Software and Systems Modeling manuscript No.
(will be inserted by the editor)

An Algorithm for Generating Model-Sensitive Search Plans for Pattern Matching on EMF Models

Gergely Varró[⋆1], **Frederik Deckwerth**[⋆⋆1], **Martin Wieber**[1], **Andy Schürr**[1]

Real-Time Systems Lab,
Technische Universität Darmstadt,
D-64283 Merckstraße 25, Darmstadt, Germany
e-mail: gergely.varro, frederik.deckwerth, martin.wieber, andy.schuerr@es.tu-darmstadt.de

Received: date / Revised version: date

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, the search plan generation algorithm itself and its runtime effects on the pattern matching engine have been evaluated by complexity analysis techniques and by quantitative performance measurements, respectively.

Key words graph pattern matching – search plan generation algorithm – model-sensitive search plan

1 Introduction

Efficient, scalable, and standard compliant tools and techniques 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 checking the application conditions in rule-based model transformation tools [11,15], bidirectional model synchronization, or 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 to 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 using the services of the modeling layer (e.g., type checks, navigation along links), while non-structural constraints are handled by other means (e.g., integer or textual comparison).

As non-structural constraints are easily manageable if attribute values in symbolic graphs [16] can be restricted in an unambiguous manner by performing user-defined operations [2], the current paper solely focuses on structural constraints that correspond to the graph pattern matching problem [26]. Although recently available pattern matching engines support type checks and link navigations as unary and binary structural constraints, respectively, practical model-driven scenarios also require the handling of n-ary constraints to express ordered references or pattern composition [14].

When constructing 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 [36], which can be efficiently evaluated.

While the majority of state-of-the-art search plan generation algorithms [9,11,24] exploits only type and multiplicity restrictions derived from the metamodel of the problem domain, two novel *model-sensitive* approaches [12,34] 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 [4], 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.

This paper is an extended version of [33], which proposed a completely new model-sensitive search plan generation algorithm, based on dynamic programming, to enable the inte-

Send offprint requests to: Gergely Varró

⋆ Supported by the Postdoctoral Research Fellowship of the Alexander von Humboldt Foundation, and associated with the Center for Advanced Security Research Darmstadt, and the DFG funded CRC 1053 MAKI.

⋆⋆ Supported by CASED (www.cased.de)

Correspondence to: gergely.varro@es.tu-darmstadt.de

grated handling of general n-ary constraints. The algorithm collects statistical data from the model under transformation via an extensible framework to improve the precision of the estimations on operation selectivity [22], which have a highly critical role in the optimization process. 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 using runtime measurements.

In this paper, as an extension of [33], (i) a *comprehensive algorithm description* is provided, which includes the presentation of *all precompilation steps* (Sec. 3.2) and *subprocedures* (Algorithms 4 to 9), (ii) all algorithmic tasks are analyzed from a complexity point of view (Sec. 4.4), (iii) the running example has been significantly extended (Sec. 4.5), (iv) the performance of our search plan generation algorithm has been quantitatively compared to other model-sensitive approaches (Sec. 5.2), (v) query optimization methods from other domains have been evaluated as related work (Sec. 6.1).

The remainder of the paper is structured as follows: Section 2 introduces basic modeling and pattern specification concepts. The general pattern matching process (including its precompilation steps) is surveyed in Sec. 3, while Sec. 4 presents the new search plan generation algorithm. Section 5 provides 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

In this section, we introduce basic (meta)modeling concepts and our notation for specifying patterns. Technical considerations related to the underlying EMF implementation are also discussed.

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 [1] (developed in the MOGENTES project [29]), 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 the 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.

EMF-Specific Issues: In EMF, fully functional Java interfaces and implementation classes can be generated from the classes of the metamodel. In this generation process, references and attributes, that are collectively referred to as *structural features*, are handled uniformly. For each navigable direction of each structural feature, an attribute and getter and setter methods are produced in the Java code representing the source class. The generated Java attribute is an indexed `List`, which stores the corresponding target objects. The generated Java interfaces and implementation classes can be instantiated at runtime, and the EMF-compliant objects on the Java heap altogether constitute the EMF model.

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).

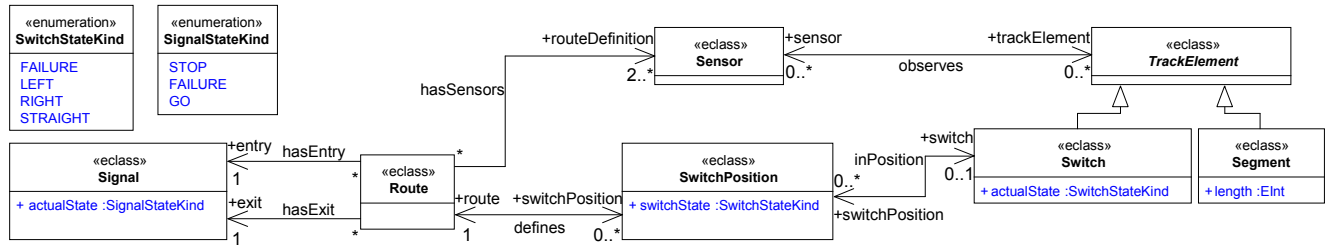
2.2 Pattern Specification

As defined in [14, 32], 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.

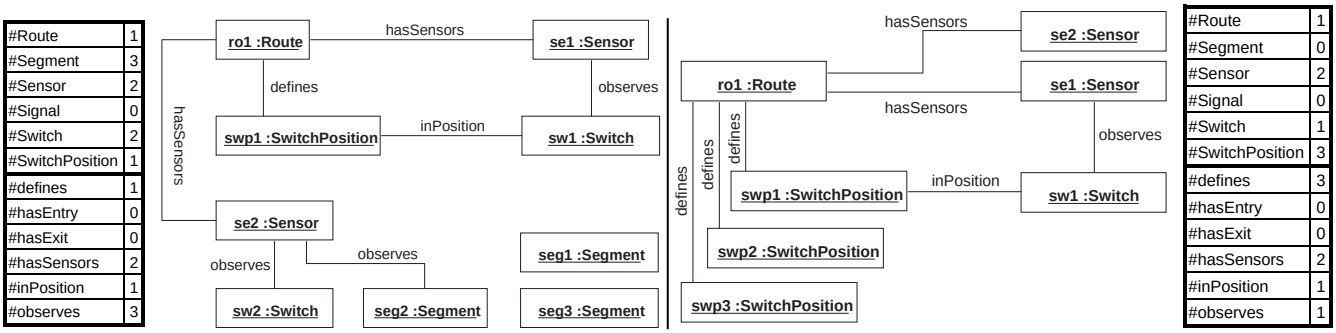
EMF-Specific Issues: Although the pattern matcher has a pluggable infrastructure for the constraints that can be used for specifying patterns, only one kind of constraint is used throughout the paper. In the following, a constraint maintains a reference to a structural feature. It also prescribes the existence of a link, which both (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 1 Pattern `routeSensor` (Fig. 2) expresses a sample requirement defined by railway domain experts. It has been simplified slightly for presentation purposes, and states that a route must have a sensor observing a switch, and that the observed switch itself must be part of the route. The pattern comprises five variables (`RO`, `IDX`, `SE`, `SW` and `SWP`), one ternary and three binary constraints, which prescribe the



(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

existence of an ordered unidirectional and three bidirectional references, respectively.

3 Pattern Matching Process

As [32] 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.

In a runtime session, a pattern matcher searches for those complete matches in the model that satisfy all constraints of the specified pattern. An initial partial match, which can already map some of the variables to objects, is used as a starting point of the recursive search process, which is characterized by a *fixed constraint sequence (i.e., the search plan)*. At each recursion level, the evaluation of the corresponding constraint in the sequence is carried out by an *operation*, which is a precompiled, atomic constraint evaluation step in the pattern matching process. An operation can only be performed if the runtime binding of the constraint parameters coincides with the specified *operation adornment*, which can be considered as an application condition. Two kinds of operations are used in the pattern matching process. An *extension operation* makes a step towards completing a partial match by using objects assigned to bound variables and binding free variables. A *check operation* filters a match if its bound variables are mapped to objects in a constraint violating manner.

The task of *search plan generation* is to find a valid operation sequence (i.e., fulfilling the application condition of each operation) that can be efficiently evaluated in the recursive search process of pattern matching.

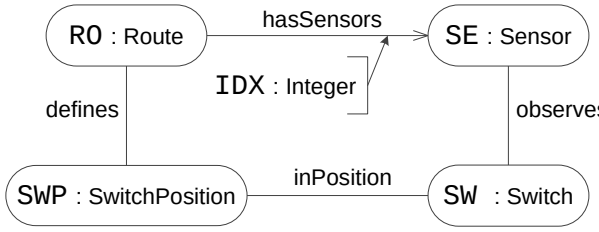
Validity of search plans. To compactly describe operation sequences in the search plan generation phase and to determine their validity, a state transition system is introduced, where the concept of *adornment* is used as a state descriptor that expresses binding information for all variables of a pattern, while the *application of an operation* can be considered as a transition. Operation applicability (i.e., expressed by the operation adornment) depends on the actual binding of the constraint parameters, which constitute a *subset* of the variables. To ease the calculations of operation applicability in the context of an adornment (i.e., which involves binding information for *all* variables), a *mask* is derived from the operation adornment.

Efficiency of search plans. The search plan generation phase uses a *(search plan) cost* to characterize the efficiency of a valid operation sequence. This cost estimates the size of the state space that would be explored during the recursive search process if the search plan was executed. The search plan cost is computed based on the weights of the operations in the sequence. An *operation weight* reflects the estimated number of objects that would have to be considered as a possible extension of a partial match if the operation was executed at a certain recursion level in the search. The operation weights are actually obtained from statistical data collected from the model.

The overall process of pattern matching is as follows:

- **Tasks at specification time.** Two tasks are performed *exactly once* for each pattern specification.

The final publication is available at http://link.springer.com/article/10.1007%2Fs10270-013-0372-2




```
pattern routeSensor(RO:Route,IDX:Integer,
    SE:Sensor,SW:Switch,SWP:SwitchPosition)={
  hasSensors(RO,IDX,SE);
  observes(SE,SW);
  inPosition(SW,SWP);
  defines(RO,SWP);
}
```

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

  - **Section 3.1.** Operations representing atomic, precompiled constraint evaluation steps in the pattern matching process are created from the pattern specification.
  - **Section 3.2.** By performing a backward reachability analysis, invalid operation sequences that could never produce complete matches are filtered out and stored in a precompiled data structure to speed up tasks performed later at runtime.
- **Tasks at runtime.** Two tasks are carried out *each time* when pattern matching is invoked.
  - **Section 3.3.** The operations are filtered and sorted by a *search plan generation algorithm* (for the details see Sec. 4) to produce efficient search plans.[1]
  - **Section 3.4.** 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.

3.1 Creating Operations

This subsection, which reuses some definitions from [14,32], introduces a compact notation for operation sequences that will control the pattern matching execution in Sec. 3.4. Additionally, the process of creating operations from the constraints in the pattern specification is also described. In the following, it is assumed that (i) a pattern has $|V|$ variables with an (arbitrary) fixed order, and (ii) the notation v_p denotes the pth variable according to this order.

An *adornment* [14] represents the 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. The *final adornment* $a_{(\mathtt{B})^*}$ contains only `B` characters, and thus, corresponds to the situation, when all the variables are bound. Considering the search process of Sec. 3.4, an adornment describes whether a variable is bound or free in all matches computed at a certain level of recursion.

Example 2 In the following, we suppose that variables `RO`, `IDX`, `SE`, `SW` and `SWP` of the `routeSensor` pattern are ordered in this specific sequence. The adornment `BFFFF` compactly describes that variable `RO` is bound, while variables `IDX`, `SE`, `SW` and `SWP` are free.

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. An operation adornment and the corresponding mask both convey the same binding information but use a syntactically different notation. A *check operation* has only bound parameters. An *extension operation* has at least one free parameter, which gets bound when the operation is executed.

The following process creates $|O|$ operations from the constraints in the pattern specification.

Maintaining references to constraints. Each operation o maintains a reference to the constraint $\mathtt{c}_o$, from which it originates.

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*, three 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*, two 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 $\star$, `B`, and `F` characters. Character $\star$ 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` or `F` in the adornment,

[1] By using caching mechanisms the search plan generation algorithm can be executed in a just-in-time manner.

the position p of the corresponding parameter v_p is looked up by using the fixed variable order, and position p is set in the mask to `B` or `F`, respectively. All other locations of the mask are set to $\star$.

Example 3 Figure 3 lists the operations that are derived from the `routeSensor` pattern. E.g., the `observes(SE,SW)` operation with (operation) adornment `BF` (highlighted by the thick frame with grey background in Fig. 3) represents the precompiled and atomic pattern matching step, which can evaluate constraint `observes(SE,SW)` when its first parameter (i.e., variable `SE`) is bound and its second parameter (i.e., variable `SW`) is free. The same application condition is also reflected in the corresponding mask $\star\star$`BF`$\star$ as `SE` and `SW` are the third and fourth variable according to the previously defined variable order, respectively. As the binding information for variables `RO`, `IDX` and `SWP` does not influence the applicability of the operation, mask $\star\star$`BF`$\star$ has the character $\star$ at the first, second, and fifth position, respectively.

In the first three tasks presented in Secs. 3.1–3.3, an operation is considered as an abstract step, which has an application condition expressed by the operation adornment and the corresponding mask. The actual implementation will only be relevant during the execution of the search plan in Sec. 3.4, when the operation (i) looks up the Sensor object that was assigned to the bound variable `SE` according to a partial match, (ii) navigates to all neighbouring Switch objects along the observes links, and (iii) a match is created for each newly explored Switch object by extending the original partial match with a mapping that assigns the Switch object to variable `SW`. As the operation binds the free variable `SW` and extends a match, it is an extension operation.

Categorizing operations. Operations can be categorized in the context of an adornment. 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. Formally, $\forall p,\ 1 \leq p \leq |V| : m_o[p] = \texttt{F} \implies a[p] = \texttt{F}$.
2. **Immediate operation applicability.** Each variable v_p, which must be *bound* according to the mask m_o of operation o, is also *bound* in adornment a. Formally, $\forall p,\ 1 \leq p \leq |V| : m_o[p] = \texttt{B} \implies a[p] = \texttt{B}$.

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.

The procedure `categorize`(o, a) (Algorithm 1) presents the categorization process in an algorithmic manner. It is initially assumed that operation o fulfills both applicability conditions (line 1). Operation mask m_o and adornment a are then compared at each position (lines 2–8). If the general operation applicability condition is violated (line 3), then operation o is immediately and irrevocably categorized as a past operation (line 4). However, if the immediate operation applicability condition is violated (line 5), then operation o is first temporarily categorized as a future operation (line 6), which turns into a final categorization result, when the cycle exits (line 9).

Applying operations. 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 $a \overset{o}{\Rightarrow} a'$) (i) binds all free variables indicated by mask m_o of operation o, and (ii) leaves the binding of all other variables unaltered.

An operation sequence $\langle o_1, \ldots, o_l \rangle$ starting from adornment a_0 is *valid*, if a sequence of adornments $a_1, \ldots, a_l$ can be derived where (i) each operation o_r is a present (applicable) operation with respect to the previous adornment a_{r-1}, and (ii) adornment a_r is produced by applying operation o_r on the previous adornment a_{r-1}. Formally,

$$a_0 \overset{o_1}{\Rightarrow} a_1 \overset{o_2}{\Rightarrow} \ldots \overset{o_{r-1}}{\Rightarrow} a_{r-1} \overset{o_r}{\Rightarrow} a_r \overset{o_{r+1}}{\Rightarrow} \ldots \overset{o_l}{\Rightarrow} a_l.$$

An adornment a is *backward reachable*, if there exists a valid operation sequence starting from adornment a that leads to the final adornment $a_{(\texttt{B})^*}$.

Example 4 The `observes(SE,SW)` operation with mask $\star\star$`BF`$\star$ can be categorized as a future operation with respect to adornment `BFFFF`, as it violates the immediate operation applicability condition at the third position. The third character in adornment `BFFFF` states that variable `SE` is free, while the character at the same position in mask $\star\star$`BF`$\star$ demands that this variable should be bound. Consequently, the operation cannot be currently applied, but it might eventually become applicable, when variable `SE` gets bound at some point in the future.

3.2 Reachability Analysis

In order to have a fast search plan generation process at runtime, backward reachable adornments have to be determined in advanced (at specification time). This is achieved by (i) introducing Boolean variables for pattern variables, (ii) preparing Boolean formulas for sets of adornments and operations to produce state and transition descriptions, respectively, and (iii) executing a backward reachability analysis on this newly defined state-transition system.

Mapping the binding information. In the following, a freedom indicator function φ is used to map binding information characters `B` and `F` to truth values `false` and `true`, respectively. Formally,

$$\varphi(\alpha) = \begin{cases} \texttt{false} & \text{if } \alpha = \texttt{B}, \text{ and} \\ \texttt{true} & \text{if } \alpha = \texttt{F}. \end{cases}$$

Boolean formulas for adornment sets. For each variable v_p in a pattern, a Boolean variable v_p is introduced. A *characteristic function $\mathcal{A}(\mathsf{v}_1, \ldots, \mathsf{v}_{|V|})$ of an adornment set A consisting of adornments of length $|V|$* is expressed as a Boolean formula over the Boolean variables $\mathsf{v}_1, \ldots, \mathsf{v}_{|V|}$. The *evaluation of the characteristic function $\mathcal{A}(\mathsf{v}_1, \ldots, \mathsf{v}_{|V|})$ of an*

Operation			*Category*	*Type*	*Boolean Formula*
Constraint	*Op. Adornm.*	*Mask*	(w.r.t. BFFFF)		
`hasSensors(RO,IDX,SE)`	`BBB`	`BBB**`	future	*check*	$(\neg\mathtt{RO} \wedge \neg\mathtt{RO}') \wedge (\neg\mathtt{IDX} \wedge \neg\mathtt{IDX}') \wedge (\neg\mathtt{SE} \wedge \neg\mathtt{SE}') \wedge (\mathtt{SW} \Leftrightarrow \mathtt{SW}') \wedge (\mathtt{SWP} \Leftrightarrow \mathtt{SWP}')$
`hasSensors(RO,IDX,SE)`	`BBF`	`BBF**`	future	**extension**	$(\neg\mathtt{RO} \wedge \neg\mathtt{RO}') \wedge (\neg\mathtt{IDX} \wedge \neg\mathtt{IDX}') \wedge (\mathtt{SE} \wedge \neg\mathtt{SE}') \wedge (\mathtt{SW} \Leftrightarrow \mathtt{SW}') \wedge (\mathtt{SWP} \Leftrightarrow \mathtt{SWP}')$
`hasSensors(RO,IDX,SE)`	`BFF`	`BFF**`	present	**extension**	$(\neg\mathtt{RO} \wedge \neg\mathtt{RO}') \wedge (\mathtt{IDX} \wedge \neg\mathtt{IDX}') \wedge (\mathtt{SE} \wedge \neg\mathtt{SE}') \wedge (\mathtt{SW} \Leftrightarrow \mathtt{SW}') \wedge (\mathtt{SWP} \Leftrightarrow \mathtt{SWP}')$
`observes(SE,SW)`	`BB`	`**BB*`	future	*check*	$(\mathtt{RO} \Leftrightarrow \mathtt{RO}') \wedge (\mathtt{IDX} \Leftrightarrow \mathtt{IDX}') \wedge (\neg\mathtt{SE} \wedge \neg\mathtt{SE}') \wedge (\neg\mathtt{SW} \wedge \neg\mathtt{SW}') \wedge (\mathtt{SWP} \Leftrightarrow \mathtt{SWP}')$
`observes(SE,SW)`	`BF`	`**BF*`	future	**extension**	$(\mathtt{RO} \Leftrightarrow \mathtt{RO}') \wedge (\mathtt{IDX} \Leftrightarrow \mathtt{IDX}') \wedge (\neg\mathtt{SE} \wedge \neg\mathtt{SE}') \wedge (\mathtt{SW} \wedge \neg\mathtt{SW}') \wedge (\mathtt{SWP} \Leftrightarrow \mathtt{SWP}')$
`observes(SE,SW)`	`FB`	`**FB*`	future	**extension**	$(\mathtt{RO} \Leftrightarrow \mathtt{RO}') \wedge (\mathtt{IDX} \Leftrightarrow \mathtt{IDX}') \wedge (\mathtt{SE} \wedge \neg\mathtt{SE}') \wedge (\neg\mathtt{SW} \wedge \neg\mathtt{SW}') \wedge (\mathtt{SWP} \Leftrightarrow \mathtt{SWP}')$
`inPosition(SW,SWP)`	`BB`	`***BB`	future	*check*	$(\mathtt{RO} \Leftrightarrow \mathtt{RO}') \wedge (\mathtt{IDX} \Leftrightarrow \mathtt{IDX}') \wedge (\mathtt{SE} \Leftrightarrow \mathtt{SE}') \wedge (\neg\mathtt{SW} \wedge \neg\mathtt{SW}') \wedge (\neg\mathtt{SWP} \wedge \neg\mathtt{SWP}')$
`inPosition(SW,SWP)`	`BF`	`***BF`	future	**extension**	$(\mathtt{RO} \Leftrightarrow \mathtt{RO}') \wedge (\mathtt{IDX} \Leftrightarrow \mathtt{IDX}') \wedge (\mathtt{SE} \Leftrightarrow \mathtt{SE}') \wedge (\neg\mathtt{SW} \wedge \neg\mathtt{SW}') \wedge (\mathtt{SWP} \wedge \neg\mathtt{SWP}')$
`inPosition(SW,SWP)`	`FB`	`***FB`	future	**extension**	$(\mathtt{RO} \Leftrightarrow \mathtt{RO}') \wedge (\mathtt{IDX} \Leftrightarrow \mathtt{IDX}') \wedge (\mathtt{SE} \Leftrightarrow \mathtt{SE}') \wedge (\mathtt{SW} \wedge \neg\mathtt{SW}') \wedge (\neg\mathtt{SWP} \wedge \neg\mathtt{SWP}')$
`defines(RO,SWP)`	`BB`	`B***B`	future	*check*	$(\neg\mathtt{RO} \wedge \neg\mathtt{RO}') \wedge (\mathtt{IDX} \Leftrightarrow \mathtt{IDX}') \wedge (\mathtt{SE} \Leftrightarrow \mathtt{SE}') \wedge (\mathtt{SW} \Leftrightarrow \mathtt{SW}') \wedge (\neg\mathtt{SWP} \wedge \neg\mathtt{SWP}')$
`defines(RO,SWP)`	`BF`	`B***F`	present	**extension**	$(\neg\mathtt{RO} \wedge \neg\mathtt{RO}') \wedge (\mathtt{IDX} \Leftrightarrow \mathtt{IDX}') \wedge (\mathtt{SE} \Leftrightarrow \mathtt{SE}') \wedge (\mathtt{SW} \Leftrightarrow \mathtt{SW}') \wedge (\mathtt{SWP} \wedge \neg\mathtt{SWP}')$
`defines(RO,SWP)`	`FB`	`F***B`	past	**extension**	$(\mathtt{RO} \wedge \neg\mathtt{RO}') \wedge (\mathtt{IDX} \Leftrightarrow \mathtt{IDX}') \wedge (\mathtt{SE} \Leftrightarrow \mathtt{SE}') \wedge (\mathtt{SW} \Leftrightarrow \mathtt{SW}') \wedge (\neg\mathtt{SWP} \wedge \neg\mathtt{SWP}')$

Fig. 3 Operations (categorized with respect to adornment `BFFFF`) and corresponding Boolean formulas

Algorithm 1 The procedure `categorize`(o, a)

```
1: cat := PRESENT
2: for (p := 1 to |V|) do
3:     if (m_o[p] = F ∧ a[p] = B) then // General operation applicability is violated
4:         return PAST
5:     else if (m_o[p] = B ∧ a[p] = F) then // Immediate operation applicability is violated
6:         cat := FUTURE
7:     end if // Both operation applicability conditions are fulfilled
8: end for
9: return cat
```

adornment set A on a given adornment a substitutes each Boolean variable v_p with a logic value $\varphi(a[p])$, which is assigned by the freedom indicator function φ to the binding information $a[p]$ of variable v_p according to adornment a. Formally, for a given adornment a, $\mathcal{A}(\varphi(a[1]), \ldots, \varphi(a[|V|]))$ is calculated. The characteristic function $\mathcal{A}(\mathsf{v}_1, \ldots, \mathsf{v}_{|V|})$ of an adornment set A is evaluated to `true` on exactly those adornments that are contained in adornment set A. Formally,

$$a \in A \Longleftrightarrow \mathcal{A}(\varphi(a[1]), \ldots, \varphi(a[|V|])) = \mathtt{true}.$$

Example 5 For instance, the final adornment `BBBBB` can be represented by the characteristic function $\mathcal{A}_0 = \neg\mathtt{RO} \wedge \neg\mathtt{IDX} \wedge \neg\mathtt{SE} \wedge \neg\mathtt{SW} \wedge \neg\mathtt{SWP}$, which is evaluated to `true` if and only if $\mathtt{RO} = \mathtt{IDX} = \mathtt{SE} = \mathtt{SW} = \mathtt{SWP} = \mathtt{false} = \varphi(\mathtt{B})$.

Boolean formulas for operations. Character $m_o[p]$ at position p in mask m_o of operation o expresses conditions and changes in the binding information for variable v_p, which can be compactly defined by a Boolean formula

$$\mathcal{R}_o^p(\mathsf{v}_p, \mathsf{v}'_p) = \begin{cases} \neg\mathsf{v}_p \wedge \neg\mathsf{v}'_p & \text{if } m_o[p] = \mathtt{B} \\ \mathsf{v}_p \wedge \neg\mathsf{v}'_p & \text{if } m_o[p] = \mathtt{F} \\ \mathsf{v}_p \Leftrightarrow \mathsf{v}'_p & \text{if } m_o[p] = * \end{cases}$$

where Boolean variables v_p and v'_p represent the binding information for variable v_p *before* and *after* the application of operation o, respectively. By considering the freedom indicator function φ, the Boolean formula $\neg\mathsf{v}_p \wedge \neg\mathsf{v}'_p$ is evaluated to `true`, if and only if variable v_p is bound before and after the application of operation o, which is exactly what $m_o[p] = \mathtt{B}$ prescribes. Similarly, in case of $m_o[p] = \mathtt{F}$, variable v_p must be free (v_p) before applying operation o and bound ($\neg\mathsf{v}'_p$) afterwards. Finally, the expression $\mathsf{v}_p \Leftrightarrow \mathsf{v}'_p$ defined for the case $m_o[p] = *$ ensures that the binding information for variable v_p remains unaltered.

Conditions for and effects of applying operation o with mask m_o of character length $|V|$ can be described by a Boolean formula $\mathcal{R}_o(\mathsf{v}_1, \ldots, \mathsf{v}_{|V|}, \mathsf{v}'_1, \ldots, \mathsf{v}'_{|V|})$ with $2\,|V|$ Boolean variables, which is produced as the conjunction of the composing Boolean formulas $\mathcal{R}_o^p(\mathsf{v}_p, \mathsf{v}'_p)$. Formally,

$$\mathcal{R}_o(\mathsf{v}_1, \ldots, \mathsf{v}_{|V|}, \mathsf{v}'_1, \ldots, \mathsf{v}'_{|V|}) = \bigwedge_{p=1}^{|V|} \mathcal{R}_o^p(\mathsf{v}_p, \mathsf{v}'_p)$$

A Boolean formula $\mathcal{R}_O(\mathsf{v}_1, \ldots, \mathsf{v}_{|V|}, \mathsf{v}'_1, \ldots, \mathsf{v}'_{|V|})$ for an operation set O can be obtained by the disjunction of the Boolean formulas $\mathcal{R}_o(\mathsf{v}_1, \ldots, \mathsf{v}_{|V|}, \mathsf{v}'_1, \ldots, \mathsf{v}'_{|V|})$ defined for all operations o in the set O. Formally,

$$\mathcal{R}_O(\mathsf{v}_1, \ldots, \mathsf{v}_{|V|}, \mathsf{v}'_1, \ldots, \mathsf{v}'_{|V|}) = \bigvee_{o \in O} \mathcal{R}_o(\mathsf{v}_1, \ldots, \mathsf{v}_{|V|}, \mathsf{v}'_1, \ldots, \mathsf{v}'_{|V|}).$$

Example 6 The Boolean formulas that correspond to the operations created for `routeSensor` pattern are depicted in the rightmost column of Figure 3. For example, the Boolean formula of the operation `observes(SE,SW)` with mask `**BF*` (highlighted by the thick frame with grey background in Fig. 3) is constructed by the conjunction of $(\mathtt{RO} \Leftrightarrow \mathtt{RO}')$, $(\mathtt{IDX} \Leftrightarrow \mathtt{IDX}')$ and $(\mathtt{SWP} \Leftrightarrow \mathtt{SWP}')$ for $*$ at positions 1, 2 and 5; $(\neg\mathtt{SE} \wedge \neg\mathtt{SE}')$ for `B` at position 3; and $(\mathtt{SW} \wedge \neg\mathtt{SW}')$ for `F` at position 4.

Backward reachability analysis using Boolean formulas. Backward reachable adornments can be computed iteratively by a backward reachability analysis (cf. Algorithm 2)

The final publication is available at http://link.springer.com/article/10.1007%2Fs10270-013-0372-2

which uses fixed point calculation on the Boolean representation of adornment sets and operations. The fixed point calculation is initialized (in line 1 of Algorithm 2) with $\mathcal{A}_0$, which is the characteristic function of the singleton set containing the final adornment $a_{(\mathtt{B})^*}$. In each iteration (line 3–5 in Algorithm 2), the set of backward reachable adornments represented by $\mathcal{A}_h$ is extended by the Boolean formula of those adornments, from which an adornment in $\mathcal{A}_h$ can be reached by applying one single operation. If the characteristic function $\mathcal{A}_h$ remains unchanged in an iteration (line 5), then it is returned (line 6), and Algorithm 2 terminates.

Example 7 The execution of Algorithm 2 on the operations of the `routeSensor` pattern is illustrated in Figs. 4(a) to 4(d), which depict the initial $\mathcal{A}_0$ and the 3 calculated characteristic functions $\mathcal{A}_1$–$\mathcal{A}_3$, respectively. The characteristic functions are presented as Karnaugh maps [35] (in the upper parts) as well as Boolean formulas (in a minimized conjunctive normal form representation in the lower parts). A Karnaugh map is the truth table representation of a Boolean function, in which each cell stores the truth value assigned to one combination of input conditions. E.g., the grey cell in Fig. 4(b) corresponds to the situation, when truth values `false` ($\neg$`RO`), `false` ($\neg$`IDX`), `false` ($\neg$`SE`), `true` (`SW`), `false` ($\neg$`SWP`) are assigned to Boolean variables `RO`, `IDX`, `SE`, `SW`, `SWP`, respectively. In this case, the characteristic function $\mathcal{A}_1$ returns the cell content 1 (i.e., the truth value `true`).

An iteration in Algorithm 2 can be demonstrated by calculating the conjunction of $\mathcal{A}_0(\mathtt{RO}', \mathtt{IDX}', \mathtt{SE}', \mathtt{SW}', \mathtt{SWP}') = \neg\mathtt{RO}' \wedge \neg\mathtt{IDX}' \wedge \neg\mathtt{SE}' \wedge \neg\mathtt{SW}' \wedge \neg\mathtt{SWP}'$ (representing the singleton set with the final adornment `BBBBB`) and the Boolean formula prepared for operation `observes(SE,SW)` with adornment `BF` (marked by the thick frame with grey background in Fig. 3). As already mentioned, $\mathcal{A}_0$ can only be evaluated to `true` if and only if $\mathtt{RO}' = \mathtt{IDX}' = \mathtt{SE}' = \mathtt{SW}' = \mathtt{SWP}' = \mathtt{false}$. When these truth values are substituted into the Boolean formula of the operation, we get $\neg\mathtt{RO} \wedge \neg\mathtt{IDX} \wedge \neg\mathtt{SE} \wedge \mathtt{SW} \wedge \neg\mathtt{SWP}$, which is evaluated to `true`, if Boolean variables `RO`, `IDX`, `SE`, `SW`, `SWP` are mapped to `false`, `false`, `false`, `true`, `false`, respectively. Note that this is again the case, which is represented by value 1 in the grey cell of the Karnaugh map in Fig. 4(b). Moreover, this case represents the adornment `BBBFB`, from which the final adornment `BBBBB` can be reached in one single step by applying operation `observes(SE,SW)` with adornment `BF`.

Implementation. In order to have an efficient implementation, reachability analysis is carried out on the reduced ordered binary decision diagram [20] (ROBDD) representation of Boolean formulas.

A binary decision diagram (BDD) is a directed acyclic graph with a single root. It consists of decision nodes and two terminal nodes (leaves). The latter two (with integers 0 and 1 inside) correspond to the truth values `false` and `true`, respectively. A decision node is characterized by a Boolean variable and it has two outgoing edges labelled by `false` and `true`, respectively. An outgoing edge of a decision node represents the assignment of the Boolean variable in the node to the truth value on the edge label. Consequently, each path leading from the root to a terminal node means one evaluation of the complete Boolean formula to the truth value of the terminal node by using the value assignments defined by the edges of the path. A BDD is ordered, if the Boolean variables appear in the same order on all paths that lead from the root to a terminal node. A sub-OBDD is a subgraph induced by a given node and all its transitively accessible child nodes. An ordered BDD is reduced, if (i) each decision node has different child nodes, and (ii) all sub-OBDDs are non-isomorphic.

Example 8 The ROBDD for the characteristic function $\mathcal{A}_3$ is shown in Fig. 5. In this figure, dashed or solid edges represent the assignment to the truth value `false` or `true`, respectively. The evaluation of this ROBBD on the adornment `BBBFB`, which corresponds to the variable assignment $\mathtt{RO} = \mathtt{false}$, $\mathtt{IDX} = \mathtt{false}$, $\mathtt{SE} = \mathtt{false}$, $\mathtt{SW} = \mathtt{true}$, $\mathtt{SWP} = \mathtt{false}$ is shown by the bold path in Fig. 5. The evaluation starts with a navigation from `RO` along the dashed edge ($\mathtt{RO} = \mathtt{false}$) to `IDX`, which is followed by a traversal of the other bold dashed edge ($\mathtt{IDX} = \mathtt{false}$) to the terminal node 1, which means that the characteristic function $\mathcal{A}_3$ evaluates to `true` in this case.

Using the results of backward reachability analysis at runtime. The ROBDD representation of the characteristic function $\mathcal{A}_h$ returned by Algorithm 2 is going to be used later in Section 4 by the search plan generation algorithm as a precompiled data structure to quickly determine at runtime whether an adornment a is backward reachable, which is indicated by the truth value `true` when the Boolean formula $\mathcal{A}_h$ is evaluated on adornment a. In formal terms, the method `isBackwardReachable`$(\mathcal{A}_h, a)$ returns

$$\mathcal{A}_h(\varphi(a[1]), \ldots, \varphi(a[|V|])).$$

Complexity analysis. When discussing complexity analysis results, it should be strongly emphasized that the *efficiency of the procedure* `isBackwardReachable`$(\mathcal{A}_h, a)$ *is of the utmost importance and significance* as (1) only this procedure is invoked by the search plan generation algorithm, and (2) search plans might need to be prepared *at each invocation* of the pattern matcher (in contrast to the complex backward reachability analysis machinery, which is carried out only once at specification time).

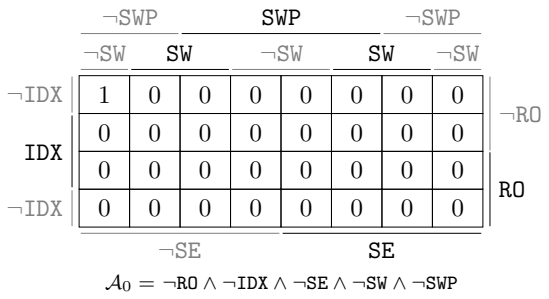
Remark 1 (Complexity of checking backward reachability at runtime) The procedure `isBackwardReachable`$(\mathcal{A}_h, a)$ requires the evaluation of the characteristic function $\mathcal{A}_h$, which can be carried out in $\mathcal{O}(|V|)$ steps.

In order to determine the complexity of Algorithm 2, basic logic operations on ROBDDs, which are described comprehensively in [20] together with their complexity analysis, must be assessed. Simple ROBDDs representing the conjunction of non-negated (v_p) or negated ($\neg\mathsf{v}_p$) Boolean variables can be produced in $|V|$ steps. In the following, the number of internal nodes in an ROBDD is denoted by $|R|$. Equality testing is linear in the number of internal nodes in the input ROBDDs (i.e., $\mathcal{O}(|R|)$) just like the unary restriction operation,

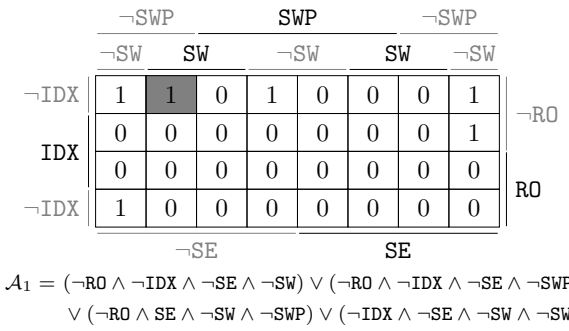
The final publication is available at http://link.springer.com/article/10.1007%2Fs10270-013-0372-2

Algorithm 2 The procedure `reachableSet`(R_O)

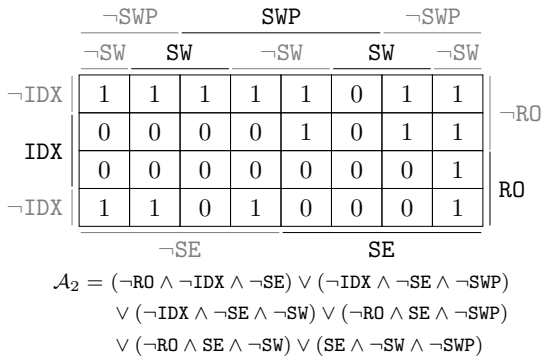
1: $\mathcal{A}_0(\mathsf{v}_1,\ldots,\mathsf{v}_{|V|}) := \neg\mathsf{v}_1 \wedge \ldots \wedge \neg\mathsf{v}_{|V|}$
2: $h := 0$
3: **repeat**
4: $\quad \mathcal{A}_{h+1}(\mathsf{v}_1,\ldots,\mathsf{v}_{|V|}) := \mathcal{A}_h(\mathsf{v}_1,\ldots,\mathsf{v}_{|V|}) \vee \exists\, \mathsf{v}'_1,\ldots,\mathsf{v}'_{|V|} : \left[\mathcal{R}_O(\mathsf{v}_1,\ldots,\mathsf{v}_{|V|},\mathsf{v}'_1,\ldots,\mathsf{v}'_{|V|}) \wedge \mathcal{A}_h(\mathsf{v}'_1,\ldots,\mathsf{v}'_{|V|})\right]$
5: **until** $(\mathcal{A}_h(\mathsf{v}_1,\ldots,\mathsf{v}_{|V|}) \neq \mathcal{A}_{h+1}(\mathsf{v}_1,\ldots,\mathsf{v}_{|V|}))$
6: **return** $\mathcal{A}_h(\mathsf{v}_1,\ldots,\mathsf{v}_{|V|})$



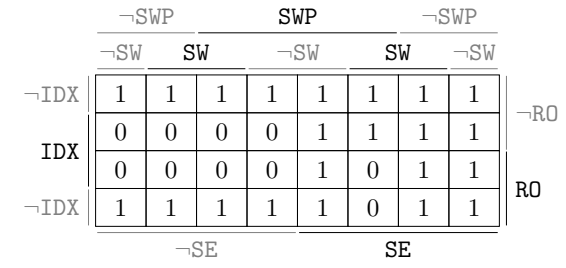
(a) The initial Boolean formula describing the final adornment



(b) The Boolean formula after the first iteration



(c) The Boolean formula after the second iteration



(d) The Boolean formula after the third (last) iteration

Fig. 4 The Boolean formulas produced by Algorithm 2 for the operations of Fig. 3

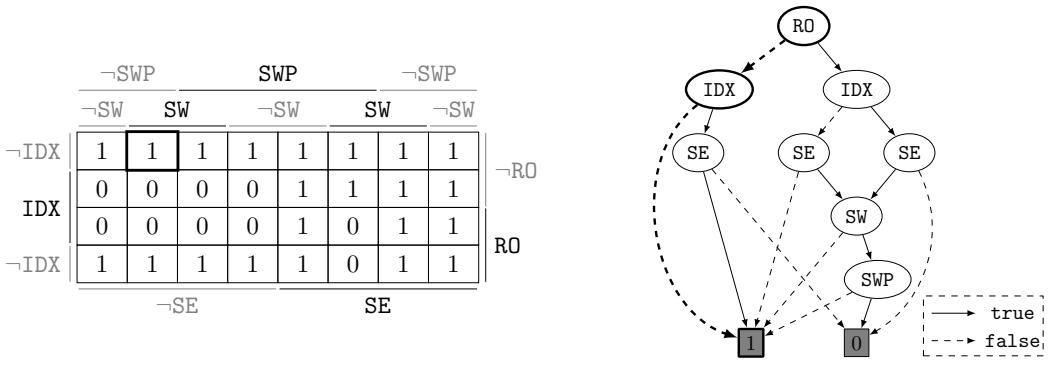


Fig. 5 The characteristic function $\mathcal{A}_3$ and its ROBDD representation

which assigns a truth value to a Boolean variable and calculates the resulting Boolean formula. The number of steps required for performing binary logic operations on ROBDDs is proportional to the product of the number of internal nodes in the operand ROBDDs ($\mathcal{O}(|R| \cdot |R|)$).

Based on these considerations, the characteristic formula $\mathcal{A}_0$ describing the final adornment $a_{(\mathtt{B})^*}$ can be constructed in $\mathcal{O}(|V|)$ steps, just like the Boolean formula $\mathcal{R}_o$ built for an operation o. The Boolean formula $\mathcal{R}_O$ that represents all the operations is calculated by $\mathcal{O}(|O|)$ disjunctions. Each iteration in Algorithm 2 (line 3) performs 1 disjunction, 1 conjunction, and 1 equality test, while the resolution of existential quantification requires 2 reductions and 1 disjunction for each quantified Boolean variable v'_p (i.e., $3 + 3|V|$ logic operations). At most $|V|$ iterations are carried out, as each cycle either increases the number of `F` characters in the added adornments by (at least) one, or termination is detected. By considering the fact that basic logic operations on ROBDDs also produce ROBDDs as a result, it can be stated that altogether $\mathcal{O}(|O| \cdot |R| \cdot |R| + |V| \cdot |V| \cdot |R| \cdot |R|)$ steps have been executed by Algorithm 2 at pattern specification time.

Although the worst case upper bound for the size of an ROBDD ($|R|$) is unfortunately exponential in the number of pattern variables, several arguments still justify the practical applicability of our approach and ROBDDs.

1. Reachability analysis is executed *only once* for each pattern *at specification time, in contrast to search plan generation, which is executed at runtime for each invocation* of the pattern matcher.
2. The *number of variables* in a pattern is typically *small* in practical application scenarios as shown by [5, 17].
3. The complexity of logic operations on ROBDDs is influenced by *the number of internal nodes* in an ROBDD, which *is always at most as large as the number of paths*. Additionally, a reduced OBDD has at most as many paths as a non-reduced OBDD, which corresponds to the truth table representation of a Boolean formula with an exponential number of cells.
4. The size of the ROBDD produced by Algorithm 2 is *frequently on a linear scale when pattern specifications only include the traditional unary and binary constraints* (i.e., type checks and link navigations), but no general n-ary constraints.
5. As the size of the intermediate ROBDDs is influenced by the order of the Boolean variables, *sophisticated techniques* like [6] *can avoid the production of large intermediate ROBDDs* in reachability analysis scenarios, if the Boolean formula $\mathcal{R}_O$ prepared for all the operations can be split into smaller independent expressions, which can be naturally and evidently done as each operation manipulates a well-identifiable set of positions in the adornments.

3.3 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_I. 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 [14].

A *search plan* $SP = \langle o_1, o_2, \ldots, o_l \rangle$*, starting from an initial adornment* a_I, is a valid operation sequence, in which each constraint of the pattern is represented by *at most one* corresponding operation. The *adornment* a_{SP} *of the search plan* SP is the last element a_l in the adornment sequence $a_I = a_0 \overset{o_1}{\Rightarrow} a_1 \overset{o_2}{\Rightarrow} \ldots \overset{o_l}{\Rightarrow} a_l = a_{SP}$ derived by using search plan SP on initial adornment a_I. A search plan is *complete*, if each constraint is represented by *exactly one* operation in the sequence, and the search plan adornment (the last adornment of the sequence) is the final adornment $a_{(B)^*}$.

Example 9 Figure 6 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 two branches with the last (check) operation, while SP2 employs a clockwise navigation along the references in the pattern.

Search plan	*Step*	*Operation*			Adornm. a_i (a_1 = 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**B**B
	(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*

Fig. 6 Search plans as sequence of operations

3.4 Search Plan Execution by a Pattern Matcher Interpreter

By conceptually following the corresponding part of [32], 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

The final publication is available at http://link.springer.com/article/10.1007%2Fs10270-013-0372-2

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 10 Figure 7 depicts two PM state spaces, which are traversed by executing search plans SP1 and SP2 on Model 2, respectively. For example, the second level of Fig. 7(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`. The leaves that are outlined represent those complete matches that pass beyond the last check operation (only shown in Fig. 6), while unframed ones fail this check. It is obvious from Fig. 7 that SP2 is better than SP1, as SP2 traverses less PM states.

4 Dynamic Programming Based Search Plan Generation

As demonstrated in Fig. 7, the search plan has a large impact on the number of produced (partial) 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 an ideal cost function should strongly correlate with the size of the PM state space.

4.1 Algorithm Data Structures

Operation weight calculation. An extension operation o is augmented by a *weight* w_o, which denotes the cost of performing the operation. From a clearly algorithmic aspect, operation weights can be arbitrarily defined.

In this paper, weight calculation uses the statistical data collected from the underlying EMF model. More specifically, a weight is defined as an average *branching factor* for that level of the PM state space tree, which represents the operation execution. 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 only algorithmic criterion is that the search plan cost c_l must be iteratively computable from the weight w_{o_l} of the last operation o_l and the cost c_{l-1} of the previous search plan (i.e., the one without the last operation).

In this paper, the search plan cost c_l 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 [34], 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 $c_0 = 0$, $\pi_0 = 1$,

$$\pi_l = \pi_{l-1} w_{o_l},$$

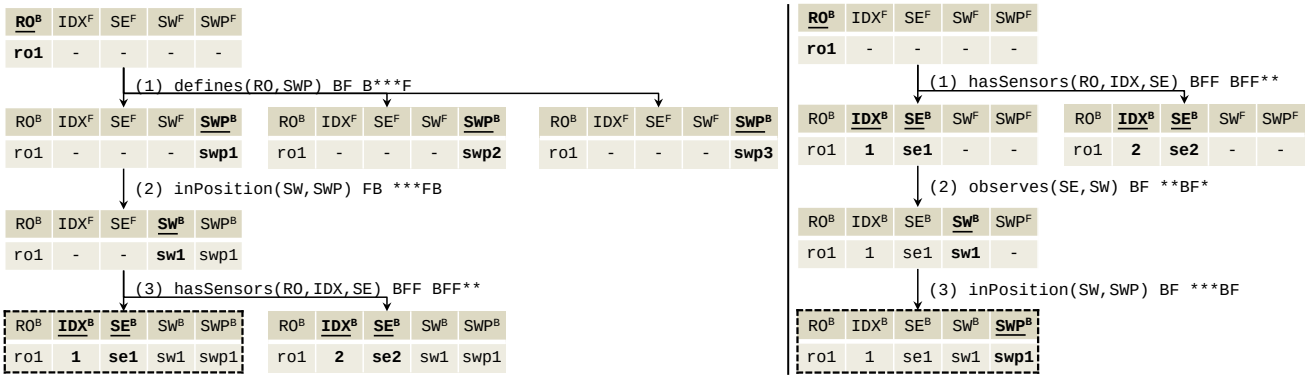
and

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

States. 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_I 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_I.

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



(a) PM state space by performing SP1 on Model 2 (b) PM state space by performing SP2 on Model 2

Fig. 7 Sample PM state spaces for Model 2

4.2 Main Algorithm

An efficient search plan is generated by a dynamic programming based algorithm (see Algorithm 3), 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 $|a_{S_0}|_{\mathrm{F}}$ 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 $n = |a_{S_0}|_{\mathrm{F}}$ 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 and 9), and (2) updates the search plan, and the sequences of present extension, future extension, and future check operations in a delayed manner in `updateOperation` (line 12), but only if the next state S' passes the insert condition (line 11), which uses the set of backward reachable adornments represented by the precalculated characteristic function $\mathcal{A}_h$ and indices $\mathbf{a}$ and $\mathbf{c}$ for decision making, which latter are calculated by `determineIndices` (line 10). If the insert condition is fulfilled, 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 $SP_{T[0][1]}$ of the state stored in $T[0][1]$ (line 18).

4.3 Subroutines

The method `calculateNextState`(S, o) (presented in Algorithm 4) 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 (line 1). The new adornment $a_{S'}$ is calculated by applying operation o on the adornment a_S of state S (i.e., $a_S \overset{o}{\Rightarrow} a_{S'}$), which is carried out by the loop in lines 2–8, which (i) binds all free variables indicated by mask m_o of operation o (lines 3–4), and (ii) leaves the binding of all other variables unaltered (line 6). 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 (line 9). More specifically, the new product value $\pi_{S'}$ is calculated as $\pi_S w_o$, while the new cost $c_{S'}$ is determined as $c_S + \pi_{S'}$.

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 $T[i'][\mathbf{c}] = \texttt{null}$).

The calculation of indices $\mathbf{a}$ and $\mathbf{c}$ is carried out by procedure `determineIndices`$(T[i'], S')$ as presented in Algorithm 5. Indices $\mathbf{a}$ and $\mathbf{c}$ are initialized to $k+1$ and 1 (lines 1–2), respectively. States in column $T[i']$ are looked up in an increasing (search plan cost) order (lines 3–14). While executing the loop, index $\mathbf{a}$ is set exactly once to the running index j, when the stored state $T[i'][j]$ at position j has the same adornment as the new state S' (lines 5–7), while index

Algorithm 3 The procedure `calculateSearchPlan`$(S_0, k, \mathcal{A}_h)$

```
1: n := |a_{S_0}|_F // number of free variables in the initial state adornment a_{S_0} is calculated
2: T[n][1] := S_0
3: for (i := n down to 1) do
4:     for (j := 1 to k) do
5:         S := T[i][j] // current state S
6:         for all (o ∈ O_S^{pe}) do
7:             // for each present extension operation
8:             S' := calculateNextState(S, o) // next state S' is partially calculated
9:             i' := |a_{S'}|_F // 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_h, 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]}
```

Algorithm 4 The procedure `calculateNextState`(S, o)

```
1: SP_{S'} := ⟨SP_S, o⟩ // Operation o is appended to the operation sequence of search plan SP_S
2: for (p := 1 to |V|) do
3:     if (m_o[p] = F) then
4:         a_{S'}[p] := B // Binding information is switched from F to B, if the corresponding character of the mask m_o of operation o is F
5:     else
6:         a_{S'}[p] := a_S[p] // Binding information is left unchanged in other cases
7:     end if
8: end for
9: (c_{S'}, π_{S'}) := f(c_S, π_S, w_o) // Product and cost values are updated as follows π_{S'} := π_S w_o, c_{S'} := c_S + π_{S'}
10: return S'
```

c increases continuously until the first such index is found for which $c_{T[i'][j]} \leq c_{S'} < c_{T[i'][j+1]}$ (lines 8–10), and in this case $j+1$ will be the final value for index **c**. Indices **a** and **c** are returned prematurely when column $T[i']$ is not completely filled with stored states (i.e., slot $T[i'][j]$ is empty) (line 12), or regularly, when the loop terminates (line 15).

The `checkInsertCondition`$(T[i'], S', \mathcal{A}_h, \mathbf{a}, \mathbf{c})$ procedure (shown in Algorithm 6) makes a positive decision, if

(1) 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 adornment $a_{S'}$ of new state S' is backward reachable as determined by evaluating characteristic function $\mathcal{A}_h$ on adornment $a_{S'}$ (`isBackwardReachable`$(\mathcal{A}_h, a_{S'}) =$ `true`), or
(2) column $T[i']$ already stores a state $T[i'][\mathbf{a}]$ at location **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 $T[i'][\mathbf{a}]$ ($\mathbf{c} \leq \mathbf{a}$).

The method `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'.

On the implementation level, operation processing is carried out in two phases as presented side-by-side in Algorithm 7. In the first phase, only extension operations O_S^{pe} and O_S^{fe} of state S are considered (lines 1–15), while the second phase only deals with the corresponding check operations O_S^{fe} (lines 16–30). In order to iterate through extension operations in an increasing weight order, the sorted sequences of present extension O_S^{pe} and future extension O_S^{fe} operations have to be merged first (in line 2) by the well-known technique [8], which is also presented as procedure `merge`(O_S^{pe}, O_S^{fe}) (Algorithm 8) for completeness.

The rest of Algorithm 7 is similar for both phases. Operations are iterated in an increasing weight order (lines 4 and 19). If the referenced constraints of operation o^* and the selected operation o differ (i.e., $\mathsf{c}_{o^*} \neq \mathsf{c}_o$ in lines 5 and 20), then operation o^* is recategorized w.r.t adornment $a_{S'}$ by Algorithm 1 (lines 6 and 21) and further processed according to the following rules:

- **Discard past operations.** If operation o^* is a past operation, then it is discarded as it violates the general operation applicability condition (lines 13 and 28).
- **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 (lines 22–24).

The final publication is available at http://link.springer.com/article/10.1007%2Fs10270-013-0372-2

Algorithm 5 The procedure `determineIndices`$(T[i'], S')$

```
1: a := k + 1
2: c := 1
3: for (j := 1 to k) do
4:     if (T[i'][j] ≠ null) then
5:         if (a_{S'} = a_{T[i'][j]}) then // When the new state S' has the same adornment as the stored state T[i'][j] at position j
6:             a := j // Index a is set (exactly once) to the running index j
7:         end if
8:         if (c_{S'} ≥ c_{T[i'][j]}) then
9:             c := j + 1 // c is incremented, until the first such index is found, for which c_{T[i'][j]} ≤ c_{S'} < c_{T[i'][j+1]}
10:        end if
11:    else
12:        return (a, c) // Indices a and c are returned prematurely when slot T[i'][j] is empty
13:    end if
14: end for
15: return (a, c) // Indices a and c are returned regularly when the loop terminates
```

Algorithm 6 The procedure `checkInsertCondition`$(T[i'], S', \mathcal{A}_h, \mathbf{a}, \mathbf{c})$

```
1: return (a = k + 1 ∧ c < a ∧ isBackwardReachable(A_h, a_{S'})) ∨ (a < k + 1 ∧ c ≤ a)
                          (1)                                          (2)
```

Algorithm 7 The procedure `updateOperations`(S', S, o)

```
1:  // Extension operations
2:  O^e_S := merge(O^pe_S, O^fe_S)
3:  i'_pe := 1, i'_fe := 1
4:  for all (o* ∈ O^e_S) do
5:      if (c_{o*} ≠ c_o) then
6:          cat := categorize(o*, a_{S'})
7:          if (cat = PRESENT) then
8:              O^pe_{S'}[i'_pe] := o*
9:              i'_pe := i'_pe + 1
10:         else if (cat = FUTURE) then
11:             O^fe_{S'}[i'_fe] := o*
12:             i'_fe := i'_fe + 1
13:         end if // Operation o* is discarded, if cat = PAST
14:     end if // Operation o* is discarded, if c_{o*} = c_o
15: end for
16: // Check operations
17:
18: i'_fc := 1
19: for all (o* ∈ O^fc_S) do
20:     if (c_{o*} ≠ c_o) then
21:         cat := categorize(o*, a_{S'})
22:         if (cat = PRESENT) then
23:             SP_{S'} := ⟨SP_{S'}, o*⟩
24:
25:         else if (cat = FUTURE) then
26:             O^fc_{S'}[i'_fc] := o*
27:             i'_fc := i'_fc + 1
28:         end if // Operation o* is discarded, if cat = PAST
29:     end if // Operation o* is discarded, if c_{o*} = c_o
30: end for
```

Algorithm 8 The procedure `merge`(O^{pe}_S, O^{fe}_S)

```
1:  i_pe := 1, i_fe := 1, i_e := 1
2:  while (i_pe ≤ |O^pe_S| ∨ i_fe ≤ |O^fe_S|) do
3:      if (i_fe > |O^fe_S| ∨ O^pe_S[i_pe] < O^fe_S[i_fe]) then
4:          O^e_S[i_e] := O^pe_S[i_pe]
5:          i_pe := i_pe + 1
6:      else
7:          O^e_S[i_e] := O^fe_S[i_fe]
8:          i_fe := i_fe + 1
9:      end if
10:     i_e := i_e + 1
11: end while
12: return O^e_S
```

- **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 w.r.t. adornment $a_{S'}$, then it is appended to the corresponding operation sequence $O^{pe}_{S'}$ (lines 7–9), $O^{fe}_{S'}$ (lines 10–12), or $O^{fc}_{S'}$ (lines 25–27) of state S', respectively.

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 (lines 14 and 29).

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 Algorithm 3.

The procedure `insert`$(T[i'], S', \mathbf{a}, \mathbf{c})$ (Algorithm 9) determines $m = \min\{\mathbf{a}, k\}$ (line 1), shifts elements between $T[i']$[$\mathbf{c}$] and $T[i'][m-1]$ downward, which wipes state $T[i'][m]$ out (lines 2–4), and inserts state S' at position $\mathbf{c}$ (line 5).

4.4 Complexity Analysis of the Search Plan Generation Algorithm

The worst case runtime complexity of Algorithm 3 can be calculated as follows.

The number of free variables in state adornments (lines 1 and 9) can be determined in $|V|$ steps. Fields of table T can be accessed (lines 2 and 5) in constant time. As $n \leq |V|$ and k is a constant parameter of the algorithm, the body of the two outermost cycles (lines 5–15) is executed $\mathcal{O}(|V|)$ times. The body of the innermost cycle (lines 7–14) is performed at most $|O|$ times. As a summary, Algorithm 3 has

$$\mathcal{O}(|V| + 1 + |V| \cdot 1 \cdot \underbrace{(1 + |O| \cdot I)}_{\text{lines 5–15}}) = \mathcal{O}(|V| \cdot |O| \cdot I),$$

runtime complexity, where I denotes the number of steps needed for one execution of the innermost cycle.

The procedure `calculateNextState`(S, o) (Alg. 4) appends operation o to search plan SP_S (line 1) in one step. Adornment $a_{S'}$ is calculated in $|V|$ steps (lines 2–8). Cost calculation (line 9) can be considered a constant time activity. The complexity of procedure `determineIndices`$(T[i'], S')$ (Algorithm 5) is $\mathcal{O}(|V|)$ as (i) its cycle is executed k (constant) times, (ii) the adornment equality check in line 5 requires $|V|$ steps, (iii) while all other comparisons and assignments are single step activities. The procedure `checkInsertCondition`$(T[i'], S', \mathbf{a}, \mathbf{c})$ (Algorithm 6) consists of constant time comparisons and Boolean operations except for the invocation of the `isBackwardReachable`$(\mathcal{A}_h, a_{S'})$ method, which fills in each Boolean variable of the precompiled characteristic function $\mathcal{A}_h$ in $|V|$ steps. In procedure `updateOperations`(S', S, o) (Algorithm 7), (i) the merge of present and future extension operations (Algorithm 8) can be carried out in $|O|$ steps, (ii) at most $|O|$ operations are (re)categorized, (iii) each categorization (i.e., each invocation of `categorize`$(o^*, a_{S'})$) is performed in $|V|$ steps, as operation mask m_{o^*} and adornment $a_{S'}$ must be compared at each variable position, and (iv) all other activities can be executed in constant time. Finally, the `insert`$(T[i'], S', \mathbf{a}, \mathbf{c})$ procedure (Algorithm 9) performs a constant number of single step rearrangements. Thus, $I = \mathcal{O}(|V|+|V|+|V|+|V|\cdot|O|+1) = \mathcal{O}(|V| \cdot |O|)$, which results in an overall $\mathcal{O}(|V|^2 \cdot |O|^2)$ runtime complexity for the search plan generation algorithm.

4.5 Running Example for the Algorithm

The execution of our algorithm on Model 2 with initial adornment `BFFFF` and parameter $k = 2$ is illustrated in Figures 8 to 11, which present the contents of table T (in the corresponding dashed boxes) after the innermost cycle (lines 6–15) of Algorithm 3 has been executed on the states stored in $T[3][1]$, $T[2][1]$, $T[2][2]$, and $T[0][1]$, respectively.

Notational guide. Figures 8 to 11 use a common notation for states, whose visual appearance is summarized in the bottom left corner of Fig. 8 as a notational guide. The headers of states have light grey backgrounds. The first line of the state headers contains the cost (typeset in bold), the adornment, and the product value of the search plan whose operations are enumerated in the bottom part of the header. The rest of the state description (already with a white background) lists sequences of present extension, future extension and future check operations in this specific order. These sequences are separated from each other by thin black lines, and each sequence is sorted based on the operation weights, which appear as numbers in the rightmost column. The 3 columns on the left present the constraint, the adornment and the mask of the operations, respectively.

Each arrow represents the derivation of a new state produced by one execution of the innermost cycle (lines 6–15). Within the range of Figures 8 to 11, solid edges mark those derivations that are new compared to the previous figure (i.e., the delta), while derivations denoted by dashed edges already appeared in one of the former figures.

States with watermark letter A were temporarily stored in the table (but later discarded due to the appearance of better states). The state with letter B failed the backward reachability analysis test, while states with watermark C were discarded as the corresponding column had already contained a better state with the same adornment.

Initialization. The initial adornment `BFFFF` has 4 free parameters, consequently, the initial state is stored at $T[4][1]$ (see lines 1–2 of Algorithm 3). The initial state has 0 as its cost, `BFFFF` as its adornment, 1 as its product value, and an empty search plan. The operations in the initial state are categorized with respect to the adornment `BFFFF`, which process was already presented in Fig. 3. However, in contrast to Fig. 3, the representation used in Figs. 8 to 11 omits the past operations (e.g., `defines(RO,SWP)` adorned by `FB`), and lists operations in a category grouped and weight sorted manner. For instance, operation `hasSensors(RO,IDX,SE)`

Algorithm 9 The procedure `insert`$(T[i'], S', \mathbf{a}, \mathbf{c})$

1: $m := \min\{\mathbf{a}, k\}$
2: **for** ($j := m - 1$ down to $\mathbf{c}$) **do**
3: $T[i'][j+1] := T[i'][j]$ // Elements between $T[i'][\mathbf{c}]$ and $T[i'][m-1]$ are shifted downward and state $T[i'][m]$ is discarded
4: **end for**
5: $T[i'][\mathbf{c}] := S'$ // State S' is inserted at position $\mathbf{c}$

with adornment `BFF`, which is the best present extension operation (with respect to state adornment `BFFFF`), has weight $\frac{\#\texttt{hasSensors}}{\#\texttt{Route}} = \frac{2}{1} = 2$ as Model 2 has 2 `hasSensors` links, and 1 `Route`.

Processing states in columns $T[4]$ and $T[3]$ (Figure 8). The first execution of the innermost cycle (lines 6–15) of Algorithm 3 processes the best present extension operation (i.e., `hasSensors(RO,IDX,SE)` with adornment `BFF`) of state stored in $T[4][1]$. The corresponding new state is inserted into $T[2][1]$ as its adornment `BBBFF` has 2 free variables, The positive decision on insertion is made according to case (1) of Algorithm 6 as column $T[2]$ is empty at this time ($1 = \mathbf{c} < \mathbf{a} = k + 1 = 3$), and adornment `BBBFF` is backward reachable ($\mathcal{A}_3(\varphi(\texttt{B}), \varphi(\texttt{B}), \varphi(\texttt{B}), \varphi(\texttt{F}), \varphi(\texttt{F})) =$ `true`). In the new state, both the cost and the product value are 2, operations originating from the constraint `hasSensors(RO,IDX,SE)` are discarded, and all other operations are recategorized w.r.t. adornment `BBBFF`. The next iteration processes the other present extension operation (`defines(RO,SWP)` with adornment `BF`) of state stored in $T[4][1]$ producing the new state appearing in $T[3][1]$. As the state in $T[4][1]$ has no more present extension operations and $T[4][2]$ is empty, Algorithm 3 continues with column $T[3]$ creating two new states with the same process and inserting them into $T[2][2]$ and $T[1][1]$, respectively, resulting in the table contents shown in Fig. 8.

Processing the state stored in $T[2][1]$ (Figure 9). When the innermost cycle of Algorithm 3 is carried out on the two present extension operations of state stored in $T[2][1]$, the previous content of $T[1][1]$ (i.e., in Fig. 8) is pushed downwards by the first newly created state (shown in $T[1][1]$ in Fig. 9), and then replaced by the second newly created state (shown in $T[1][2]$ in Fig. 9). As a consequence, the previous content of $T[1][1]$ has been wiped out, which is denoted in Fig. 9 in such a manner that the discarded state now appears below the table with watermark A.

The insertion of the first newly created state is based on case (1) of Algorithm 6 as the adornment `BBBBF` of the new state is backward reachable and the new state has a smaller cost ($\mathbf{c} = 1$) and a different adornment ($\mathbf{a} = k + 1$). The corresponding state rearrangements in the table are carried out by one execution of line 3 of Algorithm 9 with $j = 1$. The insertion of the second newly created state is prescribed by case (2) of Algorithm 6 as the table already contains a worse state with the same adornment ($2 = \mathbf{c} = \mathbf{a} < k + 1 = 3$). Note that in this situation, Algorithm 9 simply overwrites the contents of $T[1][2]$ with the new state (line 5).

Processing the state stored in $T[2][2]$ (Figure 10). When the present extension operation `observes(SE,SW)` with adornment `FB` of state stored in $T[2][2]$ is processed, a new (partially calculated) state with adornment `BFBBB` is created and discarded due to the fact that adornment `BFBBB` is not backward reachable as $\mathcal{A}_3(\varphi(\texttt{B}), \varphi(\texttt{F}), \varphi(\texttt{B}), \varphi(\texttt{B}), \varphi(\texttt{B}))$ is evaluated to `false`. The handling of present extension operation `hasSensors(RO,IDX,SE)` with adornment `BFF` results in a new state being inserted into $T[0][1]$ as shown in Fig. 10. Note that operation `observes(SE,SW)` with adornment `BB` has been recategorized (w.r.t. the final adornment `BBBBB`) in the new state to a present check operation, which is immediately appended to the corresponding search plan (see the fourth search plan line in $T[0][1]$) as prescribed by line 23 of Algorithm 7.

Processing states in column $T[1]$ (Figure 11). As shown in Fig. 11, the first of the last 4 iterations of the innermost cycle replaces the state with cost 6 in $T[0][1]$ (according to Fig. 10) by a new state having the same adornment ($\mathbf{a} = 1$) and a smaller cost ($\mathbf{c} = 1$) as prescribed by case (2) in Algorithm 6 ($1 = \mathbf{c} = \mathbf{a} < k + 1 = 3$). The remaining 3 iterations produce partially calculated states, which are discarded as the table already contains a better state (in $T[0][1]$) with the same adornment.

5 Quantitative Evaluation

In this section, we quantitatively assess and evaluate the effects of different cost models and various configurations of our proposed search plan generation algorithm on the runtime performance of the pattern matching process.

5.1 Comparison with the Domain-Specific Approach

In the first scenario, the performance effects of using different cost models in the same search plan generation algorithm were analyzed.

Experimental environment. Our model-sensitive (MS) cost model was compared to a domain-specific (DS) one, 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 [1] 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`.

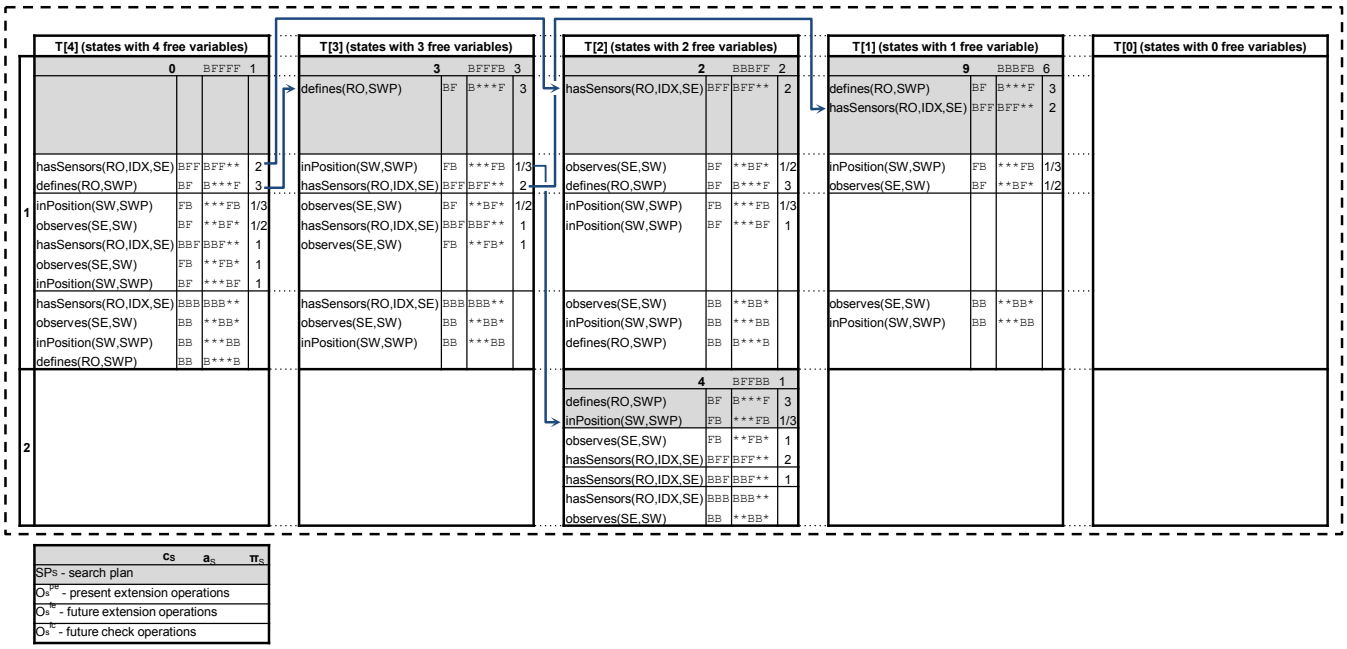


Fig. 8 Contents of table T after processing states in columns $T[4]$ and $T[3]$ when Algorithm 3 is executed on Model 2 with $k = 2$

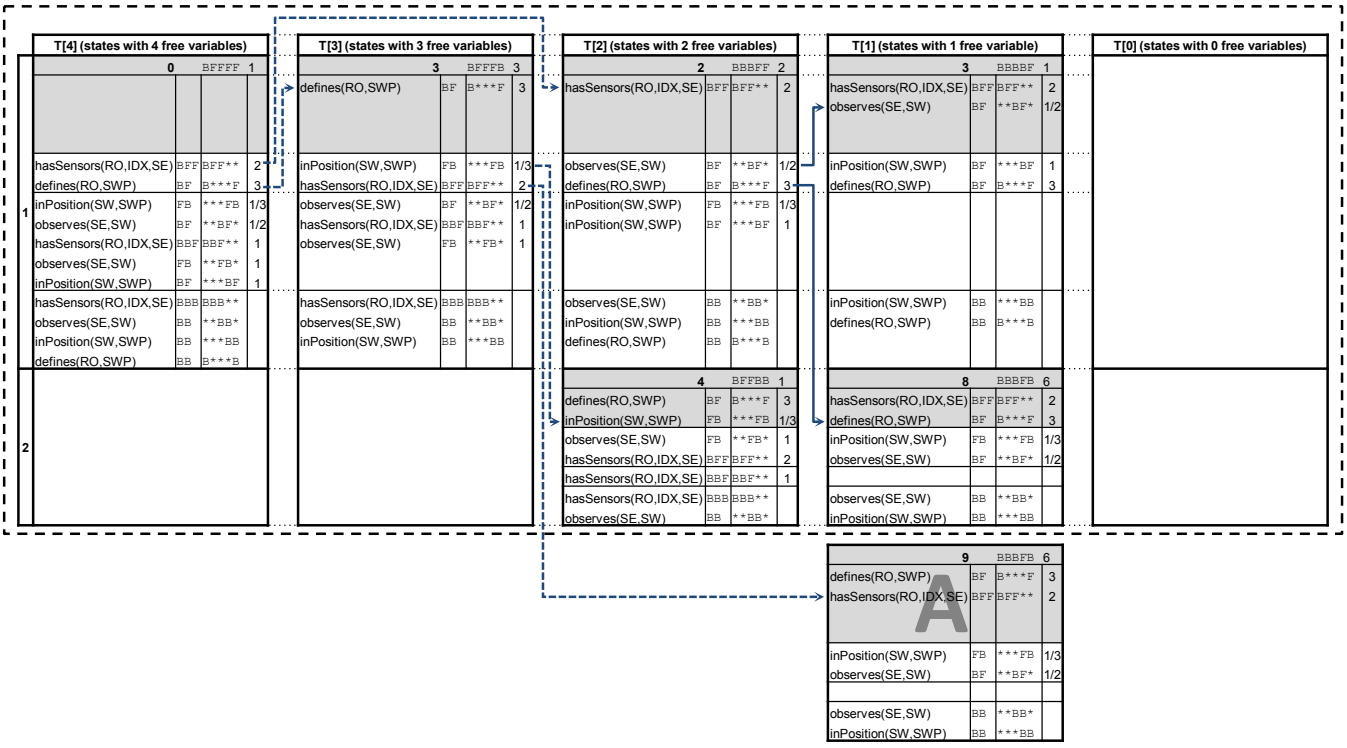


Fig. 9 Table contents and discarded states after processing the state stored in $T[2][1]$

Quantitative results. Figure 12(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 `Routes` in the model, respectively. The remaining columns show the measured values for the different configurations, which independently involve DS and 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 `Routes` 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.

Evaluation. Fig. 12(a) shows that model-sensitive search plans can clearly outperform domain-specific ones (in this case on all test models by nearly 400 steps in average) when

The final publication is available at http://link.springer.com/article/10.1007%2Fs10270-013-0372-2

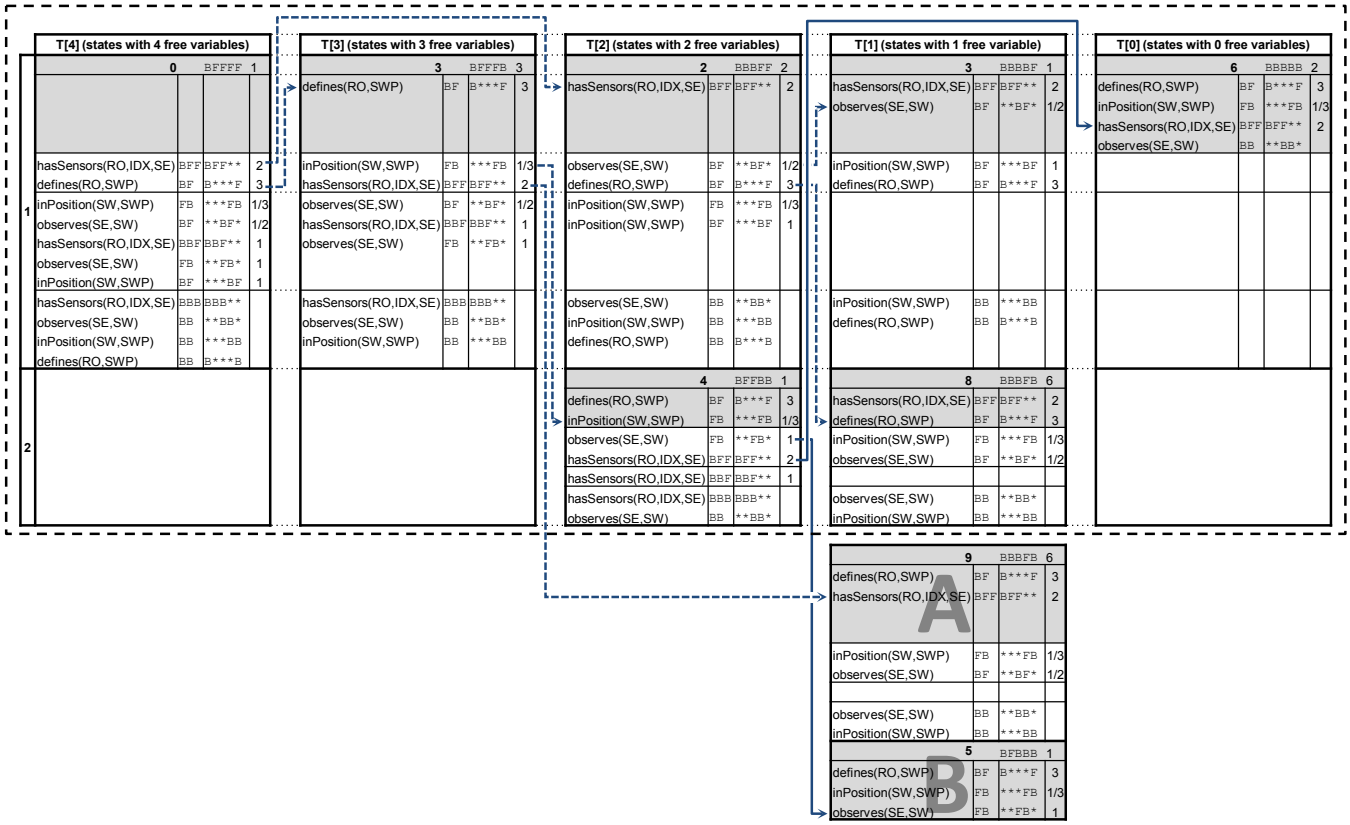


Fig. 10 Table contents and discarded states after processing the state stored in $T[2][2]$

the pattern has many structural feature constraints with arbitrary multiplicity. Our algorithm generated the very same search plan for the settings of the fifth and the seventh column, which explains the equal values there. In Fig. 12(b) the *relative frequency* distribution histogram of the PM state differences of DS and MS approaches (with parameter $k = 2$) is depicted for the case 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. 12(a)). Fig. 12(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 presumed 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 a further general characteristic, a single PM step took 51 ns in average.[3]

[3] The runtime value is an average of 20 wall clock time measurements performed on a computer with 1.57 GHz Intel Core2 Duo CPU and 2.96 GB RAM. Windows XP Professional SP 3 and Java 1.7 served as the underlying operating system and virtual machine, respectively.

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.

5.2 Comparison with the Graph-Based Model-Sensitive Approach

In the second scenario, the performance effects of using different search plan generation algorithms have been quantitatively analyzed in a common, model-sensitive setup.

Measurement setup. Our new dynamic programming algorithm (DP) with parameter settings $k = 1$ and $k = 2$ was compared to the graph-based approach (G) that was employed by the GrGen [12] and Viatra [34] model transformation tools.

In order to clearly focus on the capabilities of the search plan algorithms themselves and on their effect on the runtime performance of pattern matching in a modeling layer, programming language and virtual machine independent manner, we decided not to use GrGen and Viatra in our measurements, but to reimplement the graph-based search plan generation approach in our pattern matching engine and to employ this implementation in the performance analysis.

As the graph-based search plan algorithm cannot support general n-ary constraints, the pattern `routeSensor` of Fig. 2 was simplified by removing variable `IDX` and replacing the ternary constraint `hasSensors(RO,IDX,SE)`

The final publication is available at http://link.springer.com/article/10.1007%2Fs10270-013-0372-2

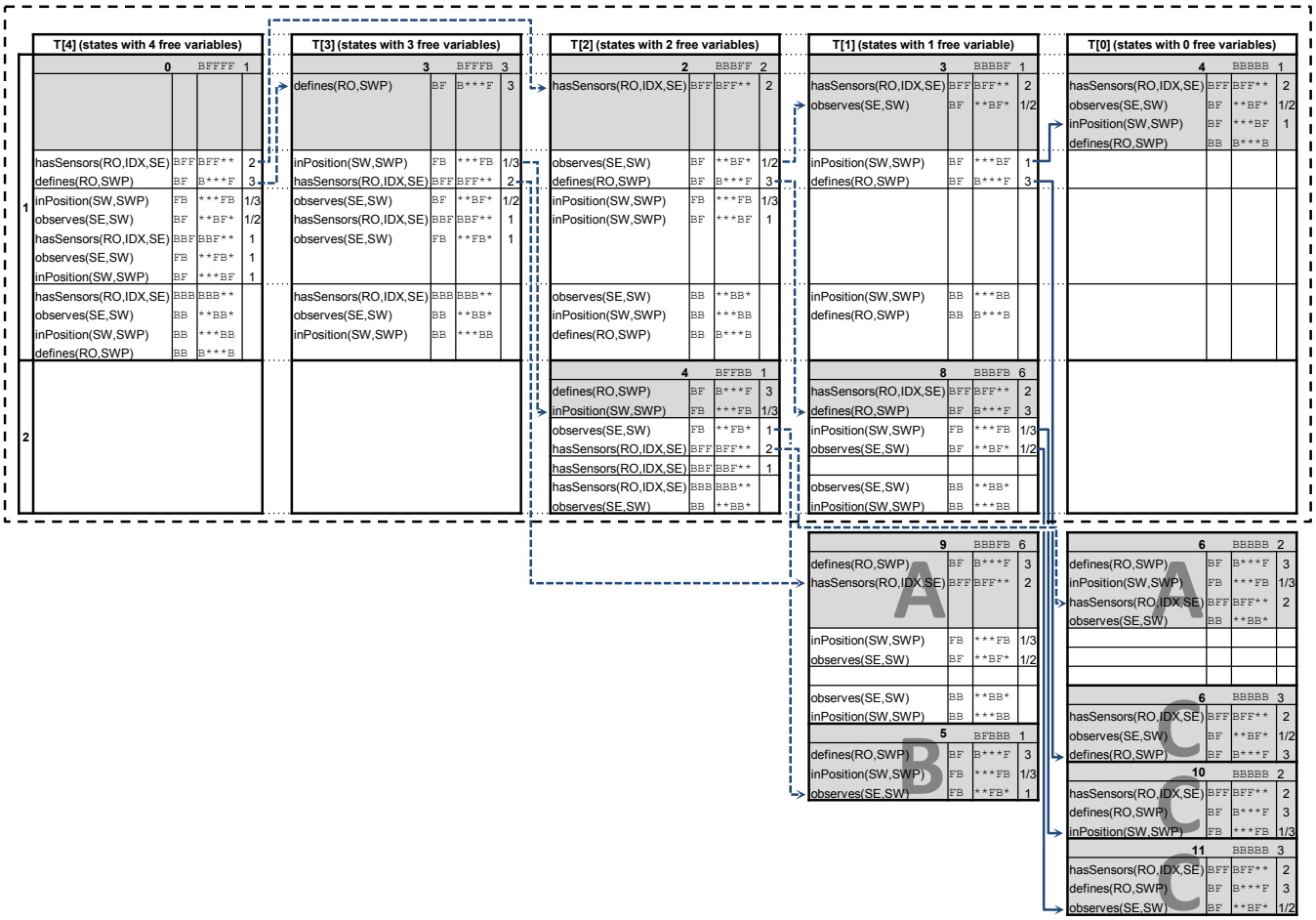
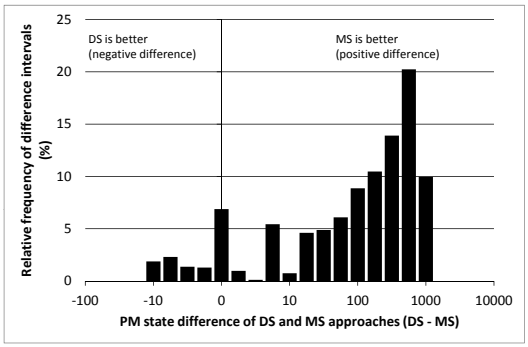


Fig. 11 Table contents and discarded states, when the execution of Algorithm 3 on Model 2 with $k = 2$ is completed

	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. 12 Measurement results for the comparison of the domain-specific and the model-sensitive approaches

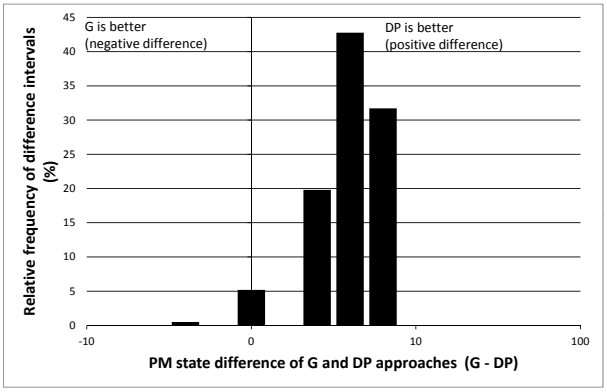
with the binary constraint `hasSensors(RO,SE)`. In this setup, only a single model with 53760 model elements (more specifically, 2560 `Routes`, 10240 `Switches`, 30720 `Sensors`, and 10240 `SwitchPositions`) was used. However, the pattern matching was executed with 3 different initial adornments. In each of the 3 cases, only a single variable (namely, `RO`, `SW`, and `SWP`) was initially bound. Similarly to Sec. 5.1, the complete process (including search plan generation) was repeated on each distinct `Route`, `Switch`, and `SwitchPosition`, respectively.

Quantitative results. Figure 13(a) presents the data measured for the comparison of model-sensitive search plan generation techniques. The first column shows the type of those objects that were designated as starting points in the measurements, while the second column indicates how many such objects existed in the model. The remaining columns present the measured values for the graph-based algorithm (G) and

our dynamic programming based approach (DP), which latter used parameter values $k = 1$ and 2. The SP columns show the average wall clock time[3] needed for the search plan generation. The PM columns again denote the number of PM states (i.e., elementary pattern matching steps), which was averaged over all distinct starting points.

Starting point type	Starting point #	G		DP (k=1)		DP (k=2)	
		SP ms	PM #	SP ms	PM #	SP ms	PM #
Route	2560	3.683	18.0	5.372	18.0	5.434	12.0
Switch	10240	1.041	4.5	5.141	4.5	5.148	4.5
SwitchPosition	10240	0.978	4.5	5.063	4.5	5.250	4.5

(a) Comparison of PM state spaces



(b) PM state difference profile

Fig. 13 Measurement results for the comparison of the graph based and the dynamic programming based algorithms

Evaluation. Figure 13(a) shows that our dynamic programming based approach generates a search plan that is at least as good as the one provided by the graph based algorithm (G). In the case highlighted by the thick frames in Fig. 13(a), the dynamic programming based technique with parameter $k = 2$ produced a 33% speed-up in average. This acceleration effect can be further analyzed using Fig. 13(b), which depicts the *relative frequency* distribution histogram of the PM state differences of G and DP approaches (with parameter $k = 2$) when these differences are calculated on a route-by-route basis for each of the 2560 starting points. Fig. 13(b) shows that the G approach was better by 2 to 4 steps only in 0.508% of the 2560 cases (first column to the left of the vertical axis), the search plan produced by our DP algorithm was faster in 94.297% of the cases (columns to the right of the vertical axis), while a draw occured in 5.195% of the cases (column on the vertical axis).

The DP algorithm needs 50% to 400% more time than the graph based approach to generate the search plan itself. For a fair evaluation, it should be emphasized that the search plan generation is still on the millisecond scale in both cases, and it is executed only once while pattern matching is carried out several times on large models. Additionally, the DP algorithm can easily handle arbitrary cost functions and non-binary constraints in an integrated manner, which are missing features in the graph based approach.

An interesting observation can be made when examining the effect of increasing the parameter value of the DP algorithm from 1 to 2. Although the size of table T is doubled, the increase in the time needed for search plan generation is relatively small, which might be explained by the costliness of the reachability analysis. As shown by Fig. 13(a), the DP approach with parameter $k = 1$ produces exactly the same search plan (and measurement results) as the graph based technique. Based on our preliminary analysis, we suspect that these algorithms always generate the same result, however, this statement is still to be proven in the future.

5.3 Limitations of the Quantitative Evaluation

The presented performance evaluation has a number of limitations. The most important and obvious restriction is that the quantitative assessment of search plan generation and pattern matching algorithms can never be complete due to the fact that these are heuristic-based approaches, whose performance can be measured, analyzed and compared in several different scenarios, but the overall best technique cannot be determined in a formally proven manner (e.g., by complexity analysis).

In this paper, our measurement setups have been limited to (i) a single metamodel, (ii) one pattern having a small size and a fixed arrangement, (iii) structurally similar models, and (iv) a restricted starting point selection for the pattern matching process.

Another unexplored issue is the quantitative comparison of our model-sensitive pattern matching technique to the Ullmann [30] and VF2 [7] algorithms. In [10], these well-known recursive backtracking algorithms were quantitatively compared to each other with a clear statement that the VF2 approach outperformed the Ullmann algorithm. As shown in [31], both the VF2 approach and our search plan based pattern matching technique traverse a state space in a conceptually similar manner. However, while the VF2 algorithm employs a larger lookahead in the syntactic and semantic feasibility rules, our approach filters the candidate pairs already in an earlier phase by considering domain-specific restrictions, which exploit the metamodel conformance of the underlying models.

6 Related Work

This section surveys those related approaches that carry out tasks that are similar to search plan generation. In this context, query optimizers in different domains are investigated first, which are followed by an overview on search plan driven pattern matching techniques used in model transformation tools.

6.1 Query Optimization

From a methodological aspect, query optimization and search plan generation aim to answer the same question, namely, how to prepare search plans (without their actual execution) in advance that can be efficiently carried out later on. For an efficient plan execution, the search space to be traversed must be as small as possible, or equivalently, the tree representing the explorable search space should be narrow at the top (i.e., near to its root). A tree of such a shape can be achieved by applying the well-justified principle to rank operations in a decreasing order according to their selectivity starting with the activity that filters out the most disallowed solutions. A key point in this process is that estimating operation selectivity is a non-trivial task even if it is based on model statistics that can be collected at various levels of granularity in the different domains.

Optimization of relational database queries. In relational databases, the table contents can be used for query optimization purposes to estimate join selectivity as already suggested by [27] decades ago. When comparing relational database approaches to model-based techniques, the gap in the abstraction level of the different data models, which is frequently bridged by an object-relational mapping [25], has to be emphasized. Consequently, the operations in the query optimization process have differing granularities. For instance, the number of links of a certain reference type defined in a superclass is immediately available in an object-oriented setup, while a query optimizer in a relational database cannot directly access this information as it is typically stored in another table. A further general characteristic difference is that relational database management systems store the table contents on disk, while our approach operates on models in the main memory.

A graph based algorithm is proposed in [19] to prepare execution plans that describe the order in which joins in an SQL query have to be performed. The algorithm calculates a directed spanning tree similarly to [12,34], and it produces a near optimal execution plan in polynomial time. The cost function in our approach uses only the estimated number of intermediate results as an optimization criterion, in contrast to [19], which additionally considers the size of the tuples in the cost calculations. Note that such a kind of extension could be easily integrated into our algorithm in the future.

Another sophisticated join optimizer is presented in [21], which uses singleton sets to represent tables participating in a join, and combines these sets in all possible ways by a dynamic programming algorithm. This technique always produces an optimal query execution plan by exploring 2^n combinations, which takes at least 7 minutes for complex joins on 20 tables, in contrast to our approach, which traverses only a limited search space if the parameter k is set to a small constant value to be able to provide good search plans in a practially reasonable amount of time. Moreover, our dynamic programming algorithm is able to handle search plan validity restrictions, e.g., caused by unidirectionally navigable references, which are non-existing setups in join opitmization.

Optimization of graph queries from the semantic web domain. In the semantic web domain, SPARQL queries can be used to formulate graph patterns. Finding the optimal join order in such queries can be interpreted as a search plan generation task. The query engines that support the SPARQL language can operate either on disk-based or on in-memory models. In the first case, the join order is determined by the underlying relational database yielding to solutions with the already mentioned advantages and disadvantages. In the other case, the in-memory models are persisted in a triple (or RDF) store, which employs a generic data model representing domain concepts in a flat (or unstructured) manner as triples that use string labels as identifiers. In contrast, a model-based approach can easily describe the same concepts in a structured manner. For example, links that are instances of a reference type in a superclass are immediately accessible in an EMF model, while looking up the same information already requires a sequence of joins to perform on a triple store. Based on these observations and on the corresponding statements of [18], more joins have to be carried out during the execution of a SPARQL query in comparison with the number of operations and joins performed for a similar graph and SQL query in a model-based pattern matcher and a query engine of a relational database, respectively.

While our approach generates a separate search plan for each pattern, the technique proposed in [18] recognizes common substructures in multiple SPARQL queries, and rewrites these queries into an equivalent form that is more efficiently executable. In order to rank joins according to their selectivity, the estimations [28] are based on precomputed statistics from the dataset in the triple store. More specifically, selectivity is represented by a number from the $[0, 1]$ interval. An unbound variable does not provide any filtering which is marked by the corresponding 1 value, while the selectivity of a bound variable depends on whether the variable plays the subject, object, or predicate role in a triple. The selectivity of a triple is defined by the product of the corresponding selectivity values calculated for the 3 variables. The experiences in [28] show that estimations based on this technique become inaccurate if the number of joins increases, which is a typical scenario in case of SPARQL queries.

To improve the precision of the selectivity estimations, which have a highly critical role in optimization of SPARQL queries, [22] introduces a selectivity measure for pairwisely joined triples, which corresponds to the situation if weights for operation sequences of length 2 were established in our approach. In this context, [22] collects statistics about triples, all of their pre-aggregated binary and unary projections, and about all triple combinations, which might be a feasible technique for flat data models, but not necessarily in a model-based setup, which has to handle hierarchical data structures in a dynamically changing environment.

Query optimization in object-oriented databases. As [23] gives an excellent survey on query processing in object-oriented database management systems (OODBMS), we focus only on the most important similarities and differences between graph pattern matching and object-oriented queries.

The first observation is that both domains use the object-oriented data model. Queries in an OODBMS are defined by object algebraic expressions, which constitute a richer language in comparison with the basic graph pattern formalism that excludes advanced pattern composition constructs [3].

Search plan generation in object-oriented databases is a complex tree transformation that processes object algebraic expressions and produces operation trees. In this method, either algebraic or path expression based techniques are applied for optimization purposes. In algebraic optimizers, which manipulates the abstract syntax tree representation of the algebraic expressions, an exhaustive search is carried out that explores and enumerates the entire search space, in contrast to our approach. Techniques using path expressions embed model navigations into the where clause of the operation tree. In this context, determining an optimal sequence of model navigations could be interpreted as a search plan generation procedure from the graph pattern matching domain, however, query optimizers in object-oriented databases cannot reverse navigations along bidirectional references.

6.2 Search Plan Driven Pattern Matching in Model Transformation Tools

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 [32].

Search plan driven pattern matchers. Fujaba [11] 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 with model-sensitive search plans. Although Fujaba [13] is a model-sensitive approach and runs on EMF models, it has only a simple greedy strategy to control pattern matching. GrGen [12] and Viatra [34], which employ model-sensitive search plans, operate on a non-standard modeling layer, which has a number of 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 paragraphs.

Analysis of model-sensitive search plan generation algorithms. In contrast to our dynamic programming search plan generation algorithm, GrGen 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 [14]. 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 weights of operations 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 [4], but not a provable fact.[4]

7 Conclusion

In this paper, we proposed a novel search plan generation algorithm based on dynamic programming together with a model-sensitive cost function to speed up pattern matching in practice. Although the approach was exemplified with a cost function defined for EMF models, the technique itself is independent from the underlying modeling layer and it can be used with an arbitrary (iteratively computable) cost function. Our comprehensive description presented the main algorithm, its precompilation steps and all the subprocedures. Additionally, the search plan generation algorithm and its runtime effects on the pattern matching engine have been evaluated by complexity analysis techniques and by (hardware and JVM independent) quantitative performance measurements, respectively.

Our future tasks are to repeat measurements in additional scenarios, and to embed the pattern matching framework into different modeling tools.

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

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. Efficient instance-level ontology validation by incremental model query techniques. http://viatra.inf.mit.bme.hu/publications/trainbenchmark, accessed: 15/10/2012
2. 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, vol. 49 (Mar 2012)
3. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: Proc. of the 21st ACM Symposium on Applied Computing. pp. 1280–1287. ACM Press, Dijon, France (April 2006)
4. 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. LNCS, vol. 5088, pp. 471–486. Springer (2008)
5. Buchmann, T., Westfechtel, B., Winetzhammer, S.: The added value of programmed graph transformations – a case study from software configuration management. In: Schürr, A., Varró, D., Varró, G. (eds.) Proc. of the 4th International Symposium on Applications and Graph Transformations with Industrial Relevance. LNCS, vol. 7233, pp. 198–209. Springer, Budapest, Hungary (Oct 2012)
6. Byröd, M., Lennartson, B., Vahidi, A., Åkesson, K.: Efficient reachability analysis on modular discrete-event systems using binary decision diagrams. In: Proc. of the 8th International Workshop on Discrete Event Systems. pp. 288–293. IEEE (2006)
7. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: An improved algorithm for matching large graphs. In: Proc. of the 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition. pp. 149–159 (May 2001)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 3rd edn. (Sep 2009)
9. 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. LNCS, vol. 1764, pp. 296–309. Springer (1998)
10. Foggia, P., Sansone, C., Vento, M.: A performance comparison of five algorithms for graph isomorphism. In: Proc. of the 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition. pp. 188–199 (May 2001)
11. Geiger, L., Schneider, C., Reckord, C.: Template- and model-based code generation for MDA-tools. In: Giese, H., Zündorf, A. (eds.) Proc. of the 3rd International Fujaba Days. pp. 57–62 (2005), ftp://ftp.upb.de/doc/techreports/Informatik/tr-ri-05-259.pdf
12. 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. LNCS, vol. 4178, pp. 383–397. Springer (2006)
13. 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. ECEASST, vol. 18 (2009)
14. Horváth, Á., Varró, G., Varró, D.: Generic search plans for matching advanced graph patterns. In: Ehrig, K., Giese, H. (eds.) Proc. of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques. Electronic Communications of the EASST, vol. 6. Braga, Portugal (Mar 2007)
15. 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. LNCS, vol. 3844, pp. 128–138. Springer (2005)
16. Lambers, L., Hildebrandt, S., Giese, H., Orejas, F.: Attribute handling for bidirectional model transformations: The triple graph grammar case. In: Hermann, F., Voigtländer, J. (eds.) Proc. of the 1st International Workshop on Bidirectional Transformations. Electronic Communications of the EASST, vol. 49. Tallinn, Estonia (Mar 2012)
17. Lauder, M.: Incremental Model Synchronization with Precedence-Driven Triple Graph Grammars. Ph.D. thesis, Technische Universität Darmstadt (Feb 2013)
18. Le, W., Kementsietsidis, A., Duan, S., Li, F.: Scalable multi-query optimization for SPARQL. In: Gehrke, J., Ooi, B.C., Pitoura, E. (eds.) Proc. of the 2012 IEEE 28th International Conference on Data Engineering. pp. 666–677. IEEE Computer Society, Arlington, Virginia, USA (2012)
19. Lee, C., Shih, C.S., Chen, Y.H.: Optimizing large join queries using a graph-based approach. IEEE Transactions on Knowledge and Data Engineering 13(2), 298–315 (2001)
20. Meinel, C., Theobald, T.: Algorithms and Data Structures in VLSI Design: OBDD Foundations and Applications. Springer (1998)
21. Moerkotte, G., Neumann, T.: Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In: Proc. of the 32nd International Conference on Very Large Data Bases. pp. 930–941. VLDB Endowment, Seoul, Korea (2006)
22. Neumann, T., Weikum, G.: Scalable join processing on very large RDF graphs. In: Binnig, C., Dageville, B. (eds.) Proc. of the 2009 ACM SIGMOD International Conference on Management of Data. pp. 627–640. ACM, Providence, Rhode Island, USA (2009)
23. Özsu, M.T., Blakeley, J.A.: Query processing in object-oriented database systems. ACM Transactions on Information Systems 8, 387–430 (1994)
24. 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. LNCS, vol. 3062, pp. 479–485. Springer (2004)
25. Roebuck, K.: Object-Relational Mapping: High-Impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors. Lightning Source Incorporated (2011)
26. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1: Foundations. World Scientific (1997)

27. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Bernstein, P.A. (ed.) Proc. of the 1979 ACM SIGMOD International Conference on Management of Data. pp. 23–34. ACM, Boston, Massachusetts, USA (1979)
28. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL basic graph pattern optimization using selectivity estimation. In: Ma, W.Y., Tomkins, A., Zhang, X. (eds.) Proc. of the 17th International Conference on World Wide Web. pp. 595–604. ACM, Beijing, China (Apr 2008)
29. The MOGENTES project: `http://www.mogentes.eu/`
30. Ullmann, J.R.: An algorithm for subgraph isomorphism. Journal of the ACM 23(1), 31–42 (Jan 1976)
31. Varró, G.: Advanced Techniques for the Implementation of Model Transformation Systems. Ph.D. thesis, Budapest University of Technology and Economics (Apr 2008)
32. Varró, G., Anjorin, A., Schürr, A.: Unification of compiled and interpreter-based pattern matching techniques. In: Tolvanen, J.P., Vallecillo, A. (eds.) Proc. of the 8th European Conference on Modelling Foundations and Applications. Lecture Notes in Computer Science, vol. 7349, pp. 368–383. Springer, Lyngby, Denmark (Jul 2012)
33. 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.) Proc. of the 5th International Conference on Model Transformation. Lecture Notes in Computer Science, vol. 7307, pp. 224–239. Springer, Prague, Czech Republic (May 2012)
34. 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. Electronic Notes in Theoretical Computer Science, vol. 152, pp. 191–205. Elsevier, Tallinn, Estonia (Sep 2005)
35. Wickes, W.E.: Logic Design with Integrated Circuits. John Wiley & Sons, Inc. (1968)
36. 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. LNCS, vol. 1073, pp. 454–468. Springer (1996)