finding time consuming operations in the pattern matching process.

6.2 Summary of contributions

I successfully adapted a local search-based pattern matching algorithm, which involves a model sensitive local search planner implementation, and two executor runtimes. During the implementation solved several design issues and integrated the result to the open-source EMF-IncQuery project. I provided support for debugging the local search execution by the Local Search Debugger View.

I assessed the performance and scalability of the solutions, and via the evaluation I showed that the local search-based algorithm provides a scalable solution for pattern matching even over large models. The measurements also shed light on a shortcoming of the current implementation of the EMF-IncQuery Base Indexer.

6.3 Future plans

The ultimate goal of the EMF-IncQuery graph pattern matching framework is to successfully, and efficiently find matches of a pattern over a given model. To improve execution times of the local search based approaches, there are several improvement possibilities.

6.3.1 Information about type cardinality

It was covered in Section 5.4 that the current implementation of the base indexer is suboptimal. In addition to optimizing the current implementation, the performance can be improved by providing an option to only maintain type cardinalities. This could return the required information with minimal overhead in execution time. It would also take up less memory than the current implementation of the base index, for it would not keep track of the model elements themselves, only just the number of instances of the types. This information is sufficient in most cases, when inverse navigation along links is not a requirement.

6.3.2 Adaptive cost calculation

The search operation costs are calculated based on the instance model properties, type of the corresponding constraint, and the types of operations, which can be extend or check. This, however, is still an estimation for the time needed to execute the operation. For this reason, this information may mislead the local search planner, when an expensive operation is declared to be cheap by the cost function.

The times needed for operations to execute can be collected runtime. We suppose that by analyzing the historical data about the execution times can significantly help search operation cost estimation by assigning different kinds of bias to search operations that take a long time to finish.

6.3.3 Advanced parallel execution

As it was emphasized in Section 4.3.3, the current parallel pattern matching execution runtime relies on a basic concept, and may not result in significant speedup. The evaluations show that on practical models, in our case obtained from software source code, this approach does not seem to be successful. In case of artificial models, a tendency was discovered that for larger sizes the parallel outperforms the sequential implementation, but the execution times are far below the theoretic possible speedup.

In order to approach the desired execution time introduced also in Section 4.3.3, *work stealing* should be applied. This would mean that at any point of the search plan, the matching process could be forked by several sequential executors. In an ideal case, this would happen in an on-demand way during search execution: if an executor seems to have significantly more work at a given extend operation, while other executors have finished their part, the work could be redistributed, thus the computational resources of the platform could be harnessed again.

6.3.4 Hybrid pattern matching

In the EMF-IncQuery framework there are two available graph pattern matching strategies for the users, and both approaches has their own advantages and drawbacks. It would be beneficial, to allow their combination in order to profit from their advantages. To accomplish this, we see the following options. (i) Local search based pattern matcher may call incremental pattern matchers for calculating matches of calls. (ii) In case of a Rete matcher, the production representing a called pattern can be provided by the result set of a local search-based matcher.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisors, Ákos Horváth and Zoltán Ujhelyi for their continuous support and invaluable advices throughout my work.

Besides my advisors, I thank Gábor Szárnyas for all the precious help with the Train Benchmark framework, and data visualization.

Finally, I would like to thank the staff working for DigitalOcean for providing me a multi-core virtual machine to run Train Benchmark.

List of Figures

2.1 The ECore metamodel 6

2.2 EMF metamodel fragment for ASGs 7

2.3 The ASG representation of the example code snippet 8

2.4 Rete network for the `catchProblemFinder` pattern 10

3.1 The pattern matcher engine of EMF-INCQUERY 13

3.2 The architecture of EMF-INCQUERY 14

3.3 The local search-based pattern matching workflow 15

4.1 Table of search plans 23

4.2 The internal structure of the new result provider 25

4.3 Local search debugger view 27

4.4 Search plan as list of search operations 28

4.5 Search space tree for search plan in Table 4.3 29

4.6 A possible parallelization of search plan execution 30

5.1 The workflow used for measurements 32

5.2 Measurement results for anti-pattern detection in ASGs 35

5.3 Search plan calculation times for different patterns 39

5.4 Times needed for the Check phase 40

A.1.1 A more detailed EMF metamodel fragment for ASGs 50

A.3.1 Train Benchmark – Read phase 53

A.3.2 Train Benchmark – Create engine phase 53

List of Tables

3.1 Search plan for the flattened pattern 17

4.1 Search plan for pattern `catchProblemFinder` 19
4.2 Search plan for pattern `handlerVariable` 20
4.3 Search plan for the flattened and normalized pattern 20
4.4 An alternative search plan for `catchProblemFinder` 21

5.1 Metrics of the analyzed ASGs 33
5.2 Main attributes of the patterns 34
5.3 Train Benchmark pattern names and complexities 38

A.1 Measurement results for Qwicap (in seconds) 51
A.2 Measurement results for Frinika (in seconds) 51
A.3 Measurement results for Hibernate (in seconds) 52
A.4 Benchmark results for the incremental algorithm 54
A.5 Benchmark result for the sequential local search-based algorithm 55
A.6 Benchmark result for the parallel local search-based algorithm 56

Bibliography

[1] Gernot Veit Batz, Moritz Kroll, and Rubino Geiß. A first experimental evaluation of search plan driven graph pattern matching. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers*, volume 5088 of *Lecture Notes in Computer Science*, pages 471–486. Springer, 2007.

[2] Gábor Bergmann. *Incremental Model Queries in Model-Driven Design*. PhD thesis, Budapest University of Technology and Economics, Budapest, 2013.

[3] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for EMF models. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations - 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, volume 6707 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2011.

[4] Franck Budinsky, David Steinberg, and Raymond Ellersick. *Eclipse Modeling Framework: A Developer's Guide*. Addison-Wesley Professional, 2003.

[5] Frank Budinsky. Moving into model-driven development. `http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/2005/0508/0508c/0508c.html`.

[6] Márton Búr. Model-based validation of Matlab-Simulink systems. Bachelor's thesis, Budapest University of Technology and Economics, Budapest, 2013, 2013.

[7] Márton Búr, Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró. Local search-based pattern matching features in EMF-IncQuery. In Parisi-Presicce and Westfechtel [24], pages 275–282.

[8] Jordi Cabot and Ernest Teniente. Incremental integrity checking of UML/OCL conceptual schemas. *Journal of Systems and Software*, 82(9):1459–1478, 2009.

[9] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA - visual automated transformations for formal verification and validation of UML models. In *17th IEEE International Conference on Automated Software*

Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK, pages 267–270. IEEE Computer Society, 2002.

[10] Eclipse Foundation. Ecore API. `https://eclipse.org/modeling/emf/`.

[11] Eclipse OCL Project. MDT-OCL website, 2014. `https://projects.eclipse.org/projects/modeling.mdt.ocl`.

[12] Christoph Eickhoff, Tobias George, Stefan Lindel, and Albert Zündorf. The sdmlib solution to the FIXML case for TTC2014. In Louis M. Rose, Christian Krause, and Tassilo Horn, editors, *Proceedings of the 7th Transformation Tool Contest part of the Software Technologies: Applications and Foundations (STAF 2014) federation of conferences, York, United Kingdom, July 25, 2014.*, volume 1305 of *CEUR Workshop Proceedings*, pages 22–26. CEUR-WS.org, 2014.

[13] Holger Giese, Stephan Hildebrandt, and Andreas Seibel. Improved flexibility and scalability by interpreting story diagrams. *ECEASST*, 18, 2009.

[14] Tassilo Horn. Model querying with funnyqt - (extended abstract). In Keith Duddy and Gerti Kappel, editors, *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, volume 7909 of *Lecture Notes in Computer Science*, pages 56–57. Springer, 2013.

[15] Tassilo Horn. Graph pattern matching as an embedded clojure DSL. In Parisi-Presicce and Westfechtel [24], pages 189–204.

[16] Ákos Horváth. Automatic generation of platform specific model transformation. Master's thesis, Budapest University of Technology and Economics, Budapest, May 2006.

[17] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Sci. Comput. Program.*, 89:144–161, 2014.

[18] Benedek Izsó, Gábor Szárnyas, István Ráth, and Dániel Varró. Train benchmark technical report. 2014.

[19] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[20] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 742–745. ACM, 2000.

[21] Object Management Group. *Object Constraint Language Specification (Version 2.3.1)*, May 2012. `http://www.omg.org/spec/OCL/2.3.1/`.

[22] Object Management Group. *UML Version 2.5*, June 2015. `http://www.omg.org/spec/UML/2.5/`.

[23] Oracle. Enterprise Manager, 2015. `http://www.oracle.com/technetwork/oem/enterprise-manager/overview/index.html`.

[24] Francesco Parisi-Presicce and Bernhard Westfechtel, editors. *Graph Transformation - 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 21-23, 2015. Proceedings*, volume 9151 of *Lecture Notes in Computer Science*. Springer, 2015.

[25] Mirko Seifert and Stefan Katscher. Debugging triple graph grammar-based model transformations. *Fujaba Days*, pages 19–25, 2008.

[26] Oszkár Semeráth. Consistency analysis of domain-specific languages. Master's thesis, Budapest University of Technology and Economics, Budapest, Dec 2013.

[27] Oszkár Semeráth, Ágnes Barta, Ákos Horváth, Zoltán Szatmári, and Dániel Varró. Formal validation of domain-specific languages with derived features and well-formedness constraints. *Software & Systems Modeling*, pages 1–36, 2015.

[28] The Eclipse Project. *EcoreTools*. `https://www.eclipse.org/ecoretools/`.

[29] The Eclipse Project. *Xtext*. `http://www.eclipse.org/Xtext/`.

[30] Massimo Tisi, Salvador Martínez Perez, and Hassene Choura. Parallel execution of ATL transformation rules. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, volume 8107 of *Lecture Notes in Computer Science*, pages 656–672. Springer, 2013.

[31] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.*, 98:80–99, 2015.

[32] Zoltán Ujhelyi, Gábor Szőke, Ákos Horváth, Norbert István Csiszár, László Vidács, Dániel Varró, and Rudolf Ferenc. Performance comparison of query-based techniques for anti-pattern detection. *Information & Software Technology*, 65:147–165, 2015.

[33] Gergely Varró, Frederik Deckwerth, Martin Wieber, and Andy Schürr. An algorithm for generating model-sensitive search plans for pattern matching on EMF models. *Software and System Modeling*, 14(2):597–621, 2015.

Appendix

A.1 A more detailed metamodel for program ASGs

This metamodel fragment in Figure A.1.1 contains all types required for the example instance model in Figure 2.3.

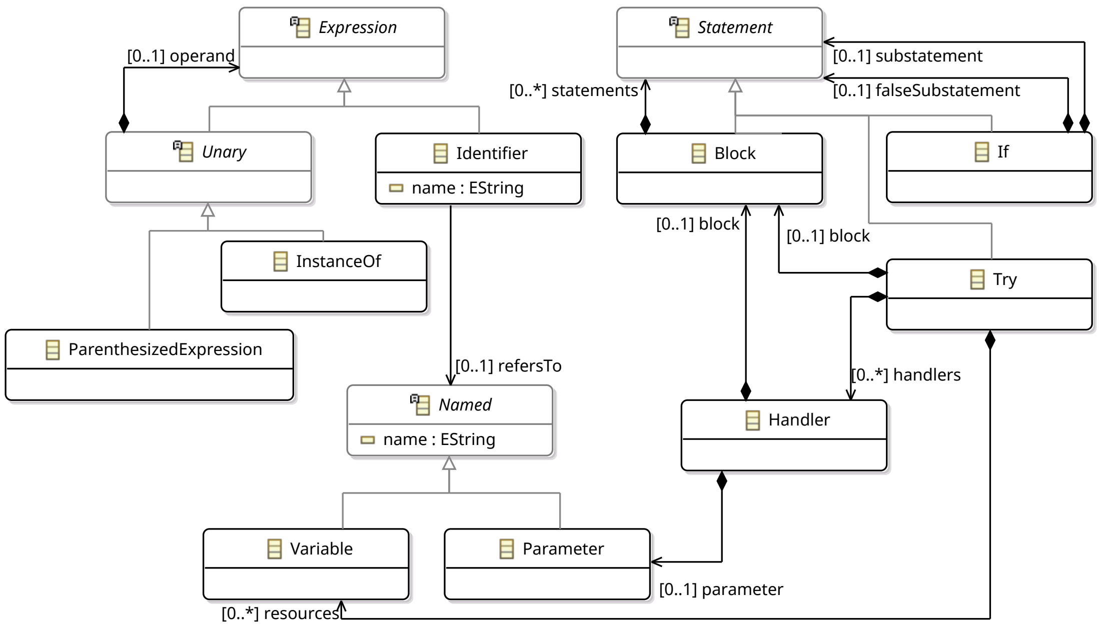


Figure A.1.1. A more detailed EMF metamodel fragment for ASGs

A.2 Detailed measurement results for code anti-patterns

Numeric output of the measurement process on the source code examples by phases is displayed in Table A.1, Table A.2, and Table A.3.

	Case	Tool	Read	Create engine	Calculate search plan	Check
1	Catch	Incremental	0.082	0.097	N/A	0.0002
2	Catch	Parallel	0.092	0.032	0.008	0.0000
3	Catch	Sequential	0.064	0.067	0.010	0.0001
4	Constant compare	Incremental	0.076	0.099	N/A	0.0002
5	Constant compare	Parallel	0.081	0.025	0.025	0.0014
6	Constant compare	Sequential	0.080	0.037	0.023	0.0007
7	No default switch	Incremental	0.077	0.091	N/A	0.0001
8	No default switch	Parallel	0.082	0.026	0.001	0.0004
9	No default switch	Sequential	0.085	0.038	0.001	0.0002
10	Unused parameter	Incremental	0.082	0.416	N/A	0.0013
11	Unused parameter	Parallel	0.092	0.126	0.086	0.0120
12	Unused parameter	Sequential	0.084	0.106	0.144	0.0063

Table A.1. Measurement results for Qwicap (in seconds)

	Case	Tool	Read	Create engine	Calculate search plan	Check
1	Catch	Incremental	2.535	1.813	N/A	0.0017
2	Catch	Parallel	2.575	0.855	0.406	0.0008
3	Catch	Sequential	2.512	0.842	0.400	0.0009
4	Constant compare	Incremental	2.448	1.906	N/A	0.0014
5	Constant compare	Parallel	2.529	0.837	0.685	0.0052
6	Constant compare	Sequential	2.385	0.867	0.575	0.0081
7	No default switch	Incremental	2.535	1.628	N/A	0.0009
8	No default switch	Parallel	2.485	0.851	0.001	0.0007
9	No default switch	Sequential	2.395	0.840	0.001	0.0005
10	Unused parameter	Incremental	2.458	3.105	N/A	0.0024
11	Unused parameter	Parallel	2.529	0.989	2.350	0.0711
12	Unused parameter	Sequential	2.406	0.912	2.283	0.0757

Table A.2. Measurement results for Frinika (in seconds)

	Case	Tool	Read	Create engine	Calculate search plan	Check
1	Catch	Incremental	17.858	10.588	N/A	0.0002
2	Catch	Parallel	16.671	4.925	2.698	0.0018
3	Catch	Sequential	16.377	4.758	2.647	0.0020
4	Constant compare	Incremental	16.854	11.408	N/A	0.0007
5	Constant compare	Parallel	16.887	5.019	4.570	0.1888
6	Constant compare	Sequential	17.903	5.101	7.730	0.0588
7	No default switch	Incremental	17.648	9.166	N/A	0.0002
8	No default switch	Parallel	16.620	4.683	0.001	0.0007
9	No default switch	Sequential	16.987	4.888	0.001	0.0003
10	Unused parameter	Incremental	16.748	12.785	N/A	0.0008
11	Unused parameter	Parallel	16.776	4.702	9.959	0.2326
12	Unused parameter	Sequential	15.965	4.939	10.557	0.2632

Table A.3. Measurement results for Hibernate (in seconds)

A.3 Detailed measurement results for Train Benchmark

Diagrams output by the Train Benchmark framework displaying the times needed for the different measurement phases by model sizes. Results for the Read and Crete engine phase is included in Figure A.3.1 and in Figure A.3.2 below. Detailed measurement data is added in Table A.4, Table A.5, and in Table A.6.

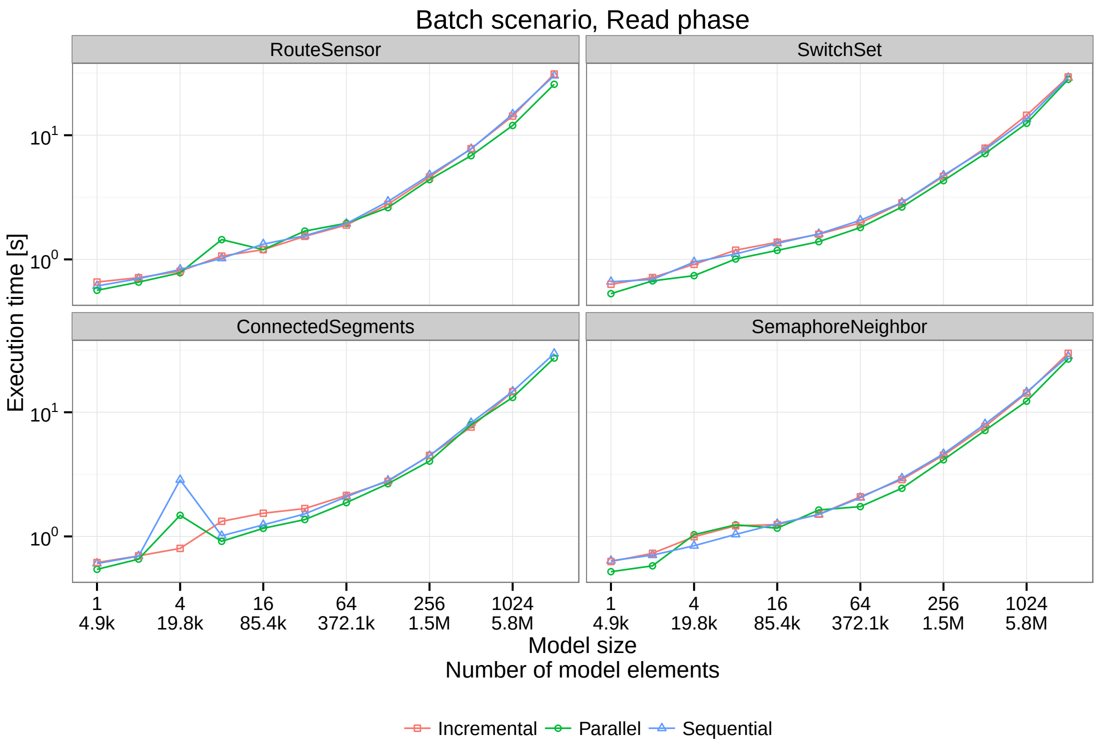


Figure A.3.1. Train Benchmark – Read phase

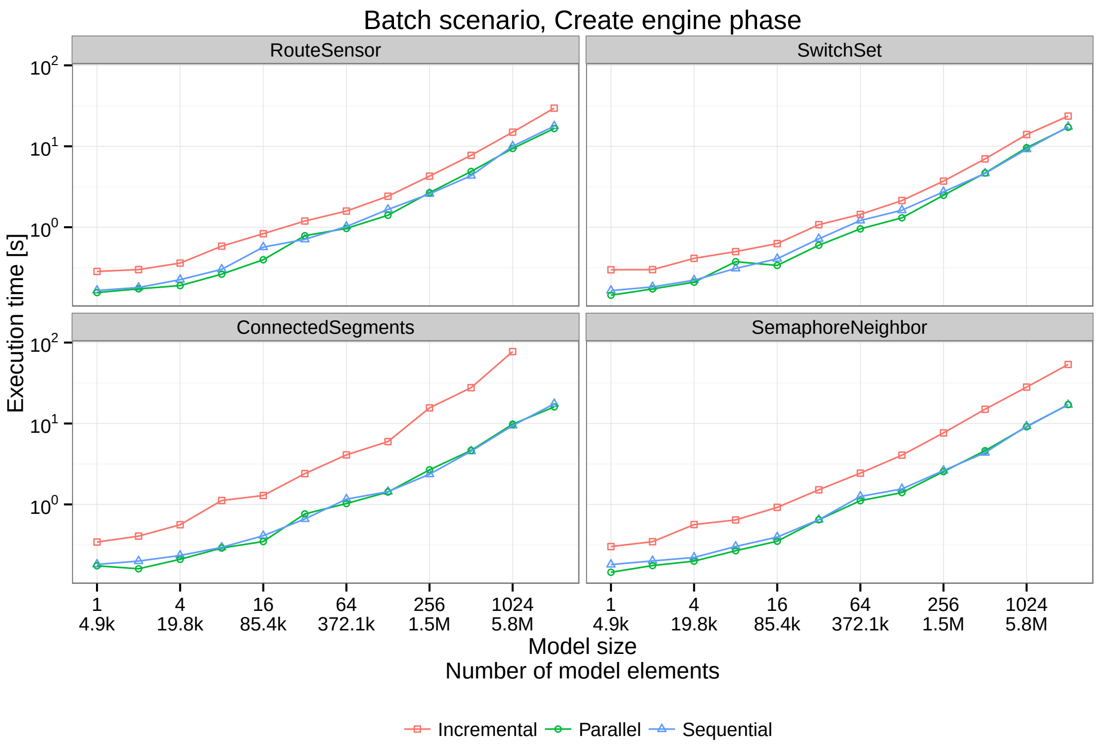


Figure A.3.2. Train Benchmark – Create engine phase

	Case	Model scale	Calculate search plan	Check	Create engine	Read
1	ConnectedSegments	1	N/A	0.005	0.343	0.6160
2	ConnectedSegments	2	N/A	0.005	0.406	0.6973
3	ConnectedSegments	4	N/A	0.006	0.561	0.8009
4	ConnectedSegments	8	N/A	0.005	1.119	1.3233
5	ConnectedSegments	16	N/A	0.005	1.289	1.5371
6	ConnectedSegments	32	N/A	0.006	2.402	1.6769
7	ConnectedSegments	64	N/A	0.008	4.095	2.1386
8	ConnectedSegments	128	N/A	0.009	5.975	2.7748
9	ConnectedSegments	256	N/A	0.010	15.595	4.4905
10	ConnectedSegments	512	N/A	0.020	27.702	7.5964
11	ConnectedSegments	1024	N/A	0.030	77.361	14.6564
11	ConnectedSegments	2048	N/A	-	-	-
12	RouteSensor	1	N/A	0.004	0.284	0.6573
13	RouteSensor	2	N/A	0.004	0.299	0.7124
14	RouteSensor	4	N/A	0.004	0.360	0.8038
15	RouteSensor	8	N/A	0.004	0.583	1.0638
16	RouteSensor	16	N/A	0.004	0.832	1.1981
17	RouteSensor	32	N/A	0.004	1.192	1.5331
18	RouteSensor	64	N/A	0.006	1.583	1.8914
19	RouteSensor	128	N/A	0.005	2.416	2.7768
20	RouteSensor	256	N/A	0.006	4.279	4.6098
21	RouteSensor	512	N/A	0.007	7.726	7.7939
22	RouteSensor	1024	N/A	0.009	14.953	14.2515
23	RouteSensor	2048	N/A	0.017	29.599	31.1415
24	SemaphoreNeighbor	1	N/A	0.004	0.302	0.6278
25	SemaphoreNeighbor	2	N/A	0.005	0.346	0.7314
26	SemaphoreNeighbor	4	N/A	0.003	0.564	0.9943
27	SemaphoreNeighbor	8	N/A	0.004	0.644	1.2172
28	SemaphoreNeighbor	16	N/A	0.005	0.920	1.2461
29	SemaphoreNeighbor	32	N/A	0.004	1.516	1.5093
30	SemaphoreNeighbor	64	N/A	0.006	2.436	2.0862
31	SemaphoreNeighbor	128	N/A	0.005	4.065	2.8686
32	SemaphoreNeighbor	256	N/A	0.006	7.648	4.5046
33	SemaphoreNeighbor	512	N/A	0.006	14.986	7.6752
34	SemaphoreNeighbor	1024	N/A	0.005	28.147	14.2688
35	SemaphoreNeighbor	2048	N/A	0.007	53.646	29.8475
36	SwitchSet	1	N/A	0.004	0.298	0.6307
37	SwitchSet	2	N/A	0.004	0.299	0.7143
38	SwitchSet	4	N/A	0.004	0.413	0.9119
39	SwitchSet	8	N/A	0.004	0.500	1.1864
40	SwitchSet	16	N/A	0.005	0.628	1.3747
41	SwitchSet	32	N/A	0.004	1.075	1.5935
42	SwitchSet	64	N/A	0.005	1.439	1.9664
43	SwitchSet	128	N/A	0.006	2.134	2.8516
44	SwitchSet	256	N/A	0.007	3.714	4.6632
45	SwitchSet	512	N/A	0.009	6.989	7.8651
46	SwitchSet	1024	N/A	0.011	13.969	14.4903
47	SwitchSet	2048	N/A	0.016	23.615	29.4733

Table A.4. Benchmark results for the incremental algorithm

	Case	Model scale	Calculate search plan	Check	Create engine	Read
48	ConnectedSegments	1	0.047	0.074	0.175	0.5428
49	ConnectedSegments	2	0.043	0.037	0.160	0.6564
50	ConnectedSegments	4	0.081	0.050	0.210	1.4798
51	ConnectedSegments	8	0.262	0.086	0.289	0.9131
52	ConnectedSegments	16	0.396	0.128	0.348	1.1620
53	ConnectedSegments	32	1.202	0.276	0.763	1.3668
54	ConnectedSegments	64	2.496	0.240	1.026	1.8711
55	ConnectedSegments	128	5.115	0.341	1.423	2.6572
56	ConnectedSegments	256	10.875	0.458	2.678	4.0401
57	ConnectedSegments	512	14.811	1.289	4.648	7.9337
58	ConnectedSegments	1024	28.935	2.672	9.810	13.1614
59	ConnectedSegments	2048	81.477	21.778	16.030	27.3225
60	RouteSensor	1	0.014	0.020	0.156	0.5635
61	RouteSensor	2	0.019	0.024	0.173	0.6561
62	RouteSensor	4	0.027	0.029	0.190	0.7796
63	RouteSensor	8	0.072	0.034	0.263	1.4431
64	RouteSensor	16	0.119	0.045	0.396	1.1981
65	RouteSensor	32	0.284	0.053	0.785	1.6905
66	RouteSensor	64	0.542	0.067	0.967	1.9556
67	RouteSensor	128	1.207	0.104	1.406	2.6108
68	RouteSensor	256	2.539	0.151	2.669	4.3887
69	RouteSensor	512	3.800	0.189	4.885	6.8383
70	RouteSensor	1024	7.025	0.285	9.403	11.9807
71	RouteSensor	2048	14.836	0.380	16.569	25.7272
72	SemaphoreNeighbor	1	0.024	0.040	0.145	0.5200
73	SemaphoreNeighbor	2	0.041	0.052	0.176	0.5795
74	SemaphoreNeighbor	4	0.066	0.083	0.199	1.0330
75	SemaphoreNeighbor	8	0.103	0.050	0.268	1.2399
76	SemaphoreNeighbor	16	0.210	0.195	0.351	1.1655
77	SemaphoreNeighbor	32	0.702	0.231	0.646	1.6327
78	SemaphoreNeighbor	64	1.534	0.422	1.113	1.7387
79	SemaphoreNeighbor	128	2.776	0.123	1.400	2.4430
80	SemaphoreNeighbor	256	5.743	1.732	2.543	4.1405
81	SemaphoreNeighbor	512	7.488	0.250	4.615	7.1392
82	SemaphoreNeighbor	1024	15.876	1.702	9.112	12.2774
83	SemaphoreNeighbor	2048	52.981	0.665	17.138	26.8228
84	SwitchSet	1	0.019	0.011	0.145	0.5297
85	SwitchSet	2	0.025	0.017	0.173	0.6714
86	SwitchSet	4	0.028	0.021	0.209	0.7397
87	SwitchSet	8	0.037	0.028	0.375	1.0071
88	SwitchSet	16	0.064	0.027	0.337	1.1823
89	SwitchSet	32	0.230	0.025	0.599	1.3891
90	SwitchSet	64	0.497	0.047	0.954	1.8049
91	SwitchSet	128	0.895	0.038	1.307	2.6413
92	SwitchSet	256	2.009	0.069	2.480	4.3071
93	SwitchSet	512	2.885	0.084	4.675	7.1133
94	SwitchSet	1024	6.073	0.131	9.606	12.4735
95	SwitchSet	2048	11.590	0.148	17.169	28.1495

Table A.5. Benchmark result for the sequential local search-based algorithm

	Case	Model scale	Calculate search plan	Check	Create engine	Read
96	ConnectedSegments	1	0.160	0.029	0.182	0.6048
97	ConnectedSegments	2	0.179	0.041	0.200	0.6937
98	ConnectedSegments	4	0.240	0.069	0.234	2.8547
99	ConnectedSegments	8	0.455	0.079	0.295	1.0115
100	ConnectedSegments	16	0.859	0.247	0.411	1.2394
101	ConnectedSegments	32	2.020	0.258	0.661	1.5176
102	ConnectedSegments	64	3.175	0.735	1.162	2.0864
103	ConnectedSegments	128	6.631	0.827	1.438	2.8166
104	ConnectedSegments	256	14.495	3.906	2.361	4.4578
105	ConnectedSegments	512	20.719	3.109	4.542	8.2206
106	ConnectedSegments	1024	39.900	6.585	9.425	14.7102
107	ConnectedSegments	2048	104.046	34.750	17.492	29.6772
108	RouteSensor	1	0.092	0.006	0.165	0.6108
109	RouteSensor	2	0.104	0.009	0.180	0.6993
110	RouteSensor	4	0.147	0.014	0.225	0.8292
111	RouteSensor	8	0.265	0.023	0.302	1.0207
112	RouteSensor	16	0.471	0.034	0.567	1.3270
113	RouteSensor	32	0.795	0.060	0.708	1.5513
114	RouteSensor	64	1.398	0.067	1.019	1.9423
115	RouteSensor	128	2.529	0.102	1.655	2.9249
116	RouteSensor	256	5.121	0.174	2.583	4.7742
117	RouteSensor	512	10.334	0.227	4.330	7.7517
118	RouteSensor	1024	19.039	0.488	10.023	14.7573
119	RouteSensor	2048	38.614	0.525	17.730	30.2251
120	SemaphoreNeighbor	1	0.138	0.036	0.181	0.6367
121	SemaphoreNeighbor	2	0.152	0.060	0.200	0.7069
122	SemaphoreNeighbor	4	0.206	0.100	0.221	0.8413
123	SemaphoreNeighbor	8	0.353	0.084	0.302	1.0385
124	SemaphoreNeighbor	16	0.663	0.133	0.395	1.2638
125	SemaphoreNeighbor	32	1.355	0.356	0.648	1.5020
126	SemaphoreNeighbor	64	1.993	0.287	1.254	2.0495
127	SemaphoreNeighbor	128	4.037	0.957	1.554	2.9425
128	SemaphoreNeighbor	256	8.168	0.994	2.621	4.6234
129	SemaphoreNeighbor	512	14.477	2.945	4.356	8.0286
130	SemaphoreNeighbor	1024	27.131	6.580	9.242	14.5130
131	SemaphoreNeighbor	2048	52.988	44.576	17.004	28.2886
132	SwitchSet	1	0.105	0.004	0.165	0.6604
133	SwitchSet	2	0.113	0.005	0.183	0.6893
134	SwitchSet	4	0.151	0.009	0.222	0.9518
135	SwitchSet	8	0.232	0.010	0.309	1.1063
136	SwitchSet	16	0.354	0.014	0.408	1.3451
137	SwitchSet	32	0.736	0.025	0.718	1.6062
138	SwitchSet	64	1.044	0.033	1.211	2.0653
139	SwitchSet	128	2.188	0.056	1.622	2.8650
140	SwitchSet	256	5.262	0.093	2.735	4.7560
141	SwitchSet	512	9.661	0.132	4.624	7.6547
142	SwitchSet	1024	18.099	0.142	9.202	13.4744
143	SwitchSet	2048	34.605	0.262	17.445	29.0464

Table A.6. Benchmark result for the parallel local search-based algorithm