Unification of Compiled and Interpreter-Based Pattern Matching Techniques

Gergely Varró*, Anthony Anjorin**, and Andy Schürr

Technische Universität Darmstadt,
Real-Time Systems Lab,
D-64283 Merckstraße 25, Darmstadt, Germany
{gergely.varro,anthony.anjorin,andy.schuerr}@es.tu-darmstadt.de

Abstract. In this paper, we propose a graph pattern matching framework that produces both a standalone compiled and an interpreter-based engine as a result of a uniform development process. This process uses the same pattern specification and shares all internal data structures, and nearly all internal modules. Additionally, runtime performance measurements have been carried out on both engines with exactly the same parameter settings to assess and reveal the overhead of our interpreter-based solution.

Keywords: model transformation, pattern matching interpreter, compiled pattern matcher.

1 Introduction

As model transformation undoubtedly plays an immense role in the process of model-driven development, efficiency and scalability are, therefore, important issues. In many state-of-the-art tools [1,2], model transformations are governed by imperative control flow statements, which apply declarative rules as basic transformation units. Such tools offer the usual advantages of declarativity like an easily understandable specification language, and readily available solutions provided by the underlying execution engine for many performance-critical tasks, whose optimal implementation requires years of specialized expertise. One such task is the efficient checking of the application conditions of rules, which requires identifying those parts in the system model on which the rule is executable.

This application condition checking process (as well as several other subtasks in bidirectional model synchronization and on-the-fly consistency checking scenarios) can be described as a general pattern matching problem. In this context, a pattern consists of constraints, and the matching process determines those parts of the underlying model that fulfill all these constraints. Structural constraints

* Supported by the Postdoctoral Fellowship of the Alexander von Humboldt Foundation and associated with the Center for Advanced Security Research Darmstadt.
** Supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 368–383, 2012.

express restrictions that can be checked by using the services of the modelling layer (e.g., type checks, navigation along links), while non-structural constraints are handled by some other means (like integer or textual comparison). The rest of the paper will focus on handling structural constraints, which corresponds to the graph pattern matching problem [3]. Nonetheless, our approach is left open w.r.t. the integration of non-structural constraints as well.

When implementing a pattern matching engine, developers must decide on several important issues (see Sec. 2) already in the early phase of design, which are hardly modifiable in later development phases as they have radical consequences on the overall architecture. One of these critical topics is the decision whether a compiled or an interpreter-based engine is to be built.

A *compiled engine* only consists of program code that is directly executable on a certain platform without an extra module for performing pattern matching. A compiled engine typically features better runtime performance, as the algorithms are represented as machine or byte code of the underlying execution system and no operation handling layer is needed. In contrast, an *interpreter-based engine* requires a specific module (the interpreter), which is responsible for executing the operations needed to perform pattern matching. Such a technique could offer more flexibility (e.g., model-sensitive performance optimization [4]) and provide additional services such as high-level debug support, as the interpreter can access and exploit runtime information more easily.

There exists a large variety of advanced compiled pattern matcher implementations [1,5], and several sophisticated interpreter-based approaches [2,6] in different rule-based model transformation tools. Although some of them provide solutions for both cases, the resulting engines can be considered to be separate programs. The following statement [7] is, therefore, still valid: "Interpretation and code generation are often seen as two alternatives, not as a continuum". In order to allow different combinations of these alternatives, techniques are needed that handle compiled and interpreter-based pattern matchers in a uniform and tightly integrated way.

In this paper, we propose a pattern matching framework that can produce both a standalone compiled and an interpreter-based engine as a result of a uniform development process, which shares (i) the pattern specification, (ii) all internal data structures, and (iii) all internal activities except for one engine-specific module. Furthermore, applying exactly the same settings in this uniform process wherever possible, runtime performance measurements are carried out on both engines to assess and reveal the overhead of our interpreter-based solution. To our best knowledge, our proposed approach can be considered the first pattern matcher to support both a compiled and an interpreter-based setup in a unified, configurable and integrated manner and can, therefore, be easily embedded and used by different rule-based model transformation tools.

The remainder of the paper is structured as follows. Related work is discussed in Section 2. Section 3 introduces basic metamodelling terminology, pattern specification constructs, and the process of pattern matching. Sections 4 and 5 present our data structures and algorithms used in the unified pattern matching engine.

Section 6 gives a quantitative assessment and performance comparison of our compiled and interpreter-based engines. Section 7 concludes our paper.

2 Design Space of Pattern Matchers and Related Work

A widely deployable pattern matching engine should support many different application scenarios like the execution of rule-based model transformations on a desktop computer, as well as performing security monitoring tasks on an embedded system. As the computational power and the amount of available resources of these architectures significantly differ, the development of a pattern matcher requires considering several design issues that influence the applicability and performance of the approach.

The design space of pattern matching engines can be characterized by the following properties:

(1) Dependency on separate pattern matching modules. The first property, which has the closest relation to the topic of this paper, expresses whether pattern matching requires a specific interpreter (I), or can be performed without any separate modules in a standalone manner as a compiled program (C).

(2) Existence and granularity of intermediate data structures. Pattern matching interpreters and code generators that produce compiled engines usually operate on data structures with different granularity. One group of solutions directly processes the declarative, pattern specification either in a low-level form as an abstract syntax tree representation (AST), or in a high-level form as a pattern definition (P). The other group operates on a preprocessed (and typically optimized) intermediate data structure, which can either be a low-level byte code representation (BC), or a high-level search plan (SP).

(3) Generation schedule of intermediate data structures. When intermediate data structures are used by the pattern matcher, their generation schedule can also be an important design decision due to its time consuming nature. Intermediate data structures can be calculated clearly at compile time (CT), at runtime in an on-demand fashion (OD) by using a caching mechanism, or at runtime (RT) before each pattern matching process.

(4) Availability of model sensitive pattern matching strategies. The size and the structure of the underlying model often influence the runtime performance of a pattern matcher. As both characteristics can significantly change as a transformation proceeds, the runtime selection of a pattern matching strategy in a model sensitive (MS) way (i.e., by using statistics from the model) is a feasible optimization compared to approaches that rely only on metamodel-level, domain-specific (DS) information.

(5) Incrementality. As matches for a given pattern are often requested several times during the life cycle of several application scenarios, exploiting the reuse of already calculated matches is a feasible optimization possibility. In this sense, batch engines (B) restart the pattern matching process from scratch at each invocation, while incremental approaches (I) store a set of (partial) matches,

and update this set according to a defined schedule that depends on changes in the underlying model.

(6) Implementation/target language. As the applicability of a pattern matcher in a specific environment is largely determined by the implementation language of the interpreter, or the target language of the code generator, this property has also been included in our survey. The categorization here indicates support for a single (1) or multiple (*) languages.

(7) Reusability in different modelling spaces. Another important factor, in the evaluation of pattern matchers, is their reusability in different modelling environments. In this sense, an engine can operate on non-standard (NS) or standard (S) (e.g., EMF, MDR) model repositories. In the latter case, a star (*) suffix is added, if a tool provides clear interfaces to several standard modelling environments.

(8) Model access. When a tool operates in a standard compliant modelling environment, the underlying model can be accessed via tailored (T) or reflective (R) interfaces, which obviously affects both the runtime performance, and the resource (disk and memory) demand of an approach.

As a categorization of general model transformation tools is already available [8], this survey, which cannot be complete due to space restrictions, focuses on the pattern matching modules of state-of-the-art, rule-based transformation engines, and systematically compares them based on the previously listed criteria, which has been preceded by a manual inspection of the available source code (or a related publication). Table 1 presents the evaluation of these pattern matchers, which are enumerated in alphabetical order.

Table 1. Tool comparison

Tool name	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
ATL [2]	I	BC	CT	DS	B	1	S*	R
Epsilon [9]	I	AST	N/A	DS	B	1	S*	R
Fujaba @ KS [1]	C	SP	CT	DS	B	1	S*	T
Fujaba @ PO [10]	I	SP	RT	MS	B	1	S	T – R
GReAT [11]	I C	P	N/A	DS	B	1	NS	T
GrGen [5]	C	SP	OD	MS	B	1	NS	N/A
Groove [12]	I	SP	OD	MS	B I	1	NS	N/A
Henshin [13]	I	P	N/A	DS	B	1	S	R
PROGRES [14]	I C	BC	OD	DS	B I	*	NS	N/A
VIATRA [15,6]	I	SP	OD	DS MS	B I	1	NS S	T
Our approach	I – C	SP	CT OD – RT	DS	B	1	S	T
Perfect tool	I – C	BC,SP	CT – OD – RT	DS – MS	B – I	*	NS – S*	T – R

The N/A mark shows if a categorization is non-applicable, while the '–' notation is used to express that a tool is able to cover the whole range of values in an integrated and configurable manner. The last two lines represent the evaluation of our current approach, and a hypothetic ideal pattern matching engine that could be deployed in many different application scenarios.

Table 1 clearly shows that many aspects of the ideal solution have already been solved separately by the different existing tools; however, the coverage of *design space ranges* along several properties is still not satisfactory. The main challenge here is that each of the above-mentioned design space properties represents a decision that is hard-wired into the tool architecture making reengineering tasks difficult in this context.

3 Modelling Concepts and Data Structures

In this section, we introduce basic metamodelling terminology required to present our approach, define concepts related to pattern specification, and illustrate the process of pattern matching and its runtime data structures.

3.1 Metamodels and Models

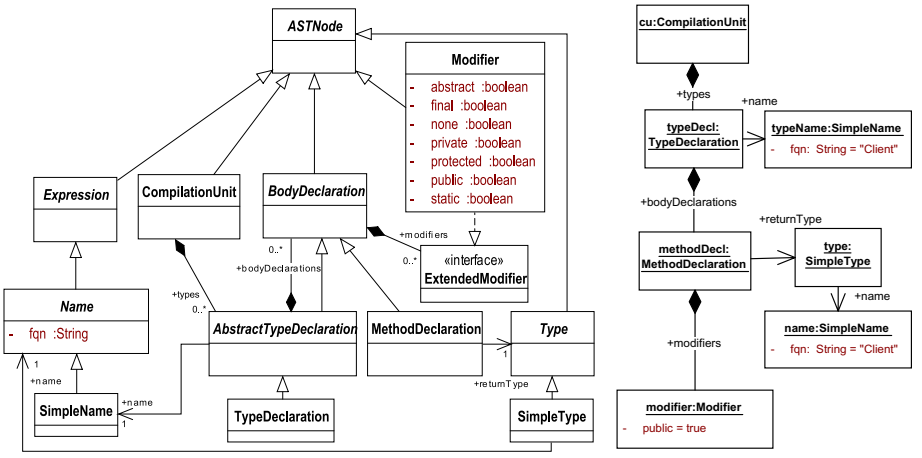
A *metamodel* is the specification of the concepts and relationships in a certain domain. Figure 1(a) depicts an excerpt of the metamodel from a case study [16] for the GraBaTs'09 transformation tool contest [17], which poses a program comprehension challenge based on the Eclipse Java Development Tools (JDT) API [18]. Using the common UML class diagram notation, parts that are relevant for our running example are depicted. The metamodel has been taken unchanged from the case study, and defines concepts as *classes* (e.g., a `CompilationUnit`). Classes can *inherit* from other classes (e.g., every `MethodDeclaration` is a `BodyDeclaration`), can contain *attributes* (every `Name` has an `fqn` as an attribute of type `String`), and can *reference* other classes (`CompilationUnit`s contain arbitrary many `AbstractTypeDeclaration`s, which each have exactly one `SimpleName`). Attributes and references are referred to as *structural features* in the rest of the paper.

A *model* is an abstraction of a system, created with an intended goal in mind. In alignment with the UML notation, nodes and edges are referred to as *objects* and *links*, respectively. A model that is expressed using concepts specified in a metamodel is said to *conform to* the metamodel. Figure 1(b) depicts a model, which corresponds to the Eclipse JDT representation of a Java class `Client` with a single public method, which returns a `Client`.

3.2 Pattern Specification

This subsection introduces the concepts needed for specifying patterns. The following definitions are based on [19].

A *pattern* is a set of constraints over a set of variables. A *variable* is a placeholder for an object in a model. *Interface variables* constitute a subset of all variables used in a pattern, and represent the variables that can be accessed outside of the pattern. All other *local* (i.e., non-interface) variables can only be



(a) An excerpt of the metamodel of the Eclipse JDT (b) A sample model

Fig. 1. An excerpt of the metamodel of the Eclipse JDT API and a sample model

accessed and used internally in the pattern. A *constraint* specifies a condition on a set of variables that must be fulfilled by the objects, which are assigned to the variables during pattern matching. A constraint consists of a *constraint type* and a set of variables (also referred to as *parameters* in this context), for which the constraint must hold. In the following, an explicit reference to the type of a constraint shall be denoted by adding a 'type' suffix.

The pattern matcher has a pluggable infrastructure for the constraints that can be used for specifying patterns.[1] In this paper, only a subset of constraints is presented. The support for extending the framework with constraints for attribute handling, positive and negative pattern invocations is already available, however, the implementation for pattern calls is still a task for the future.

A *class constraint* (`cls`) restricts the type of the objects that can be assigned to its single parameter. A *structural feature constraint* (`sf`) prescribes the existence of a link, which connects the assigned objects and conforms to a given structural feature. Both constraints `cls` and `sf` have references to types in the corresponding metamodel. A *Boolean constraint* must evaluate to `true` for the assigned values to its single parameter.

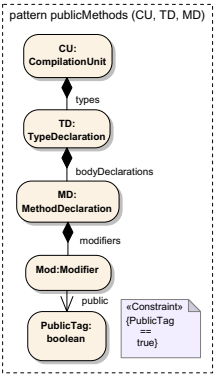
The textual specification of patterns, used in this paper, is defined by the following simplified EBNF grammar:

```
patternSpecification ::= "pattern" signature "=" body
signature ::=  NAME "(" interfaceVariables ")"
body ::= "{" constraint* "}"
constraint ::=  NAME typeReference? "(" variables ");"
typeReference ::= "<" NAME ">"
```

[1] Quantifiers can be defined at runtime in our framework, when the pattern matching is invoked, and consequently, they are not part of the pattern specifications.

```
interfaceVariables ::= variables
variables ::= ( NAME ",")* NAME
NAME ::= [a-zA-Z]+
```

Example. The graphical representation and the textual specification of pattern `publicMethods`[2] are presented in Fig. 2. This pattern requires the existence of a compilation unit `CU`, which has a type declaration `TD` with a public method declaration `MD`. Pattern `publicMethods` has three interface variables `CU`, `TD`, and `MD` (line 1). The class constraint on line 3 prescribes that variable `CU` must be mapped to a `CompilationUnit`. The structural feature constraint on line 10 requires a `types` reference that connects the objects that are assigned to variables `CU` and `TD`. The Boolean constraint on line 16 prescribes that the object assigned to variable `PublicTag` must be `true`. Please note that (i) the order of constraints (rows) in the textual representation of a pattern is arbitrary, (ii) constraints on references and attributes are handled in a uniform way (line 13), and (iii) the pattern matcher has access to all the properties of a metamodel element (information whether a class is abstract, a reference is a composition, etc.) via the references maintained in the class and structural feature constraints.




```
1  pattern publicMethods(CU,TD,MD) = {
2      // Class constraints
3      cls<CompilationUnit>(CU);
4      cls<TypeDeclaration>(TD);
5      cls<MethodDeclaration>(MD);
6      cls<Modifier>(Mod);
7      cls<boolean>(PublicTag);
8
9      // Structural feature constraints
10     sf<types>(CU,TD);
11     sf<bodyDeclarations>(TD,MD);
12     sf<modifiers>(MD,Mod);
13     sf<public>(Mod,PublicTag);
14
15     // Boolean constraints
16     true(PublicTag);
17 }
```

Fig. 2. Pattern `publicMethods` in a graphical and textual representation

3.3 Pattern Matching and Runtime Data Structures

Pattern matching is the process of determining mappings for all variables in a given pattern, such that all constraints in the pattern are fulfilled. The mappings of variables to objects are collectively called a *match*, which can be a *complete match* when all the variables are mapped, or a *partial match* in all other cases.

An *adornment* represents binding information for a sequence of variables and is indicated in the textual syntax by a sequence of letters B and F of the same length, which appears as a superscript on the name of the concept to which

[2] A more complex example scenario can be found in [4].

the adornment is attached. The letter B or F in an adornment, means that the variable in that position is bound or free, respectively.

When pattern matching is invoked, interface variables can be already bound to objects to restrict the search. The corresponding binding information of all interface variables is called a *pattern adornment*.

An *operation* represents a single atomic step in the matching process and it consists of a constraint and a constraint adornment. A *constraint adornment* prescribes which parameters must be bound when the operation is executed. A *check operation* has only bound parameters. An *extension operation* has free parameters, which get bound when the operation is executed.

Example. Suppose a matching process for the pattern `publicMethods` (Fig. 2) is to be run on the model of Fig. 1(b), with the interface variable `CU` bound to the compilation unit `cu` at pattern invocation. This single mapping itself constitutes a partial match, and the corresponding pattern adornment is BFF,[3] since only the first interface variable has been bound. When the pattern matching process terminates, a complete match is returned, which maps variables `CU`, `TD`, `MD`, `Mod`, and `PublicTag` to objects `cu`, `typeDecl`, `methodDecl`, `modifier`, and a Boolean `true` value, respectively.

4 Workflow of Compiled Pattern Matching

This section presents the workflow for generating a compiled pattern matching engine. In this process, a pattern matcher class is generated for every pattern. Although an adornment is part of runtime binding information, the generated engine must be prepared to handle a fixed set of pattern adornments, which are selected in advance at compile time. For each selected pattern adornment, a method that performs the actual pattern matching is generated into the corresponding pattern matcher class according to the approach depicted in Fig. 3, which operates as follows:

Section 4.1 For each constraint type used in the pattern specification, the set of allowed constraint adornments is calculated.
Section 4.2 For each pair of constraint and allowed constraint adornment, an operation is loaded.
Section 4.3 Operations are filtered and ordered by a search plan algorithm to produce an efficient search plan.
Section 4.4 Based on the search plan, generators are created, which control the code generation process.

When the pattern matcher is invoked at runtime, the pattern adornment is determined and used to select and execute the corresponding generated method.[4] The selected method performs the complete pattern matching process, collecting

[3] The pattern adornment only contains binding information for interface variables (`CU`, `TD` and `MD` in this example).

[4] If no such method exists, an exception is thrown, which might initiate a pattern matcher regeneration process.

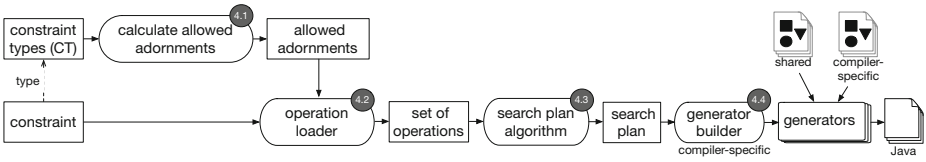


Fig. 3. Process for producing a compiled pattern matching engine

complete matches in a result set. The following subsections discuss the steps of the process in detail.

4.1 Allowed Adornment Calculation

In general, not every possible adornment is valid for every constraint type. Our framework allows the underlying modelling layer to define the set of allowed adornments for every available constraint type in a configurable way. For presentation purposes, standard Ecore/EMF is assumed as the modelling layer in the following: A structural feature constraint type, referring to a bidirectional reference, would allow adornments `BB`, `BF`, and `FB`, where `BB` means checking the existence of a link, while `BF` and `FB` denote forward and backward navigations, respectively. In the case of unidirectional references, only `BB` and `BF` adornments make sense. Analogously, only `BB` and `BF` adornments are allowed for structural feature constraint types referring to an attribute. Only the adornment `B` is allowed for class and Boolean constraint types.[5]

Example. The framed part of Figure 4(a) shows the complete list of constraint types and allowed adornments for pattern `publicMethods`. Lines 1–5 show that class constraint types have the adornment `B` as type checks can be performed only on a bound parameter. Line 6 represents a check for the existence of a `types` link, while line 7 means a forward navigation along a link of type `types`.

4.2 Operation Loading

The operation loader prepares all the operations that might be needed to execute pattern matching by iterating through all constraints of a pattern. It looks up the allowed adornments for the type of the constraint, and creates a new operation for each combination of constraint and allowed adornment.

Example. Figure 4(a) lists the set of operations that might be needed to calculate matches for pattern `publicMethods`. For example, line 1 shows that variable `CU` must already be mapped to an object before a corresponding type check can be performed. Line 7 expresses that a forward navigation along `types` links is executable only when an object has already been bound to variable `CU`.

[5] The set of allowed adornments for standard MOF/JMI, in contrast to Ecore/EMF, would also allow `FF` for associations, and `F` for class constraint types. Similarly, a modelling layer with additional EMF services could also support an extended set of allowed adornments for EMF supporting e.g., `FB` for indexed attributes.

```
1  cls<CompilationUnit>^B(CU)
2  cls<TypeDeclaration>^B(TD)
3  cls<MethodDeclaration>^B(MD)
4  cls<Modifier>^B(Mod)
5  cls<boolean>^B(PublicTag)
6  sf<types>^BB(CU,TD)
7  sf<types>^BF(CU,TD)
8  sf<bodyDeclarations>^BB(TD,MD)
9  sf<bodyDeclarations>^BF(TD,MD)
10 sf<modifiers>^BB(MD,Mod)
11 sf<modifiers>^BF(MD,Mod)
12 sf<public>^BB(Mod,PublicTag)
13 sf<public>^BF(Mod,PublicTag)
14 true^B(PublicTag)
```

(a) Operations

```
1  publicMethods^BFF(CU,TD,MD) = [
2    cls<CompilationUnit>^B(CU);
3    sf<types>^BF(CU,TD);
4    cls<TypeDeclaration>^B(TD);
5    sf<bodyDeclarations>^BF(TD,MD);
6    cls<MethodDeclaration>^B(MD);
7    sf<modifiers>^BF(MD,Mod);
8    cls<Modifier>^B(Mod);
9    sf<public>^BF(Mod,PublicTag);
10   cls<boolean>^B(PublicTag);
11   true^B(PublicTag);
12 ]
```

(b) Search plan for the sample pattern `publicMethods`

Fig. 4. Data structures used by both engines

4.3 Search Plan Generation

Operations are filtered and ordered by a search plan algorithm to produce efficient search plans. Due to space restrictions, search plans generation techniques like [1,4,6] and their details (e.g., cost functions, optimization algorithms) are not discussed in this paper.

A *search plan* is defined as a sequence of operations satisfying the following conditions:

1. Each constraint in the pattern has exactly one corresponding operation in the search plan.
2. Each variable that must be *bound* according to the constraint adornment of an operation is either already bound in the pattern adornment, or appears as a free variable in one of the preceding operations.
3. Each variable that must be *free* according to the constraint adornment of an operation is not bound in the pattern adornment and does not appear in any of the preceding operations as free variables.
4. Each extension operation must be immediately followed by class check operations that perform the type checking of the free variables of the extension operation.

Example. Figure 4(b) shows a search plan for pattern `publicMethods`, when the first interface variable (`CU`) is bound when pattern matching is invoked. The search plan has been derived from the set of operations (Fig. 4(a)) by filtering and reordering, and it fulfills all conditions 1–4. The constraint adornment on line 3, for example, is valid, as `CU`, which must be bound, is indeed bound in the pattern adornment. Similarly, `TD`, which must be free, is free in the pattern adornment, and does not appear in any preceding operation as a free variable.

4.4 Code Generation for a Compiled Pattern Matcher

The final step is code generation, which is controlled by a set of different *generators* that each maintain a link to a template.[6] As depicted in detail in Fig. 5 for the pattern adornment `BFF`, a *method generator* references a series of *operation generators*, responsible for navigation in the model (e.g., lines 2 and 5), which in turn reference *variable generators*, that store the determined values for variables on the JVM stack (e.g., lines 3–4 and 6–7). A *match generator* is responsible for the code that should be executed when a (complete) match is found (line 15). Code generation is initiated by starting the template of the method generator.

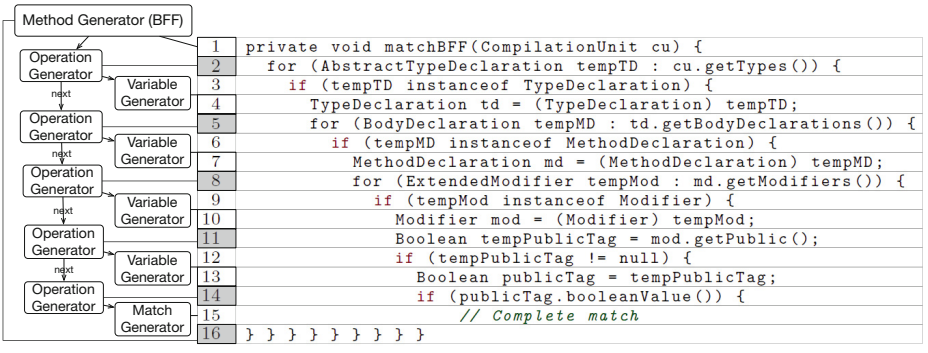


Fig. 5. Generated code to handle pattern matching

5 Workflow of Interpreter-Based Pattern Matching

Interpreter-based engines typically carry out all their pattern matching related tasks at runtime. As such, the pattern specification should be considered as a starting point when the matching process is invoked. Furthermore, in order to avoid any dependencies on generated data structures, interpreter-based pattern matchers typically use a reflective API of the modelling layer to access objects and to navigate along links.

In contrast to this general tendency, our interpreter-based solution (whose workflow is presented in Fig. 6) uses tailored interfaces (just like the compiled pattern matcher) to completely eliminate the performance effects that would be caused by the different model access strategies, and thus, to enable a fair quantitative comparison of our compiled and interpreter-based engine variants. This requires generated operation classes (and a loader class), which are subclasses of their generic counterparts and are produced at compile time (as shown by the solid arrows). A generated operation, thus, represents an atomic step in the pattern matching process and uses a tailored interface for model access purposes.

[6] Velocity is used as a template language and engine.

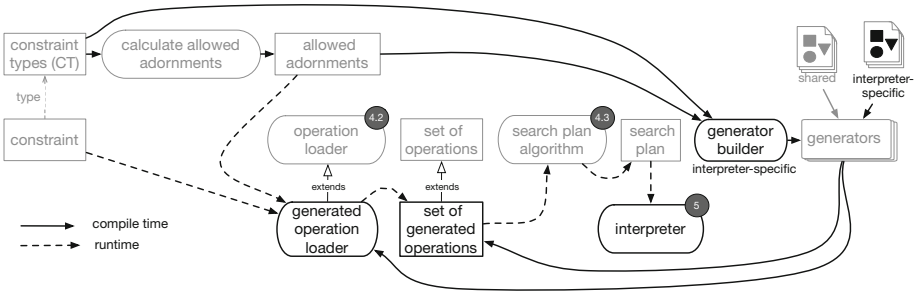


Fig. 6. Process for producing an interpreter-based engine

Although generated operations are produced at compile time, *all the remaining activities* are executed at *runtime* as highlighted by dashed arrows in Fig. 6. The runtime part of the interpreter-based pattern matching approach, which reuses all parts from Sec. 4 that are shaded in grey, works as follows:

Section 4.2. When the pattern matcher is invoked, a (generated) operation is loaded for each pair of constraint and allowed constraint adornment. As constraints are part of the pattern, this behaviour completely aligns with the expectation that an interpreter-based engine should start processing the pattern specification at runtime.

Section 4.3. Operations are filtered and ordered by a search plan algorithm to produce a search plan. The algorithm is obviously influenced by the pattern adornment, that has been determined by examining which interface variables have been mapped to objects in the model at pattern invocation.[7]

Section 5. Finally, the interpreter performs pattern matching by executing the search plan. The details of this interpreter module are presented in the remaining part of this section.

The interpreter uses a *match array* for storing the current match and the operations in the search plan. Every operation has a link to the next operation and a mapping, that identifies the slots in the match array, which correspond to the parameters of the operation.

When the interpreter is invoked, it prepares the initial match array, whose size is determined by the number of variables in the pattern. The initial match array is filled according to the input mapping of interface variables to objects in the model, and is passed to the first operation in the search plan. When an extension operation is performed, the structural feature is traversed forwards (`BF`) or backwards (`FB`) to bind the corresponding free variable, the type of the accessed object is validated, and the execution is passed on to the following operation for subsequent processing together with the extended match array. For a check operation, the operation is performed and, in case of success, the current match array is simply left unchanged and passed on. When the match

[7] Search plans can be cached and reused depending on the configuration settings.

array passes beyond the last operation, it represents a complete match and is stored in a result set. A single match array is used for storing all (partial) matches, and only complete matches are copied and stored in a result set.

This behaviour is a depth-first traversal[8] of a state space (just like the compiled approach), where a *state* represents the processing of a partial match by an operation. The state space can alternatively be described as a tree, whose root is the initial match, while internal nodes and leaves correspond to partial and complete matches, respectively.

6 Measurement Results

In this section, we present measurement results from comparing our compiled engine with our interpreter-based pattern matching engine, both produced via a uniform process, using our framework as discussed in Sec. 4 and Sec. 5. The pattern used for the measurements is from [16] and describes all classes, excluding inner classes, that contain at least one public static method, whose return type is the same as the class itself. Five models (Set0 – 4) of different size were taken unchanged from the same case study.

A 1.57 GHz Intel Core2 Duo CPU with 2.96 GB RAM was used for all measurements. Windows XP Professional SP 3 and Java 1.6 served as the underlying operating system and virtual machine, respectively. The maximum heap size was set to 1536 MB. *User time*, which is the amount of time the CPU spends performing actions for a program, was measured. On the used machine, this could only be measured with a precision of ± 12.625 ms. As a single pattern matching task takes less time than this value, each measurement was performed as a series of blocks. In a measurement block, the pattern matching task of the compiled engine was performed 100 times, while in the case of the interpreter-based engine, search plan generation and pattern matching were executed 500 times, and 20 times, respectively.[9] This block-based measurement was repeated 50 times in all cases to provide stable average values.

The generated search plans and resulting matches in both cases were validated manually to be equivalent. To obtain a fair comparison, search plan generation and pattern matching were measured separately for the interpreter.

Table 2 presents the measured execution times. The first column indicates the model used (Set0 – 4), the second and third columns the size of the model in number of Java classes and objects, respectively. The fourth column denotes the corresponding state space size in number of states, the fifth column the time (ms) for the compiled engine, while the last two columns show the time (ms) for search plan generation and pattern matching for the interpreter-based engine.

[8] Alternative strategies (e.g., breadth-first traversal) would typically duplicate match arrays during the execution of extension operations, which would cause an increased memory consumption.

[9] The only reason for selecting different block length values was to have raw data approximately on the same scale, which already belongs to the measureable range.

Table 2. Measurement results

	Java classes	Model size	State space	Compiled PM	Interpreter-based SP	Interpreter-based PM
	#	#	#	ms	ms	ms
Set0	14	70447	232	0.03	0.12	0.19
Set1	40	198466	549	0.08	0.12	0.53
Set2	1605	2082841	37348	12.99	0.12	41.91
Set3	5769	4852855	94300	31.17	0.41	142.34
Set4	5984	4961779	103122	36.01	0.84	230.38

Table 2 shows that the compiled engine, which represents algorithms in a byte code form, and thus, requires no operation handling tasks to be executed, outperformed the interpreter-based engine in all cases, and was about 4–6 times faster. For the interpreter-based engine, the time spent for search plan generation increased for the larger models (Set3 and Set4), which are loaded into memory prior to search plan generation. We believe this is caused by garbage collection, which becomes necessary due to the substantial difference in size of the models as compared to Set0 or Set1.

In order to clarify the exact reason for the large execution times of the interpreter-based engine in case of large models, and to justify our assumption on the role of garbage collection, further measurements should be performed in the future with larger heap size settings, on more patterns and on different application domains e.g., like the ones mentioned in [20]. An additional comparison using reflective interfaces for our engines would also be interesting. Furthermore, other dimensions could be measured including memory footprint, number of loaded classes and RAM consumption.

Note that the main goal of our measurements was neither to quantitatively analyze pattern matchers with search plans [4,20], nor to draw any conclusion regarding the performance of compiled and interpreter-based engines in general, nor to repeatedly justify the obvious comparative statements about the runtime superiority of compiled techniques, but to assess the *exact extent* of performance differences between our pattern matcher variants. In our measurements, both variants accessed the EMF-based modeling layer via the same tailored interfaces. Additionally, they applied the same algorithm to produce the same search plan, which resulted in the same state space traversed in the same depth-first order. We think that this unified measurement setup could only be achieved by using a framework-based solution, and any modification in this setup would introduce performance influencing factors that are independent from the main issue under investigation (i.e., the exact effects caused by the selection of our compiled or interpreter-based engine).

7 Conclusion

In this paper, we proposed a pattern matching framework that can produce both a standalone compiled and an interpreter-based engine in a uniform process that

shares all internal data structures and the majority of modules. As main advantages, our framework-based solution (1) eases the task of reengineering a tool with respect to its pattern matcher module, and (2) enables a switching possibility between the compiled and interpreter-based engines at runtime. Additionally, we carried out performance measurements on both engines with the same parameter settings to assess the overhead of our interpreter-based solution.

The proposed approach has been implemented in the context of the Democles project, whose goal is to provide a model-based pattern matcher implementation, which integrates several advanced pattern matching algorithms in one framework, and can be embedded into different tools. Contributions of this paper cover one aspect of this project, which was to present the unified process for handling compiled and interpreter-based pattern matchers. The model-sensitive search plan algorithm of the pattern matcher has been published in [4]. The interpreter additionally supports (a yet unpublished) step by step execution possibility, which can be the basis of a high-level debugger in the future.

References

1. Geiger, L., Schneider, C., Reckord, C.: Template- and modelbased code generation for MDA-tools. In: Giese, H., Zündorf, A. (eds.) Proc. of the 3rd International Fujaba Days, Paderborn, Germany, pp. 57–62 (2005), `ftp://ftp.upb.de/doc/techreports/Informatik/tr-ri-05-259.pdf`
2. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
3. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1: Foundations. World Scientific (1997)
4. Varró, G., Deckwerth, F., Wieber, M., Schürr, A.: An Algorithm for Generating Model-Sensitive Search Plans for EMF Models. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 224–239. Springer, Heidelberg (2012)
5. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 383–397. Springer, Heidelberg (2006)
6. Varró, G., Varró, D., Friedl, K.: Adaptive graph pattern matching for model transformations using model-sensitive search plans. In: Karsai, G., Taentzer, G. (eds.) Proc. of International Workshop on Graph and Model Transformation. ENTCS, vol. 152, pp. 191–205. Elsevier (2005)
7. Voelter, M.: Best practices for DSLs and model-driven development. Journal of Object Technology 8(6), 79–102 (2009)
8. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–645 (2006)
9. Kolovos, D., Rose, L., Paige, R.: The Epsilon book, `http://www.eclipse.org/gmt/epsilon/doc/book/`
10. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. In: Margaria, T., Padberg, J., Taentzer, G. (eds.) Proc. of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques, ECEASST, vol. 18 (2009)

11. Agrawal, A., Karsai, G., Neema, S., Shi, F., Vizhanyo, A.: The design of a language for model transformations. Software and Systems Modeling 5(3), 261–288 (2006)
12. Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
13. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010)
14. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES Approach: Language and Environment. In: Handbook on Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages and Tools, pp. 487–550. World Scientific (1999)
15. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA model transformation system. In: Karsai, G., Taentzer, G. (eds.) Proc. of the 3rd International Workshop on Graph and Model Transformation, pp. 25–32. ACM (2008)
16. Sottet, J.S., Jouault, F.: Program comprehension, `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2009reverseengineering.pdf`
17. Levendovszky, T., Rensink, A., van Gorp, P.: 5th International Workshop on Graph-Based Tools: The Contest, `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/`
18. Beaton, W., des Rivieres, J.: Eclipse platform technical overview. Technical report, The Eclipse Foundation (2006)
19. Horváth, Á., Varró, G., Varró, D.: Generic search plans for matching advanced graph patterns. In: Ehrig, K., Giese, H. (eds.) Proc. of the 6th Int. Workshop on Graph Transformation and Visual Modeling Techniques, vol. 6 of ECEASST (2007)
20. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. In: Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, Dallas, Texas, USA, pp. 79–88. IEEE Computer Society Press (2005)