Automated Software Engineering Journal manuscript No.
(will be inserted by the editor)

A model-driven framework for guided design space exploration

Ábel Hegedüs, Ákos Horváth, and Dániel Varró

Received: date / Accepted: date

Abstract Design space exploration (DSE) aims at searching through various models representing different design candidates to support activities like configuration design of critical systems or automated maintenance of IT systems. In model-driven engineering, DSE is applied to find instance models that are (i) reachable from an initial model with a sequence of transformation rules and (ii) satisfy a set of structural and numerical constraints. Since exhaustive exploration of the design space is infeasible for large models, the traversal is often guided by hints, derived by system analysis, to prioritize the next states to traverse (selection criteria) and to avoid searching unpromising states (cut-off criteria). In this paper, we define an exploration approach where selection and cut-off criteria are defined using dependency analysis and algebraic abstraction of transformation rules. Additionally, we apply different state encoding techniques to identify recurring states and reduce the number of visited states. Finally, we illustrate our approach on a cloud infrastructure configuration problem and provide detailed evaluation on both synthetic and real applications. This evaluation includes (i) the comparison of several exploration techniques, (ii) performance measurements on multiple state encoding techniques and (iii) comparing two implementation architectures of our design space exploration framework.

Keywords Design space exploration · Model-driven engineering · Search-based software engineering

This work was partially supported by the CERTIMOT (ERC_HU-09-01-2010-0003) project, the TÁMOP (4.2.2.B-10/1–2010-0009, 4.2.2.C-11/1/KONV-2012-0001) grants and the János Bolyai Scholarship.

Á. Hegedüs · Á. Horváth · D. Varró
Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar tudósok krt. 2. – Hungary
E-mail: {abel.hegedus,akos.horvath,varro}@mit.bme.hu

1 Introduction

Design space exploration (DSE) is a process to analyze several "functionally equivalent" implementation alternatives, which meets all *design constraints* in order to identify the most suitable design choice (*solution*) based on quality metrics such as cost or dependability. Design space exploration often appears as a challenging problem in application areas, such as dependable embedded systems [32, 40] and IT system management, where model-driven engineering (MDE) techniques are already popular. DSE can be performed either during the design process to find optimal designs or during runtime to help dynamic reconfigurations.

In traditional DSE problems, the design constraints and quality metrics are numeric attributes to express cost, time or memory limits etc. However, systems with modular software and hardware architectures (like AUTOSAR [2] in the automotive domain or large reconfigurable architectures) introduced *complex structural constraints* that express restrictions on the graph-based model of the system under design. These constraints may include restrictions related to the communication architecture or allocation of software and hardware resources. Furthermore, during the design of dynamically changing systems (e.g. reconfiguration of virtual servers over physical ones), the design space exploration also requires the *dynamic creation and deletion* of elements.

Existing DSE approaches usually apply model checking with exhaustive state space exploration [4, 44, 14] or solve finite domain constraint satisfaction problems (CSP) [25, 13]. In both cases, the high-level system models are often mapped to low-level formalisms that can be used as inputs for model checking tools or CSP solvers. Exhaustive approaches are well suited to problems where most of the design space is traversed to identify rare solutions and explores states are efficiently stored. CSP solvers are capable of efficiently apply branch-and-bound or other numerical techniques to solve high number of equations that share variables. However, neither approach can effectively handle structural constraints and dynamic manipulation of elements.

To better align generic exploration techniques with specific problems, designers often provide additional information (*hints*) about the system (e.g. from earlier experience or by some analysis) that can reduce the design space to a more feasible size [32]. The design process is often complemented with different design and analysis and verification tools, which can also provide (mathematically well-founded) hints about the model in the early stages of development. These hints may express additional system properties, which can be incorporated in the DSE process to assist the evaluation of alternate solutions.

Guided model-driven design space exploration aims to explore alternative system designs efficiently by making use of advanced model-driven techniques (e.g. incremental model transformations) and hints (obtained by analysis tools or provided by the designer). These hints are interpreted during the exploration to continue along promising search paths (using selection criteria) and to avoid the traversal of unpromising designs (by cut-off criteria). Additionally, the

Fig. 1 Inputs and outputs of guided DSE

use of incremental techniques leads to exploration strategies that are able to find additional (alternative) solutions, which are close to an earlier solution. Figure 1 illustrates the inputs (goals, constraints, operations, initial design, hints and guidance) and outputs (alternative designs as possible solutions) of guided design space exploration.

In our paper, we propose a *model-driven framework for guided design space exploration*, where the system states are graphs, operations are defined as graph transformation rules, while goals and constraints are defined as graph patterns. We extend our previous work on model-driven design space exploration [23] by incorporating hints during the exploration strategy, which are derived from dependency analysis of transformation rules and algebraic analysis on the Petri net abstraction of the system [55]. Cut-off and selection criteria are defined based on these hints [20], and their evaluation guides the design space exploration by identifying dead end states and prioritizing possible operations, respectively.

Major contributions of this paper with respect to our previous work [23, 20, 22, 21] are (1) the formal definitions of the concepts of guided design space exploration, (ii) the new implementation architecture of our model-driven framework based on the Eclipse Modeling Framework, (iii) a comparison of state encoding techniques used in our framework and (iv) a detailed evaluation of our framework with multiple scenarios using relevant case studies.

2 Overview of the Approach

In our paper, we describe a novel framework that combines the *model-driven* approach of design space exploration (DSE) with *guided exploration* techniques building on hints from analysis and guidance through cut-off and selection criteria. The schematic overview of the framework for guided design space exploration is illustrated in Figure 2.

First, the *design problem description* specifies the domain where the exploration takes place to produce solutions. It includes: (1) the initial state of the system at the start of the exploration, (2) the set of manipulation operations (called labeling or exploration rules) defined on the system, (3) goals described as structural or numerical constraints, which need to be satisfied by solution states found by the exploration, and (4) global constraints, which are satisfied by the initial and solution states and all intermediate states on the trajectory

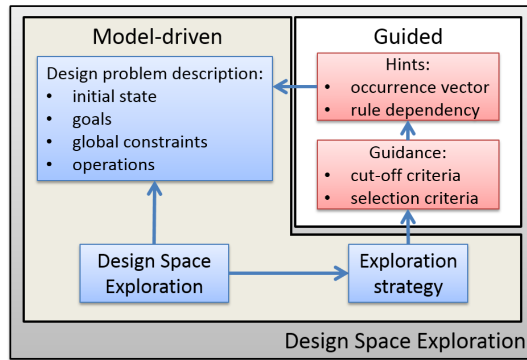

Fig. 2 Model-driven Guided Design Space Exploration

between them. The detailed discussion of the problem description is found in Section 4

The *design space exploration* performs the search for solutions by exploring the design (or state) space of the problem description. It starts from the initial state and traverses reachable states by applying the operations on the system (see Section 3). In order to find a solution quickly exploration is often aided by an *exploration strategy* (detailed in Section 7). A simple strategy (as proposed in [23]) may use random selection in a depth first search or statically assign priority levels to operations. However, a more advanced strategy should also determine whether a given state will never lead to a valid solution (i.e. it is a dead end) and states reachable from it should not be traversed. In a guided approach, the exploration strategy relies on *guidance*, which uses *hints* for driving the traversal and identifying dead ends.

Hints are information originating from the designer or (as in our paper) from some automated analysis carried out using formal methods that often abstract the design problem description. The result of the analysis can be information regarding the number of operation applications (called as an occurrence vector), partial ordering of operations, restricting the set of required operations etc. These results are often generated before the exploration in a preprocessing phase. Our guided approach uses *occurrence vectors* and *dependency relations* between rules as hints (see Section 5).

Finally, the guidance calculates and interprets hints and provides decision support for the exploration strategy (see details in Section 6). In our approach, guidance is defined as the evaluation of cut-off and selection criteria based on the current state and the hints (as defined in [20]). *Cut-off criteria* identify dead end states and bound the exploration, while *selection criteria* prioritize available rules in a state by their likelihood of leading to a final (solution) state.

2.1 Challenges of Guided Design Space Exploration

While existing model-driven frameworks (e.g. GROOVE [36]) are able to explore the design space of smaller problems by exhaustively traversing reachable states and checking global constraints and goals in each state, they use no global information when selecting the applied labeling rules. Our guided approach, however, takes advantage of hints and guidance that help the exploration and addresses the following challenges:

- *identify decisions in the exploration:* the framework should clearly separate the guidance from the exploration strategy to easily allow the modification of both parts of the framework.
- *soundly reduce traversed design space:* the guidance should reduce the number of traversed states before finding solutions, but it must ensure that no valid solutions are removed by the cut-off criteria.
- *provide optimal solutions:* the guided framework should find the solutions that are optimal (with respect to a user-defined metric). Moreover, the framework should be able to continue exploration to find other (less optimal) solutions if necessary.
- *extensibility:* the approach should be easily applicable on different design problems and the set of criteria should be extensible. This is a key feature for adapting the framework to various domains.

3 Guided Design Space Exploration

The guided design space exploration approach is based on a general search process, which traverses the design space starting from the initial state. This general process includes a step (*Evaluate criteria*), which relies on the guidance and hints provided by system analysis to the different exploration strategies (*identify decisions* challenge). The search process, depicted in Figure 3, consists of the following steps:

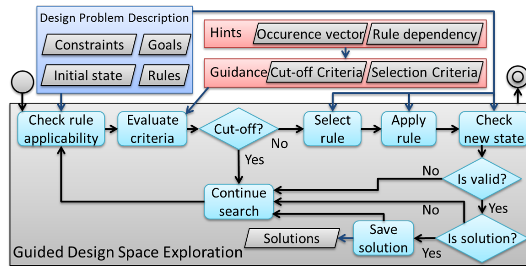

Fig. 3 Workflow of the guided design space exploration

1. *Check operation applicability.* First, labeling rules (of the design problem description) are checked for executability (i.e. whether they can be executed

in the current state of the model) and this information is passed to the criteria evaluation.
2. *Evaluate criteria.* The cut-off and selection criteria are evaluated using the hints (the rule dependencies and the occurrence vector) and the results are stored.
3. *Cut-off?* If at least one of the cut-off criteria were satisfied during the evaluation, or there are no applicable rules, the state is a dead end and the branch is cut.
4. *Select rule.* The design space exploration then selects the next applicable rule based on the evaluation results.
5. *Apply rule.* The selected rule is applied to the model resulting in a new model state.
6. *Check new state.* The global constraints and goals are checked on the new state to decide whether it is an invalid or solution state.
   (a) *Is valid state?* If any of the constraint are violated, the state is invalid and the exploration continues from the previous state. Note, that a state is also considered invalid if the exploration has visited it earlier, since in this case the reachable states are already explored from this state.
   (b) *Is solution found?* If all of the goals are satisfied, the state is a solution.
7. *Save solution.* When a solution model is found, the trajectory (with the executed rules and corresponding model state information) is saved to a solution list.
8. *Continue search.* Once the new model state is checked, the next applicable rule is selected from a valid new state, otherwise from the previous state.

Design space exploration terminates either once a predefined number of solutions are found (or if the found solution is acceptable by other, user-defined metrics) or if there are no applicable rules within the limited search space. Since a hint does not always represent a feasible trajectory, the exploration is restarted with an alternative vector if more solutions are required to be found.

4 Design Problem Description

4.1 Motivating Example: Cloud Configuration

Today services are often built on top of a *cloud middleware* (CM) using components as building blocks to be able to scale dynamically to meet demands. *Servers* (S) and high-availability *clusters* (Cl) can be deployed on the cloud middleware, while *databases* (DB) are installed on servers and *applications* (App) are executed over databases. Finally, servers can also be deployed on clusters and *storage* (St) subsystems can only operate over clustered servers.

In order to provide an appropriate infrastructure for clients, the configuration of the cloud infrastructure must meet certain requirements (including complex structural constraints), e.g. an application and a storage subsystem

is required for a cloud-based web service. Such an infrastructure is shown in Figure 4.

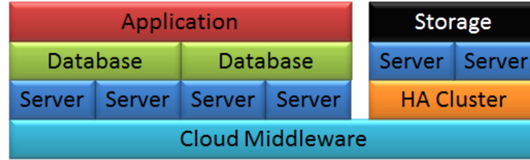

Fig. 4 An example system providing reliable service

To satisfy this constraint the cloud configuration has to be designed in an appropriate way. We assume that regular change management commands (including deletion or creation, e.g. deploying a new database) are issued by some middleware service broker. If the current infrastructure of the cloud detects that the required parameters cannot be satisfied by the actual cloud configuration, reconfiguration operations are to be initiated, which lead the system into a state where all constraints are met. To deal with changes of requirements and possible commands, guided design space exploration is used to find command sequences that should be executed to create a valid configuration.

4.2 Initial State

States are represented as instance models that conform to a metamodel. This metamodel describes the problem domain and the initial state defines where the design space exploration starts from.

Definition 1 (Graph) A *graph* $G = (N, E, src, trg)$ is a 4-tuple with a set N of nodes, a set E of edges, a source and a target function $src, trg : E \to N$.

Definition 2 (Type and instance graphs) A *type graph* TG is a graph. An *instance graph* G is typed over TG by a typing morphism $type : G \to TG$.

We assume that type inheritance is also allowed in type graphs, where a given type can be a specialization of other types, while those types are generalizations of the given type. We refer to type graphs as *metamodels* and to instance graphs as *instance models* of a metamodel. We will use this simple structure for describing models, while more complete and formal treatment of metamodeling can be found e.g. in [1, 28, 53].

The left part of Figure 5 shows the metamodel for the cloud case study. The metamodel contains a cloud component *Node* designated graphically as a rectangle. The specific components *Socket*, *Server*, *Database*, *Application* and *Storage* are specialized from this node, *Socket* is a generalization of *Cloud MW* and *Cluster*. Edge *deployedOn* is a relation that connects two different components denoting that the source node is deployed on the target node of

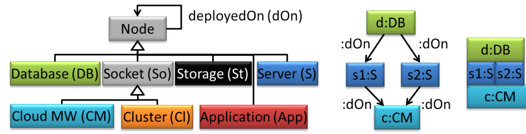

Fig. 5 Metamodel and instance model of the cloud infrastructure

this relation. The right part of Figure 5 illustrates an instance model containing a database d deployed on two servers $s1, s2$ that are on cloud c. Note that in the rest of the paper, we omit *deployedOn (dOn)* relations by illustrating the relation using vertical arrangement of components.

4.3 Goals and Global Constraints

Goals and global constraints of the design problem description are defined as functions evaluated over matches of graph patterns [15].

Graph patterns (query) represent conditions that have to be fulfilled by a part of the model and are frequently considered as the atomic units of model transformations [52]. A *match* of a graph pattern is a set of nodes in the instance model that satisfies all conditions defined by the graph pattern. Formally defined as follows:

Definition 3 (Graph pattern) A *graph pattern* $gp = (sc, \vee_{j \in J} nac_j)$ consists of (i) *structural conditions sc* prescribing the existence of type conformant nodes and edges and (ii) *negative application condition* $nac = \neg gp$, defined by a negative subpattern, prescribes conditions which are forbidden in a specific context of sc.

Definition 4 (Match) A *match* m for a graph pattern $gp = (sc, \vee_{j \in J} nac_j)$ in an instance model G denoted by $m : gp \longrightarrow G$ means that: (i) $\exists m : sc \mapsto G$ there exists an injective, type conformant total morphism m from the graph sc to the instance model G, (ii) $\forall j \in J \nexists m' \supseteq m$ where $m' : gp_{nac_j} \longrightarrow G$: there is no match for any of its embedded NACs that extends the match of the pattern gp.

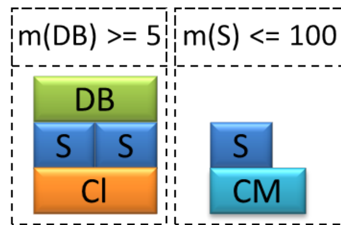

Fig. 6 Example goal and global constraint

The left part of Figure 6 shows a graph pattern describing a database (*DB* node) deployed on two servers (*S*) that are both deployed on the same cluster (*Cl*). The example goal specifies using this pattern requires that a solution model includes at least 5 databases deployed on clusters, while the right part shows a global constraint, that allows maximum 100 servers deployed on clouds altogether.

The examples in Figure 6 use very simple numeric bounds on the size of the match set but for the purposes of DSE more complex functions may be used. We have experimented with goals that evaluate interdependent and symmetric matches as well. However, it is important to note that the computationally expensive part of constraint and goal evaluation is done by pattern matching that can work fully incrementally [9]. Additionally, graph patterns with embedded NACs provide expression power equal to first-order logic [37], while advanced features (e.g. transitive closure [11], match counting and evaluation of expressions) further increase the usability of the language.

4.4 Operations

The operations that define the possible elementary manipulations on the problem state are represented by graph transformation (GT) rules [15].

Definition 5 (Graph transformation rule) A *graph transformation rule* is a pair $r = (pre, post)$, where *pre* is the *precondition* (or left-hand side - LHS) pattern determining the applicability of the rule and *post* is the *postcondition* (or right-hand side - RHS) pattern that specifies the result model declaratively.

Definition 6 (Activation) An *activation* $act(rule, m)$ of a transformation rule *rule* is a match m of the precondition pattern *pre* of the rule.

The reconfiguration actions of the ongoing example are captured by a set of graph transformation rules in Figure 7. An overview on using graph transformations for software architecture reconfigurations can be found in [5].

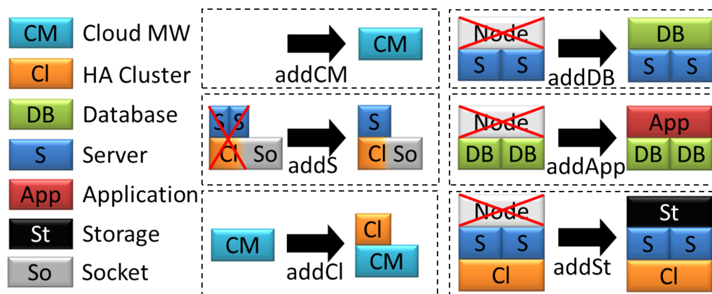

Fig. 7 Graph transformation rules

The *addCM* rule adds a new cloud *CM*, *addS* creates a new server *S* deploying it on top of a *CM* or cluster *Cl*, however, a *Cl* cannot have more than two *S* deployed on it. Rule *addCl* produces a new *Cl* deploying it on top of a *CM*, *addDb* adds a new database *DB* deploying it on top of two *S* that have no other *Node* deployed on them, *addApp* creates a new application *App* deploying it on top of two *DB* that have no other *Node* deployed on them. Finally, *addSt* adds a new storage *St* deploying it on two *S* that are deployed on the same *Cl* and have no other *Node* deployed on them.

It is important that the set of goals, constraints and rules are easily extensible by the designer (*extensibility* challenge). The design problem description is not hard-coded into the exploration and can be modified using a high-level textual language [52]. Our framework also supports dynamic handling of goals, constraints and rules, e.g. to generate solutions for different subsets of rules.

Executing an activation $act(rule, m)$ alters the model by replacing the pattern defined by LHS with the pattern of the RHS of the transformation rule *rule* (illustrated in Figure 8). This is performed by (1) taking the *match* m of the LHS in the model (2) *checking the negative application conditions*, (3) *removing* a part of the model that can be mapped to the LHS but not the RHS yielding an intermediate graph and (4) *adding* new elements to the intermediate graph, which exist in the RHS but not in LHS or *updating* existing elements, yielding the derived graph.

Definition 7 (Exploration step) An *exploration step* $G \stackrel{act}{\Longrightarrow} G'$ is the execution of the activation *act* on the instance model G resulting in the modified model G'.

4.5 Design Space Exploration

The design space traversed by the guided exploration approach is represented by a graph transition system [35] containing the states, which are reachable from the initial state by executing the operations.

Definition 8 (Exploration sequence) An *exploration sequence* $G_0 \stackrel{act_1}{\Longrightarrow} G_1 \stackrel{act_2}{\Longrightarrow} G_2 \Longrightarrow \ldots$ is a sequence of exploration steps (executing an activation of a given transformation rule).

An exploration sequence starting from G and yielding G' is denoted shortly by $G \stackrel{*}{\Longrightarrow} G'$, where $*$ denotes that one or more exploration steps may belong to the sequence.

Definition 9 (Design space exploration problem) The *design space exploration problem* is a 4-tuple $DSE = (G_0, Op, Goal, Cons)$, where G_0 is the initial state, Op is the set of possible operations, $Goal$ is the set of goals, $Cons$ is the set of global constraints.

Definition 10 (Solutions of design space exploration) The *solutions* of DSE is a pair $Sol = (DSE, ES)$, where ES is a set of exploration sequences and for each sequence $G_0 \stackrel{*}{\Longrightarrow} G_i \in ES$, the final state G_i contains matches of the patterns in $Goal$, each state in the sequence is reached by exploration steps from Op and does contain matches of any pattern in $Cons$.

The solutions are found by constructing the possible execution sequences starting from the initial state of the DSE problem.

Definition 11 (Design space) A *design space* of a DSE problem is a graph $DS = (N_{im}, E_{es}, src, trg, G_{curr})$, where the nodes $G_i \in N_{im}$ are instance models, edges are exploration steps $G_j \stackrel{act}{\Longrightarrow} G_k \in E_{es}$ and the source and target of an edge are the instance models before and after the execution of the step. The exploration starts from the initial state G_0 of DSE and the exploration steps use only operations from Op in DSE. Finally, $G_{curr} \in N_{im}$ is the *current state* of the design space which is also the target of the last exploration step.

A path in the design space is an exploration sequence also called a *trajectory* between two states. A state G_i is *reachable* from G_0 iff there is a trajectory in DS from G_0 to G_i.

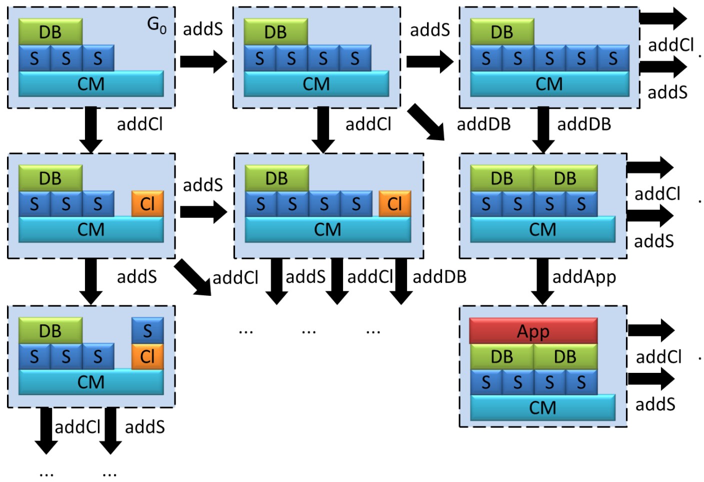

Fig. 8 A part of a design space

In Figure 8 an extract of the design space of the running example is shown. On the left, the root of the design space is the start graph G_0 where the system configuration contains a *CM*, three *S*, and one *DB* components. Operations *addS*, *addCl*, and *addCM* are applicable to G_0, here we follow the execution of *addS* and *addCl*.

5 Hints

The design space exploration framework uses exploration sequences to reach solution states. In order to guide the exploration efficiently, both the amount and order of operation executions are useful hints.

5.1 Graph Transformation Rule Dependency

Given the precondition-postcondition nature of GT rules used as operations, it is possible to derive which rules might be affected by the execution of a given operation. For example, executing an activation of GT rule r can alter the model in a way that other rules, which were disabled before, become enabled (or the other way around), thus the application of these rules *depend* on the application of r. The dependencies between rules are independent of the instance models, and can be derived from the rule definitions. This analysis can be carried out using various techniques, such as critical pair analysis [19] or conditional transformation-based dependency analysis [31], and results in a matrix of dependencies between rules.

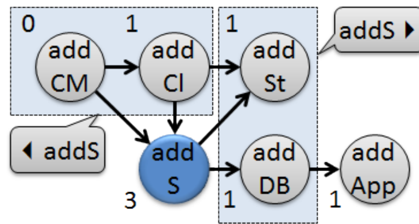

Fig. 9 Dependency graph example

Definition 12 (Dependency graph) A dependency graph for a DSE problem is a graph $Dep = (N_{op}, E_{sd}, src, trg)$, where the nodes N_{op} are the possible operations Op of DSE and the edges E_{sd} denote *sequential dependency* (i.e. for an edge e the application of the source operation ($src(e)$) may affect the activations of target operation $trg(e)$).

The result of the analysis is used to create a *dependency graph* (Dep, illustrated in Figure 9). Note that there may be arcs in both directions between two rules. As illustrated on Figure 9, rule *addS* depends on rules *addCM*, *addCI*, while rules *addSt*, *addDB* depend on *addS* (the sets are represented by $\blacktriangleleft r_{addS}$ and $r_{addS} \blacktriangleright$, respectively).

5.2 Transformation Rule Occurrence Vector

We use a Petri net abstraction technique introduced for GTS in [54], which provides hints that estimate how many times each rule is applied in order to reach a given state.

Definition 13 (Occurrence vector) A candidate occurrence vector σ is a solution of the analysis of the Petri net abstraction, where σ(i) is the number of times that rule r_i is applied during the execution.

During the design space exploration, the number of times rule r_i has been applied in a given path is stored in the *application vector* ($\overline{v}_a$) as $\overline{v}_a$(i).

Definition 14 (Compliant exploration sequence) An exploration sequence of the design space exploration is compliant with σ if $\overline{v}_a \leq \sigma$ (the number of applications is less than or equal to σ(i) for each rule r_i).

Throughout the paper we use the difference $\sigma(i) - \overline{v}_a(i)$ as the *remaining application number* $\#_i$ of rule r_i. This number is stored as an attribute for nodes in *Dep* (see Figure 9) together with the *state of* r_i that is either enabled or disabled in a given state.

Note that in some design problems, the occurrence vectors provided by the analysis could be used as the least amount of executed rules instead of maximum. In such cases, the guidance would be able to efficiently reach states where those rules are executed in the predefined number of times. Naturally, it is possible to prepare design problems where such guidance is not entirely beneficial (e.g. there are rules that are not part of the occurrence vector but have to be executed earlier than those that are part of the vector). In our paper, we use occurrence vectors as upper limit of execution on the rules and instead apply alternative occurrence vectors when no solution is found for a given one.

5.3 Using Dependency Graphs in Design Space Exploration

The model state and the dependency graph are tightly connected for a given initial graph and occurrence vector. Figure 10 illustrates how the application of a GT rule affects the current state and the remaining application number. First, the current state is depicted as the model M (representing the current cloud configuration) and remaining application number and state of each node in the dependency graph *Dep* (in short, the *current dependency graph*). The color of the nodes (e.g. n_{addS}) of *Dep* represents the state of the corresponding GT rules (r_{addS}), green (dark) background for enabled, grey (light) for disabled. The number near each node is the remaining application number (e.g. $\#_{addS} = 3$).

In the course of design space exploration, the next GT rule, which is applied (r_{addS} in the example) is *selected* from the set of enabled rules. The *application*

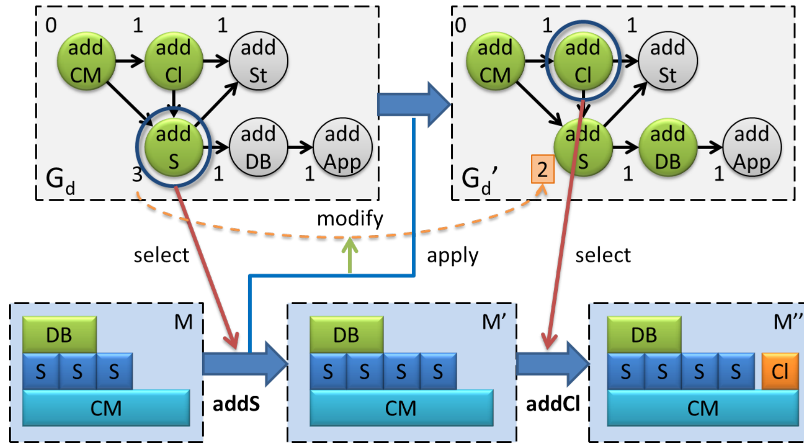

Fig. 10 Executing an operation and its effects on the dependency graph

has the following effects on the models: (a) model M changes according to the rule definition (here, a new server S is added to cloud CM), the new model is illustrated as M' (b) the $\#_{addS}$ is *modified* to represent that the rule is applied (it decreases from 3 to 2) (c) Dep is also changed to Dep', as $\#_{addS}$ decreased and the applicability of GT rules may change (here r_{addDB} becomes enabled). The design space exploration then continues from M' by selecting a rule based on the dependency graph Dep'.

Note that both the Petri net abstraction and the dependency analysis of rules are techniques more tailored for design problems with dynamic creation and deletion of elements. While they can be applied in special problems where a graph-based model is used but the operations do not create or delete elements, they may be less efficient in guiding the exploration.

6 Guidance

6.1 Overview of Cut-off and Selection Criteria

Cut-off and selection criteria are used as guidance to decide in which order the states of the design space are explored. We define formal criteria over the current dependency graph, which are evaluated to support decisions:

Definition 15 (Cut-off criteria) A cut-off criterion is a function $cut : (DS, Dep) \mapsto bool$, where DS is the design space and Dep is the current dependency graph, which returns true if further exploration from the current state G_{curr} of DS cannot lead to a goal state with a compliant trajectory.

When a cut-off criterion returns true, the exploration continues from another state instead of executing an operation in the current state.

Definition 16 (Selection criteria) A selection criterion is a function $sel : (DS, Dep) \mapsto Op$, where DS is the design space and Dep is the current dependency graph, which returns an ordered list of operations that have activations in the current state G_{curr} of DS.

A given rule r_i is placed before an other rule r_j, if the execution of r_i is more promising, based on Dep and the current state, than the execution of r_j.

6.2 Criteria for Guided Design Space Exploration

We used the following cut-off and selection criteria [20], which are meaningful when dealing with guided DSE.

- *Non-compliant path (Look-ahead) cut-off criterion.* If the application of any GT rule would make the current execution path non-compliant with the occurrence vector, it can be cut.
- *Permanently disabled rule cut-off criterion.* The current path can be cut if there is a disabled rule, which still has to be applied based on the occurrence vector, but rules that may enable it will not be applied.
- *Independent rule application selection criterion.* Applicable rules with no forward dependency should be applied as early as possible to reduce the number of different applicable operations later in the trajectory.
- *Maximal forward-dependant application path selection criterion.* Among the applicable rules at any given state of the exploration, the rule that affects more applications should be applied earlier in the trajectory.

The criteria defined over the dependency graph are evaluated at every state using an algorithm described in [20] (*interpret hints* challenge). The main steps of the algorithm are: (1) a starting point is selected from the criterion, (2) the list of nodes satisfying the starting point are created, (3) the operations of the criterion are applied on each node and (4) the result is assembled as a boolean value (cut-off criteria) or an ordered list of rules (selection criteria).

7 Exploration Strategy

Guided exploration strategies can be categorized by the used hints and guidance. We specified two guided strategies (see Figure 11), the first uses occurrence vectors only as hints (*occurrence*), while the other uses rule dependency as well (*full guidance*). Note that the full guidance strategy uses rule priorities only if two labeling rules were evaluated as equal by the guidance. These strategies are compared to the *fixed priority* depth-first search strategy.

Figure 12 illustrates the design space exploration for these techniques on a simple example. The circles denote the traversed states which are numbered according to the traversal order, while the applicable rules are listed beside them. Downward arrows illustrate rule applications, while upward (and dotted) arrows represent backtracking from invalid or cut-off states. The same rule can

Exploration strategy		Used hints		
		Rule dependency	Occurrence vector	Rule priority
Fixed priority		No	No	Yes
Guided	Occurrence	No	Yes	No
	Full guidance	Yes	Yes	Yes

Fig. 11 Comparison of exploration strategies by used hints

be applied multiple times at a given state if more than one applicable match is found in the graph (see state 2 on the right side). The termination of the exploration is done based on the evaluation of found solutions. In a very simple case, we terminate if the path leading to the last found solution contains a total number of rule applications equal to a problem-specific limit (i.e. it is the shortest trajectory to a solution model).

Note that many other, more complex termination techniques are possible, for example by extending labeling rules with costs and terminating when solutions identified by a total cost less then a predefined limit are found [55]. Additionally, in DSE it is often required to provide multiple solutions and apply an evaluation on the quality of these afterwards. This evaluation may define quantitative goals instead of simple boolean function, e.g. in the case study it would be possible to count the number of components, evaluate possible redundancy structures or optimize resource usage (if such information is added to the model). Finally, similar evaluation is done in flexible CSPM [23], where hard and soft goals are defined and solutions have to (i) satisfy all hard goals and (ii) their quality is calculated as a linear function on the weight of the fulfilled soft goals (similarly to weighted CSP).

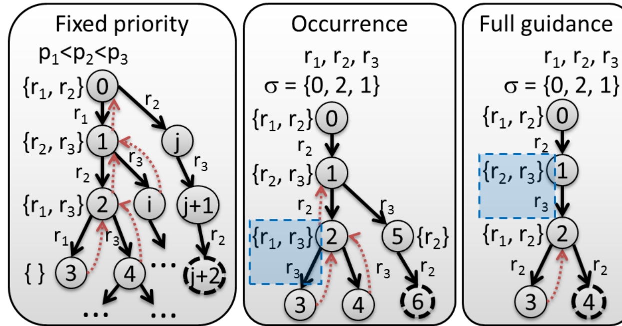

Fig. 12 Comparison of exploration strategies

Comparison of exploration strategies In the case of the *fixed priority* strategy, the next applied operation is the one with the highest priority among the applicable ones. In the example, first r_1 is applied then r_2. From state 2, first r_1 is applied leading to state 3 without applicable rules. After backtracking, r_3 is applied instead. Note that after this point all reachable states from state

2 and state 1 are explored before trying r_2 in state 0 (which finally leads to an optimal solution). Moreover, as the depth-first technique is used in the fixed priority exploration strategy, the first solution found by that strategy is often several times longer than the optimal, suboptimal solutions are used as depth limits to force the exploration to find shorter solutions.

The *occurrence* strategy applies operations based on the occurrence vector provided by the system analysis. The example in Figure 12 shows that r_2 should be applied twice and r_3 once. Therefore, r_1 is not applied in state 0 or 2 (highlighted) in order to be compliant with the occurrence vector. In states 3 and 4, the exploration backtracks (as no more rule applications are allowed by the vector) and then continues to find the solution in state 6.

The *full guidance* exploration strategy (illustrated in the right side of Figure 12) takes the dependency relations between rules into account in addition to the occurrence vector. Therefore, in state 1 (highlighted) it selects r_3 for the next application. Rule r_2 is applicable on two matches in state 2, the first leading to a dead-end state, while the second application leads to a solution in state 4. Note that the selection in state 1 leads to a reduced traversed design space compared to the occurrence exploration strategy (*reduce traversed design space* challenge).

8 Implementation Details

We implemented the first version of our model-driven framework for guided design space exploration on top of the VIATRA2 model transformation framework [22]. In this paper we present a new implementation architecture that builds on the Eclipse Modeling Framework (EMF) (Section 8.1). We also discuss the challenge of identifying recurring states during the exploration using state encoding techniques (Section 8.2).

8.1 EMF-based Impementation

The Eclipse Modeling Framework [48] has become a de facto standard modeling representation in the Eclipse ecosystem. *EMF* provides metamodeling capabilities and handles instance models, however it does not include model transformation and model query support that are required for design space exploration. In the following we describe the technologies that were integrated to provide guided design space exploration over EMF models.

Metamodels and instance models. The metamodel of a domain is created using the Ecore metametamodel which defines the concept of EClasses (types) and EReferences (relations). Based on the metamodel, EMF uses a generative approach to provide capabilities to create, manipulate, store and load instance models for the defined metamodel.

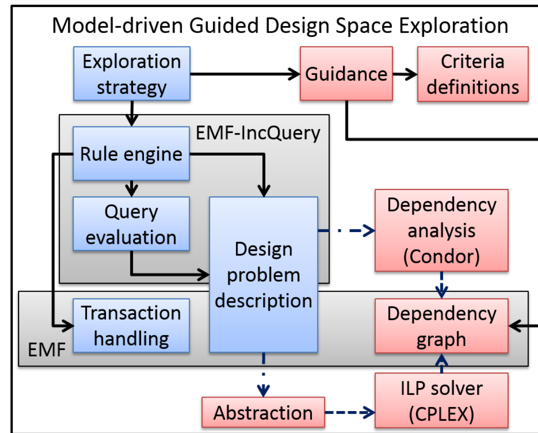

Fig. 13 Overview of the EMF-based guided DSE framework

Model queries for goals and global constraints. There are several tools available for querying EMF instance models, including EMF-Query2 [49], EMF Search [50] and Eclipse-OCL [51]. We choose *EMF-IncQuery* [10], which is an incremental *query evaluation* framework that uses the pattern matching technology (RETE network) of VIATRA2 over EMF models. Incremental evaluation is important for the DSE framework for efficiency including evaluating goals, global constraints, and operation preconditions in each state.

Operations. We define operations using EMF-IncQuery model query definitions as preconditions and simple Java code for model manipulation that is parameterized by the match of the precondition query.

For DSE, we need to efficiently manage the possible activations in an instance model of an arbitrary set of operations that have model queries as preconditions and to provide a common way for executing an operation with a selected activation. EMF-IncQuery includes an event-driven *rule engine* (where events are the incremental changes in query results) which supports the execution scenarios including the manual selection and execution of activations required for DSE.

Backtracking and exploration. The EMF *transaction framework* supports the execution of complex commands that are composed of a series of primitives (such as create, add, remove, set) and are undoable or redoable as required. The transaction handling is used by creating composite commands for each operation execution. These commands can be undone when the exploration backtracks.

Dependency graph and criteria evaluation The representation of the *dependency graph* uses EMF and was developed to be independent from the representation of instance models. The *criteria definitions* and the criteria evaluation

algorithm (*guidance*) are implemented in Java as separate components, and are connected to the guided design space *exploration strategy*. Therefore, it can be used with the EMF-based DSE implementation. However, we do not have an automated way of inspecting the simple Java code to perform the *abstraction* from transformation rules to Petri Nets. Note that the precondition queries can be evaluated automatically as the query language of EMF-IncQuery is declarative and query specifications can be processed as models. We used the industry leading IBM *CPLEX*[1] optimization tool, which supports the calculation of alternate solutions (occurrence vectors used for initializing the *dependency graph*). The edges of *Dep* are computed from the transformation rules using the *Condor*[2] dependency analyzer tool.

8.2 State Encoding Techniques

The exploration may encounter the same model state on different trajectories and has to identify such states in order to avoid the re-exploration of states reachable from that state (see Step 6.a in Section 3). In the graph transition system illustrated in Figure 8 several states are reached through different trajectories, for example by adding a server or a cluster on the same middleware in different order.

8.2.1 Identifying Recurring States

Recurring states can be identified by iterating through each already visited state in the search space and comparing the current state to them. However, there are multiple reasons that make this approach infeasible:

- The exploration is performed over a single instance model and (a) it would have to be copied in each explored state for comparison (infeasible for memory limitations) or (b) each comparison would require the re-exploration of each explored state (infeasible for runtime limitations).
- Model comparison itself as a single operation is also challenging and often slow.
- The complete search space does not fit into available memory, therefore fully explored parts of the search space are deleted to free up memory for the part that is explored currently.

Due to these limitations, we need to represent the current state of the exploration in a concise way that can be stored and efficiently compared to previously stored states. The concise representation of the current state is called *state encoding*, while the result for a given state is called the *state code*.

In the following we describe the challenges in state encoding and present possible techniques that can be used for DSE.

[1] `http://www.ibm.com/software/integration/optimization/cplex-optimizer/`
[2] `http://roots.iai.uni-bonn.de/research/condor/`

8.2.2 Challenges of State Encoding

There are a number of challenges related to the specification of state encoding methods:

- **Deterministic**: applying the encoding to the same state must always result in the same state code.
- **Under-approximating**: if two different states have the same state code, then each solution reachable from one state is reachable from the other as well.
- **Fast**: the time to calculate the state code for a given state and check whether it was already explored should not be orders of magnitude slower than one iteration of the search process (see Section 3).
- **Minimal**: state codes should not contain redundant information or data that is common in all state codes. This is important as the DSE framework has to store a large number of state codes.

8.2.3 Comparison of State Encoding Techniques

We developed several different state encoding techniques for the DSE framework, which are compared in Figure 14.

State encoding	Op	OpIncr	Mod	ModFull	Ind	IndIncr
Metamodel dependent	Yes				No	
State information	Operations		Instance model			
Incremental calculation	No	Yes	No			Yes
Precision	Partial			Full		

Fig. 14 Comparison of state encoding techniques

The state encoding techniques are classified along the following aspects:

- Metamodel dependent state encoding techniques (*Op*, *OpIncr*, *Mod* and *ModFull*) are customized for encoding instance models of a given metamodel, while the remaining techniques (*Ind* and *IndIncr*) can be applied on an arbitrary metamodel. While the former techniques can take advantage of the specific structure of the metamodel and provide faster computation or smaller state codes, the latter are reusable in the framework for any design problem description.
- The information that identifies a state can be the operations that were executed to reach the encoded state (*Op* and *OpIncr*) or the instance model in the current state (*Mod*, *ModFull*, *Ind*, *IndIncr*). Techniques that use the operations calculate state codes by evaluating the current trajectory from the initial state and can identify equivalent trajectories. When the instance model is used for encoding, the state code usually includes information on each element in the model and the relations between them.

– Since the execution or backtracking of applying an operation involves model manipulations that change only a small part of the model (or the trajectory), it is possible to create incremental state encoding techniques that update the state code of the previous state based on the change (*OpIncr* and *IndIncr*). The other techniques perform the encoding without prior knowledge of the previous state, state code or the last change and calculate the state code only from the current state (*Op*, *Mod*, *ModFull* and *Ind*).
– Finally, some techniques are only partially precise (*Op*, *OpIncr* and *Mod*), which means that they can associate the same state code to states that are significantly different and depending on the design problem description, they may violate the *under-approximating* challenge of state encoding. On the contrary, there are techniques that are fully precise (*ModFull*, *Ind* and *IndIncr*) and will ensure that two states will only have equal state codes if the states themselves are also equal.

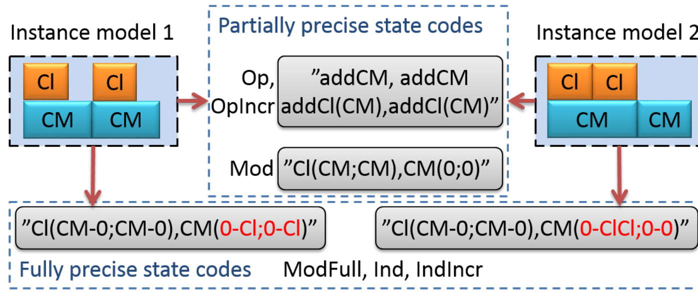

Fig. 15 State encoding example

Figure 15 illustrates the state codes calculated by the different state encoding techniques we listed in Figure 14. The two instance models on each side contain two middleware and two cluster nodes, but on the left the clusters are on different middleware nodes, while on the right they are on the same one. Techniques that are only partially precise associate the same state code for the two instance models.

Operation-based encoding. Techniques *Op* and *OpIncr* take the execution of operations in the trajectory and encode them in a sorted list. This means that if two trajectories include the same operations in different order, then the two states have the same state code. The state code also includes the activation of each operation as well. However, if the specific elements in the activation are used, then equal states will get different state codes, since a cluster node created by one application of the *addCl* operation will be different from the one created by another application of the same operation. On the contrary, if only the types of nodes are used, then the encoding will only be partially precise, as illustrated by Figure 15.

Partially precise model encoding. The partially precise technique *Mod* uses the instance model for encoding. It stores the type of each element and the type of the elements that are targets of *deployedOn* relations to the given element. For example, the instance models in Figure 15 contain two cluster nodes deployed on middleware nodes, which is encoded as $Cl(CM;CM)$. Furthermore, the middleware nodes are not deployed on any nodes, which is encoded as $CM(0;0)$. Note that the state code of both instance models is the same although they are clearly different. That is why we specify the fully precise state encoding technique *ModFull*.

Fully precise model encoding. The *ModFull* technique extends *Mod* by identifying that storing the *deployedOn* relations only in one direction leads to partially precise encoding. To ensure that instance models with the same state code are truly structurally the same, state codes calculated by *ModFull* include the inverse of *deployedOn* relation as well. This means that for each element, the state code will include the types of elements that are deployed on the given element. Therefore, the state code for the instance model on the left side of Figure 15 will include $CM(0-Cl;0-Cl)$, while the same part for the right side will contain $CM(0-ClCl;0-0)$.

Metamodel independent encoding Finally, all the above techniques were customized for the metamodel of the cloud case study, but we can also define techniques that calculate state codes for instance models of arbitrary metamodels, as long as the metamodels themselves are also available at the time of the encoding. The technique Ind calculates the state code of an instance model by taking each element and finding their type in the provided metamodel. Then the possible relations for the given type and their inverses are encoded similarly to *ModFull*. Instead of iterating through each element every time the state encoding is performed, the incremental technique IndIncr stores the partial state codes corresponding to the elements in the instance model and updates these stored values based on the model changes related to the execution or backtracking of an operation. This incremental approach can add a bit of overhead for handling model changes but also means that calculating the state code takes less time.

9 Evaluation of the Approach

We evaluate our model-driven framework for guided design space exploration in four different measurement scenarios. Apart from synthetic benchmarks $(1,3-4)$, we also evaluate a real application of guided DSE (2).

1. First, we demonstrate that the full guidance strategy can be more efficient than the other strategies (namely, fixed priority and occurrence, which we used for previous measurements in [23]) as it traverses considerably fewer states and does not introduce significant overhead, thus provides better runtime in most test sets than the other approaches.

2. Next, we evaluate a different kind of guided exploration that uses local violations of structural constraints to generate quick fixes for domain-specific modeling languages and demonstrate that the added information makes the approach feasible in a live modeling scenario, unlike exploration without guidance.
3. We compare the different state encoding techniques on the cloud case study and demonstrate that by identifying equivalent states they allow the exploration to scale to larger design spaces.
4. Finally, we show our initial results with an EMF-based implementation of our framework and compare it to the existing VIATRA2-based framework.

In each scenario, we introduce the test sets used in the evaluation and the environment and method used for the measurement, then we evaluate the results. The reader is directed to [23] for comparison of (the previous version of) the DSE framework with other tools (e.g. SICStus Prolog CLP(FD), KORAT and GROOVE).

9.1 Scenario 1: Dependency Graph Guided Exploration

9.1.1 Test Sets Used in the Evaluation

For evaluation, we used the cloud case study presented in Section 4.1 and a service configuration case study (presented in [55]). These cases are relevant in the context of model-driven DSE as they represent both design time and runtime exploration problems, respectively, and it allows comparison with previous results [55, 23].

Both case studies included multiple test sets (see Figure 16). *PowerOn* test sets deal with empty initial models, while *Reconfigure* test sets deal with existing models which must be modified to satisfy goals. In the cloud test sets, the goals describe the number of required components (e.g. 2 applications and 2 storage in *PowerOn Small*). Furthermore, global constraints are raised to give some limit to the priority based strategy (e.g. a cloud middleware should have at most 100 nodes installed). Finally, the *Clustered Database* test set requires databases to be deployed on clusters (see Figure 6).

In the service configuration test sets, the models represent a set of services that are reconfigured runtime (e.g. removing faulty or starting new instances) to meet some QoS requirements. The constraints in these test sets define the maximum number of services, while goals describe the number of active services and that faulty services are removed.

The size of the models are given after the name of the problem, in the cloud test sets the required applications and storage subsystems, while in the service test sets the maximum number of services, faulty and active services in the initial model and active services in solutions.

9.1.2 Evaluation Environment and Method

The evaluation was carried out 5 times for each test set and strategy in the following way[3]: (1) the initial model is loaded into VIATRA2, (2) the goals, constraints and operations are added to the framework, (3) the exploration component is initialized and runtime measurement is started (using wall time with OS-level nanotime precision). Next, (4) the design space exploration framework looks for an optimal solution. Finally, (5) the runtime measurement is stopped and the results are saved. The exploration is limited to 1 million visited states.

9.1.3 Evaluation of Results

The table in Figure 16 shows the results of measurements using the case study models. For each test set, we measured the average length of the shortest discovered solution trajectory (the number of applied rules), the average number of visited states during the design space exploration and the average runtime of the exploration.

Problem			Exploration strategy	Optimal trajectory found [length]	Visited states	Runtime [ms]
Cloud infrastructue	1	PowerOn Small (2/2)	Fixed priority	23	205 393	108 408
			Occurrence	23	715	676
			Full guidance	23	23	77
	2	PowerOn Large (5/5)	Fixed priority	66	154 669	94 902
			Occurrence†	-	-	-
			Full guidance	56	56	147
	3	Clustered Databases (5/5)	Fixed priority	28	662 425	360 418
			Occurrence	27	39 096	22 444
			Full guidance	27	6 543	4 533
Service reconfiguration	4	Reconfigure Small (8/2/1/3)	Fixed priority	5	372	451
			Occurrence	5	52	271
			Full guidance	5	5	170
	5	Reconfigure Medium (15/3/5/8)	Fixed priority	9	34 639	17 312
			Occurrence	9	1 475	1 318
			Full guidance	9	607	759
	6	Reconfigure Large (20/8/0/8)	Fixed priority	21	441 640	203 122
			Occurrence	21	716 671	359 152
			Full guidance	21	558 976	268 042

Fig. 16 Results for exploration until optimal solution († denote test sets where exploration did not terminate in all tests)

[3] For measurements we used a computer with Intel Centrino Duo 1.66 GHz, 1.5 GB memory (Java heap size), Windows 7 Professional 32 bit, Eclipse 3.6.1, VIATRA2 3.2

We made the following observations based on the results from the different cases:

Find optimal solution. We observed that the usage of occurrence vectors as hints in the exploration ensures that the first solution found by such strategies is optimal as well (*optimal solutions* challenge). In our observations, the *fixed priority* strategy, finds longer solutions first and traverses a large number of states even in case 4 (which is the smallest), before finding an optimal solution.

Low overhead of criteria evaluation. The evaluation of cut-off and selection criteria is performed at every new traversed state, and it might (in principle) slow down the exploration considerably. However, our observation is that criteria evaluation has very low overhead (less than 5% of the overall runtime). The full guidance strategy requires some initial bookkeeping (building dependency graph and initializing criteria), but afterwards, it traverses 1000 states in roughly $600ms$ (similarly to the other strategies).

Rule dependency increases efficiency. In all test sets, the *full guidance* strategy traverses significantly fewer states than the *occurrence* strategy. Note that the only test set when the *fixed priority* strategy traverses less states is test set 6, where the occurrence vector is recalculated at least 20 times before finding a feasible solution.

It is important to note that in these test sets, the full guidance approach outperforms the occurrence strategy by identifying infeasible occurrence vectors with less exploration. Figure 17 illustrates how the number of traversed states for these two strategies when exploring infeasible occurrence vectors in test set 5. The graph clearly shows that the full guidance strategy explores half of the states in average that the occurrence does. Note that in test set 2, the occurrence strategy did not find a solution inside the limit in some instances.

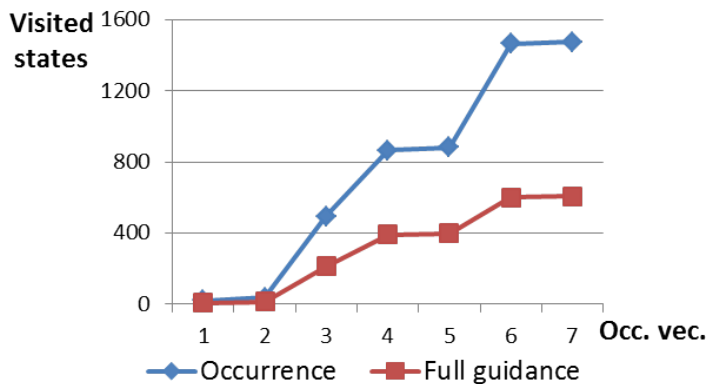

Fig. 17 Reduction in visited states by the full guidance strategy

To sum up the results of the evaluation, we observed that:

- The combined use of occurrence vectors and rule dependency for cut-off and selection criteria based guidance outperforms our previously published strategies [23]. The full guidance strategy finishes in less time than (1) the occurrence strategy in all 6 test sets (with at least 25%) and (2) the fixed priority strategy in 5 out of 6 test sets (with at least 60%).
- The added computation required for criteria evaluation increase runtime only by 5% in average.
- The under-approximation of the occurrence vector based analysis ensures that guided exploration strategies always find optimal solutions first.

9.1.4 Limitations

Our guided DSE relies on the quality of the hints provided for the design problems. This may be a limitation in the following cases: (1) if the occurrence vector is infeasible and it includes a large number of rule applications (similarly to test set 6) and (2) if the dependency graph (*Dep*) is close to a complete directed graph, the guidance of the cut-off and selection criteria is less effective. Finally, a large *Dep* (in case of large set of operations) may increase the overhead of criteria evaluation.

9.2 Scenario 2: Quick Fix Generation for Domain-specific Modeling Languages

9.2.1 Test Sets Used for the Evaluation

We defined the generation of quick fixes that correct inconsistencies in domain specific models as a design space exploration problem in a previous paper [21]. In this scenario, the initial state of the exploration is an instance model containing a number of inconsistencies. The operations are elementary model manipulations that preserve syntactic correctness (by e.g. *syntax-driven editing*). The goal of the exploration is to eliminate the inconsistencies corresponding to a given model element e without introducing additional inconsistencies into the model.

We use the Business Process Model And Notation (BPMN [34]) as an illustrative case study. BPMN is a well-known and widely used standard, flowchart-like notation system for specifying business processes. We evaluate the approach on two real BPMN projects, obtained from an industrial partner from the banking sector. One project is a corporate customer registering workflow, composed of five processes and approximately 250 model activities in total. The other project is a corporate procurement workflow, composed of three processes and around 70 model activities.

In [21] we presented a guided exploration strategy that uses the inconsistencies in the model as hints for the selection criteria. In each iteration of the search process, each model element included in the inconsistencies that also

include the selected model element e are collected into a set V. Then the possible activations of the operations are collected and only those activations that include an element in V are selected. Between those activations, we use simple priorities for each operation. Since the execution of the operation modifies the model, the set V may change.

To demonstrate that this guidance increases the efficiency of the exploration, we use a breadth-first search exploration strategy that only uses priorities and imitates a fixed quick fix strategy encoded into a development environment. We show that the guided strategy finds possible quick fixes in less time and thus makes our approach applicable as an assistance for model editing.

9.2.2 Evaluation Environment and Method

The evaluation was carried out by adding inconsistencies to each process and running the quick fix generation approach independently. We performed measurements[4] five times for each test set including different total and local number of inconsistencies in the model.

The measurement of a given test set was done as follows: the inconsistent BPMN model is loaded into VIATRA2, the inconsistency rules and operations are added to the framework, the DSE framework is initialized and time measurement is started. Next, the exploration looks for three different solutions and gathers them in a list, once it is done the time measurement is stopped. Finally, the results are saved and the framework is disposed to return the environment to the initial state. We limited the measurement to 300000 states for test sets with one local inconsistency and 1000000 states for other test sets.

9.2.3 Evaluation of Results

The table in Figure 18 shows the results of our measurements using the case study models (with the size of the models given under their name). For each model we measured the performance for the given number of total and local inconsistencies. For each test set, we measured the number of visited states and the time of quick fix generation for both the guided and BFS exploration strategies. Finally, measurement results are given with the mean values along with deviations.

We made the following observations based on the results from the different models:

One local violation ($\#1{-}2, 5{-}10, 13{-}16$) In these test sets the guided strategy generated quick fixes in less than 2.2 seconds in all test sets except #14, where finding three different solutions takes 19 seconds. However, the BFS strategy

[4] All measurements were carried out on a computer with Intel Core i5 2.3 GHz processor, 2.5 GB DDR3 memory (Java heap space), Windows 8 Professional 64 bit, Eclipse 3.8, BPMN 1.2, VIATRA2 3.3

Project	#	Model (N/E/S/P)	|V(M)|	|V$_e$(M)|	Guided				BFS with priority			
					T [ms]	D_T	S	D_S	T [ms]	D_T	S	D_S
Procurement	1	Delivery	1	1	251	0,10	140	0,00	5 271	0,04	5 621	0,00
	2	(8/8/0/2)	1	1	219	0,11	109	0,00	4 196	0,03	2 814	0,00
	3	PurchaseOrder	5	3	73 879	0,03	51 460	0,00	-	-	-	-
	4	(14/15/0/1)	5	2	979	0,45	330	0,01	12 010	0,04	9 572	0,00
	5	PurchaseRequest	3	1	159	0,06	96	0,00	3 440	0,01	2 926	0,00
	6	(13/13/0/3)	3	1	78	0,01	107	0,01	1 749	0,35	2 916	0,00
Centralized register	7	Macro	5	1	301	0,41	133	0,00	4 533	0,11	3 328	0,00
	8	(16/12/4/1)	5	1	242	0,07	206	0,00	2 665	0,05	3 318	0,00
	9	Soliciting	4	1	666	0,15	261	0,00	8 881	0,13	4 810	0,00
	10	(20/25/0/1)	4	1	2 114	0,05	855	0,02	225 277	0,19	111 489	0,18
	11	Instructing	13	2	172 378	0,20	92 245	0,00	-	-	-	-
	12	(40/45/2/1)	13	3	748 179	0,06	340 090	0,00	-	-	-	-
	13	Deciding	3	1	544	0,13	113	0,00	3 857	0,03	2 810	0,00
	14	(9/10/0/1)	3	1	18 965	0,02	13 289	0,00	-	-	-	-
	15	Contracting	9	1	407	0,12	49	0,00	5 407	0,04	5 496	0,00
	16	(36/43/2/1)	9	1	1 117	0,05	158	0,00	12 608	0,02	5 511	0,00

Fig. 18 Quick fix generation for DSMLs ($N/E/S/P$: no. of nodes/edges/subprocesses/ pools, $|V(M)|$: total number of violations, $|V_e(M)|$: no. of violations for selected element, T: time [ms], D_T: standard deviation of time, S: no. of visited states, D_S: standard deviation of visited states)

performs at least one order of magnitude slower in most test sets. In test set #10 the exploration takes more than 100 seconds, while in test set #14 it is unable to find three solutions within the measurement limits.

Locality (#5, 12, 14, 15) The higher number of local violations for the selected element leads to slower fix generation, while the total number of violations in the model does not affect the performance significantly. For example, finding three solutions for test set #3 takes more than one minute and the exploration of more than 50000 states with the guided strategy, while in test set #12 the exploration takes more than 12 minutes. However, we found that even with complex DSMLs such as BPMN visiting one state only takes between $2ms$ and $4ms$, independently of the number of states explored before (at least in the scope of the measurements this held).

Model size ($\#3, 10 - 12, 14$) The guided strategy is less sensitive to the size of the instance model than the BFS strategy. This is a direct consequence of our guided approach, which applies operations on elements specified by local violations. The operations have a higher number of activations in larger models, which means that the BFS strategy has to try each activation, while the guided strategy can focus on local modifications.

To summarize, it is feasible to generate quick fixes for DSMLs with the guided strategy, in most cases without interrupting the editing process (i.e. with a response within 3 seconds). However, exploring the same design space with a simple BFS strategy is much slower and often infeasible.

9.2.4 Limitations

The guided exploration strategy of the quick fix generation assumes that each well-formedness constraint includes all elements in the violations that are related to the constraint violation. Additionally, if the set of possible operations is not representative of the model editing of the domain, then the quick fixes found by the exploration may not be helpful to the user (e.g. if the quick fix consists of deleting the neighborhood of the selected element).

9.3 Scenario 3: State Encoding Techniques

9.3.1 Test Sets Used for the Evaluation

We used the cloud case study metamodel for comparing the different state encoding techniques detailed in Sec. 8.2. The design space exploration uses the same set of operations as before, but we have removed the goals and any guidance from the design problem description to be able to measure the performance of the encoding techniques by exploring the complete design space of the cloud case study to a specific depth.

9.3.2 Evaluation Environment and Method

The evaluation was carried out using the EMF-based implementation architecture by starting the exploration from the empty model and traversing all reachable states with a depth-first search that has limited depth[5]. Each measurement was performed at least 10 times for each encoding technique.

9.3.3 Evaluation of Results

We present measurement results for the number of different visited states in the design space to a given depth (Figure 19) and the total time the exploration took to traverse the complete design space to a given depth (Figure 20).

	Number of visited states					
Search depth	**4**	**7**	**10**	**13**	**16**	**18**
Without state encoding	33	4 508	1 990 528	-	-	-
Partially precise (Op, OpIncr, Mod)	25	231	1 131	3 872	10 557	18 820
Fully precise (ModFull, Ind, IndIncr)	25	327	2 567	14 429	64 444	158 818

Fig. 19 Visited states by depth first search

We made the following observations based on the results in Figure 19 for the different encoding techniques:

[5] All measurements were carried out on a computer with Intel Core i7 3.4 Ghz processor, 2.5 GB DDR3 memory (Java heap size), Windows 7 Professional 64 bit, Eclipse 4.2

Often recurring states. We found that while the exploration traverses almost 2 million states even for a depth limit of 10, the number of different state codes is orders of magnitude lower, 1131 for the partially precise techniques and 2567 for the fully precise techniques. This also shows that most of the operations in this case study can be performed in different order and still reach the same state (similar to the interleaving of concurrent events in distributed systems).

Scaling to deeper search. We can see that while the number of visited states increases super-exponentially, the number of different states scales well even to a 18 depth. As discussed, the state codes are kept in memory while most of the search tree can be disposed during a depth-first search. This means that by identifying equivalent states, the memory needs of the exploration are lowered while the possible depth of the exploration is increased.

	Total time (Encoding time) [ms]					
Search depth	**4**	**7**	**10**	**13**	**16**	**18**
Without state encoding	27(0)	637(0)	271 138(0)	-	-	-
Op	22(1)	53(5)	192(14)	633(54)	1 830(175)	3 371(349)
OpIncr	19(1)	45(4)	186(12)	618(32)	1 713(75)	3 139(133)
Mod	24(3)	55(7)	197(27)	698(110)	1 995(348)	3 704(685)
ModFull	25(4)	74(12)	481(91)	2 849(634)	13 710(3 415)	35 537(9 371)
Ind	27(7)	88(33)	712(326)	4 744(2 518)	25 381(14 925)	70 340(43 642)
IndIncr	28(3)	75(12)	533(94)	3 139(626)	15 012(3 341)	38 923(9 226)

Fig. 20 Exploration and encoding time with different state encoding techniques

We also evaluate the results on the exploration time for the different techniques, shown in Figure 20:

Low overhead for state encoding. While the application of a state encoding technique means that the state code is calculated in each state, this calculation does not cause considerable overhead. Combined with the fact that most of the design space is not explored when a state is identified as a recurring state means that the total time of the exploration is much lower than without using encoding.

Reusable encoding techniques. We can see that using a metamodel dependent encoding technique (*ModFull*) can be twice as fast as a metamodel independent technique (*Ind*). On one hand, by defining the encoding on the specific metamodel, it is possible to take into account the characteristics of such instance models and optimize the encoding accordingly. On the other hand, a metamodel independent technique is reusable without modification for any design problem description while still performing in the same order of magnitude.

Incremental techniques. We found that updating the state codes incrementally results in a faster state encoding even with the increased overhead on changes. While in the case of operation encoding (*OpIncr*) the gain is minimal, incremental encoding of the instance model (*IndIncr*) can be almost twice

as fast as calculating the complete state code each time (*Ind*). The difference in gain is caused by the fact that even in greater depth, the length of the trajectory (the number of operations) is quite low, while the size of the instance model can get larger. The incremental model encoding would be especially useful when the initial state already contains a large instance model, while the model modifications of a given operation execution is relatively few.

9.3.4 Limitations

Four out of the total six state encoding techniques were specifically developed for the case study used for the evaluation. Therefore state encoding techniques that follow similar approach may not work well for different metamodels or problems. Additionally, the test sets used in the measurements did not include industrial size models, therefore the scalability of the techniques is not measured with regards to model size. It is possible that only an incremental technique would be acceptable in such cases.

9.4 Scenario 4: Viatra2-based versus EMF-based Implementation

9.4.1 Test sets used for the evaluation

The comparison between the Viatra2-based and EMF-based implementations of the DSE framework uses the cloud case study as well. Similarly to the first scenario, the exploration is performed with different goals. In the *Clustered DB Small* and *Clustered DB Big* test sets, the goal is to have two or three database nodes deployed on clustered servers, with an optimal trajectory consisting of 9 or 13 operations, respectively. In the *Simple Power On* test set, the goal is to have one application and one storage node deployed, with an optimal trajectory consisting of 14 operations.

9.4.2 Evaluation Environment and Method

The main goal of the measurements[6] was to get an overview about the performance characteristic of the simple EMF-based solution opposed to the Viatra2-based one. Both engines used a depth-first search exploration strategy and were executed without guidance and also with priorities. Note that state encoding is not used in these measurements.

Each test set was measured multiple times and recorded the length of the shortest solution found and whether it is the optimal one, the number of visited states for the given exploration (with a limit of 500000 states) and the total runtime of the exploration (in milliseconds).

[6] All measurements were carried out on a computer with Intel Core i5 2.5 GHz processor, 2.5 GB DDR3 memory (Java heap size), Windows 7 Professional 64 bit, Eclipse 4.2

9.4.3 Evaluation of Results

Fig. 21 presents the measurement results for the three scenarios.

Problem	Exploration strategy	Shortest trajectory found [length]	Visited states	Runtime [ms]
Clustered DB Small	VIATRA2 no guidance	9 (optimal)	435 217	42 968
	EMF no guidance	9 (optimal)	421 588	77 056
	VIATRA2 priority	9 (optimal)	206 864	21 056
	EMF priority	9 (optimal)	207 893	35 141
Clustered DB Big	VIATRA2 no guidance	15	500 000	50 612
	EMF no guidance	14	500 000	99 244
	VIATRA2 priority	16	500 000	59 176
	EMF priority	16	500 000	100 521
Simple Power on	VIATRA2 no guidance	16	500 000	51 085
	EMF no guidance	15	500 000	102 652
	VIATRA2 priority	16	500 000	55 151
	EMF priority	16	500 000	110 294

Fig. 21 Comparison of VIATRA2-based and EMF-based implementation

Our key observations from the results are the following:

Similar functionality. In all test sets, the shortest trajectory leading to a solution and the number of visited states is very close in both implementations. This shows that the DSE framework can be realized using different model representations and technologies. The slight difference between the results is due to the fact that there is random choice in the selection of the next activation.

Slower transaction handling. The exploration time is about $1.5-2$ times longer for the EMF-based solution than it is for the VIATRA2-based one. Based on further profiling, it is clear that the transaction handling of the VIATRA2 framework, that is specifically designed for model transformations, is much more efficient than the EMF Transactions framework (based on the times spent in the *lock/unlock/commit* calls during transaction handling). Additionally, change propagation in EMF about model manipulations involves a lot of operations with collections and those increase the exploration time.

9.4.4 Limitations

The EMF-based solution is only a proof of concept at the current state of development which needs further optimization and incorporation of additional techniques in order to reduce the state space and cut off unnecessary branches in the search tree. A more efficient transaction handling or just simply an efficient operation redo functionality (as there are no parallel access present which would require transactions) in EMF model management would result in better performance characteristics.

9.5 Summary

We selected these scenarios when evaluating our DSE framework to measure the applicability and performance from different perspectives. To summarize our main observations are:

- The results of the guided exploration using a dependency graph showed that our criteria-driven approach can reduce the design space further thus increasing the efficiency of the exploration, while also ensuring the optimal solutions are found early (Section 9.1).
- The quick fix generation for BPMN processes is a real application that demonstrated that a guided approach can support the editing process by acceptable response times (Section 9.2).
- The evaluation of state encoding has shown that the application of encoding techniques can reduce the number of visited states far more than the overhead that their calculation adds to the overall exploration runtime (Section 9.3).
- Finally, the comparison between the new EMF-based architecture and the previous, Viatra2-based implementation indicates that the framework can be applied to offer DSE over EMF models (Section 9.4).

It is important to note that in each scenario we also identified certain limitations (Sections 9.1.4,9.2.4,9.3.4 and 9.4.4) that require future work and additional evaluation on other case studies to further enhance the described exploration strategies and techniques.

10 Related Work

Model-driven guided design space exploration implemented over graph transformations is a novel idea in the field, however, similar approaches are not unprecedented in a broader research scope as described below.

The formal foundations of model-driven DSE used in our approach is described in [23]. In our previous work, [55] introduces the usage of occurrence vectors for hints in the optimization of GT systems, while [20] defines the dependency graph and the evaluation algorithm for arbitrary cut-off and selection criteria. We introduced the concept of quick fixes for DSMLs in [21]. Finally, this paper extends our ASE 2011 paper [22] that introduced the design of the guided DSE framework and guided exploration strategies based on the dependency graph as hint. The major contributions of the paper are (i) the formal definitions of the concepts of guided design space exploration, (ii) the new implementation architecture based on the Eclipse Modeling Framework, (iii) the description of different state encoding techniques and (iv) a detailed evaluation of our framework with multiple scenarios.

10.1 Graph Transformation based Approaches

The approach in [16] is similar to our approach as it also exploits the dependencies between GT rules using critical pair analysis. Here, GT systems are enhanced with control flow as well and the dependency information helps in discovering possible runtime problems. Model checking approaches to analyze GT systems are similar to our approach as they also perform state space exploration. One can categorize them as *compiled approaches* such as [44, 14, 6, 41, 4], which translate graphs and GT rules into off-the-shelf model checkers to carry out verification, and *interpreted approaches* like [36, 3, 27], which store system states as graphs and directly apply transformation rules to explore the state space, similarly to our approach. They place emphasis on exhaustive traversal (e.g. by optimizing the storage of individual states), while we aim at finding solutions quickly using guidance and hints.

10.2 Model-driven Design Space Exploration Techniques

The *DESERT* tool suite [33] provides model synthesis and constraint-based DSE for DSMLs with structural semantics using ordered binary decision diagrams for encoding and pruning the design space. [42] presents a generic DSE framework extending upon DESERT by supporting arbitrary analysis tools and includes model transformations for mapping design problems to intermediate and low-level formats.

The *OCTOPUS Toolset* [7] uses an intermediate representation for design problem specification and performs DSE using integrated analysis tools. It has been successfully applied to design software-intensive embedded systems [8].

The Gaspard *Framework* [17] is specifically focused on the design of massively parallel embedded systems and uses multilevel modeling where high-level UML models are automatically refined to allow design space exploration to evaluate performance characteristics through simulations.

These are all compiled approaches, where the design problems are specified as models and model transformations are applied to derive inputs for tools that will execute the exploration. These tools are often specifically designed for efficient exploration, however some information from the exploration (e.g. the explicitly explored states) may not always be available or back-translatable. On the contrary, we use model transformations as a way to perform the exploration itself. We proposed using information from analysis tools to guide this model transformation based exploration, which provides opportunity to apply conceptually different methods for guidance (e.g. dependency analyzer).

An efficient design space exploration approach was also presented built on the *FORMULA* framework in [24]. The design problem is described using domain-specific languages, exploration is done with symbolic execution and an SMT solver is used to check the satisfiability of a set of constraints generated by the symbolic execution.

Schätz et al. [43] developed an interactive, incremental process using declarative transformation rules for driving the exploration. The rules are modified interactively to improve the performance of the exploration, which can be considered as a guidance. However, the hints do not originate from analysis, contrary to our approach.

[29] presents a framework for the automatic deployment of software components to hardware architecture that uses design space exploration to find deployment alternatives that offer near-optimal reliability characteristics. The design problem consists of architecture models annotated with reliability-relevant properties, while the exploration uses an evolutionary algorithm to find possible alternatives. Similarly to our approach, global constraints prevent the exploration of infeasible solutions.

10.3 Guided Design Space Exploration Techniques

Existing DSE techniques sometimes use guidance information to reduce the number of alternatives that are evaluated.

Mohanty et al. [32] use "human in the loop" guidance in addition to symbolic search techniques for finding candidates, which are then analyzed using low-level simulation to find the final design. In [40], different chip design alternatives are evaluated using implementation specific information from earlier designs (e.g. cycle counts and energy consumption) or estimates by experienced designers. The hints are a collection of values, while guidance is used for selecting optimal mappings. These approaches use hints and guidance for reducing the design space, although hints originate from earlier experience or human interaction, not formal mathematical analysis of the design problem.

10.4 Structural Constraint Solving

Structural constraint solving aims to find object graphs that satisfy given constraints both on attributes and (object) structures by exploring a (usually) bounded number of possible object graphs. The CUTE [45] framework uses a combination of symbolic and concrete execution to derive path constraints for each separate execution paths. Java PathFinder [56] is based on Generalized Symbolic Execution that first introduced the use of model checkers for solving structural constraints.

KORAT [13] performs specification based testing by using a predicate representing the properties (constraints) of the desired output structures and explores the input state space of predicates using bounded exhaustive testing.

In all of these approaches hints are given in the form of explicit bounds on the size of the state space. However, they cannot restrict how solutions are achieved from the initial model, meaning that no constraints can be defined to hold on states visited during a solution trajectory. In our case it is supported by global constraints and explicit rule definitions, thus resulting in fundamentally different search strategies.

10.5 Metaheuristic based Search Strategies

There are several single-solution based metaheuristic techniques used in search based software engineering for the optimization of various design space exploration problems [47].

Guided local search based techniques [57] uses a predefined schema to inject penalties into their guidance functions. Simulated annealing based techniques [12] are similar to hill climbing approaches with the ability to avoid local optimum solutions by permitting moves to less fit states, with a decreasing probability over time.

Common in these techniques that they use an iterative traversal algorithms to improve candidate solutions with regards to their measure of quality (e.g., guidance function). However, with no hints available about the global optimum these techniques rely only on neighboring states when selecting the next step on the contrary our approach uses hints like the occurrence vector for finding the optimum solution.

10.6 State encoding techniques in graph-based validation tools

State encoding techniques are often used in graph-based validation tools where the exhaustive exploration of all possible graphs also introduces the challenge of storing a large number of graphs and identifying equal or isomorphic states.

The *GROOVE* model checker [36] stores changes or *deltas* between states similarly to our incremental state encoding techniques [38]. The idea was proposed earlier by Mens [30], while the *GRAS database* [26] has also used this approach. GROOVE also three steps for checking equality, by introducing graph certificates (similar to state codes), isomorphism checking (with an improved approach presented in [39]) and graph equality.

Godefroid et al. [18] presented a state caching method that identifies the possible interleaving of concurrent executions and ensures that most states are visited only once during the exploration. In [46] hash compaction is combined with state caching to further increase efficient storing of the explored states.

11 Conclusion and Future Work

Design space exploration can be used in Model-driven engineering for problems that involve searching through a large number of possible alternatives and selecting solutions that satisfy design goals and global constraints.

Guided DSE exploration uses hints to reduce the number of states traversed when searching for solutions. Hints are used (i) to identify dead-end states (cut-off criteria) and (ii) to order applicable rules in a given state (selection criteria).

In the current paper, we defined a model-driven framework for guided DSE, which uses rule dependency and occurrence vectors as hints for the exploration

strategy. Evaluation of the exploration strategies using a cloud configuration problem showed that our criteria-driven approach can reduce the design space further thus increasing the efficiency of the exploration.

Guided exploration is also applied in a real application for generating quick fixes to help business process designers in correcting violations of well-formedness constraints. The evaluation results show that guidance is required to achieve response time acceptable to use during the editing process.

We presented a new implementation architecture that builds on the de facto industrial standard Eclipse Modeling Framework and the EMF-IncQuery incremental query evaluation framework. We have evaluated the new architecture against the previous version of our DSE framework based on the VIATRA2 model transformation framework and argue that even in an early phase it performs well.

Our DSE framework uses state encoding to identify states that were already explored earlier in the process and thus further reduce the number of traversed states. We specified six encoding techniques that are evaluated on the cloud reconfiguration case study. The evaluation showed that these techniques can significantly reduce the design space.

Future work. We are actively working on improving our EMF-based architecture and are planning to explore techniques to further increase the scope of guided exploration. The dependency graph can be further extended with additional information such as including the goals and constraints in the graph or adorning dependency relations with the cause of the dependency.

We are investigating ways for better reusing the design space when exploring subsequent occurrence vectors to identify states where the traversal should continue. We are also working on defining problem-specific criteria and specialized algorithms to increase the efficiency of the approach.

We are also interested in evaluating similarity of solutions based on automorphism groups as an extension to state encoding and guide the exploration to find dissimilar solutions, as suggested by one of our reviewers.

Finally, we are planning to apply design space exploration to new problems that we encounter in our cooperation with industrial partners.

Acknowledgements The authors would like to thank their students Miklós Földényi and Tamás Szabó, who helped in the evaluation of state encoding techniques and comparing the VIATRA2 and EMF-based DSE framework implementations and the anonymous reviewers for their excellent comments and suggestions.

References

1. Atkinson C, Kühne T (2003) Model-driven development: A metamodeling foundation. IEEE Softw 20(5):36–41, DOI 10.1109/MS.2003.1231149
2. AUTOSAR Consortium (2013) The AUTOSAR Standard. http://www.autosar.org/

3. Baldan P, König B (2002) Approximating the behaviour of graph transformation systems. In: Proc. ICGT 2002, Springer, LNCS, vol 2505, pp 14–29
4. Baresi L, Spoletini P (2006) On the Use of Alloy to Analyze Graph Transformation Systems. In: Graph Transformations, LNCS, vol 4178, Springer, pp 306–320
5. Baresi L, Heckel R, Thöne S, Varró D (2006) Style-based modeling and refinement of service-oriented architectures. Journal of Software and Systems Modelling 5
6. Baresi L, Rafe V, Rahmani AT, Spoletini P (2008) An efficient solution for model checking graph transformation systems. ENTCS 213
7. Basten T, van Benthum E, et al (2010) Model-driven design-space exploration for embedded systems: The octopus toolset. In: Margaria T, Steffen B (eds) Leveraging Applications of Formal Methods, Verification, and Validation, LNCS, vol 6415, Springer, pp 90–105
8. Basten T, Hendriks M, Trčka N, Somers L, Geilen M, Yang Y, Igna G, de Smet S, Voorhoeve M, van der Aalst W, et al (2013) Model-driven design-space exploration for software-intensive embedded systems. In: Model-Based Design of Adaptive Embedded Systems, Springer, pp 189–244
9. Bergmann G, Ökrös A, Ráth I, Varró D, Varró G (2008) Incremental pattern matching in the viatra model transformation system. In: Proceedings of the Third International Workshop on Graph and model transformations, ACM, pp 25–32
10. Bergmann G, Hegedüs Á, Horváth Á, Ujhelyi Z, Ráth I, Varró D (2012) Integrating efficient model queries in state-of-the-art EMF tools. In: 50th International Conference on Objects, Models, Components, Patterns (TOOLS Europe), Springer, Prague, LNCS, DOI 10.1007/978-3-642-30561-0_1
11. Bergmann G, Ráth I, Szabó T, Torrini P, Varró D (2012) Incremental pattern matching for the efficient computation of transitive closures. In: Sixth International Conference on Graph Transformation, Bremen, Germany
12. Bouktif S, Sahraoui H, Antoniol G (2006) Simulated annealing for improving software quality prediction. In: Proc. of GECCO '06, ACM
13. Boyapati C, Khurshid S, Marinov D (2002) Korat: Automated testing based on Java predicates. In: International Symposium on Software Testing and Analysis (ISSTA), ACM
14. Edelkamp S, Jabbar S, Lluch-Lafuente A (2006) Heuristic search for the analysis of graph transition systems. In: Proc. ICGT 2006, Springer, LNCS, vol 4178
15. Ehrig H, Engels G, Kreowski HJ, Rozenberg G (eds) (1999) Handbook on Graph Grammars and Computing by Graph Transformation, vol 2: Applications, Languages and Tools. World Scientific
16. Ermel C, Gall J, Lambers L, Taentzer G (2011) Modeling with plausibility checking: Inspecting favorable and critical signs for consistency between control flow and functional behavior. In: Proc. FASE '11, Springer, LNCS

6603
17. Gamatié A, Le Beux S, Piel É, Ben Atitallah R, Etien A, Marquet P, Dekeyser JL (2011) A model-driven design framework for massively parallel embedded systems. ACM Transactions on Embedded Computing Systems (TECS) 10(4):39
18. Godefroid P, Holzmann GJ, Pirottin D (1995) State-space caching revisited. Formal Methods in System Design 7(3):227–241
19. Heckel R, Küster JM, Taentzer G (2002) Confluence of Typed Attributed Graph Transformation Systems. In: Proc. ICGT 2002. LNCS, Springer
20. Hegedüs Á, Horváth Á, Varró D (2010) Towards guided trajectory exploration of graph transformation systems. ECEASST 40, Proc. of PNGT '10
21. Hegedüs Á, Horváth Á, Ráth I, Branco MC, Varró D (2011) Quick fix generation for DSMLs. In: IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC, IEEE Computer Society, DOI 10.1109/VLHCC.2011.6070373
22. Hegedüs Á, Horváth Á, Ráth I, Varró D (2011) A model-driven framework for guided design space exploration. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE Computer Society, pp 173 –182, DOI 10.1109/ASE.2011.6100051
23. Horváth Á, Varró D (2011) Dynamic constraint satisfaction problems over models. Software and Systems Modeling 10.1007/s10270-010-0185-5
24. Jackson EK, Simko G, Sztipanovits J (2013) Diversely enumerating system-level architectures. In: EMSOFT, pp 1–10
25. Kang E, Jackson EK, Schulte W (2010) An approach for effective design space exploration. In: Calinescu R, Jackson EK (eds) Monterey Workshop, Springer, LNCS, vol 6662, pp 33–54
26. Kiesel N, Schuerr A, Westfechtel B (1995) GRAS, a graph-oriented (software) engineering database system. Information Systems 20(1):21 – 51
27. König B, Kozioura V (2006) Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In: TACAS, pp 197–211
28. Kühne T (2006) Matters of (meta-) modeling. Software and Systems Modeling 5:369–385, 10.1007/s10270-006-0017-9
29. Meedeniya I, Buhnova B, Aleti A, Grunske L (2011) Reliability-driven deployment optimization for embedded systems. Journal of Systems and Software 84(5):835–846
30. Mens T (1999) A formal foundation for object-oriented software evolution. PhD thesis, Vrije Universiteit Brussel
31. Mens T, Kniesel G, Runge O (2006) Transformation dependency analysis - a comparison of two approaches. In: Langages et Modèles à Objets (LMO 2006)
32. Mohanty S, Prasanna VK, Neema S, Davis J (2002) Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. SIGPLAN Not 37:18–27

33. Neema S, Sztipanovits J, Karsai G, Butts K (2003) Constraint-based design-space exploration and model synthesis. In: Alur R, Lee I (eds) Embedded Software, LNCS, vol 2855, Springer, pp 290–305
34. Object Management Group (2013) Business Process Model and Notation (BPMN) Version 1.2. http://www.omg.org/spec/BPMN/1.2/
35. Rensink A (2003) Towards model checking graph grammars. In: Workshop on Automated Verification of Critical Systems (AVoCS), Technical Report DSSE–TR–2003–2, Citeseer, pp 150–160
36. Rensink A (2004) The GROOVE simulator: A tool for state space generation. In: Applications of Graph Transformations with Industrial Relevance (AGTIVE), Springer, LNCS
37. Rensink A (2004) Representing first-order logic using graphs. In: International Conference on Graph Transformations (ICGT), LNCS 3256, Springer, pp 319–335
38. Rensink A (2005) Time and space issues in the generation of graph transition systems. In: Mens T, Schürr A, Taentzer G (eds) Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004), Electronic Notes in Theoretical Computer Science, vol 127, pp 127–139
39. Rensink A (2007) Isomorphism checking in GROOVE. Electronic Communications of the EASST 1
40. Ristau B, Limberg T, Fettweis G (2008) A mapping framework for guided design space exploration of heterogeneous MP-SoCs. In: Proc. of the conf. on Design, automation and test in Europe, ACM, DATE '08
41. dos Santos OM, Dotti FL, Ribeiro L (2004) Verifying object-based graph grammars. ENTCS 109:125–136
42. Saxena T, Karsai G (2010) MDE-based approach for generalizing design space exploration. In: Petriu D, Rouquette N, Haugen y (eds) Model Driven Engineering Languages and Systems, LNCS, vol 6394, Springer, pp 46–60
43. Schatz B, Holzl F, Lundkvist T (2010) Design-space exploration through constraint-based model-transformation. In: Engineering of Computer Based Systems (ECBS), pp 173 –182
44. Schmidt Á, Varró D (2003) CheckVML: A tool for model checking visual modeling languages. In: Proc. UML 2003, Springer, LNCS, vol 2863
45. Sen K, Marinov D, Agha G (2005) CUTE: a concolic unit testing engine for C. SIGSOFT Softw Eng Notes 30:263–272
46. Stern U, Dill DL (1996) Combining state space caching and hash compaction. Methoden des Entwurfs und der Verifikation digitaler Systeme 4:81–90
47. Talbi EG (2009) Metaheuristics: From Design to Implementation. Wiley
48. The Eclipse Project (2012) Eclipse Modeling Framework. http://www.eclipse.org/emf
49. The Eclipse Project (2012) EMF Model Query 2. http://wiki.eclipse.org/EMF/Query2
50. The Eclipse Project (2012) EMFT Search. http://www.eclipse.org/modeling/emft/?project=search

51. The Eclipse Project (2012) MDT OCL. `http://www.eclipse.org/modeling/mdt/?project=ocl`
52. Varró D, Balogh A (2007) The model transformation language of the VIATRA2 framework. Science of Computer Programming 68(3):214–234
53. Varró D, Pataricza A (2003) VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Software and Systems Modeling 2(3):187–210
54. Varró D, Varró-Gyapay S, Ehrig H, Prange U, Taentzer G (2006) Termination analysis of model transformations by petri nets. In: Proc. ICGT 2006, Springer, Brazil, LNCS, vol 4178
55. Varró-Gyapay S, Varró D (2006) Optimization in Graph Transformation Systems Using Petri Net Based Techniques. ECEASST 2, Proc. of PNGT '06
56. Visser W, Păsăreanu CS, Khurshid S (2004) Test input generation with Java PathFinder. SIGSOFT Softw Eng Notes 29(4):97–107, DOI http://doi.acm.org/10.1145/1013886.1007526
57. Voudouris C, Tsang EPK, Alsheddy A (2010) Effective application of guided local search. Wiley Encyclopedia of Operations Research and Management Science