An Algorithm for Generating Model-Sensitive Search Plans for EMF Models

Gergely Varró*, Frederik Deckwerth, Martin Wieber, and Andy Schürr

Technische Universität Darmstadt,
Real-Time Systems Lab,
D-64283 Merckstraße 25, Darmstadt, Germany
{gergely.varro@es,f.deckwerth@stud}.tu-darmstadt.de,
{martin.wieber,andy.schuerr}@es.tu-darmstadt.de

Abstract. In this paper, we propose a new model-sensitive search plan generation algorithm to speed up the process of graph pattern matching. This dynamic programming based algorithm, which is able to handle general n-ary constraints in an integrated manner, collects statistical data from the underlying EMF model, and uses this information for optimization purposes. Additionally, runtime performance measurements have been carried out to quantitatively evaluate the effects of the search plan generation algorithm on the pattern matching engine.

Keywords: graph pattern matching, search plan generation algorithm, model-sensitive search plan

1 Introduction

Efficient, scalable, and standard compliant techniques and tools are still undoubtedly needed to promote the spread of model-driven technologies in an industrial context. As numerous scenarios in the model-based domain, such as (i) checking the application conditions in rule-based model transformation tools [1, 2], (ii) bidirectional model synchronization, or (iii) on-the-fly consistency validation, can be described as a general pattern matching problem, its efficient implementation is undisputedly an important task.

In this general pattern matching context, a pattern consists of constraints, which place restrictions on variables, and the number of variables involved in a constraint is referred as its arity. The pattern matching process determines a mapping of variables to the elements of the underlying model in such a way that the assigned model elements must fulfill all constraints. Structural constraints 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 (e.g., integer or textual comparison).

As non-structural constraints are easily manageable [3], the current paper only focuses on structural constraints, which correspond to the graph pattern

* Supported by the Postdoctoral Fellowship of the Alexander von Humboldt Foundation and associated with the Center for Advanced Security Research Darmstadt.

The final publication is available at http://link.springer.com/chapter/10.1007%2F978-3-642-30476-7_15

matching problem [4]. Although available pattern matching engines support type checks and link navigations as unary and binary structural constraints, respectively, practical model-driven scenarios additionally require the handling of n-ary constraints to express ordered references or pattern composition [5].

When building a pattern matching engine, its performance highly depends on the order in which the constraints of a pattern are evaluated (cf. the impact of the variable ordering in general backtracking). This rationale motivates the construction of heuristics-based algorithms for generating constraint sequences or search plans [6], which can be efficiently evaluated.

While the majority of state-of-the-art search plan generation algorithms [1, 7, 8] exploits only type and multiplicity restrictions derived from the metamodel of the problem domain, two novel *model-sensitive* approaches [9, 10] take, for optimization purposes, the potential structure of *instance models* into account as further domain-specific knowledge. Although the inherent performance advantages of model-sensitive search plan generation techniques have already been clearly shown [11], the applicability of the tools themselves in a more general modeling context is hindered by the fact that both engines (i) operate on non-standard (tool specific) model representations, and (ii) apply graph-based algorithms for search plan generation, which can handle only unary and binary constraints in an integrated manner.

In this paper, we propose a completely new model-sensitive search plan generation algorithm, based on dynamic programming, to enable the integrated handling of general n-ary constraints. The algorithm collects statistical data from the model under transformation via an extensible framework and uses this information for optimization purposes. The pluggable collection of statistical data is exemplified on Eclipse Modeling Framework (EMF) compliant models. Finally, the effects of the search plan generation algorithm on the performance of pattern matching are quantitatively evaluated by using runtime measurements.

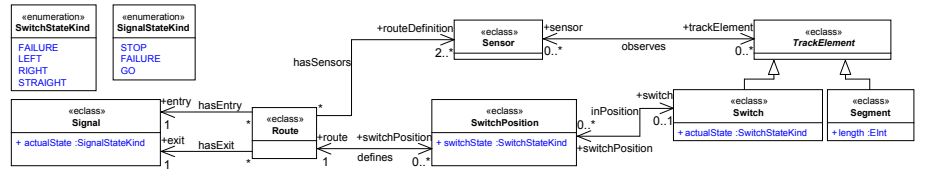
The remainder of the paper is structured as follows: Section 2 introduces basic modeling and pattern specification concepts. The general pattern matching process is sketched in Sec. 3, while Sec. 4 presents the new search plan generation algorithm. Section 5 gives a quantitative assessment and performance comparison. Related work is discussed in Sec. 6, and Sec. 7 concludes our paper.

2 Metamodel, Model and Pattern Specification

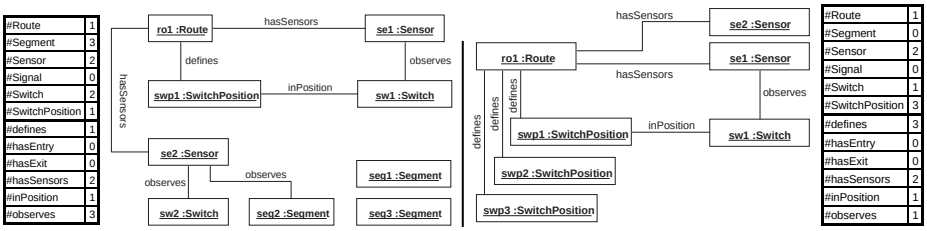
2.1 Metamodels and Models

A *metamodel* represents the core concepts of a domain. In this paper, our approach is demonstrated on a real-world running example from the railway domain [12] (developed in the MOGENTES project [13]), whose metamodel is depicted in Fig. 1(a). *Classes* are the nodes in the metamodel: Routes, Sensors, Signals, SwitchPositions, and TrackElements, which can either be Switches or Segments. *References* are the edges between classes, which can be uni- or bidirectionally navigable as indicated by the arrows at the end points. A navigable end is labelled with a *role name* and a *multiplicity*, which restricts the number of target

objects that can be reached via the given reference. In our example, a Route has at least 2 Sensors (as shown by the unidirectional reference hasSensors), and defines an arbitrary number of SwitchPositions, which is a bidirectional reference. *Attributes* (depicted in the lower part of classes) store values of primitive or enumerated types, e.g., the length integer in a Segment, or the actualState of a Switch whose possible values are listed in the *enumeration* SwitchStateKind. Figures 1(b) and 1(c) depict two *models* from the domain, whose nodes and edges are called *objects* and *links*, respectively.



(a) The metamodel of the railway track domain



(b) Model 1 (c) Model 2

Fig. 1. Metamodel of the railway track domain and two sample models

EMF-Specific Issues: References and attributes are collectively referred to as *structural features* and handled uniformly in EMF. Each navigable direction of a structural feature is represented by an indexed `List` in the source class, which stores corresponding target objects.

Our approach collects statistical data from the model at runtime via EMF adapters. An *object* and *link counter* is introduced for each class and structural feature, which stores the number of type conforming objects and links, respectively, as shown by the tables in Figures 1(b) and 1(c).

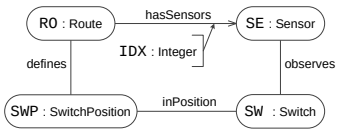
2.2 Pattern Specification

As defined in [5, 14], a *pattern* is a set of constraints over a set of variables. A *variable* is a placeholder for an object in a model, and it has a reference to a class from the metamodel, which defines the type of the objects that can be assigned to the variable during pattern matching. A *constraint* specifies a condition on a set of variables (which are also referred to as *parameters* in this context) that must be fulfilled by the objects, which are assigned to the parameters.

EMF-Specific Issues: Although the pattern matcher has a pluggable infrastructure for the constraints that can be used for specifying patterns, only one kind of constraints is used throughout the paper.[1] In the following, a constraint maintains a reference to a structural feature, and it prescribes the existence of a link, which (i) conforms to the referenced structural feature and (ii) connects the source and the target object assigned to the first and last parameter, respectively.

An *ordered* or *unordered* structural feature can be modeled by a *binary* constraint in the pattern specification, when *the order information is irrelevant* in the pattern matching process. In contrast, *ternary* constraints should be used for *ordered unidirectional* structural features, where the second parameter is an integer index, which prescribes the location of the target object in the list of the source object containing links that conform to the structural feature.

Example. Pattern `routeSensor` (Fig. 2) expresses a sample requirement defined by railway domain experts, which has been slightly simplified for presentation purposes. It states that a route must have a sensor observing a switch, and the observed switch itself must be part of the route. The pattern has 5 variables (`RO`, `IDX`, `SE`, `SW` and `SWP`), 1 ternary and 3 binary constraints, which prescribe the existence of an ordered unidirectional and 3 bidirectional references, respectively.




```
1  pattern routeSensor(RO:Route, IDX:Integer,
2    SE:Sensor, SW:Switch, SWP:SwitchPosition) =
3  {
4    hasSensors(RO, IDX, SE);
5    observes(SE, SW);
6    inPosition(SW, SWP);
7    defines(RO, SWP);
8  }
```

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

3 Pattern Matching Process at Runtime

As [14] states, *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. The overall process of pattern matching is as follows:

Section 3.1 Operations representing atomic steps in the pattern matching process are created from the pattern specification.

Section 3.2 The operations are filtered and sorted by a *search plan generation algorithm* (for the details see Sec. 4) to produce efficient search plans.

Section 3.3 The search plan is then used by an interpreter to control the actual execution of pattern matching, which is carried out as a depth-first traversal.

[1] Type restrictions for variables are going to be represented as constraints only in a future version of the pattern matcher.

3.1 Creating Operations

This subsection, which reuses some definitions from [5, 14], describes the process of creating operations from the constraints in the pattern specification. In the following, it is assumed that an (arbitrary) order is fixed for the variables in the pattern, and the notation v_p denotes the pth variable according to this order.

An *adornment* [5] represents binding information for *all variables* in the pattern by a corresponding character sequence consisting of letters `B` or `F`, which indicate that the variable in that position is *bound* or *free*, respectively.

An *operation* represents a single atomic step in the matching process. It consists of a constraint, an operation adornment, and a mask, which is derived from the operation adornment. An *operation adornment* prescribes which *parameters* must be bound when the operation is executed, while a *mask* represents the same binding information, but projected on *all variables* in the pattern. A *check operation* has only bound parameters. An *extension operation* has free parameters, which get bound when the operation is executed.

Setting operation adornments. For presentation purposes, we assume that operations use the standard EMF services, which restricts the set of operations created for a constraint in the following manner.

For each *binary constraint referring to a bidirectional structural feature*, 3 operations with the corresponding `BB`, `BF`, and `FB` adornments are created. The check operation (`BB`) verifies the existence of a link, while the other two, adorned by `BF` and `FB`, denote forward and backward navigations, respectively. Analogously, for each *binary constraint referring to a unidirectional structural feature*, 2 operations with the corresponding `BB` and `BF` adornments are prepared.

For each *ternary constraint (referring to an ordered unidirectional structural feature)*, operations adorned by `BBB`, `BBF`, and `BFF` are prepared (adornment `BFB` is disallowed for presentation purposes). The check operation (`BBB`) verifies that (i) a link connects the source and the target object mapped to the first and the third parameter, respectively, and (ii) the target object is stored in the appropriate `List` of the source object at the index assigned to the second parameter. The operation with the `BBF` adornment is a forward navigation along the *single* link, which is stored at the index assigned to the second parameter. Finally, the operation adorned by `BFF` is a forward navigation along *all* links that conform to the structural feature of the constraint, and that retain the source object mapped to the first parameter.

Mask derivation. A *mask* m_o is a sequence of `*`, `B`, and `F` characters. Character `*` at position p means that the binding of variable v_p is irrelevant, while letters `B` or `F` at position p explicitly prescribe the corresponding variable v_p to be bound or free, respectively. For each letter `B` (`F`) in the adornment, the position p of the corresponding parameter v_p is looked up by using the fixed variable order, and position p is set to `B` (`F`) in the mask. All other locations of the mask are set to `*`.

Categorizing and applying operations. In the context of an adornment, operations can be categorized. An operation o is a *present* (or applicable) *operation* with respect to an adornment a, if the following conditions hold:

1. **General operation applicability.** Each variable v_p, that must be *free* according to the mask m_o of operation o, is also *free* in adornment a.
2. **Immediate operation applicability.** Each variable v_p, that must be *bound* according to the mask m_o of operation o, is also *bound* in adornment a.

An operation o is a *past operation*, if the first condition on general operation applicability is violated. An operation o is a *future operation*, if only the second condition on immediate operation applicability is violated.

If an operation o is a present (or applicable) operation w.r.t. adornment a, then *applying the operation o on adornment a resulting in an adornment a'* (denoted by $a \overset{o}{\Longrightarrow} a'$) (i) binds all free variables indicated by mask m_o of operation o, and (ii) leaves the binding of all other variables unaltered.

Example. Figure 3(a) lists the operations derived from the routeSensor pattern. In the following, we suppose that variables `RO`, `IDX`, `SE`, `SW` and `SWP` are ordered in this specific sequence. For instance, operation `observes(SE,SW)` adorned by `BF` is an extension operation, and it is only applicable if variable `SE` is bound, and variable `SW` is free, which is also reflected in mask `**BF*` as `SE` and `SW` are the third and fourth variable, respectively. This operation can be categorized as a future operation with respect to adornment `BFFFF`, as it violates the immediate operation applicability condition at the third position.

Operation			*Applic.*	*Type*
Constraint	*Op. Adornm.*	*Mask*		
hasSensors(RO,IDX,SE)	BBB	BBB**	future	*check*
hasSensors(RO,IDX,SE)	BBF	BBF**	future	**extension**
hasSensors(RO,IDX,SE)	BFF	BFF**	present	**extension**
observes(SE,SW)	BB	**BB*	future	*check*
observes(SE,SW)	BF	**BF*	future	**extension**
observes(SE,SW)	FB	**FB*	future	**extension**
inPosition(SW,SWP)	BB	***BB	future	*check*
inPosition(SW,SWP)	BF	***BF	future	**extension**
inPosition(SW,SWP)	FB	***FB	future	**extension**
defines(RO,SWP)	BB	B***B	future	*check*
defines(RO,SWP)	BF	B***F	present	**extension**
defines(RO,SWP)	FB	F***B	past	**extension**

(a) Operations

Search plan	*Step*	*Operation*			Adornm. a_i (a_0 = BFFFF)
		Constraint	*Op. Adornm.*	*Mask*	
Search plan 1 (derived from model 1)	(1)	defines(RO,SWP)	BF	B***F	BFFF**B**
	(2)	inPosition(SW,SWP)	FB	***FB	BFF**BB**
	(3)	hasSensors(RO,IDX,SE)	BFF	BFF**	B**BB**B
	(4)	observes(SE,SW)	BB	**BB*	BB*BB* B
Search plan 2 (derived from model 2)	(1)	hasSensors(RO,IDX,SE)	BFF	BFF**	B**BB**FF
	(2)	observes(SE,SW)	BF	**BF*	BBB**B**F
	(3)	inPosition(SW,SWP)	BF	***BF	BBBB**B**
	(4)	defines(RO,SWP)	BB	B***B	*B*BBB*B*

(b) Search plans as sequence of operations

Fig. 3. Operations and search plans for the `routeSensor` pattern

3.2 Search Plan Generation

When pattern matching is invoked, variables can already be bound to objects to restrict the search. The corresponding binding information of all variables is called *initial adornment* a_0. By using the initial adornment, a search plan generation algorithm filters and sorts the operations to produce a search plan. The current search plan formalism is a precise and extended variant of [5].

A *search plan* $SP = \langle o_1, o_2, \ldots, o_l \rangle$*, starting from an initial adornment* a_0, is a sequence of operations satisfying the following conditions:

1. **No multiple constraint checks.** Each constraint in the pattern has *at most one* corresponding operation in the search plan.
2. **Valid adornment sequence.** An adornment sequence $a_0, a_1, \ldots, a_l$ can be derived in such a way that $a_0 \overset{o_1}{\Longrightarrow} a_1 \overset{o_2}{\Longrightarrow} \ldots \overset{o_l}{\Longrightarrow} a_l$. The last element a_l in this adornment sequence is referred as the *adornment of the search plan.*

A search plan is *complete*, if each constraint is represented by *exactly one* operation in the sequence, and its adornment has only `B` characters.

Example. Figure 3(b) depicts two search plans generated by our algorithm for Models 1 and 2, when variable `RO` is initially bound and, thus, the initial adornment is `BFFFF`. The rightmost column presents the adornment *after* applying the operation in the same line. SP1 extends the partial match along two separate directions before joining the branches with the last (check) operation, while SP2 employs a clockwise navigation along the references in the pattern.

3.3 Search Plan Execution by a Pattern Matcher Interpreter

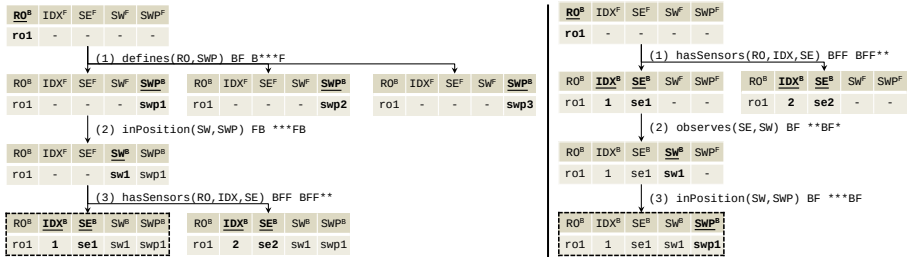
By conceptually following the corresponding part of [14], the interpreter uses a *match array* for storing the matches, and the search plan for guiding the pattern matching process. The size of the match array is determined by the number of variables in the pattern. Each operation has a mapping, which identifies the slots in the match array that correspond to the parameters of the operation.

When pattern matching is invoked, the initial match array is filled in by the objects that are initially assigned to the variables, and it is passed on to the first operation in the search plan. When an extension operation is executed, the structural feature of its constraint is navigated in forward (`BF`, `BBF`, `BFF`) or backward (`FB`) direction depending on the operation adornment, then each accessed object is type checked and bound to the corresponding free variable, and the execution is passed on to the following operation for subsequent processing together with the extended match array. A check operation simply passes on the unchanged match array, if the actual check succeeded, and stops triggering further processing steps otherwise. If a match array passes beyond the last operation, then it represents a complete match, which is copied and stored in the result set.

This pattern matching (PM) process implements a depth-first traversal of a PM state space, where a *PM state* represents a partial match that is produced by an extension operation during pattern matching. The PM state space can be described by a tree, whose root is the initial match, while internal nodes and leaves correspond to partial and complete matches, respectively. Note that each tree level is produced by a corresponding extension operation, and check operations do not influence the tree structure as they do not bind any variables.

Example. Figure 4 depicts two PM state spaces, which are generated by performing search plans SP1 and SP2 on Model 2, respectively. E.g., the second level of Fig. 4(a) represents the partial matches that are prepared when navigating along `defines` links from route `ro1` to switch positions `swp1`, `swp2`, and `swp3`, as prescribed by operation `defines(RO,SWP)` with adornment `BF`. Framed

leaves represent those complete matches that pass beyond the last check operation (only shown in Fig. 3(b)), while unframed ones fail this check. It is obvious from Fig. 4 that SP2 is better than SP1, as SP2 traverses less PM states.



(a) PM state space by performing SP1 on Model 2

(b) PM state space by performing SP2 on Model 2

Fig. 4. Sample PM state spaces for Model 2

4 Dynamic Programming Based Search Plan Generation

As demonstrated in Fig. 4, the search plan has a large impact on the number of produced matches, and consequently, on the performance of pattern matching. As such, the production of a good search plan is an essential issue, and that is why a quantitative characterization of operations and search plans is introduced for optimization purposes by means of weights and costs. Note that a cost function should ideally have a strong correlation with the size of the PM state space.

Operation weight calculation. An extension operation o is augmented by a *weight* w_o, which denotes the cost of performing the operation. In our approach, a weight is defined as an average *branching factor* for that level of the PM state space tree, which represents the operation execution, and is calculated using the statistical data collected from the underlying model. The weights of ternary operations with the `BBF` adornment are set to 1 (irrespective of the model), as these operations never induce any branching in the matching process. For binary and ternary operations with the corresponding `BF` and `BFF` adornments (forward navigation), the structural feature referenced by the constraint of the operation is determined, and the weight is the ratio of the link and object counters defined for this structural feature and its *source* class, respectively. For binary operations with `FB` adornment (backward navigation), the link counter of the structural feature is divided by the object counter of the *target* class to define the weight.

Search plan costs. The search plan cost c_l used in this paper estimates the size of the PM state space tree via the $c_l = \sum_{j=1}^{l} \prod_{i=1}^{j} w_{o_i}$ expression [10], which sums up the estimated number of PM states on a level-by-level basis (excluding the root). To support an iterative search plan cost calculation, the

cost c_l is complemented by a product value π_l and the calculation is rearranged as $(c_l, \pi_l) = f(c_{l-1}, \pi_{l-1}, w_{o_l})$, where $c_0 = 0$, $\pi_0 = 1$, $\pi_l = \pi_{l-1} w_{o_l}$, and

$$c_l = \sum_{j=1}^{l} \prod_{i=1}^{j} w_{o_i} = \overbrace{w_{o_1} + \ldots + w_{o_1} w_{o_2} \cdots w_{o_{l-1}}}^{c_{l-1}} + \underbrace{\overbrace{w_{o_1} \cdots w_{o_{l-1}}}^{\pi_l}}_{\pi_{l-1}} \cdot \underbrace{w_{o_l}}_{w_{o_l}} = c_{l-1} + \pi_l.$$

Algorithm data structures. To avoid unnecessary recalculations in our approach, a state stores only the best of those search plans that share the same adornment. A *state* S contains a *search plan* SP_S with its *adornment* a_S and *costs* (c_S, π_S); and sequences of *present extension* O_S^{pe}, *future extension* O_S^{fe}, and *future check* O_S^{fc} operations[2] (w.r.t. adornment a_S), which are (i) pairwise disjoint by definition, and (ii) ordered based on their weights. Two states are *adornment disjoint*, if they have different adornments.

The initial state S_0 has an empty operation sequence as its search plan, the initial adornment a_0 as its adornment, and its cost values are set as $c_{S_0} := c_0, \pi_{S_0} := \pi_0$. Its operations are categorized w.r.t. the initial adornment a_0.

Algorithm. An efficient search plan is generated by a dynamic programming based algorithm (see Algorithm 1), which iteratively fills states into an initially empty table T with $n+1$ columns and k rows, where n is the number of free variables $|a_{S_0}|$ in the adornment a_{S_0} of the initial state S_0 and $k \geq 1$ is a user-defined parameter that influences the trade-off between efficiency and optimality of the algorithm. In general, the column $T[i]$ stores the best k adornment disjoint states (in an increasing cost order), which have i free variables in their adornment, while $T[i][j]$ is the jth best from these adornment disjoint states.

The two key features of the algorithm can be summarized as follows. (i) The table *only stores adornment disjoint states* with the consequence of keeping only the best search plan from those ones that share a common prefix. (ii) Additionally, the table *only stores a constant number* of adornment disjoint states *in each column*, immediately discarding costly search plans, which are not among the best k solutions, and implicitly all their possible continuations. This avoids the production of all search plans, which could alone result in the same (exponential) complexity as the match calculation process.

First, the algorithm determines the number of free variables $n = |a_{S_0}|$ in the adornment a_{S_0} of the initial state S_0 (line 1), and stores this state S_0 in $T[n][1]$ (line 2). Then, the table is traversed by processing columns in a decreasing order based on the number of free variables in the state adornments (lines 3–17). In contrast, the inner loop (lines 4–16) proceeds in an increasing state cost order starting from the best state $T[i][1]$ in each column $T[i]$. For each present extension operation o in each stored state S (lines 6–15), the next state S' is prepared in a two-phase process, which (1) calculates the search plan $SP_{S'}$, the adornment $a_{S'}$ and the cost $c_{S'}$ of the next state S' immediately in `calculateNextState` (lines 8–9), and (2) updates the search plan, and the sequences of present extension, future extension, and future check operations in

[2] Note that past and present check operations need not be stored as they will be immediately processed by the algorithm.

Algorithm 1 The procedure `calculateSearchPlan`(S_0, k)

```
1: n := |a_{S_0}| //number of free variables in the initial state adornment a_{S_0} is calculated
2: T[n][1] := S_0
3: for (i := n to 1) do
4:   for (j := 1 to k) do
5:     S := T[i][j] //current state S
6:     for each (o ∈ O_S^{pe}) do
7:       // for each present extension operation
8:       S' := calculateNextState(S, o) // next state S' is calculated
9:       i' := |a_{S'}| // next state S' has i' free variables in its adornment a_{S'}
10:      (a, c) := determineIndices(T[i'], S')
11:      if (checkInsertCondition(T[i'], S', a, c)) then
12:        updateOperations(S', S, o)
13:        insert(T[i'], S', a, c)
14:      end if
15:    end for
16:  end for
17: end for
18: return SP_{T[0][1]}
```

a delayed manner in `updateOperation` (line 12), but only if the next state S' passes the insert condition (line 11), which uses indices $\mathbf{a}$ and $\mathbf{c}$ for decision making, which are calculated by `determineIndices` (line 10). In the latter case, the complete next state S' is inserted into the column $T[i']$ by using indices $\mathbf{a}$ and $\mathbf{c}$ (line 13). Finally, the algorithm returns the search plan $SP_{T[0][1]}$ (line 18).

The procedure `calculateNextState`(S, o) partially calculates the new state S' from state S and operation o. The new search plan $SP_{S'}$ is determined by appending operation o to the search plan SP_S of state S. The new adornment $a_{S'}$ is calculated by applying operation o on the adornment a_S of state S (i.e., $a_S \overset{o}{\Longrightarrow} a_{S'}$). The new costs $c_{S'}$ and $\pi_{S'}$ are computed from the costs c_S and π_S of state S, and the weight w_o of operation o according to the cost function f.

The procedure `determineIndices`$(T[i'], S')$ calculates indices $\mathbf{a}$ and $\mathbf{c}$. Index $\mathbf{a}$ marks the position of that stored state $T[i'][\mathbf{a}]$, which has the same adornment $a_{S'}$ as state S'. Index $\mathbf{a}$ is set to $k+1$, if no such stored state exists. Index $\mathbf{c}$ marks the position at which state S' should be inserted based on its cost. Index $\mathbf{c}$ is set to $k+1$, if state S' is not among the best k adornment disjoint states. Formally, $\mathbf{c}$ is the smallest index for which $c_{S'} < c_{T[i'][\mathbf{c}]}$ holds (or $c_{T[i'][\mathbf{c}]} = \texttt{null}$).

The procedure `checkInsertCondition`$(T[i'], S', \mathbf{a}, \mathbf{c})$ makes a positive decision, (1) if column $T[i']$ does not contain any states with the adornment $a_{S'}$ of new state S' ($\mathbf{a} = k+1$), new state S' is among the best k adornment disjoint states ($\mathbf{c} < \mathbf{a}$), and a reachability analysis[3] determines that the search plan $SP_{S'}$ can be completed in a valid manner, or (2) if column $T[i']$ already stores a state $T[i'][\mathbf{a}]$ at location $\mathbf{a}$ with the adornment $a_{S'}$ of new state S' ($\mathbf{a} < k+1$), and this new state S' is better than the stored state $T[i'][\mathbf{a}]$ ($\mathbf{c} \leq \mathbf{a}$).

[3] The reachability analysis is only discussed in [15] due to space limitations.

The procedure `updateOperations`(S', S, o) processes all operations o^* of present extension O_S^{pe}, future extension O_S^{fe}, and future check O_S^{fc} sequences of state S in an increasing weight order by also recategorizing these operations with respect to the adornment $a_{S'}$ of new state S'.

- **Discard operations causing multiple checks.** If operation o^* originates from the same constraint as the selected operation o, then operation o^* is discarded to avoid checking a constraint more than once. This can be easily checked as each operation maintains a reference to its constraint.
- **Discard past operations.** If operation o^* is a past operation, then it is discarded as it violates the general operation applicability condition.
- **Append present check operations to the search plan.** If operation o^* is a present check operation, then it is immediately appended to the search plan to perform the corresponding checks as soon as possible.
- **Append present extension, future extension, and future check operations to the corresponding list.** If operation o^* is a present extension, a future extension or a future check operation, then it is appended to the corresponding operation sequence $O_{S'}^{pe}$, $O_{S'}^{fe}$, or $O_{S'}^{fc}$ of state S', respectively.

As operation application can only change variables from free to bound, a past operation can never be recategorized in any states derivable from S', (hence, its immediate disposal is justified) while a future operation might eventually become a present or past operation in a later phase of the algorithm.

The procedure `insert`$(T[i'], S', \mathbf{a}, \mathbf{c})$ determines $m = \min\{\mathbf{a}, k\}$, removes state $T[i'][m]$, shifts elements between $T[i'][\mathbf{c}]$ and $T[i'][m-1]$ downward, and inserts state S' at position $\mathbf{c}$.

Example. The dashed box of Fig. 5 presents the contents of table T (with 3 empty fields in the second row) after running our algorithm on Model 2 with initial adornment `BFFFF`. Each arrow represents the derivation of a new state, which was produced by one execution of the innermost cycle (lines 6–15). States with watermark A were temporarily stored in the table (but later discarded due to the appearance of better states). The state with letter B failed the reachability analysis, while states with watermark C were discarded as the corresponding column had already contained a better state with the same adornment.

For instance, the first execution of the innermost cycle processes operation `hasSensors(RO,IDX,SE)` with adornment `BFF`, whose weight is $\frac{\#\texttt{hasSensors}}{\#\texttt{Route}} = \frac{2}{1} = 2$ as Model 2 has 2 `hasSensors` links, and 1 `Route`. The corresponding new state is inserted into $T[2][1]$ as its adornment `BBBFF` has 2 free variables, and column $T[2]$ is empty at this time. In this new state, both costs are 2, operations with constraint `hasSensors(RO,IDX,SE)` are discarded, and all other operations are recategorized w.r.t. adornment `BBBFF`.

5 Measurement Results

In this section, we quantitatively assess the effects of different cost models and various configurations of our proposed search plan generation algorithm on the

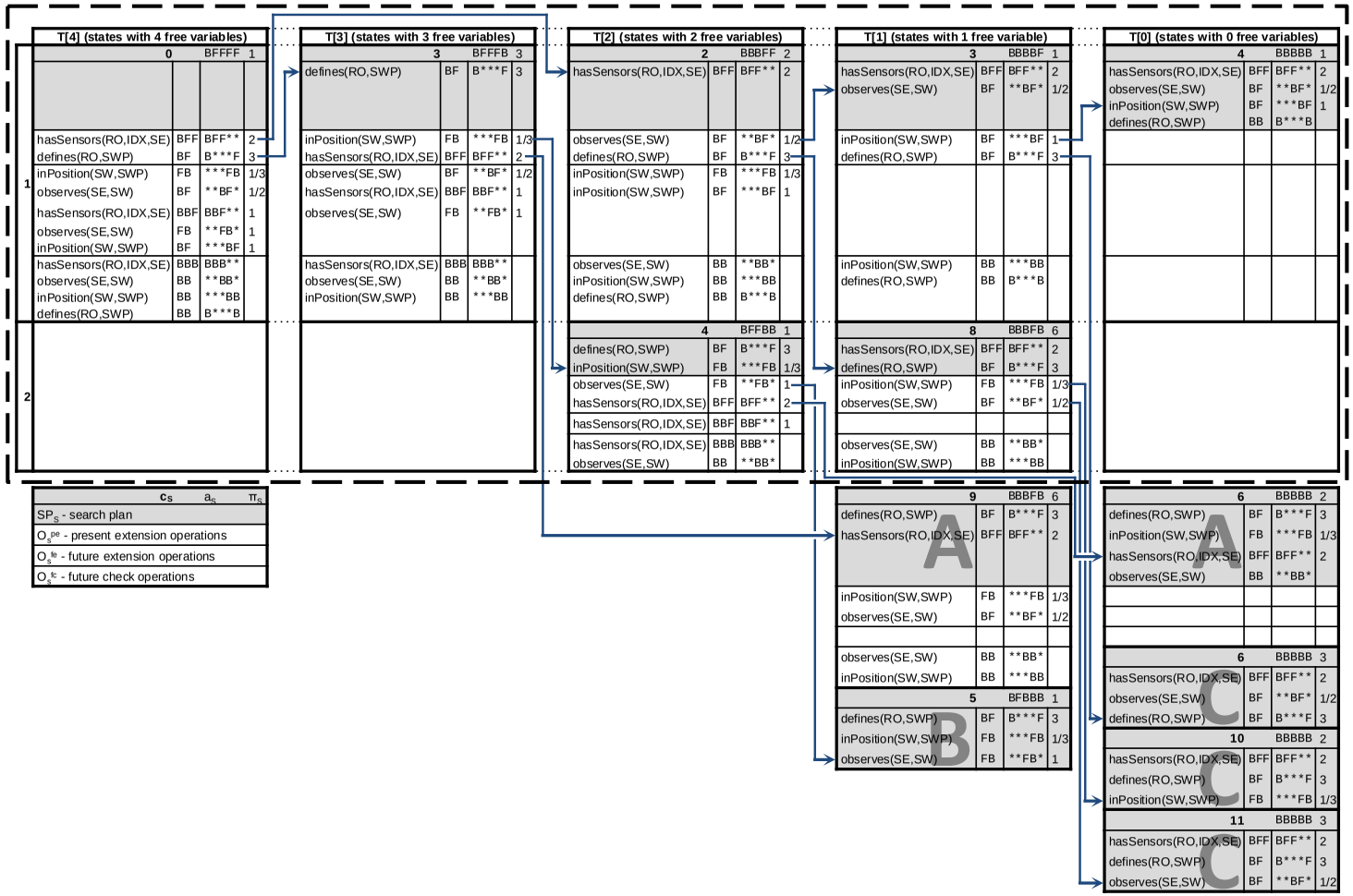


Fig. 5. Execution of the algorithm on Model 2 with $k = 2$

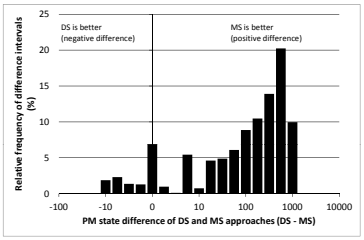
runtime performance of the pattern matching process. More specifically, our model-sensitive (MS) cost model was compared to a domain-specific (DS) approach, which latter used operation weights 1 and 10 for constraints representing structural features with at most one (1) and arbitrary (*) multiplicity, respectively. For configuring our algorithm, its parameter k was set to 1 and 2.

The pattern `routeSensor` of Fig. 2 and 10 models of different size from the case study [12] were used for experimentation purposes. Pattern matching was always restricted to a given `Route` in the model, which was assigned to variable `RO` in the initial match and used as a starting point. The complete process (including search plan generation) was repeated on each distinct `Route`.

Figure 6(a) presents the measured data. The first column indicates the model identifier, the second and third columns the model size and the number of distinct `Route`s in the model, respectively. The remaining columns show the measured values for the different configurations, which independently involve domain-specific (DS) and model-sensitive (MS) cost models, and algorithm parameter values $k = 1$ and 2. The PM columns denote the number of PM states (i.e., elementary pattern matching steps), which was averaged over all distinct `Route`s in the model. The SP columns show the cost of the (model-sensitive) search plan that was considered the best by the search plan generation algorithm and that was actually used to control pattern matching.

	Model size	Routes	DS (k=1) PM	DS (k=2) PM	MS (k=1) SP	MS (k=1) PM	MS (k=2) SP	MS (k=2) PM
	#	#	#	#	$c_{T[0][1]}$	#	$c_{T[0][1]}$	#
1	1450	20	1128.55	579.80	430	579.80	115	118.50
2	2601	40	885.15	456.75	349	456.75	102	104.78
4	5234	80	881.15	454.94	355	454.94	101	105.19
8	10627	160	912.64	470.81	361	470.81	102	106.29
16	21186	320	939.48	483.93	357	483.93	104	107.36
32	42202	640	936.80	482.44	353	482.44	104	107.21
64	85428	1280	960.83	494.65	362	494.65	105	108.51
128	171030	2560	955.14	491.77	362	491.77	105	108.78
256	339490	5120	943.89	486.02	356	486.02	104	107.93
512	685830	10240	953.93	491.18	364	491.18	106	109.18

(a) Comparison of PM state spaces



(b) PM state difference profile

Fig. 6. Measurement results

Fig. 6(a) shows that model-sensitive search plans have the capability to clearly outperform domain-specific ones (in this case on all test models by nearly 400 steps in average) when the pattern has many structural feature constraints with arbitrary multiplicity. Our algorithm generated the same search plan for the settings of the fifth and the seventh column, which explains the equal values there. Fig. 6(b) presents the *relative frequency* distribution histogram of the PM state differences of DS and MS approaches (with parameter $k = 2$) when these differences are calculated on a route-by-route basis for each of the 2560 starting points of model 128 (see the thick frames in Fig. 6(a)). Fig. 6(b) shows that the DS approach was better by 6 to 10 steps in 1.875% of the 2560 cases (first

column), the MS search plan was faster by 562 to 1000 steps in nearly 10% of the cases (last column), while a draw occured in 6.875% of the cases (fifth column).

In contrast to our preliminary expectations, which assumed that it was sufficient to set parameter k only to 1 in practical cases, it can be seen that a more thorough analysis with $k = 2$ can already pay off for small and simple patterns.

Unfortunately, the models of this case study were structurally similar, since all the MS search plans (irrespectively of the different models) were the same for a given parameter, which should not necessarily be the case. As further general characteristics, the average wall clock time[4] for search plan generation was 50 μs (for all configurations), and a single PM step took 51 ns in average. Neither the search plan generation, nor the pattern matching is affected by the model-sensitive nature of the approach, as object and link counters are initialized and incrementally updated, when the model is loaded and changed, respectively.

6 Related Work

Numerous useful model transformation tools are now surveyed, which internally perform search plan driven pattern matching. A more detailed comparison of pattern matcher engines is provided in [14].

Search plan driven pattern matchers. Fujaba [1] uses a search plan generation strategy that solely exploits type and multiplicity restrictions, which are derived from the metamodel. According to the used strategy, a navigation along an edge with an at most one multiplicity precedes navigations along edges with arbitrary multiplicity. Fujaba originally operated on top of a non-standard model representation, but recent versions can handle EMF models as well.

Pattern matchers driven by model-sensitive search plans. Although Fujaba [16] is a model-sensitive approach and runs on EMF models, it has only a simple greedy strategy to control pattern matching. GrGen [9] and Viatra [10], which employ model-sensitive search plans, operate on a non-standard modeling layer, which has several consequences. On one hand, these tools can use an arbitrary and optimized model representation, which can already have an integrated support for statistical data collection. On the other hand, if these tools aim to manipulate EMF-compliant models, then they have to be converted by import and export mechanisms, which (i) is not always possible for legacy EMF-based systems, and (ii) results in the inherent duplication of the complete model, which has a significant negative impact on the memory consumption. Since all other similarities and distinctions of GrGen, Viatra, and our approach are related to the employed search plan generation algorithms, these are evaluated in the following separate paragraphs.

Analysis of model-sensitive search plan generation algorithms. In contrast to our dynamic programming search plan generation algorithm, GrGen

[4] A 2.93 GHz Intel Pentium Dual-Core CPU with 3.7 GB RAM was used for all measurements. A 64-bit Ubuntu 11.04 with kernel 2.6.32–33 and Java 1.6.0_20 served as the underlying operating system and virtual machine, respectively. Measurements that result in time values were repeated 50 times for each starting point.

and Viatra use graph based techniques, which are obviously sufficient for sorting and filtering unary and binary constraints, which are the most widespread restriction types, but these solutions lack the integrated handling of general n-ary constraints, which are required for ordered references and pattern composition [5]. Both GrGen and Viatra support the construction of complex patterns from simpler ones, but the calculation of matches along pattern composition is scheduled by a separate piece of code and not the core search plan algorithm.

Search plan costs are calculated from the operation weights as a sum $\sum_i w_{o_i}$ in Viatra, and as a product $\prod_i w_{o_i}$ in GrGen, which can also be restructured to a sum by using the logarithm operator (i.e., $\sum_i \ln w_{o_i}$). As a graph based algorithm provides a provably optimal solution with these cost functions, they are perfect for filtering operations, but completely useless for sorting due to the insensitivity of these cost functions to the operation order.

A dynamic programming algorithm can cope with more complex cost functions, and it can provably find the optimum, if the whole solution space is explored when $k = \binom{n}{\lfloor \frac{n}{2} \rfloor}$. For a smaller k, the optimality is no longer guaranteed as the optimal search plan might have a prefix that is not among the best k adornment disjoint solutions at some point, and thus, this solution is discarded. In this sense, the selection of k can be considered as a trade-off between the polynomial runtime of the algorithm and the proven optimality of the solution.

Finally, it must be emphasized that the overall success of model-sensitive search plan generation algorithms highly rely on a strong correlation between the search plan cost and the size of the actually traversed state space, which is only a hypothesis that was thoroughly analyzed in [11], but not a provable fact.[5]

7 Conclusion

In this paper, we proposed a novel search plan generation algorithm based on dynamic programming together with a model-sensitive cost function for EMF models to speed up pattern matching in practice. Additionally, performance measurements have been carried out in a hardware and JVM independent manner to assess the effects of search plan generation on the pattern matching process.

Our future tasks are to repeat measurements in additional scenarios, to give a quantitative performance comparison of our approach to other pattern matchers, and to embed the pattern matching framework into different modeling tools.

Acknowledgements. The authors acknowledge the help of Benedek Izsó, István Ráth and Dániel Varró in providing us the railway scenario for the measurements.

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

[5] This means that the execution of the optimal search plan does not necessarily result in the traversal of the smallest state space.

Fujaba Days. (2005) 57–62 ftp://ftp.upb.de/doc/techreports/Informatik/tr-ri-05-259.pdf.
2. Jouault, F., Kurtev, I.: Transforming models with ATL. In Bézivin, J., Rumpe, B., Schürr, A., Tratt, L., eds.: Proc. of the International Workshop on Model Transformation in Practice. Volume 3844 of LNCS., Springer (2005) 128–138
3. Anjorin, A., Varró, G., Schürr, A.: Complex attribute manipulation in TGGs with constraint-based programming techniques. In Hermann, F., Voigtländer, J., eds.: Proc. of the 1st International Workshop on Bidirectional Transformations. Electronic Communications of the EASST (2012) Accepted paper.
4. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations. World Scientific (1997)
5. 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. Volume 6 of ECEASST. (2007)
6. Zündorf, A.: Graph pattern matching in PROGRES. In Cuny, J., Ehrig, H., Engels, G., Rozenberg, G., eds.: Proc. 5th Int. Workshop on Graph Grammars and Their Application to Computer Science. Volume 1073 of LNCS., Springer (1996) 454–468
7. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the Unified Modeling Language. In Engels, G., Rozenberg, G., eds.: Proc. of the 6th International Workshop on Theory and Application of Graph Transformation. Volume 1764 of LNCS., Springer (1998) 296–309
8. Rensink, A.: The GROOVE simulator: A tool for state space generation. In Pfalz, J.L., Nagl, M., Böhlen, B., eds.: Proc. of the 2nd International Symposium on the Applications of Graph Transformations with Industrial Relevance. Volume 3062 of LNCS., Springer (2004) 479–485
9. Geiß, R., Batz, V., Grund, D., Hack, S., Szalkowski, A.M.: GrGen: A fast SPO-based graph rewriting tool. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: Proc. of the 3rd International Conference on Graph Transformation. Volume 4178 of LNCS., Springer (2006) 383–397
10. 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. Volume 152 of ENTCS., Elsevier (2005) 191–205
11. Batz, G.V., Kroll, M., Geiß, R.: A first experimental evaluation of search plan driven graph pattern matching. In Schürr, A., Nagl, M., Zündorf, A., eds.: Proc. of the 3rd International Symposium on the Applications of Graph Transformation with Industrial Relevance. Volume 5088 of LNCS., Springer (2008) 471–486
12. Izsó, B.: Ontology based verification of system models. Master's thesis, Budapest University of Technology and Economics (2011) In Hungarian.
13. The MOGENTES project. http://www.mogentes.eu/
14. Varró, G., Anjorin, A., Schürr, A.: Unification of compiled and interpreter-based pattern matching techniques. Technical Report 2922, Technische Universität Darmstadt (March 2012) http://tuprints.ulb.tu-darmstadt.de/2922/.
15. Deckwerth, F.: Model-sensitive search plan algorithm for EMF models. Master's thesis, Technische Universität Darmstadt (January 2012)
16. 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 Int. Workshop on Graph Transformation and Visual Modeling Techniques. Volume 18 of ECEASST. (2009)