Graph Transformation Planning via Abstraction

Steffen Ziegert
Department of Computer Science
University of Paderborn
steffen.ziegert@uni-paderborn.de

Modern software systems increasingly incorporate self-* behavior to adapt to changes in the environment at runtime. Such adaptations often involve reconfiguring the software architecture of the system. Many systems also need to *manage* their architecture themselves, i.e., they need a *planning component* to autonomously decide which reconfigurations to execute to reach a desired target configuration. For the specification of reconfigurations, we employ graph transformations systems (GTS) due to the close relation of graphs and UML object diagrams. We solve the resulting planning problems with a planning system that works directly on a GTS. It features a domain-independent heuristic that uses the solution length of an abstraction of the original problem as an estimate. Finally, we provide experimental results on two different domains, which confirm that our heuristic performs better than another domain-independent heuristic which resembles heuristics employed in related work.

1 Introduction

Modern software systems increasingly incorporate self-* behavior to adapt to changes in the environment at runtime. Such adaptations often involve reconfiguring the software architecture of the system. However, the ability to *perform* reconfiguration might not be enough. Many systems also need to *manage* their architecture themselves, i.e., they need a *planning component* to autonomously decide which reconfigurations to execute to reach a desired target configuration [KM07].

Graph transformations systems (GTS) [EEPT06] have been deemed a suitable formalism for the specification of reconfigurations due to their close relation to graphs and UML object diagrams [LM98, WF02]. They have been used for the verification of reconfiguration operations in various approaches, e.g., [BBG^{+}06, Ren08]. Planning of architecture reconfiguration has also been covered before, e.g., in [EW11] for coordinating behavior in cyber-physical systems and in [TK11] for planning a self-healing process in automotive systems. The involved planning problems are usually solved by one the following two approaches: either a translation into a dedicated planning language, i.e., the Planning Domain Definition Language (PDDL) [FL03], is performed or a planning system is developed that works directly on a GTS.

Both approaches have their drawbacks. Translation-based approaches suffer from a different expressiveness of GTSs and PDDL: while the creation and deletion of objects is a fundamental feature of GTSs, there is no such thing in PDDL. By not allowing to de-/instantiate objects, PDDL maintains a finite state space. To handle de-/instantiation of objects nevertheless, a modeling workaround can be used that declares all uninstantiated objects in the initial state, but uses a predicate to state their actual existence, as in [TK11, ZW13]. However, the workaround is based on the assumption that a maximal number of objects is known beforehand or can be deduced from the GTS.

Planning systems for GTSs on the other hand are not highly evolved. Up until today, there are only few systems that use domain-independent heuristics to guide their search through the state space of a

To appear in EPTCS.

GTS. They employ simple domain-independent heuristics that compute values for the structural similarity of the current configuration and the target configuration. Multiple such *similarity-based* heuristic functions are presented in [EJLL06]. Several of them are different variants of counting the nodes and edges that have to be created or deleted to reach the target configuration, e.g., they differ in whether or not it is allowed to rely on the identity of nodes and edges, and then using this number as a distance measure. Another slightly different variant of such a heuristic has been presented in [Sni11].

In this paper, we propose a new GTS-based planning system. It employs a domain-independent heuristic function that can be used in different search algorithms. The heuristic function computes the solution length of an abstraction of the original problem as an estimate for a given state. The abstract problem reinterprets certain parts of the rules' application conditions and is thus easier to solve than the original problem. As part of our contribution, we compare the performance of our heuristic against the performance of a similarity-based heuristic.

The next section introduces graph transformations systems and the notion of planning problems on graph transformations systems. In Section 3 we present an application example, which is used to explain our heuristic approach in Section 4. The evaluation of our approach is given in Section 5. We discuss related work in Section 6 before concluding in Section 7.

2 Planning with Graph Transformations

A graph transformation system (GTS) consists of a set of graph transformation rules and an initial graph. The graph transformation rules can be applied to the initial graph and its resulting successor graphs to construct the state space of the GTS. The underlying theory of GTS is based on graphs and graph morphisms.

Definition 1 (Graph, Graph Morphism)**.** *A* graph $G = (V_G, E_G, src_G, tgt_G)$ *consists of a set of nodes* V_G*, a set of edges* E_G*, and source and target functions* $src_G, tgt_G : E_G \to V_G$*. A* graph morphism $f : G \to H$ *between two graphs is a pair of mappings* $f = (f_E, f_V)$ *with* $f_E : E_G \to E_H$ *and* $f_V : V_G \to V_H$ *such that* $f_V \circ src_G = src_H \circ f_E$ *and* $f_V \circ tgt_G = tgt_H \circ f_E$*. A graph morphism* $f = (f_E, f_V)$ *is* injective *if* f_E *and* f_V *are injective.*

A graph morphism is a mapping of nodes and edges of one graph to nodes and edges of another graph such that the source and target nodes of edges are preserved. Such morphisms are used in graph transformation rules to define which nodes and edges are created, deleted, or preserved when the rule is applied to a graph.

Definition 2 (Graph Transformation Rule)**.** *A* graph transformation rule $p = (L, R, r)$ *consists of two graphs* L *and* R*, called* left-hand side (LHS) *and* right-hand side (RHS)*, and an injective partial graph morphism* $r : L \to R$*, called* rule morphism*. Given a graph transformation rule* p *and a match* $m : L \to G$ *of its LHS into a* host graph G*, the* direct derivation *from* G *with* p *at* m*, written* $G \overset{p,m}{\Longrightarrow} H$*, is the pushout of* r *and* m *in* $Graph^P$*, the category of graphs and partial graph morphisms, as shown below. [EHK*$^+$*97]*

$$\begin{tikzcd}
L \arrow[r, "r"] \arrow[d, "m"'] \arrow[dr, phantom, "(PO)"] & R \arrow[d, "m'"] \\
G \arrow[r, "r'"] & H
\end{tikzcd}$$

Whether a graph transformation rule can be applied to a graph depends on whether a match of its LHS to the graph can be found. Multiple such matches result in multiple direct derivations and thus in multiple successor graphs. When a rule is applied that specifies the deletion of a node, dangling edges might occur. The above definition follows the single pushout approach (SPO), which results in dangling edges being deleted as well. Furthermore, we employ injective matchings: each node (and edge) of the rule maps to a distinct node (or edge) of the graph.

To restrict the applicability of a rule, negative application conditions (NAC) can be used that forbid specific graph structures from being present in the graph.

Definition 3 (Negative Application Condition)**.** *Let $p = (L, R, r)$ be a graph transformation rule, G a graph, and $m : L \to G$ a match. A* negative application condition (NAC) *is a tuple $NAC = (N, n)$ with $n : L \to N$ and n being injective. If $\neg\exists q : N \to G$ such that $q \circ n = m$, then m* satisfies *NAC, written $m \models NAC$.*

We also write $p = (L, R, r, \mathscr{N})$, where $\mathscr{N}$ is a set of NACs, when we want to explicitly refer to the NACs of a graph transformation rule p. Now we have everything we need for the definition of graph transformation systems.

Definition 4 (Graph Transformation System)**.** *A* graph transformation system *$S = (\mathscr{R}, G_0)$ consists of a set of graph transformation rules $\mathscr{R}$ and an initial graph G_0.*

A planning problem on a graph transformation system also involves a graph pattern, which defines valid target configurations. The specification of graph patterns also support NACs.

Definition 5 (Graph Pattern)**.** *A* graph pattern *$P = (L, \mathscr{N})$ consists of a graph L and a set of NACs $\mathscr{N}$ where each $NAC \in \mathscr{N}$ is a tuple $NAC = (N, n)$ with $n : L \to N$ and n being injective.*

Having a means of specifying valid target configurations of a planning problem, we can now define the planning problem on a graph transformation system.

Definition 6 (Planning Problem)**.** *A* graph transformation planning problem *$\mathscr{P} = (\mathscr{R}, G_0, P_{tgt})$ consists of a set of graph transformation rules $\mathscr{R}$, an initial graph G_0, and a target graph pattern $P_{tgt} = (L_{tgt}, \mathscr{N}_{tgt})$. A* plan *for $\mathscr{P}$ is a sequence of direct derivations $G_0 \Rightarrow \ldots \Rightarrow G_k$ such that the target graph pattern P_{tgt} has a match in G_k.*

3 Application Example

We consider the reconfiguration of Electronic Control Units (ECUs) in automotive systems as an application scenario. In current development, software components are deployed on ECUs at design-time. The AUTOSAR consortium proposed a component-based software architecture standard[1] for the development of ECU software. Following the AUTOSAR standard, a Runtime Environment (RTE) is generated out of a predefined set of components. The RTE acts as a middleware that connects the software components with Basic Software (BSW) that controls the hardware. We expect that in the future software components can be deployed at ECUs at runtime. This allows to react to hardware failures or adapt to low levels of energy by runtime reconfiguration. Such runtime reconfigurations are executed according to reconfiguration plans, which are computed in soft real-time. In the context of our application example, a planning task might be to shut down an ECU due to a recognized hardware failure or reboot an ECU due to a re-deployment of the RTE middleware after a software upgrade.

Consider the graph in Figure 1 as the initial state for the planning problem. There are two ECUs, `n1` and `n2`, and two software components, `c1` and `c2`. For each component there is an instance that is running on one of the ECUs.

---

[1] AUTOSAR specifications are available at `http://www.autosar.org/index.php?p=3`.

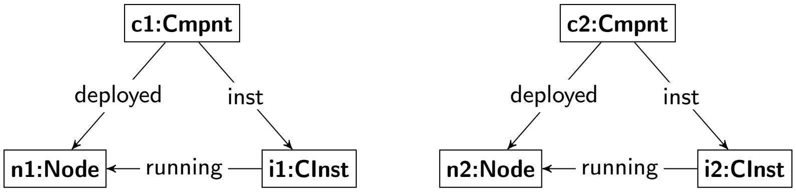
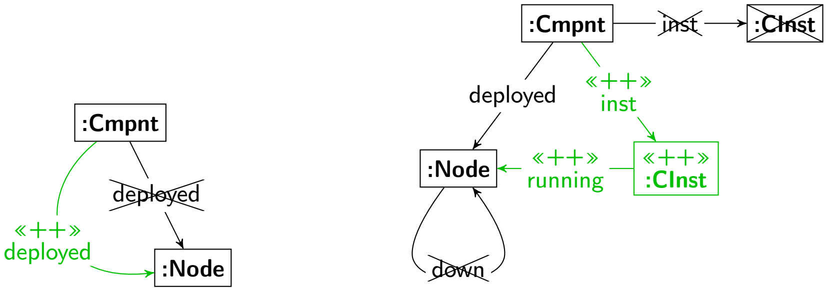
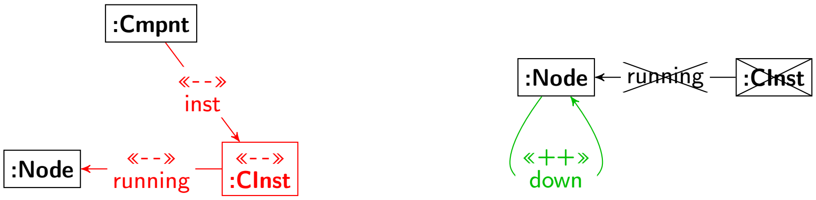


Figure 1: Initial configuration



(a) `deployComponent` (b) `createInstance`



(c) `destroyInstance` (d) `shutdownNode`

Figure 2: Graph transformation rules

Figures 2a to 2d show the rules that are relevant for our application scenario. We use the stereotype «++» to denote elements that are only available in the RHS and «--» to denote elements that are only available in the LHS. Crossed out elements denote NACs. If crossed out elements are adjacent to each other, they belong to the same NAC. In Figure 2a component data is deployed on an ECU such that the component can be instantiated. Figure 2b specifies the creation of a component instance. In Figure 2c an instance that is running on an ECU is destroyed. Figure 2d specifies shutting down an ECU if no instances are running on it.

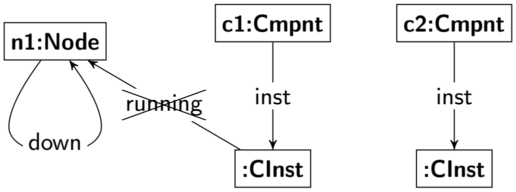


Figure 3: Target graph pattern

The target is specified as a graph pattern in Figure 3. It states that ECU `n1` should be shut down and components `c1` and `c2` should both be instantiated. Since a component instance of `c1` is running on `n1` in the initial state, we added a NAC disallowing component instances of `c1` to run on `n1` in goal states.

A sample plan arriving in a valid target configuration deploys component `c1` at ECU `n2` in the first step, then destroys the component instance `i1`, then shuts down `n1`, and at last creates a new instance of `c1` that runs on `n2`. To compute such plans, our planning system employs a heuristic function that solves for each encountered state in its state space an abstraction of the original problem.

4 Solving the Planning Problem via Abstraction

Our technique to solve the GT planning problem is an informed search in the state space of the GTS. The main feature of the system is the heuristic function that tells the search algorithm, e.g., A*, Best-First (BF), or Enforced Hill-Climbing (EHC), which state to expand next. The heuristic function itself solves a planning problem that is an abstraction of the original problem considered from a given state. Information gained during solving the abstract problem is used to guide the search of the original problem.

4.1 Abstract State Sequences

The abstraction used by the heuristic function combines two ideas. The first idea is to *relax* the applicability of a rule, i.e., to apply only changes that enable subsequent rule applications, but discard changes that disable subsequent rule applications. In essence, this is realized by not deleting elements when rules are applied (to relax LHS matching) and the use of labels, which allow to reinterpret (and thus relax) NAC matching. The second idea is to apply all applicable transformations *in parallel* to construct the next (abstract) state, instead of choosing one state to expand next. This results in a linear (abstract) state space and thus allows to solve the abstract problem efficiently. Two applicable transformations cannot be in conflict with each other due to the applied relaxation.

Figure 4 shows the abstract state sequence from the initial state of the problem to the first state that satisfies the target graph pattern. Each transition corresponds to one parallel and relaxed application of all applicable transformations. In the first transition, all reconfigurations deploying components are executed in parallel, as well as all reconfigurations destroying component instances. In the second transition, new instances are created and both ECUs are shutdown.

The relaxation used during the construction of the abstract state sequence is supposed to discard the deletion of elements and reinterpret certain parts of the rules' application conditions. Therefore, each element that is supposed to be deleted according to the rule morphism of an applicable transformation, is maintained in the successor graph G_{succ}, but is labeled with *deleted* instead. Each element that is supposed to be created, is added to G_{succ} as usual and labeled with *created*. Elements labeled with *deleted* and *created* correspond to the dashed and dotted elements of Figure 4, respectively. Labeling these elements gives the option to disregard them when considering the applicability of transformations in later iterations. This has happened in the second transition of Figure 4 with the component instances `i1` and `i2`: since they have been labeled as *deleted* by the first transition, new instances can be created during the second transition by applying the `createInstance` rule. In general terms, in order for a transformation to be considered applicable (despite a matching NAC), there has to be at least one element that is labeled with *deleted* or *created* in the part of the host graph that is matched by the NAC. If there are multiple NACs or a NAC has multiple matches, then each NAC match has to contain at least one element that is labeled with *deleted* or *created* for the transformation to be applicable.

The generation of the abstract state sequence stops as soon as the abstract planning problem is solved, i.e., the abstract state sequence reached a state that satisfies the target graph pattern. However, it is also

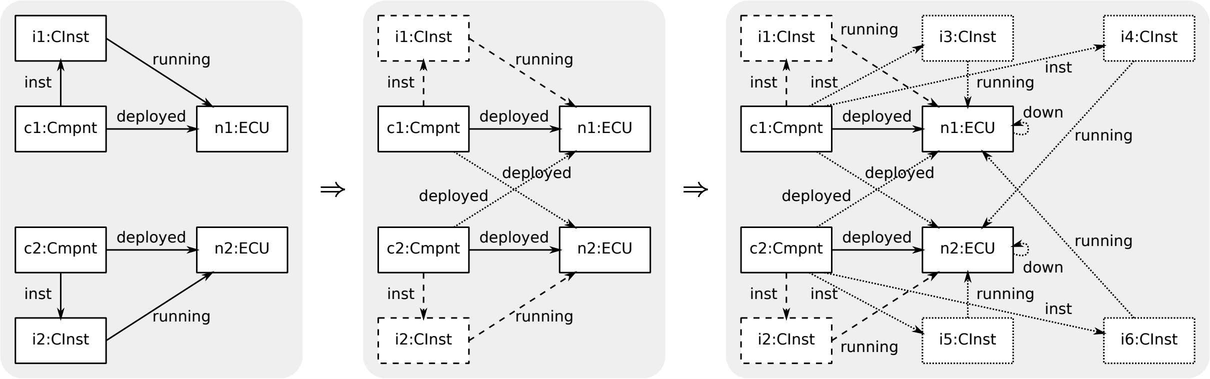


Figure 4: An abstract state sequence

possible that there is no path from the initial abstract state to a state that satisfies the target graph pattern. Therefore, we also abort the generation of the abstract state sequence after we generated x successor states, where x is two times the heuristic value of the initial state of the concrete planning problem.

4.2 Heuristic Values

Our planning system performs a state space exploration by successively choosing a state and expanding it, i.e., applying each rule at each possible match to generate its successor states. To decide which state to expand next, the system calculates a heuristic value for each unexpanded state. Calculating a heuristic value for a state involves generating the abstract state sequence starting in this state until we reach a state that satisfies the target graph pattern.

Having constructed the abstract state sequence, a naive idea would be to use its length as heuristic estimate. Although the abstract state sequence is expected to be shorter for states which are near to a goal state and longer for states which are further away from a goal state, this value is still rather imprecise. A better idea is to give the approximate number of *individual* reconfigurations needed for reaching the goal state. However, we cannot simply count all applied reconfigurations per transition to calculate this number, because this would include a lot of reconfigurations that were *not* needed to reach the goal state. The reconfigurations that *were* needed to reach the goal state are called a *relaxed plan* and their number is called the *length* of the relaxed plan.

Our approach to calculate this number incorporates rule application information into the newly created elements of each successor graph. Each created element is labeled with information about the transformation that caused its creation. This label consists of the iteration number of the successor graph creation loop, the name of the applied rule, and a distinct identifier for the match of the rule to the host graph. As an example, the `deployed` edge from component `c1` to ECU `n2` in Figure 4 is labeled with *(iteration #1, '`deployComponent`', match #1)*.

When the goal match is found, we can count the number of distinct rule application labels that are contained in the elements of the goal match. This number is the number of transformations needed to create the elements in the goal match. However, these labels contains only labels of elements that appear *directly* in the goal match. It does not yet contain labels of elements that were needed to *arrive* at the goal match. An example for this is the label of the aforementioned `deployed` edge from component `c1` to ECU `n2`. While the `deployed` edge is not contained in the goal match, its creation during the first transition was necessary for the application of *another* transformation during the second transition to create an element in the goal match. In this example, the application of `createInstance` that creates component instance `i4` during the second transition required the `deployed` edge from `c1` to `n2`. Our approach includes labels of such elements, i.e., elements that were needed to arrive at the goal match, when counting the rule application labels in the goal match: each element created by a transformation—in addition to its own label—inherits the labels of all elements in the LHS match of the rule application. In the example of Figure 4, the rule application label of the `deployed` edge created during the first transition is propagated to the newly created instance `i4` during the second transition.

Elements that have been marked as *deleted* are handled similarly. For example, the component instance `i1` receives the rule application label *(iteration #1, '`destroyInstance`', match #1)* when it is marked as *deleted* by the application of the `destroyInstance` rule. Labels of elements being marked as *deleted* are propagated to newly created (or deleted) elements if the labeled element is contained in a NAC match, e.g., the label of `i1` is propagated to the `down` edge at ECU `n1` when `shutdownNode` is applied during the second transition. Note, that elements being marked as *deleted* inherit labels in the same manner as elements marked as *created*: they inherit labels of created elements if contained in the LHS match and labels of deleted elements if contained in the NAC match. By inheriting the labels of other elements, the elements in the goal match do not only contain labels of transformations that directly created them, but also about all prior transformations that made their creation possible (whether by means of element creation or deletion).

The heuristic value is now simply defined as the number of labels that are attached to all elements in the goal match. If there are multiple goal matches, we use the smaller value. Applied to the example of Figure 4, the goal match containing `i4` and `i2` would result in a heuristic value of 4. There are other goal matches, e.g., containing `i5` instead of `i2`, but they would result in larger heuristic values.

5 Evaluation

Heuristics We compared our heuristic function (h_{abs}) against a similarity-based heuristic which resembles heuristic functions employed in related work [EJLL06, Sni11]. Both heuristics have been implemented in GROOVE [KR06], a tool for state space generation and verification of graph grammars, to conduct the experiments.

The similarity-based heuristic (h_{sim}) counts the number of nodes and edges that exist in both the current configuration and the target configuration. It relies on the types of nodes and edges to judge whether a node or edge is counted as existing. More precisely, it puts the type of each node and edge of a configuration into a multiset and takes the cardinality of the intersection of the current configuration's multiset and the target configuration's multiset as a similarity measure. The heuristic value is then defined as the additive inverse of this measure.

Search algorithms Both heuristic functions are evaluated in combination with greedy best-first and a variant of enforced hill-climbing.

Greedy best-first (GBF) is a well-known search algorithm for informed search. It uses a closed list and an open list of states. After expanding a state, i.e., all successor states have been generated, the state is placed in the closed list. For each new successor state found, its heuristic value is computed and it is placed into the open list. The decision which state to expand next is based on the heuristic values of the states in the open list.

Enforced hill-climbing (EHC) is a local search algorithm. In each iteration it performs a breadth-first search from the current state until it finds a state with a better heuristic value. When such a state is found, it updates the current state and continues with the next iteration. We use a modified EHC that applies greedy best-first search instead of breadth-first search in each iteration.

Problem domains We used two problem domains for our experiments: Blocks World and ECUs.

Blocks World is a classical problem domain in the area of AI planning. It constitutes of a table with a set of cubes that can be stacked upon each other. A cube can only be moved if there are no other cubes on top of it and there is only one arm that can hold a cube, i.e., two cubes cannot be moved at the same time. Finding an optimal solution in this domain has been shown to be NP-hard [GN92].

The ECUs domain functions as explained in Section 4. In contrast to the Blocks World domain which does not involve the instantiation of objects, the ECUs domain contains rules creating new objects.

Experiment setup For the Blocks World domain, we used 8 different problem sizes (4, 6, 8, 10, 12, 14, 16, and 18 blocks) and 4 different problem instances (cross product of 2 random initial and target configurations) per problem size.

For the ECUs domain, we used 4 different problem sizes (2, 3, 4, and 5 ECUs), each with 4 different problem instances. Two of these problem instances had the same number of component instances running in the initial configuration as ECUs were available. The other two problem instances used an additional component instance. Each target configuration specified every second ECU (rounding down at odd numbers of ECUs) to be shut down.

The experiments were conducted on a Dual Intel Xeon E5520 compute server with 16 (virtual) cores running at 2.27GHz. Each experiment was given 4 cores and 4GB of RAM. If no plan could be computed within 20 minutes, the job was terminated.

Results First, we give an overview of the number of generated states for each combination of heuristic function and search algorithm. The number of generated states means the number of states that have been found during the generation of the concrete planning problem's state space. This number is a measure for how well the employed heuristic prunes the state space. It does not include abstract states generated by h_{abs}. Figure 5 shows a histogram of the average number of states for the BlocksWorld domain, Figure 6 for the ECUs domain. Note the logarithmic scale in both histograms. With increasing problem size h_{abs} makes its superiority clear. Combinations with h_{sim} failed to provide a solution within 20 minutes for the problems of size 10 blocks and above (on the BlocksWorld domain) and 5 ECUs (on the ECUs domain). There is no significant difference in performance between GBF and EHC.

Considering the average planning times in Figures 7 and 8, we can observe that h_{sim} performs better than h_{abs} on small domains. The performance of h_{abs} on small domains is worse than that of h_{sim} because the computation cost of finding a relaxed plan is in general much higher than the computation costs of counting the number of nodes and edges in a state. While h_{sim} consumes only approx. 4% of the planning time, h_{abs} consumes over 89% of the planning time (independently of whether used by GBF or EHC). The performance changes to the favor of h_{abs} as the problem size increases. The planning time of h_{abs} scales better than the planning time of h_{sim}, which is expected since the number of generated states also scales better. Note that we can only make an appropriate comparison of the scaling behavior, not the *absolute* planning times, because our implementation is not optimized for efficiency.

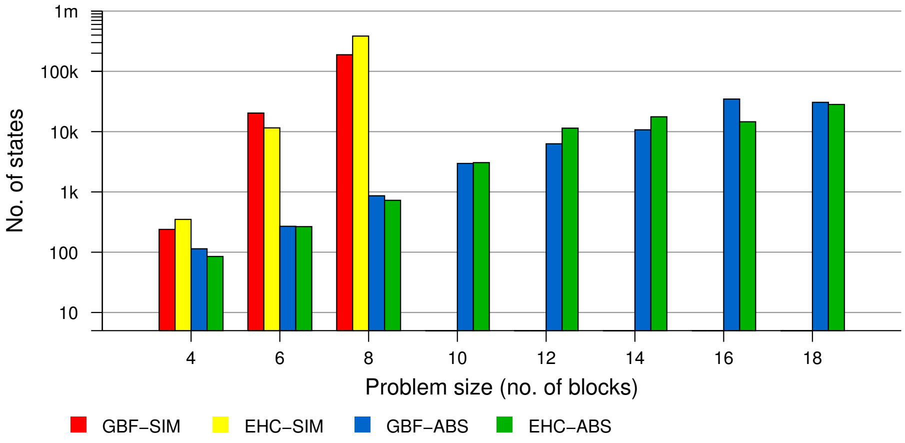


Figure 5: Histogram of the average number of states in Blocks World domains

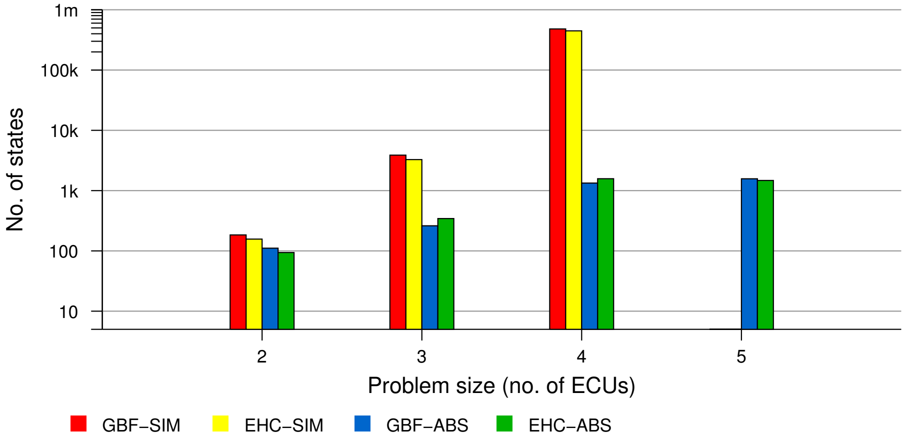


Figure 6: Histogram of the average number of states in ECUs domains

6 Related Work and Further Discussion

Our heuristic function is mainly inspired by the planning system Fast-Forward (FF) [HN01]. FF is a forward-chaining planner with a heuristic function that uses the solution length of a relaxed problem as one of the main features. It won the 2nd International Planning Competition (IPC-2000), which led to a shift of planning research towards heuristic-guided approaches. Variants of its techniques are used in many of today's state-of-the-art planners, like SGPlan [CWH06] or LAMA [RW10].

In contrast to our system, which uses label propagation to find the relaxed plan, FF computes the relaxed plan by a backward search on a structure called the *planning graph* [BF97]. Such a planning graph roughly resembles our list of successor graphs in the abstraction. By applying this idea to graph transformation systems instead of PDDL's propositional state representations, we face two important differences, especially with the combination of dynamic object creation and the support for NACs.

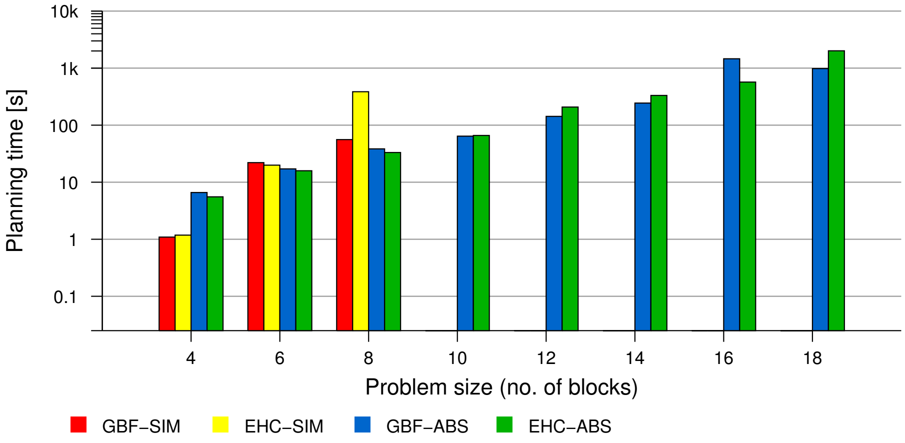


Figure 7: Histogram of the average planning times in Blocks World domains

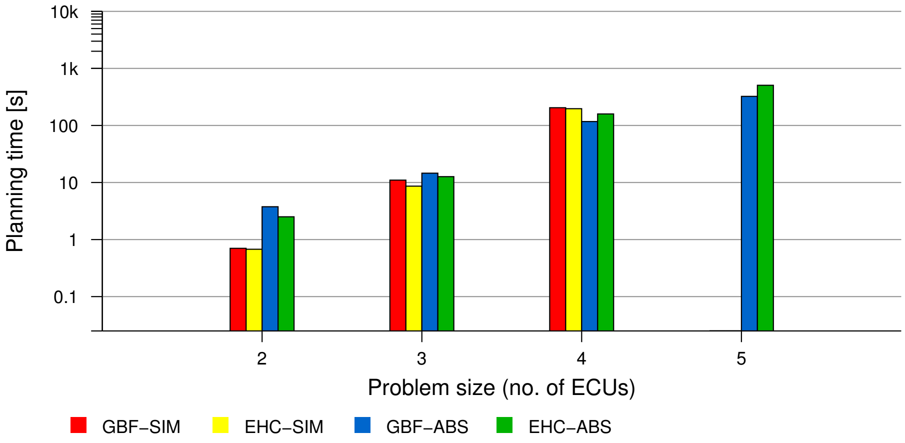


Figure 8: Histogram of the average planning times in ECUs domains

- Dynamic object creation can lead to an explosion of the graph size during the creation of successor graphs in the abstraction. This is not an issue in PDDL-based planners because they do not support dynamic object creation.
- The equivalent to NACs in PDDL are negative existential quantifications over conjunctive facts. They are usually solved by compiling them away, i.e., translating them into DNF, which results in a blowup of the propositional domain representation, cf. [HN01]. In our approach, we avoid such a blowup by building the support for NACs directly into the abstract planning algorithm.

Another approach that directly plans on graph transformation system is [EW11]. They also use search algorithms like A* or Best First to search through the state space, but they employ a *domain-specific* heuristic. An obvious disadvantage is that a heuristic suitable for the given application domain has to be developed first. To overcome the disadvantage, they use an approach to learn heuristic functions automatically. A learning algorithm derives a *regression function* that predicts the costs of solving the problem from a given state. To derive the regression function, the learning algorithm needs a predefined declaration of state features and a training set with problem instances. While this solution is an improvement from developing heuristic functions manually, it still requires the developer to declare a set of state features that is suitable for the given application domain.

In the introduction, we mentioned approaches that translate the GT planning problem into PDDL as an alternative to planning directly on graph transformation systems. This has been done in [TK11] for the story pattern formalism and in [ZW13], an extension of the former by a transactional concept to support durations that are specified for reconfigurations. In both of them, the straightforward solution to support NACs is to translate them into negative existential quantifications over all objects of the forbidden objects' types. The major disadvantage of these approaches is the restricted expressiveness of PDDL. Because PDDL does not support the dynamic creation of objects, these approaches use a modeling workaround, which requires a fixed maximal number of objects per type.

7 Conclusion and Future Work

In this paper, we presented an approach to planning with graph transformations. It features a domain-independent heuristic function that uses the solution of an abstraction of the problem as guidance. We further showed that it performed better than a similarity-based heuristic function which resembles heuristic functions employed in related work [EJLL06, Sni11].

In future work we plan to further improve our heuristic planning approach. One idea is to adapt the notion of *helpful actions*, cf. [HN01], to graph transformation planning. An action is called helpful in the concrete planning problem, if it is applied in the first step of the parallel relaxed plan. By considering only helpful actions when expanding a state, we expect to achieve a strong performance improvement.

When we finished developing a GTS-based planner that is satisfyingly performant, we intend to do a detailed evaluation comparing its efficiency to the efficiency of running off-the-shelf planning systems on PDDL models that have been generated out of the GTS by a translator. We are specifically interested in finding out whether one approach dominates the other one in general or whether this depends on properties of the application domain.

Acknowledgement

The author would like to thank Amir Shayan Ahmadian and Johannes Geismann for major contributions to the implementation and valuable discussions.

References

[BBG$^+$06] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein & Daniela Schilling (2006): *Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation*. In: *28th Int. Conf. on Software Engineering (ICSE 2006)*, ACM Press.

[BF97] Avrim Blum & Merrick L. Furst (1997): *Fast Planning Through Planning Graph Analysis*. *Artificial Intelligence* 90, pp. 281–300.

[CWH06] Yixin Chen, Benjamin W. Wah & Chih-Wei Hsu (2006): *Temporal Planning using Subgoal Partitioning and Resolution in SGPlan*. *Journal of Artificial Intelligence Research (JAIR)* 26, pp. 323–369.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science, Springer.

[EHK$^+$97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner & A. Corradini (1997): *Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach*. In G. Rozenberg, editor: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, chapter 4, World Scientific Publishing, pp. 247–312.

[EJLL06] Stefan Edelkamp, Shahid Jabbar & Alberto Lluch Lafuente (2006): *Heuristic Search for the Analysis of Graph Transition Systems*. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro & Grzegorz Rozenberg, editors: *3rd Int. Conf. on Graph Transformation (ICGT 2006)*, *Lecture Notes in Computer Science (LNCS)* 4178, Springer, pp. 414–429.

[EW11] H.-Christian Estler & Heike Wehrheim (2011): *Heuristic Search-Based Planning for Graph Transformation Systems*. In: *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS 2011)*, pp. 54–61.

[FL03] Maria Fox & Derek Long (2003): *PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains*. *Journal of Artificial Intelligence Research (JAIR)* 20, pp. 61–124.

[GN92] Naresh Gupta & Dana S. Nau (1992): *On the Complexity of Blocks-World Planning*. *Artificial Intelligence* 56, pp. 223–254.

[HN01] Jörg Hoffmann & Bernhard Nebel (2001): *The FF Planning System: Fast Plan Generation Through Heuristic Search*. *Journal of Artificial Intelligence Research (JAIR)* 14, pp. 253–302.

[KM07] Jeff Kramer & Jeff Magee (2007): *Self-Managed Systems: an Architectural Challenge*. In: *Workshop on the Future of Software Engineering (FOSE 2007)*, IEEE Computer Society, pp. 259–268.

[KR06] Harmen Kastenberg & Arend Rensink (2006): *Model Checking Dynamic States in GROOVE*. In Antti Valmari, editor: *13th Int. Workshop on Software Model Checking (SPIN 2006)*, *Lecture Notes in Computer Science (LNCS)* 3925, Springer, pp. 299–305.

[LM98] Daniel Le Métayer (1998): *Describing Software Architecture Styles Using Graph Grammars*. *IEEE Transactions on Software Engineering* 24(7), pp. 521–533.

[Ren08] Arend Rensink (2008): *Explicit State Model Checking for Graph Grammars*. In: *Concurrency, Graphs and Models*, *Lecture Notes in Computer Science (LNCS)* 5065, Springer, pp. 114–132.

[RW10] Silvia Richter & Matthias Westphal (2010): *The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks*. *Journal of Artificial Intelligence Research (JAIR)* 39, pp. 127–177.

[Sni11] Erik Snippe (2011): *Using Heuristic Search to Solve Planning Problems in GROOVE*. In: *14th Twente Student Conference on IT*, University of Twente.

[TK11] Matthias Tichy & Benjamin Klöpper (2011): *Planning Self-Adaptation with Graph Transformations*. In Andy Schürr, Dániel Varró & Gergely Varró, editors: *Int. Symp. on Applications of Graph Transformation with Industrial Relevance (AGTIVE 2011)*, *Lecture Notes in Computer Science (LNCS)* 7233, Springer.

[WF02] Michel Wermelinger & José Luiz Fiadeiro (2002): *A Graph Transformation Approach to Software Architecture Reconfiguration*. *Science of Computer Programming* 44(2), pp. 133–155.

[ZW13] Steffen Ziegert & Heike Wehrheim (2013): *Temporal Reconfiguration Plans for Self-Adaptive Systems*. In S. Kowalewski & B. Rumpe, editors: *Software Engineering (SE 2013)*, *Lecture Notes in Informatics (LNI)* , Gesellschaft für Informatik e.V. (GI), Bonn.