Utrecht University

Master Artificial Intelligence

Generating and Verifying Planning Sketches using Temporal Logic

Anneline Daggelinckx
Student number: 2060481

First examiner: Brian Logan
Second examiner: Natasha Alechina

July, 2023

Abstract

Sketches express the subgoal structure of a set of planning problems with a similar goal. A sketch consists of sketch rules that indicate how the environment has to change in a particular situation to reach the goal. Using sketches effectively speeds up the search for a solution to a planning problem. While a previous technique for learning sketches has been investigated, this thesis proposes a novel, modular approach to automatically find all 'good' sketches with n rules and m features given a set of domain instances with similar goals. Our approach involves lazily generating a pool of potential sketches and verifying them using temporal logic constraints. We show that our method can find good sketches for eight domains, including those learned in previous research. Because our approach allows us to generate all good sketches with a certain number of rules, we can analyze patterns of equivalent sketches, which opens up a wide range of future work.

Contents

1 Introduction

When humans want to achieve a goal, for example, getting a cup of coffee, baking a cake, or building a house, we try to take actions that bring us closer to reaching our objective. This is precisely the process that AI planning aims to automate. Planning in AI is the problem of finding which sequence of actions to take, starting from an initial state of the environment, to reach a goal. The first approaches to solving this problem are what we now call classical planning. Classical planners often work by trying out action sequences until a path to the desired goal is found. The possible paths and environment states are explored using algorithms like breath-first search or iterated width search. But with large problem instances, the search space gets large, and solving a planning problem gets intractable.

A way to speed up this search is by exploiting the subgoal structure of a planning problem. For example, when baking a cake, as a subgoal, you can gather all the ingredients. This extra information can significantly reduce the search space. Different methods were introduced for using subgoals to speed up the search. Examples are hierarchical task networks (HTNs) [24] and landmarks [26]. Recently, Bonet and Geffner introduced a new approach to exploiting subgoal structures, named sketches [4].

A sketch consists of one or more sketch rules. Sketch rules C $\rightarrow$ E express that when condition C is true, effect E is a desired subgoal. C and E are expressed in terms of features. For example, consider a table with blocks and a robot arm that can pick up, stack, and unstack blocks. If the goal is to unstack all blocks, we can define feature n as the number of blocks that are stacked upon each other and define a sketch rule $\{n > 0\} \mapsto \{n \downarrow\}$. This sketch rule states that if some blocks are stacked upon each other, a subgoal is to decrease the number of stacked blocks, i.e., to pick up a block that is stacked upon another block. This subgoal doesn't have to be reached in one step. In this example, if the robot is already holding a block, it first needs to put the block on the table before it can unstack a new block.

Sketches seem promising to speed up the search for a plan [9]. Initially, the construction of sketch rules involved expert handcrafting. Therefore Drexler et al. [10] proposed a method that automatically learns these rules. Their approach uses answer set programming (ASP) to find a coherent set of m sketch rules.

As an alternative approach, we will present an automata-based method for verifying sketches. We will use temporal logic to express candidate sketch rules and filter the valid sketches using automata-based model checking. The motivation for examining this method is threefold. First, our approach will be modular. We will first generate a pool of candidate sketches and use model checking to verify them. Because of this modularity, our method will also be able to verify handwritten sketches or candidate sketches generated by any other algorithm. Our method will also enable the exploration of different preprocessing steps to filter the candidate sketch rules. Secondly, our method inherits the guarantees of model checking. Automata-based model checking and its correctness are already widely studied [20]. Lastly, only one method of learning sketches has been examined before. Because of the different nature of our approach, we might find other sketches that give us more insight into the concept.

We will start this thesis by formalizing the notion of planning. After that, we will discuss related work on exploiting subgoal structures in planning problems and constructing these structures automatically. We give extra attention to sketches, which we will define formally. In section 4, we will formalize our approach for generating and verifying sketches. We will first explain how we generate a pool of candidate sketches to verify. Then we will define what we think a good sketch is. After that, we will give the syntax and semantics of CTL_f^* and use it to translate the definition of a good sketch into temporal logic. Section 5 will describe the different parts of our Python implementation of this method. To test our method, we will try to reconstruct the sketches found by Drexler et al. [10]. Our experiments and results are presented in section 6. Lastly, we discuss our findings, elaborate on directions for future work, and draw our conclusions in section 7 and 8.

2 Planning

In planning, we differentiate between *domains* and more detailed *instances* of a domain. A domain is a general description of an environment and its workings without details of the specific objects or information about a goal we want to reach. Starting from a domain, we can make different instances. These instances will specify which objects are in the environment, in what situation we start, and what goal we want to reach. As an example, we will consider the Blocksworld domain. In Blocksworld, there is a table with blocks and a robot that can pick up blocks from the table and put them down again, stack a block on top of another one, or unstack them. The actions the robot can execute can be expressed in *action schemas.*

Definition 1 (Action Schema) *An action schema is a quadruple* $a = (name, precond, effects^+, effects^-)$ *in which*

- $name = n(x_1, ..., x_k)$ *consists of a unique name* n *and a list of variables* $(x_1, ..., x_k)$ *that appear somewhere in* a*.*
- *precond denotes the preconditions of* a*, which is a set of literals.*
- $effects^+$ *and* $effects^-$ *represent the positive and negative effects of* a*, respectively. They are both sets of literals.*

For an action schema a, we will write $name(a), precond(a), effects^+(a)$, and $effects^-(a)$ to refer respectively to its name, preconditions, and positive and negative effects.

Without knowing the specifics of how many blocks there are, their names, how the blocks are stacked initially, or what goal we want to reach, the previous description of the blocksworld already gives a good idea of the environment. This is the purpose of a *planning domain.*

Definition 2 (Planning Domain) *A planning domain is a tuple* $D = (V, A)$ *in which*

- V *is a set of predicate symbols* (p, k) *with* p *the name of the predicate symbol, and* k *its arity.*
- A *is a set of action schemas, such that the variables in each action schema are predicate symbols from* V*.*

As an example, take the blocksworld domain. Two of its predicates are $(holding, 1)$, which states whether the robot is holding a block, and $(on, 2)$, which states that two blocks are stacked upon each other. An action schema is $(pickup(x), \{on\text{-}table(x), arm\text{-}empty(x), clear(x)\}, \{holding(x)\}, \{on\text{-}table(x)\})$, which defines the action of picking up a block that is lying on the table with no other blocks stacked on top of it.

By adding specific objects, an initial state, and a goal, a planning domain D can be extended to a *(domain) instance* (also called *problem instance* or *planning problem*).

Definition 3 (Domain Instance) *A (domain or problem) instance is a tuple* $P = (D, O, I, G)$ *in which*

- $D = (V, A)$ *is the domain*
- O *is a (finite) set of object names*
- I *is the initial state; a set of ground predicates* $p(c_1, ..., c_k)$ *in which* $(p, k) \in P$ *and* $c_i \in O$.
- G *is the goal, also a set of ground predicates.*

For a domain instance, we can construct a set of *(grounded) predicates* by adding n objects to each predicate symbol (p, n). For example, if $O = \{block1, block2\}$, the predicate symbol $(\textit{on-table}, 1)$ produces two grounded predicates $\textit{on-table}(block1)$ and $\textit{on-table}(block2)$. We will denote these grounded predicates as $V(O)$ for a set of predicate symbols V and objects O. Similarly, we can construct a set of *(grounded) actions* $A(O)$.

A *state* $s \subseteq V(O)$ defines how the environment looks at a specific moment. Formally, it is a set of grounded predicates that are true at that moment. Following the closed world assumption, the predicates not mentioned in this set are considered false in this state. A state $s \subseteq V(O)$ is a *goal state* for an instance $P = (D, O, I, G)$ if $G \subseteq s$.

An instance can also be represented by a *transition system.*

Definition 4 (Transition System) *A transition system of an instance* $P = (D, O, I, G)$ *with* $D = (V, A)$, *is a triple* $\Sigma = (S, A(O), \gamma)$ *such that*

- $A(O)$ *is the set of grounded actions,*
- $\gamma : S \times A(O) \to S$ *such that*

$$\gamma(s, a) = \begin{cases} (s \setminus effects^{-}(a)) \cup effects^{+}(a) & \textit{if } preconditions(a) \in s \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

- $S \subseteq 2^{V(O)}$ *such that* $I \in S$ *and* $s \in S \iff \exists a_1, ..a_n \in A(O)$ *such that* $\gamma(...(\gamma(\gamma(I, a_1), a_2), ...), a_n) = s$.

A *dead state* is a state from which it is impossible to reach the goal, i.e., $s_0 \in S$ is a dead state if there exists no sequence of states $(s_0, s_1, s_2, ..., s_n)$ and $a_i \in A(O)$ with $\gamma(s_i, a_i) = s_{i+1}$ and $s_n \in G$. States that are not dead are called *alive.*

As a complexity measure of a problem instance, Lipovetzky and Geffner introduced the notion of *problem width.*

Definition 5 (Problem Width) *[23] The width $w(P)$ of a (problem) instance P is the minimum $k \in \mathbb{N}$ for which there is a sequence $t_0, t_1, ..., t_m$ of atom tuples t_i, each with at most k atoms, such that:*

1. *t_0 is true in the initial state of P,*
2. *any optimal plan for t_i can be extended into an optimal plan for t_{i+1} by adding a single action, $i \in \{1, ..., n-1\}$,*
3. *any optimal plan for t_m is an optimal plan for P.*

As an example, consider the *Blocksworld domain* with blocks A, B, and C, an initial state in which C is on the table, A is stacked on B and B on C, and a goal to have block C on top of block A. This is a problem of width 2. A sequence of atom tuples that adheres to the three conditions in the definition of problem width can be:

$$\begin{tikzcd}
t_0 = (clearA, arm\text{-}empty) \arrow[d, "\text{unstack(A, B)}"] \\
t_1 = (holdingA, clearB) \arrow[d, "\text{putdown(A)}"] \\
t_2 = (clearA, clearB) \arrow[d, "\text{unstack(B, C)}"] \\
t_3 = (clearA, holdingB) \arrow[d, "\text{putdown(B)}"] \\
t_4 = (clearA, ontableB) \arrow[d, "\text{pickup(C)}"] \\
t_5 = (clearA, holdingC) \arrow[d, "\text{stack(C, A)}"] \\
t_6 = (clearA, ConA)
\end{tikzcd}$$

If we want to make such a sequence with tuples of one atom, we can, for example, try this sequence

$$\begin{tikzcd}
t_0 = (clearA) \arrow[d, "\text{unstack(A, B)}"] \\
t_1 = (holdingA) \arrow[d, "\text{putdown(A)}"] \\
t_2 = (ontableA) \arrow[d, "\text{unstack(B, C)}"] \\
t_3 = (holdingB) \arrow[d, "\text{putdown(B)}"] \\
t_4 = (ontableB) \arrow[d, "\text{pickup(C)}"] \\
t_5 = (holdingC) \arrow[d, "\text{stack(C, A)}"] \\
t_6 = (ConA)
\end{tikzcd}$$

This sequence does not satisfy the second condition from the definition of problem width, since not every optimal plan for $t_5(holdingC)$ can be extended into an optimal plan for $t_6(ConA)$. For example, $holdingC$ can be accomplished optimally by unstacking A and putting it on the table, unstacking B and stacking it on A, and then picking up C. At that point, B is stacked upon A, and $ConA$ cannot be reached optimally anymore. The same goes for other sequences with tuples of one atom. Therefore, the Blocksworld problem instance we consider here is of width two.

3 Literature

3.1 Related Work

Different approaches to exploiting the subgoal structure of a planning problem have been explored. Many of these approaches speed up the search for a solution but come at the cost of human expert knowledge. Providing expert knowledge is often time-consuming. Research has been done on constructing this expert knowledge automatically, sometimes with the help of extra information like (partial) solutions. In the following, I will review some of these approaches and how the extra knowledge they need can be learned.

Classical planning A classical planning problem consists of an initial state (predicates that are initially true), a set of action schemas with their preconditions and effects, and a goal condition. Classical planners use no additional information about the nature of the problem. Example algorithms to solve classical planning problems are forward searches like breath-first-search or depth-first search, starting from the initial state, and backward searches, where one starts from the goal state and searches for the initial state by applying actions backward. For problem instances with a large search space, these algorithms can be slow to find a solution.

Popular approaches to classical planning are STRIPS-style approaches [13]. The STRIPS algorithm attempted to reduce the search space size at the cost of completeness. In each state, only the transitions of actions that are "relevant for the goal" are considered. An action is only relevant for the goal if it reduces the differences between the current and goal states. This idea can be applied to both forward searches and backward searches.

HTN The plans that humans make have a hierarchical nature. For example, when baking a cake, we can plan that we will first collect all ingredients, then mix them following the recipe, and finally put the cake in the oven. These steps are all high-level tasks that can be subdivided into more concrete actions. Collecting the ingredients consists of getting the butter, getting eggs, getting milk, etc. Getting butter then consists of going to the fridge, opening it, grabbing the butter, and so on. *Hierarchical task networks* are a way to speed up the search for a solution to a planning problem by exploiting this hierarchical structure of tasks. Primitive tasks are the actions that an agent can execute directly, while compound tasks are higher-level tasks that can be decomposed into primitive or other compound tasks. An HTN planning problem consists of a high-level task that needs to be executed and a set of predefined compound tasks and the lower-level tasks they require.

In contrast to classical planning, an HTN goal is not to be in a state where something is true but to do a certain task. While in classical planning, we express that our goal is to be in a state where there is a cake, in HTN planning, the goal is to bake a cake. During execution, compound tasks are decomposed into simpler tasks until a sequence of primary tasks is reached.

Compared to classical planning, HTNs find a solution to planning problems faster [24] because they have access to more domain knowledge to reduce the search space. A downside to this approach is the need for an expert to define the task networks, which is very time-consuming.

To handle this downside, research has been done on finding methods to learn the task networks automatically. The difficulty is that any sequence of tasks could make up a compound task. Finding useful compound tasks is something humans can naturally do, but it is more difficult for a computer. Therefore most of the learning methods use additional input. Research has been done on learning task networks from traces [32, 25, 18], partial decomposition trees [31, 32], solutions to planning problems [18], and combinations of them.

Landmarks Given a planning problem, landmarks are facts that always need to be true at some point to reach the goal, starting from the initial state [26]. For example, when baking a cake, it always holds that at some point, all ingredients are in the same bowl. Landmarks are thus subgoals that always need to be reached before it is possible to reach the goal. Additionally, different landmarks can be connected with order relations [26, 17, 27]. It can be the case that landmark B relies on landmark A, such that A always needs to be reached before B. For example, all ingredients should be in the same bowl before the cake is in the oven.

From the start of landmark research, methods have been examined to extract landmarks from the planning problem. The learning goes in two steps. First, landmark candidates and approximated orders between them are found by relaxing the planning task. Second, all landmark candidates that cannot be proved to be landmarks are removed [17, 28, 27, 26].

Using landmarks speeds up search compared to classical planning because they exploit information about the subgoal structure and can therefore do better than blind search.

Logics to express control knowledge Another type of domain knowledge that can help to speed up search is information on the dynamics of a domain. For example, if you want to paint a room blue, you can express that after you've painted a wall blue, you do not want to do an action that paints the wall in a different color. A way to express this type of information is by using logical languages. Research has been done on expressing this domain-specific control knowledge in both linear temporal logics (LTL) [1] and temporal action logics (TAL) [21].

This domain knowledge can be used to eliminate options during forward planning. In forward planning, one starts from the initial state and executes actions to make paths until one of the paths reaches a goal state. Using the control knowledge, every potential partial solution is checked to satisfy the knowledge formulas. If a partial solution violates one of the rules, it is not further examined. This speeds up the search compared to classical planning because some partial plans that are certain to fail or be suboptimal can be eliminated before they are fully worked out.

Unlike HTNs and landmarks, this control knowledge gives information about good and bad transitions between states instead of focussing on subgoals.

Similar to HTNs, experts are needed to provide control knowledge. Some research has been done on learning LTL control knowledge using a set of optimal and suboptimal plan solutions [7].

General Policies A policy is a function that maps states to actions, specifying the action to take in a given state. A policy solves an instance if, starting from the initial state, sequentially applying the actions given by the policy results in achieving the goal. A general policy is a policy that solves multiple instances of a planning domain [19, 3].

Different ways to learn general policies are explored. Bonet et al. [2] learned general policies using an SAT-solver and the theory of qualitative numerical planning problems (QNP). A QNP is an abstraction over a set of problem instances that consists of features, which are functions over states of the problem instances that express properties of these states, and actions in which pre- and postconditions are described in terms of these features. An SAT-solver is a program that can compute the satisfiability of propositional logic formulas. Bonet et al. use an SAT-solver to learn a qualitative numerical planning problem (QNP) from sample plans and solve this QNP with a FOND planner, which results in a general policy.

Francès et al. [15] took inspiration from this approach but skipped the phase of learning the QNPs. Instead, they generate a pool of features, which they use together with a set of domain instances to express the original planning problem as a weighted Max-SAT problem. By solving this Max-SAT problem, they find a general policy. Lastly, research has been done on learning general policies using graph neural networks (GNNs) [30, 29].

Sketches A more recent approach to adding subgoal structures to a planning problem is the concept of sketches [4]. Sketch rules express under which conditions a certain subgoal is desired. For example, if the goal is to bake a cake, a sketch rule could be: if the number of missing ingredients is larger than 0, you want to decrease this number. General policies can also be considered sketches in which the subgoals must be reached by applying only one action.

The sketch rules must be followed while constructing a solution to a planning problem. Using sketches significantly reduces the search space w.r.t. classical planning [9].

Sketch rules depend on the domain and the desired class of problems but do not depend on domain instances. If the goal in the blocks domain is to $clear(x)$, the sketch rules are the same, independent of which block x is, independent of the initial state and independent of the number of blocks in the domain.

Research is done on learning sketches by Drexler et al. [10]. Inspired by the approaches of learning general policies [2, 15], Drexler et al. use answer set programming (ASP) to find a sketch of a certain width, given a set of features and a set of domain instances. An ASP program consists of a set of facts and rules provided in propositional logic. Answer set solvers then find a set of truth values that satisfy the ASP program. Drexler et al. translate the problem of finding a sketch for a set of problem instances to an ASP program and return one sketch.

3.2 Sketches

3.2.1 Definitions

A *sketch* [4] defines subgoals we want to achieve to reach a goal. These subgoals are captured in rules. The idea is that a sketch is not instance dependent but works over a set of instances with similar goals. We call a set of problem instances drawn from the same domain and with similar goals a *class of problems Q*. In the literature, the notion of similar goals is not formally defined.We assume goals to be similar if there is a mapping from objects to goals, such that the goal of each problem instance can be constructed by applying this mapping to objects from the instance. Examples for the Blocksworld domain are achieving a state where a specific block is clear and the robot arm is empty (achieving $\mathit{clear}(x) \land \mathit{arm\text{-}empty}$), stacking two blocks (achieving $\mathit{on}(x, y)$), or having all blocks next to each other on the table ($\mathit{on\text{-}table}(x_1) \land \mathit{on\text{-}table}(x_2) \land ... \land \mathit{on\text{-}table}(x_n)$ with n the number of blocks.)

Sketches are defined over a set of features Φ. *Features* are functions from states in Q to booleans or non-negative integers. For a state s and a feature f, $f(s)$ is the *feature valuation* of f in s , i.e., feature f has value $f(s)$ in state s.

An example of a numerical feature in the blocksworld domain is the number of blocks currently stacked upon each other. As you can see, this features only depend on the domain description and not on specific instances. Sketch rules indicate how these features should change to achieve the goal.

As mentioned, a sketch consists of multiple sketch rules, in which a sketch rule expresses that under certain conditions, we want to manipulate features in a certain way. A sketch rule $C \rightarrow E$ consists of a set of *conditions* and a set of *effects*. If these conditions are true in a state, the subgoal is to reach a state where the effects hold.
The conditions C of a sketch rule consist of *feature conditions* c_i, defined over boolean and numerical features. Feature conditions are statements that are either true or false in a state. Let $n \in \Phi$ be a numerical feature and $b \in \Phi$ be a boolean feature. The semantics of a feature condition c_i are the following:

$$c_i ::= \quad n{=}0 \quad | \quad n > 0 \quad | \quad b \quad | \quad \neg b$$

For a numerical feature $n \in \Phi$, a feature condition can state that this feature has to be equal to zero ($n{=}0$), or this feature has to be greater than zero ($n > 0$). For a boolean feature $b \in \Phi$, a feature condition can indicate that the feature has to be true (b) or false ($\neg b$).

A numerical feature condition c_i *holds* in state s iff

- $c_i = (n = 0)$ and $n(s) = 0$
- $c_i = (n > 0)$ and $n(s) > 0$
- $c_i = b$ and $b(s) = true$
- $c_i = \neg b$ and $b(s) = false$

A condition $C = \{c_1, c_2, ..., c_n\}$ *holds* in a state s if c_i holds in s for $1 \leq i \leq n$.

The effects of a sketch rule E consist of *feature value changes* e_i over a set of features Φ. These feature value changes indicate how features should change over time.

$$e_i ::= \quad n\uparrow \quad | \quad n\downarrow \quad | \quad n_{=} \quad | \quad b \quad | \quad \neg b \quad | \quad b_{=}$$

For a numerical feature n, feature value changes can indicate that the value of a feature needs to increase ($n\uparrow$), decrease ($n\downarrow$), or shouldn't change ($n_{=}$). For a boolean feature b, a value change can indicate that b needs to be true (b) or false ($\neg b$). Additionally, an effect can require a boolean to keep its value regarding the condition state ($b_{=}$). If an effect is not mentioned, we assume it doesn't matter how its value changes.

We say that a feature value change e_i *holds* for a transition between two states (s, s') iff

- $e_i = n\uparrow$ and $n(s) < n(s')$
- $e_i = n\downarrow$ and $n(s) > n(s')$
- $e_i = (n_{=})$ and $n(s) = n(s')$
- $e_i = b$ and $b(s') = true$
- $e_i = \neg b$ and $b(s') = false$
- $e_i = (b_{=})$ and $b(s) = b(s')$

An effect $E = \{e_1, e_2, ..., e_n\}$ *holds* for a transition (s, s') iff every feature value change e_i holds for (s, s') for all $1 \leq i \leq n$.

A transition from a state s to state s' *follows a rule* $R : C \rightarrow E$ iff C holds in s and E holds for (s, s'). *Applying a rule* $R : C \rightarrow E$ in state s means following a path $(s, s_1, s_2, ..., s')$ such that (s, s') follows R.

A *sketch* is a collection of sketch rules.

Definition 6 (Sketch) *A* sketch *over a class of problems Q and a set of features Φ is a finite set of sketch rules $R_i : C_i \rightarrow E_i$ over Φ.*

For example, consider the Blocksworld in which the goal is to clear a block. A block is clear if no other blocks are stacked upon it and the robot is not holding it.

The sketch rules for this domain that are presented in [9] use two features. The boolean feature H states whether the robot is holding a block. The numerical feature n counts the number of blocks stacked above the block we want to clear. The sketch rules are

$$r_1 = \{\neg H, n > 0\} \rightarrow \{H, n\downarrow\}$$
$$r_2 = \{H, n > 0\} \rightarrow \{\neg H, n_=\}$$

Let's call the block we want to clear block x. The first rule states that if the robot is not holding any block, and there are blocks above block x, our subgoal is to find a state where the robot is holding a block and the number of blocks above x has decreased. In other words, the subgoal is to unstack a block on the tower above x. The second rule states that if the robot is holding a block, the subgoal is to not hold a block so that the number of blocks above x doesn't change. In other words, the robot should put the block down on the table or on top of another tower of blocks that doesn't contain x.

Another example is an environment with a robot vacuum and different rooms that need to be cleaned. If the goal is to clean every room, we can define feature n as the number of dirty rooms and define a sketch rule $\{n > 0\} \rightarrow \{n\downarrow\}$. This sketch rule states that if rooms are still dirty, a subgoal is to decrease the number of dirty rooms, i.e., to clean at least one room. This subgoal doesn't have to be reached by only applying one action. In this example, the robot vacuum first has to do one or more move actions before it can do the cleaning action and reach the subgoal.

Based on the notion of problem width, Bonet and Geffner introduced a complexity measure on sketches, named *sketch width* [4]. The width of a sketch is the maximum width of the subproblems induced by its rules.

Definition 7 (Sketch width) *The width $w_S(P[s])$ of a sketch S at state s of a problem P, is the width of the subproblem $P[s]$ that is like P but with initial state s and goal states g such that the pair (s, g) follows one of the rules of S. The width of a sketch S for a class of problems Q is $w_S(Q) = max_{P \in Q} max_{s \in \{\text{ reachable state in } P\}} w_S(P[s])$*

3.2.2 Differences in definitions with previous papers

Any vs Equal In the previous papers about sketches [4, 9, 10], the syntax of feature effects is slightly different, but the semantics are equivalent to our definition. We define $f_=$ for a feature $f \in \Phi$ to indicate that this feature should keep its value relative to the condition state. If a feature is not mentioned in the effect, we assume its value does not matter. Bonet and Geffner define this the other way around. To express that the value of a feature doesn't matter, they introduce the feature value change $f?$. A feature not mentioned in the effect means its value should stay the same.

We changed the notation because it is more suitable for both our method and its implementation. The notation we use aligns with how sketches are represented and saved to files in the DLPlan library, which we will use for our implementation. Furthermore, it eases the translation to temporal logic. When a predicate is not mentioned in a logic formula, its truth value doesn't matter, which is similar to our syntax for sketches.

Distinguishing the goals In Bonet and Geffner [4], the definition of a sketch requires the set of features Φ to *distinguish the goals* for the class of problems Q. A set of features Φ *distinguishes* or *separates* the goals for Q if it is possible to separate goal states from non-goal states by only using the feature valuations of the features in Φ. Drexler et al. do not require goal separation in their paper about learning sketches [10]. To show this, consider the width-0 sketch for the gripper domain, presented in [10]. We have $\Phi = \{B, c\}$ where B is a boolean feature true if the robot is in room B, and c is a numerical feature, counting the number of balls the robot is carrying.

$$\begin{aligned} r_1 &= \{c = 0\} \rightarrow \{\neg B\} \\ r_2 &= \{B\} \rightarrow \{c \downarrow\} \\ r_3 &= \{\neg B, c > 0\} \rightarrow \{c{=}\} \end{aligned}$$

Drexler et al. consider this a well-formed sketch, while the features in Φ do not distinguish the goals. For example, the goal state where all balls are in room B, the robot is in room B, and the robot is not carrying any balls. For this state, $B = \mathit{true}$ and $c = 0$. For a non-goal state where all balls are in room A, the robot is in room B and not carrying any balls, we get the same feature valuations. These features thus don't distinguish goal states from non-goal states. Since we use the work of Drexler et al. [10] as a reference for our results, we decided not to consider goal separation. Nevertheless, our method allows for adding goal separation in future work.

4 Approach

Our overall goal is to find sketches for a set of planning problems such that consecutively following sketch rules results in achieving the goal. We will call such sketches *good sketches* and formally define them in this chapter.

Our method to find good sketches will start with generating a pool of candidate sketches by combining features into conditions and effects and combining these conditions and effects into sketches. We will translate the constraints from the definition of a good sketch into temporal logic formulas and use a model checker to verify whether the constraints hold for a given sketch and instance in order to find all good sketches.

4.1 Sketch Generation

To generate a candidate sketch pool, we start with a pool of features Φ, similarly to Drexler et al. [10]. For a sketch S, let's define $|S|$ as the number of rules the sketch has, $F(S)$ as the set of features that are mentioned in the sketch, and $|F(S)|$ the number of features used by the sketch. We aim to construct sketches of maximum f_{max} features and r_{max} rules. Our goal is thus to construct a set of candidate sketches.

$$CS := \{S \mid |F(S)| \leq f_{max}, |S| \leq r_{max}, F(S) \in \Phi\}$$

Given this finite feature set Φ, there are only finite options to combine them in conditions and effects to create sketch rules and sketches.

To calculate an upper bound on the number of candidate sketches, we start by calculating how many sets there are containing f features. This amount is equal to:

$$C_f^{|\Phi|} = \frac{|\Phi|!}{(|\Phi| - f)!f!}$$

There are three possibilities to use a feature in a feature condition. We can not mention the feature or use $n = 0$ or $n > 0$ for a numerical feature and b or $\neg b$ for a boolean feature. Each feature can be used in the effects in four ways; not mentioning the feature, $n\uparrow, n\downarrow, n$= for a numerical feature, and $b, \neg b, b$= for a boolean feature. Each feature can thus be used in a rule in $3 \cdot 4 = 12$ ways. Therefore, the number of possible rules with maximum f_{max} features can be bounded by:

$$n_{rules} \leq C_{f_{max}}^{|\Phi|} \cdot 12^{f_{max}}$$

Note that this is an upper bound and not an equality. Some rules are considered multiple times. For example, the rule $\{\} \rightarrow \{\}$ is constructed for each feature set. Moreover, a rule that only mentions one feature is constructed for each feature set that contains this feature.

An upper bound on the number of candidate sketches with r_{max} rules and maximum f_{max} features is:

$$|CS| \leq C_{r_{max}}^{n_{rules}}$$

4.2 Good sketch

Not all assemblies of conditions, effects, and sketch rules will create a sketch such that following its sketch rules guarantees reaching the goal. Therefore we will define the notion of a *good sketch.* The idea of a good sketch is that for every sequence $s_0, s_1, s_2, ...$ starting from the initial state s_0, such that every pair (s_i, s_{i+1}) follows one of the rules R_j, there will be an n such that s_n is the goal.

We will define three constraints that ensure a sketch is good.

Definition 8 (Good sketch) *For a sketch over a class of problems Q to be* good*, the following constraints should hold for all instances in Q (in which the initial state is not a dead state);*

1. *For every alive, non-goal state s of the instance, a sketch rule should exist such that its condition is true in s and its effect can be reached from s.*
2. *A sketch rule should not lead to a dead state.*
3. *Every path starting in the initial state and consisting of a chain of rule applications should eventually reach the goal.*

The first constraint assures that if it is still possible to reach the goal, we can always follow one of the rules, no matter in which state we are. This way, we can never end up in a state where no other rule can be applied. This constraint is a bit too demanding since there might be a sketch with rules that never lead to state s, so there is no need for a rule that can be applied from s. On the other hand, since we want that a sketch works for a class of problems, we can argue that it needs to be good with any alive state of the instance as the initial state. In [10], this constraint is also required.

The second constraint is straightforward. If a pair of states (s, s') follows a rule, we don't want s' to be a dead state. This way, we ensure that following rules keeps the possibility of reaching the goal. One can argue whether this constraint is necessary since the third constraint already requires every chain of rule applications to reach the goal. For example, look at figure 1. The first constraint is satisfied; from every alive, non-goal state, a rule can be applied. The third constraint is also satisfied, because the only sequence of rule applications starting from the initial state is first applying rule 1 and then rule 2 (following the blue arrows). One can argue whether this sketch is good because applying rules will always lead to the goal starting from the initial state, but intuitively rule 2 is not acceptable since it leads into a dead state. We decided to keep this constraint.

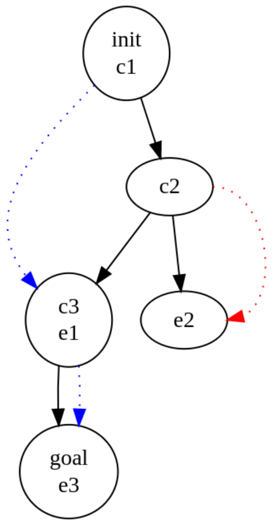

Figure 1: *Transition system with sketch rules. c_i represents that the conditions of rule i hold. e_i represent that the effects of rule i hold, respectively to the states where c_i held. The black arrows are the transitions, and the colored arrows indicate state pairs that follow a rule.*

The third constraint states that we should eventually reach the goal if we follow consecutive rules. The first constraint implies that there also exists such a path given that the initial state is not a dead state.

The constraints we set on a *good sketch* are inspired by but differ from those given in previous work. Bonet and Geffner [4] define a sketch as *well-formed* if the set of rules in the sketch are *terminating*. Intuitively, termination means you cannot get stuck in a cycle when following rules. Additionally, in Bonet and Geffner the features in a sketch should distinguish the goals, as mentioned in section 3.2.2. They prove that when a sketch is terminating and has bounded width, it will eventually reach the goal using the SIW_R algorithm.

We define a *good sketch* in a different way to facilitate translation to temporal logic and to put extra constraints on them, similar to [10].
A sketch can be good but not well-formed because a good sketch doesn't require goal separation as defined by Bonet and Geffner. On the other hand, a sketch can be well-formed but not good if it doesn't hold that for every state, a sketch rule can be applied.

In order to know whether a given sketch is good, we need to solve a *sketch verification problem.*

Definition 9 (Sketch Verification Problem) *Given a sketch over a class of problems Q, verify whether this sketch is good.*

In the remainder of this chapter, we will describe how we can translate the constraints of a good sketch to CTL_f^*. This way, we can use a model checker to *verify* whether sketches from our candidate sketch pool are good.

4.3 Translation of Sketch Constraints into CTL_f^*

Our objective is to create logical formulas that express the three constraints, allowing us to evaluate whether a sketch is "good". In order to do so, we will represent the conditions and effects of sketch rules in CTL_f^*.

4.3.1 CTL_f^*

CTL^* [12] is a logical language that combines state formulae of computational tree logic (CTL) and path formulae of linear temporal logic (LTL). CTL_f^* has the same syntax as CTL^*, but considers only finite paths. In addition to the standard operators of CTL_f^*, we add operators to reason about the past in a path [22].

We define PV as a set of *propositional variables* (also called *atomic propositions*). A *Kripke model* is a tuple $M = (St, R, V)$ in which St is a set of states, $R \subseteq St \times St$ is a set of transitions between these states and $V : PV \to 2^{St}$ is a *valuation function* which indicates in which states a propositional variable is true. We assume that if a propositional variable is not true in a state, it is false. A *finite path* π over a Kripke model $M = (St, R, V)$ is a sequence of states $\pi = (s_0, s_1, ..., s_n)$ with $n \in \mathbb{N}$ such that $\forall i : 0 \leq i < n$ holds that $(s_i, s_{i+1}) \in R$.

The grammar of CTL_f^* consists of *state formulae* φ, whose truth values are evaluated in a specific state, and *path formulae* γ, whose truth values are evaluated on a specific finite path.

Let $p \in PV$ be a propositional variable. The syntax of CTL_f^* is the following:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid E\gamma,$$
$$\gamma ::= \varphi \mid \neg\gamma \mid \gamma \wedge \gamma \mid X\gamma \mid \gamma U \gamma \mid Y\gamma \mid O\gamma.$$

State formula $E\gamma$ is true in a state if there exists a finite path, starting from that state such that γ holds for that path. Path formula $X\gamma$ is true on a path if γ is true in the next state on that path. $\gamma_1 U \gamma_2$ holds on a path if γ_2 is true in a future moment on the path, and γ_1 is true until γ_2 becomes true. On a path, $Y\gamma$ holds if γ was true in the previous step, and $O\gamma$ holds if γ was true somewhere in the past.

Let $M = (St, R, V)$ be a Kripke model, and $q \in St$ a state. We define the semantics of the state formulae as follows.

$M, q \models p$ for $p \in PV$ if $q \in V(p)$
$M, q \models \neg\varphi$ iff $M, q \not\models \varphi$
$M, q \models \varphi_1 \wedge \varphi_2$ iff $M, q \models \varphi_1$ and $M, q \models \varphi_2$
$M, q \models E\gamma$ iff there exists a finite path $\pi = (q_0, q_1, ...q_n)$ and $j : n \geq j \geq 0$
such that $q_j = q$ and $\pi, j \models \gamma$

The path formulae are evaluated on finite paths. Let $\pi = (q_1, .., q_n)$ be a finite path in M, $i \in \{1, .., n\}$ and $\pi(i) := q_i$ be the i^{th} state on that path.

$\pi, i \models \varphi$ iff $M, \pi(i) \models \varphi$
$\pi, i \models \neg\gamma$ iff $\pi, i \not\models \gamma$
$\pi, i \models \gamma_1 \wedge \gamma_2$ iff $\pi, i \models \gamma_1$ and $\pi, i \models \gamma_2$
$\pi, i \models X\gamma$ iff $i < n$ and $\pi, i + 1 \models \gamma$
$\pi, i \models \gamma_1 U \gamma_2$ iff there is a j with $i \leq j \leq n$ such that $\pi, j \models \gamma_2$
and for all k with $i \leq k < j$, we have that $\pi, k \models \gamma_1$
$\pi, i \models Y\gamma$ iff $i > 0$ and $\pi, i - 1 \models \gamma$
$\pi, i \models O\gamma$ iff there is a k with $1 \leq k \leq i$ such that $\pi, k \models \gamma$

Additionally, we define state formula $A\gamma$ (for all paths that start in this state γ holds), and path formulae F (somewhere in the future γ holds) and G (γ holds in all states) in terms of the other operators.

$$\begin{aligned} A\gamma &\equiv \neg E \neg\gamma \\ F\gamma &\equiv True U \gamma \\ G\gamma &\equiv \neg F \neg\gamma \end{aligned}$$

4.3.2 Expanded Sketch

Consider a sketch S that consists of n rules over a set of features Φ:

$$S = \{R_i : C_i \rightarrow E_i | i \in \{1, ..., n\}\}$$

with $C_i = \{c_{i_1}, c_{i_2}, ..., c_{i_k}\}$ a set of feature conditions and $E_i = \{e_{i_1}, e_{i_2}, ..., e_{i_l}\}$ a set of feature value changes. In a state s, each feature condition c_{i_j} is true or false, making it suitable to be used as propositional variables in logical formulas. This is not the case for feature value changes. The value of a feature value change e over a feature f can depend on both the current state and a previous state. To solve this, we write the sketch rule for all possible values of f and will substitute the conditions and feature value changes by *feature variables*. We define a *feature variable* as $(n = x)$, $(n > x)$ or $(n < x)$ for n a numerical feature and x a non-negative integer, or b, $\neg b$ for b a boolean feature. Feature variables can be evaluated in a single state and can thus be used as propositional variables in logical formulas.

We define an *expanded sketch* as a set of *expanded sketchrules*

$$S^{\varepsilon} = \{R_i^{\varepsilon} : C_i^{\varepsilon} \rightarrow E_i^{\varepsilon}\}$$

in which the conditions C_i^{ε} and the effects E_i^{ε} are sets of feature variables. These conditions and effects can be used in logical formulas by making a conjunction of the feature variables in the set. To *expand* a sketch means to translate a sketch to an expanded sketch.

Sketch expansions are instance specific. To expand a sketch, we use a domain instance to determine the upper and lower bound of a feature n and split the rule into formulas for all possible values of n.

Consider a sketch rule $R_i : \{c_1, c_2, ..., c_k\} \rightarrow \{e_1, e_2, ..., e_l\}$. Let $e_i = n\downarrow$ for a numerical feature n. For St a set of states, and the values of n bounded by

$$n_{min} \leq n(s) \leq n_{max} \qquad \forall s \in St$$

we can split this rule into multiple rules R_{i_m} with $n_{min} \leq m \leq n_{max}$. Each rule R_{i_m} is defined as follows:

$$R_{i_m} : \{c_1, ..., c_k, (n = m)\} \rightarrow \{e_1, ..., e_{i-1}, (m > n), e_{i+1}, ..., e_l\}.$$

If the effects contain any other feature value change of the form $n\downarrow$, $n\uparrow$, n= for a numerical feature n or b= for a boolean feature b, we can repeat this process for the new rules R_{i_m} until all feature value changes are substituted with feature variables.

For an expanded sketch, we can write the condition and effect of its rules as a conjunction of its feature variables.

Next, we will translate the three constraints of a good sketch to CTL_f^*. We will translate each constraint separately and require their conjunction to hold for a good sketch.

4.3.3 Constraint translation

We introduce a proposition *goal*, which is true in all the goal states of the problem instance.

The first constraint states that for every state s in the instance, a sketch rule should exist such that its condition holds in s and its effect can be reached from s, unless s is a goal state or a dead state. This can be expressed by saying that on every path, either *goal* is true and we are in the goal state, or it is no longer possible to reach the goal, or there exists a rule with a condition that holds at present, and there exists a path such that its effect holds in the future. In CTL_f^*, we express this constraint as:

$$\mathbf{AG}\Big(\bigvee_i \big(C_i \land \mathbf{EX}(\mathbf{EF}(E_i \land \mathbf{EF}goal))\big) \lor goal \lor \neg\mathbf{EF}goal\Big) \tag{1}$$

The second constraint states that a sketch rule must never lead to a dead state. In other words, for every rule it must hold that if a condition is true at any point, there cannot be a future where the effect is true in a dead state.

$$\bigwedge_i \mathbf{AG}\Big(C_i \rightarrow \neg\mathbf{EX}(\mathbf{EF}(E_i \land \neg\mathbf{EF}goal))\Big) \tag{2}$$

The third constraint states that if a path, starting in the initial state, consists of a sequence of rule applications, the goal should eventually be reached. We express this in CTL_f^* as follows:

$$\mathbf{A}\Bigg(\Bigg(\underbrace{\bigvee_i (C_i \land \mathbf{XF}E_i)}_{f_1} \land \mathbf{G}\Bigg(\underbrace{\big(\bigvee_i (\mathbf{YO}C_i \land E_i)\big) \rightarrow \big(\bigvee_j (C_j \land \mathbf{XF}E_j) \lor goal\big)}_{f_2}\Bigg)\Bigg) \rightarrow \mathbf{F}goal\Bigg) \tag{3}$$

f_1 expresses that the path starts by applying one of the rules, and f_2 states that from that point onward, it must always hold that if we reach a state by applying a rule, it is either possible to apply another rule or the goal is reached. This formula can be expressed without the past operators $\mathbf{Y}$ and $\mathbf{O}$, but we use them for readability.

As the last constraint is an implication, it also holds when no paths satisfy the antecedent (i.e., no paths exist consisting of a sequence of rule applications). However, the first and second constraints imply that there should always be such a path. The first constraint asserts that a rule can be followed in each alive state. The only scenario where no rule is applicable is when we reach a dead state. However, the second constraint specifies that a rule cannot lead to a dead state, ensuring that, given a good sketch and satisfaction of the first two constraints, a sequence of rule applications always exists.

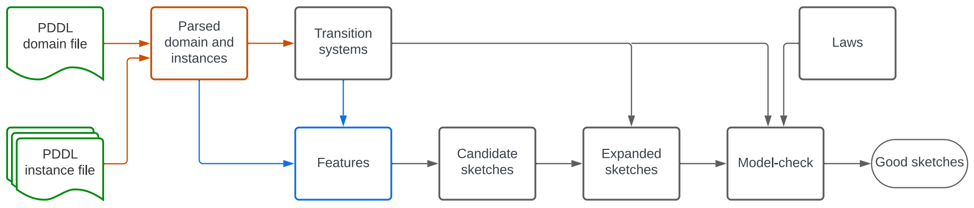

Figure 2: *This figure shows a simplified schematic overview of our implementation. Green boxes represent files. Orange boxes and arrows represent that we used objects and functionalities of the Tarski library for this part of the program. Blue arrows and boxes mean we used objects and functionalities from the DLPlan library.*

5 Implementation

This section will review how we implemented our method for generating and verifying sketches [1]. Figure 2 shows a simplified schematic overview of our implementation. The full overview can be found in Appendix C.

We start by parsing files that contain information about the domains and instances we want to use. Next, we use this information to build transition systems and state spaces for all instances. We then use the states of the instances to generate a feature pool. These features are used to create a pool of candidate sketches lazily. For each candidate sketch, we start by model-checking the instances with the smallest number of states and work toward the largest. As soon as one of the constraints of a good sketch doesn't hold for a sketch on one of the instances, we continue to the following sketch without verifying this sketch on the remaining instances. If all constraints hold on all instances for a sketch, we add this sketch to our pool of good sketches.

5.1 Language and Libraries

We implemented our method for generating and verifying sketches in Python (3.10). We chose Python because of the availability of three libraries:

1. The Tarksi library [14] allows us to parse PDDL domains and instances into Python objects;
2. The DLPlan library [8] provides functionality to generate a pool of features from domain instances;
3. The PyNuSMV library [5] provides Python bindings for the NuSMV model checker.

Due to the lack of CTL_f^* model checkers with past, we opted to alter our formulas into CTL and LTL formulas and use a suitable model checker for this new problem. We used the NuSMV model checker [6] because it supports CTL

[1]The implementation is available online at https://github.com/AnnelineD/TLSketch/

and LTL model checking and allows for past operators in LTL. Additionally, the PyNuSMV library made it easy to access the NuSMV functionalities in our Python code.

Additionally, we made a small logic library that supports representations of LTL and CTL. Several other Python libraries implement LTL and/or CTL (for example, $ltlf_2dfa$, $NL2LTL$, $pyModelChecking$, ...). Still, all focus on broader goals like model checking, translating natural language to logic, or finite automata translation. Since we merely needed representations and no additional functionality, these libraries were more challenging to use than necessary for our purpose. Additionally, we found no libraries that support past LTL.

5.2 Input-PDDL

As input for our program, we use domains and instances represented in the Planning Domain Definition Language (PDDL) [16]. PDDL is a language to express planning domains and instances. A planning domain is described by its predicate schemas, action schemas, and, optionally, types and constants. Types can tag objects and variables in predicate symbols and action schemas such that only the objects with corresponding types can be used to ground predicates and actions. Constants are objects that are present in every instance. A planning instance of such a domain is described by a set of (typed) objects, the set of propositions that are true in the initial state, and a logical formula that represents the goal. In our examples, only conjunctions are allowed in the goal. The goal states are the states of the instance in which this goal conjunction is true.

Example PDDL descriptions of the domain and an instance of the Blocksworld can be found in Appendix A.

5.3 Building the Transition Systems

The Tarski library [14] is a helpful tool for parsing PDDL files and representing the corresponding domains and instances. It provides functionalities for representing states and actions of problem instances, as well as calculating the resulting state $\gamma(s, a)$ after applying an action $a \in A$ to a state $s \in S$.

Besides parsing PDDL files, we utilize the Tarski library to build a transition system given an instance. To build such a transition system, we iteratively apply grounded actions to calculate reachable states, starting at the initial state.

We represent this transition system as a directed graph with nodes labeled by numbers. We maintain a separate list of states, indicating which state corresponds to each node in the graph. This approach allows for easy translation of states between representations of different libraries without modifying the graph structure. Similarly, we represent the initial state and goal states using the indices of the states in this list. This ensures they don't need to be changed when translating between different state representations.

We will represent states by sets of strings, which can be easily translated to and from both the state representations from the Tarski and the DLplan library.

This representation enables efficient caching, which will be discussed in detail later.

Furthermore, we use the Tarski library to parse the goal of the instance into a set of propositions (which is possible because the goal is a logical formula with only conjunctions) that we can use to find all goal states. For each goal state, we add a self-loop in the transition system. We do this because we use a model checker for infinite paths instead of finite paths. By adding a self-loop in the goal state, we can mimic finite paths that end in the goal.

5.4 Sketch generation

5.4.1 Feature extraction

We use the DLPlan library [8], made by Drexler et al., to calculate a feature pool and to evaluate features in states. Like the Tarski library, it has a representation of states and instances but also of features and sketches (called policies).

As we saw, features are functions from states to positive numbers or boolean values. In DLPlan, features are Description Logic (DL) formulas over the predicate symbols of a domain. Description logic consists of concepts and roles, which are respectively unary and binary relations over the set of objects of an instance. In addition to the roles and concepts in Description logics, Drexler et al. add a numerical grammar rule $n\_count(X)$ that counts the number of objects that adhere to a description logic formula and a boolean grammar rule $b\_empty(X)$ that expresses that the set of objects for which the DL formula X holds is empty. A feature's *complexity* is the number of nested concepts and rules used to construct the feature.

DLPlan can generate features based on a set of states. Although unclear in [10] how many states are used and of which instances, we use all states of all provided instances. As input, we needed to translate states in our representation to states in the DLplan representation.

In the Tarski library, instances are represented by predicate symbols and action schemas from a domain, and objects as we defined an instance in the theory. On the contrary, DLPlan represents an instance using a set of grounded predicates. In DLPlan, this instance information is necessary before it is possible to build a state. To translate states from the representation of the Tarski library to a representation of the DLPlan library, we needed to ground all predicates.

Additionally, we add a static goal predicates for each predicate that is (part of) the goal, as in [9, 10]. A *static predicate* is a predicate that is true in every state of an instance. These goal predicates allow us to build features that note whether some goal predicate is reached. For example in the blocksworld, if the goal is to stack *blockA* on *blockB*, we add grounded predicate $on_g(blockA, blockB)$ as a static predicate. This goal predicate allows us to make a feature that checks whether the goal is reached. Since we use the exact same naming for all propositions and objects in the Tarski, DLplan, and our own representation, the translation between states of the different representations is straightforward.

Once we have the DLPlan representation of all DLPlan states, we can use the DLPlan library to generate a feature pool and calculate the values of these features for each state. From this feature pool, we delete four features; "*b_empty*(*c_top*)", "*b_empty*(*c_bot*)", "*n_count*(*c_top*)", "*n_count*(*c_bot*)". These features are domain independent and have the same value in each state. Therefore, they are not informative.

Once all feature valuations are calculated, we will continue working with these features in a string representation. This way, our sketch representation does not rely on the DLPlan library. Once the features and their values are cached (more about caching later), there is no need to recalculate the instances or states in the representation of the DLPlan library. Additionally, this facilitates testing.

5.4.2 Sketch representation

The DLPlan library contains representations for policies that can also be used to represent sketches. We chose to make a new representation for sketches for three reasons. First, to generate a pool of candidate sketches, we start by making sketch rules that we combine into different sketches. The DLPlan library has no representation for separate rules but only for complete policies, which made it cumbersome to combine sketch rules into new sketches. Secondly, the conditions and effects in these policies are represented such that we can only know which condition or effect a policy uses by comparing its string representation, which is not elegant. Lastly, the way policies are constructed is inconvenient for the way we create a candidate sketch pool.

To represent sketches, we started by creating representations for each possible feature condition and feature value change. Besides the conditions mentioned previously ($n = 0$, $n > 0$, b, $\neg b$), we add a feature condition $f?$ for a numerical or boolean feature f. This condition represents that it doesn't matter which value f has and contains, thus the same meaning as not mentioning this feature in the conditions or effects. We add this feature condition only to simplify the implementation of the feature generation process, where we start with a set of features and assign a condition to each. For the same reason, we added $f?$ to the possible feature value changes ($n\uparrow$, $n\downarrow$, $n_=$, b, $\neg b$, $b_=$). After the sketch pool is generated, all feature conditions and value changes of the form $f?$ can be removed from the sketch.

5.4.3 Generation

We generate sketches starting with a pool of features Φ of maximal complexity k. For the sketch generation, we use two different parameters. Let f_{max} be the maximum number of features a sketch can use and r_{max} the maximum number of rules a sketch can have. We start by constructing all subsets of Φ of size f_{max} and smaller $\mathbb{F} = \{X \subseteq \Phi : |X| \leq f_{max}\}$.

Each feature can occur in three different conditions; $n = 0$, $n > 0$, $n?$ for a numerical feature n, and b, $\neg b$, $b?$ for a boolean feature b. For every set $\phi \in \mathbb{F}$, we create all possible conditions in which the features in ϕ can occur. For a feature set of, for example, two features, we would get $3^2 = 9$ possible conditions. For each of these possible conditions, we then calculate the possible effects. In this process, we already eliminated some options based on the conditions. For example, if the condition contains the numerical feature condition $n = 0$, we don't allow the feature value change $n\downarrow$ in the same rule. For a boolean feature condition b (resp. $\neg b$), we don't allow the effect $b_=$, since, in this case, it is equivalent to the effect b (resp. $\neg b$). For a feature condition $f?$, we don't allow the feature value change $f?$ since the feature could, in this case, be left out of the rule and is equivalent to a rule which uses fewer features. We also don't allow an effect to only have value changes of the form $f?$, since this would result in a rule in which the effect is empty.

Another possible constraint to set here is to allow the effect $n\downarrow$ only if $n > 0$ is mentioned in the conditions. This is because a feature cannot decrease if its value is not bigger than zero, so a rule $\{\} \rightarrow \{n\downarrow\}$ has essentially the same meaning as $\{n > 0\} \rightarrow \{n\downarrow\}$. We chose not to use this constraint since some of the sketches generated by Drexler et al. [10] are from this first form, and using this constraint would disallow us to reconstruct their learned sketches.

Once we have all possible conditions and all possible effects that match these conditions, we also have all possible sketch rules. By combining them in sets of r_{max} rules, we receive all sketches of size r_{max}.

In our code, we implemented these steps as generators. The number of candidate sketches grows fast when we increase f_{max} or r_{max}. Increasing f_{max} enlarges the number of possible rules since more combinations of features can be used in each rule. When increasing r_{max}, more combinations of rules are possible, increasing the number of sketches. Therefore, creating all candidate sketches is not always feasible or necessary. Using generators, we lazily create the feature sets, conditions, and rules.

5.5 Sketch verification

5.5.1 CTL_f^* to CTL and LTL

Linear temporal logic (LTL) and *Computational tree logic* (CTL) are both different subsets of CTL^*. LTL is a subset defined on traces, and CTL is a subset defined on states. The syntax of LTL is the following:
Let $p \in PV$ be a propositional variable.

$$\gamma ::= p \mid \neg\gamma \mid \gamma \wedge \gamma \mid X\gamma \mid \gamma U \gamma \mid Y\gamma \mid O\gamma.$$

Given a Kripke model $M = (St, R, V)$, and an infinite path $\lambda = (q_0, q_1, ...)$ with $q_i \in St$, the semantics of LTL with past are defined by the following clauses:

$\lambda, i \models p$ iff $\lambda[i] \in V(p)$
$\lambda, i \models \neg\gamma$ iff $\lambda, i \not\models \gamma$
$\lambda, i \models \gamma_1 \wedge \gamma_2$ iff $\lambda, i \models \gamma_1$ and $\lambda, i \models \gamma_2$
$\lambda, i \models X\gamma$ iff $\lambda, i+1 \models \gamma$
$\lambda, i \models \gamma_1 U \gamma_2$ iff there is a j with $i \leq j$ such that $\lambda, j \models \gamma_2$
 and for all k with $i \leq k < j,$ we have that $\lambda, k \models \gamma_1$
$\lambda, i \models Y\gamma$ iff $i > 0$ and $\lambda, i-1 \models \gamma$
$\lambda, i \models O\gamma$ iff there is a k with $0 \leq k \leq i$ such that $\lambda, k \models \gamma$

Analogous as before, we define additional operators:

$$F\gamma \equiv True U \gamma$$
$$G\gamma \equiv \neg F \neg \gamma$$

We define that LTL formula γ holds for a Kripke model M in state q iff $\lambda, 0 \models \gamma$ for every infinite path λ in M that starts in q.

Just like CTL_f^*, CTL has added path operators. The difference is that in CTL, path operators can only occur when followed by a state formula.

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid EX\varphi \mid EG\varphi \mid E\varphi U \varphi.$$

Its semantics is, just like the semantics of state formulae in CTL_f^*, defined on states. Let $M = (St, R, V)$ be a Kripke model and $q \in St$ a state.

$M, q \models p$ for $p \in PV$ if $q \in V(p)$
$M, q \models \neg\varphi$ iff $M, q \not\models \varphi$
$M, q \models \varphi_1 \wedge \varphi_2$ iff $M, q \models \varphi_1$ and $M, q \models \varphi_2$
$M, q \models EX\varphi$ iff there exists an infinite path λ starting from q
  such that $M, \lambda[1] \models \varphi$
$M, q \models EG\varphi$ iff there exists an infinite path λ starting from q
  such that $M, \lambda[i] \models \varphi$ for all $i \geq 0$
$M, q \models E\varphi_1 U \varphi_2$ iff there exists an infinite path λ starting from q
  such that $M, \lambda[i] \models \varphi_2$ for some $i \geq 0$ and $M, \lambda[j] \models \varphi_1$
  for all j with $0 \leq j \leq i$

Now we have defined LTL with past and CTL on infinite traces, we will translate the CTL_f^* constraints presented in section 4.3.3 into separate LTL and CTL formulas. The first two constraints can be translated into CTL, and the last into LTL.

The CTL_f^* formula of the first constraint (formula (1) in section 4.3.3) only uses state formulae directly combined with path formulae. Therefore we can interpret it directly as a CTL formula. The same goes for the second constraint (formula (2) in section 4.3.3).

The CTL_f^* formula of the third constraint doesn't contain any state formulae except the **A** wrapped around the entire formula. It can directly be interpreted as an LTL formula since LTL formulas are model checked on all paths that start from the initial state.

Given an instance with transition system M and initial state q. Using the NuSMV model checker, a sketch is good if the formulas of the first and second constraintboth hold as CTL statements on M in the initial state, and the third constraint holds in q as an LTL statement.

5.5.2 SMV-files

The NuSMV model checker uses smv-files as input. An smv-file specifies the transition system and initial state on which we want to model check. Additionally, the smv file can contain variables and their values in each state, which can be used in logical formulas. We will add the features and their feature values in each state as variables to this smv file. We also define a variable "goal" that we define to be only true in goal states. Lastly, we also add the conditions and effects of sketch rules as variables. This makes the logic constraints more readable, as we will see in section 5.5.4.

5.5.3 Expanded sketch

To translate a sketch rule $R : C \to E$ to a set of expanded sketch rules, we make a list of possible feature values such that we can combine them in the new conditions. We start by finding all feature value changes that rely on previous states and putting their features in a set Δ. These feature value changes are all numerical, plus the boolean case where a feature needs to keep its value.

$$\Delta := \{f : (f \uparrow \in E) \vee (f \uparrow \in E) \vee (f_= \in E)\}$$

For each feature, we search in the condition whether this feature has constraints (e.g. it needs to be bigger than 0) and write out all possible values this feature can have, considering the conditions. Next, we take all combinations of these possible feature values and make a new, expanded condition for each combination. To this condition, we add the feature conditions of all features that were not relevant before. Next, we create a new, expanded effect for each new, expanded condition, using feature variables.

The NuSMV model checker supports statements like "$n > 0$", which makes it possible to use feature variables directly in the input for the model checker.

5.5.4 Laws

In our code, we refer to the LTL and CTL translations of the constraints of a good sketch as *laws*. These laws contain disjunctions over the number of expanded sketch rules. Therefore, we created functions that take an integer k as input, and return a law, considering there are k expanded sketch rules. We call these functions *abstract laws*. We do not use the conditions and effects of the expanded rules directly into the law but use variables c_i and e_i for the conditions and effects of the i^{th} rule, which we define in the SMV file. For example, take an imaginary constraint $\bigwedge_i c_i \to e_i$. The abstract law is defined as $L(n) = \bigwedge_{0<i\leq n} c_i \to e_i$. If the expanded sketch we want to check has, e.g., two rules, we can use $L(2)$ and define c_1, c_2 and e_1, e_2 separately.

5.5.5 Model-checking

To verify our sketches, we use the NuSMV model checker and its Python bindings provided by PyNuSMV [5]. We order the laws such that CTL formulas are checked before the LTL formulas. We do this because, for most instances, it is computationally less expensive to check a CTL formula than it is to check an LTL formula. If one of the formulas fails, we no longer check the following formulas. Instead of writing the SMV encodings to files, we deliver them to the designated method directly.

5.6 Caching

During the process, we must often access the transition systems, features, and feature valuations. Building transition systems can take long, and it can require a lot of memory to keep them all in memory if the state spaces get big.

Therefore, we cache the transition systems and states encoded as strings and the init and goal states. Additionally, we also cache features and feature valuations. By caching the feature valuations, there is no need to keep the DLplan representation of all states as soon as the feature valuations are calculated once. Lastly, we also cache the generated good sketches, together with timings. When we generate all sketches with maximum n rules and maximum m features, we can use this result when generating sketches with maximum $n + 1$ rules and maximum m features without having to recreate the results.

	width 1				width 2			
	C	$\|F\|$	$\|R\|$	Found	C	$\|F\|$	$\|R\|$	Found
Blocksworld-Clear	4	1	1	✓	4	1	1	✓
Blocksworld-On	4	2	2	✓	4	1	1	✓
Delivery	4	2	2	✓	4/5	1	1	✓
Gripper	4	2	2	✓	4	1	1	✓
Miconic	2	2	2	✓	2	1	1	✓
Reward	2	1	1	✓	2	1	1	✓
Spanner	5/6	1	1	X	5/6	1	1	X
Visitall	2	1	1	✓	2	1	1	✓

Table 1: *The maximum feature complexity (C), number of features (F) and number of rules (R) of the sketches found by Drexler et al. [10] per sketch width. A checkmark (✓) means we were able to reconstruct and verify the sketch. A cross (X) means we were not able to verify this sketch. When two different complexities are reported, this means that one of the features used in Drexler et al.'s sketch was not present in our feature pool, and we used an equivalent feature of complexity+1.*

6 Experiments

6.1 Specifications

To verify our method, we aimed to reconstruct the sketches that Drexler et al. generated in their paper about learning sketches [10]. While they only mention some of their generated sketches in the paper, the remaining sketches can be found in their supplementary material [11]. Drexler et al. generate sketches of widths zero, one, and two for nine planning domains. The domains, number of features, maximum complexity of a feature, and number of rules each of these sketches has can be found in table 1. Out of the seventeen sketches of widths one and two, twelve contain only one rule and one feature, and four sketches contain two rules and two features. Therefore, we ran two experiments:

- Experiment 1: Generate and verify all possible candidate sketches with one rule and one feature.
- Experiment 2: Generate and verify all possible candidate sketches with two rules and two features.

We ran the first experiment for eight of the nine domains Drexler et al. used. We did not use the Childsnack domain. Drexler et al. constructed only one sketch for this domain which contained four features and five rules. We did not have the computational resources to run an experiment of this size. For the second experiment, we used the Blocksworld-On, Delivery, Gripper, and Miconic domains. For the other domains, none of the Drexler sketches had two rules and two features, so that no Drexler sketch could be found in the second experiment.

We did not reconstruct the width zero sketches due to reasons explained in section 6.4. We used the feature complexity reported by Drexler et al. for each domain as the maximum feature complexity. Exceptions are the Delivery and Spanner domain, for which the features used by Drexler et al. did not show up in our feature pool, due to changes in the DLPlan library. Therefore we had to use equivalent features that had complexity +1. For experiment 1, we used the same instances as Drexler et al. used in their experiments. Some domains contained duplicate instances, which we removed before running our experiments. We will provide more details about the used instances per domain in the remainder of this section. The pool of candidate sketches and the model check time per sketch increased for the experiments with two features and two rules. Our initial idea was to set a time-out on model checking a sketch such that it would skip the sketch and go to the next one in case the model checking took too long. Unfortunately, PyNuSMV did not support the early stopping of the model-checking process. Therefore, we only ran the experiments on instances where a sketch could be checked in less than ten minutes.

We set an overall maximum time of 24 hours per domain.

We ran our experiments on a laptop with an AMD CPU with 6900HS on a Fedora 37 Linux distribution. We had access to 32 GB RAM but never used more than 2GB per domain.

6.2 Domains

Blocksworld-Clear The Blocksworld domain consists of a table with blocks and a robot arm. The robot arm can pick up one block at a time to stack and unstack the blocks on the table. In Blocksworld-Clear, the goal is to reach a state where no other blocks are stacked upon a specific block.

Blocksworld-On The Blocksworld-On problem is based on the Blocksworld domain. The goal is to have two specific blocks stacked on top of each other.

Delivery In the delivery environment, a truck has to pick up packages and deliver them to a target destination.

Gripper In the Gripper domain, several balls are lying in room A. A robot with two arms can pick up one ball per arm and move between rooms A and B. The goal is to bring all balls to room B.

Miconic In Miconic, passengers on different floors wait for an elevator to pick them up and bring them to their desired floor. When the elevator is on the floor of a passenger, they can board it, and only when it is at their goal destination, they can depart.

Reward In the Reward domain, some rewards are placed around the floor. The goal is for an agent to pick up all the rewards. The environment is implemented as a square grid, in which only specific cells are accessible.

	$\|I\|$	C	$\|F\|$	$\|CS\|$	$\|GS\|$	$\overline{t_{failed}}$	$\overline{t_{good}}$	$\overline{t_{good}}/I$	t_{total}
Blocksworld-Clear	237	4	26	199	16	0.2	143.24	0.6	2333.78
Blocksworld-On	231	4	28	215	1	0.18	751.05	3.25	794.54
Delivery	10	5	29	229	5	0.01	0.26	0.03	4.73
Gripper	5	4	34	263	5	0.01	3.08	0.62	18.67
Miconic	503	2	9	69	1	0.2	155.54	0.31	170.29
Reward	71	2	8	62	5	0.02	1.78	0.03	10.28
Spanner	346	6	68	535	6	0.78	417.55	1.21	2950.89
Visitall	177	2	5	39	2	0.06	102.26	0.58	206.98

Table 2: *Data of experiment 1: generating and verifying sketches with max 1 feature and max 1 rule.*
$|I|$ is the number of domains we used to verify the candidate sketches.
C is the maximum feature complexity.
$|F|$ is the number of features generated.
$|CS|$ is the number of candidate sketches generated and tested.
$|GS|$ is the number of good sketches found.
$\overline{t_{failed}}$ is the time in seconds it takes on average to check a sketch that is not good.
$\overline{t_{good}}$ is the time in seconds it takes on average to check that a sketch is good.
$\overline{t_{good}}/I$ is the average time in seconds to check when a sketch is good per instance.
t_{total} is the total time in seconds that was needed to verify all candidate sketches.

	$\|I\|$	C	$\|F\|$	$\|CS\|$	$\|GS\|$	$\overline{t_{failed}}$	$\overline{t_{good}}$	$\overline{t_{good}}/I$	$t_{total}(h)$
Blocksworld-On	13	4	28	2073802	6209	0.02	0.74	0.06	17.16
Delivery	10	4	19	1021086	9189	0.02	0.61	0.06	9.76
Gripper	2	4	34	2761231	37228	0.01	0.05	0.02	24.0
Miconic	19	2	9	196355	2914	0.02	1.54	0.08	3.39

Table 3: *Data of experiment 2: generating and verifying sketches with max two features and two rules. The symbols in the heading have the same meaning as in table 2. The total time is represented in hours.*

Spanner In the spanner domain, there is a man who can only move forward standing at the start of a corridor. At the end of the corridor is a gate with some nuts. On the way to the gate, spanners are laid out on the floor, which the man can pick up. The goal is to use the spanners to tighten all nuts. Each spanner can only be used once.

Visitall In Visitall, an agent has to visit specific cells of a rectangular grid at least once.

6.3 Results

To have an overall idea of the number of candidate sketches, the number of good sketches, and the time our method takes to verify good sketches, we provide an overview of this information for both experiments. In table 2, we show data for experiment 1. One thing we report is the number of instances used to verify

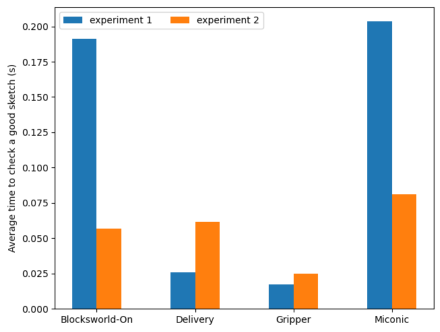

Figure 3: *Average time it takes to verify a good sketch on an instance for sketches with one sketch rule and one feature (experiment 1), and two sketch rules and two features (experiment 2) over the instances used in both experiments.*

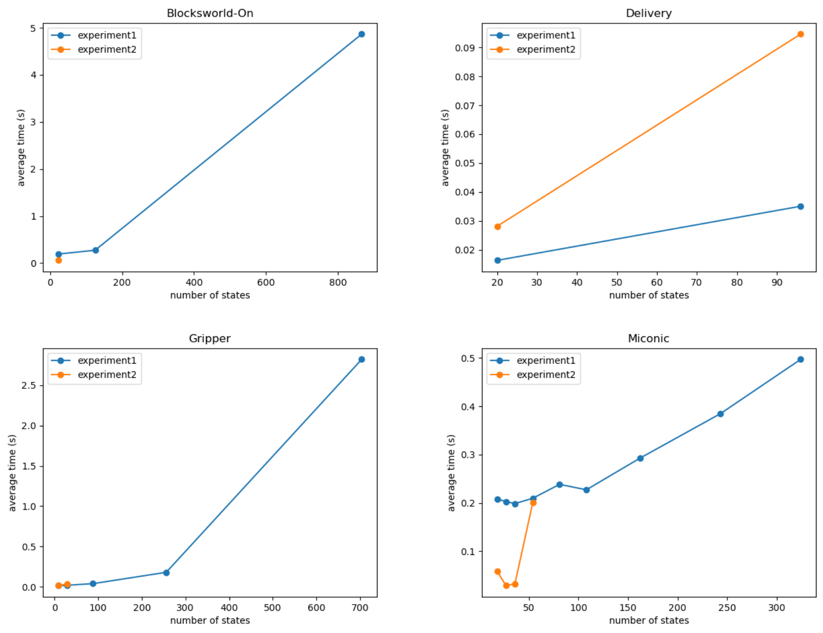

Figure 4: *Average time to verify a good sketch on an instance per number of states of the instance.*

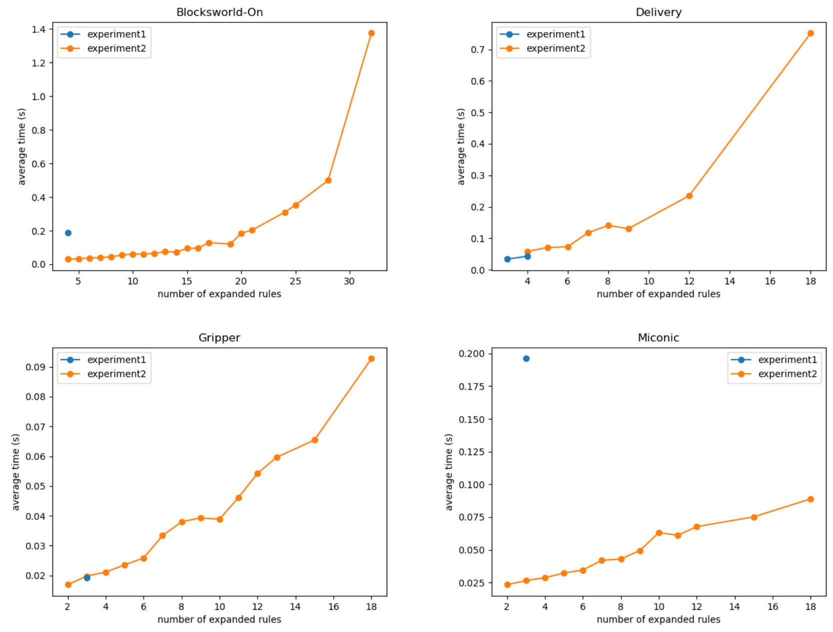

Figure 5: *Average time to verify a good sketch over one specific instance per number of expanded sketch rules. For each of the domains, we use the timings of verifying sketches over largest instance that was used for both experiment 1 and experiment 2.*

the candidate sketches for each domain. More details about these instances are discussed later in the chapter. For each candidate sketch, we saved the time to model-check this sketch over each used instance. We used these timings to calculate the average amount of time it takes to model-check a sketch that is not good and the average amount of time it takes to verify a good sketch. Since every domain uses a different number of instances, we also provide the average time to check a good sketch per instance. The total time reported in this table is the time to run our program, given that the transition systems, features, and feature valuations are already cached. For experiment 1, we report all good sketches we found in Appendix B.

Table 3 shows the same information for experiment 2, generating and verifying sketches with two rules and two features. Again, we will give more details about the used instances in the remainder of this chapter. For this experiment, we report the total time in hours. Since there were thousands of good sketches for each instance, we don't provide a list of them, but we will discuss a few in the remainder of this chapter.

After running both experiments, we obtained a set of good sketches for each experiment and each domain. In these sets, we checked whether the sketches learned by Drexler et al. [10] were present. Table 1 gives an overview of which Drexler sketches we found and which we didn't. One can see that we found the width-1 and width-2 sketches for all domains except for the Spanner domain.

To fairly compare the time it takes to verify a good sketch with one rule and one feature with the time it takes to verify a sketch with two rules and two features, we created a bar chart in which we took an average of the timings for model-checking sketches only over instances used in both experiments. This chart can be found in figure 3. We see that for the Gripper domain and the Delivery domain, it takes longer, on average, to verify sketches with two rules and two features (experiment 2) than it takes to verify one sketch and one feature (experiment 1), as we would expect. For domains Blocksworld-On and Miconic, it is the opposite way around.

To capture the effect of the size of an instance on the time to model-check a good sketch, we show the average time it takes to verify a good sketch on an instance, given the number of states of its transition system, for both experiments in figure 4. This figure was created by calculating the number of states per instance. For each good sketch, we calculated the average time it takes to verify this sketch over instances of size n. Next, we took the average of these numbers for all sketches.

To examine the effect of the number of expanded sketch rules on the time it takes to verify a good sketch, we made figure 5, which shows the average time it takes to verify a good sketch, per number of expanded sketch rules. These results were obtained using a single instance, as the instance size affects both the number of expanded rules and the time required for model checking. To achieve the broadest range of expanded rule numbers, we selected the largest instance shared between experiment 1 and experiment 2 for each domain. For each sketch, we calculated the number of expanded sketch rules for all instances and took the average time needed to verify a sketch over all instances in which the sketch has n expanded sketch rules. For the same n, we then took the average of these averages over all sketches.

In the remainder of this chapter, we will discuss the Drexler sketches we found and didn't verify for each experiment and discuss some of the other good sketches we found.

6.3.1 Experiment 1

Blocksworld-Clear For the Blocksworld domain, Drexler et al. generated 600 instances; 200 with three blocks, 200 with four blocks, and 200 with five blocks. After removing duplicate instances, we ran our experiments on 237 instances, of which 13 with three blocks, 71 with four blocks, and 153 with five blocks.

The sketches of width one and width two found by Drexler et al. have one rule and one feature, so we aimed to find them in this first experiment. The width-1 sketch for this domain is the following:

$$\{\} \rightarrow \{n \uparrow\}$$

in which n is a numerical feature that counts the number of blocks are clear and should be clear in a goal state. The sketch thus says that the number of blocks that should be clear and are clear should always increase.

The width-2 sketch for this domain found by Drexler et al. is the following:

$$\{b\} \rightarrow \{\neg b\}$$

With b a boolean feature that states that none of the blocks that need to be cleared are clear. Both these sketches were present in our generated set of good sketches.

Besides the two Drexler sketches, we generated 14 other good sketches. We present them in the appendix B. Interestingly, one of the other sketches that we found is equivalent to Drexler's width-1 sketch:

$$\{n = 0\} \rightarrow \{n \uparrow\}$$

with n the same feature as defined above. This sketch states that if no goal blocks are clear, it is desired to increase the number of clear goal blocks. In Blocksworld-Clear, there is only one block that needs to be cleared. Therefore n is always zero except in a goal state, and this sketch is equivalent to the width-1 Drexler sketch.

Analogous, we found an equivalent sketch for Drexler's width-2 sketch:

$$\{\} \rightarrow \{\neg b\}$$

with b the same feature as before. This sketch states that it is always desired to reach a state where some goal blocks are clear.

Blocksworld-On Similarly to Blocksworld-Clear, Drexler et al. generated 600 instances in total, divided over instances with three, four, and five blocks. After removing duplicate instances, we ran our experiments on 231 instances, of which 13 with three blocks, 68 with four blocks, and 150 with five blocks.

For this domain, we found only one sketch with one rule and one feature, which is the same sketch as the width-2 sketch of Drexler et al.

$$\{\} \rightarrow \{n \uparrow\}$$

With n a numerical feature that counts the number of blocks that are correctly stacked upon each other.

Delivery Like Drexler et. al, we used only instances where the environment is a 2x2 grid with one or two packages. Five instances were generated for each configuration, resulting in 10 instance files.

For Delivery, Drexler et. al found following width-2 sketch:

$$\{\} \rightarrow \{n \uparrow\}$$

with n the number of objects that are at their goal location. This sketch expresses that the number of objects at one of the goal locations should always increase.

When generating features using the DLPlan library, our feature pool did not include the exact feature used in Drexler et. al's sketch. This is because, due to an update in the DLPlan library, the generator does not create concepts for predicates of arity two and higher. Therefore our feature pool did not include this feature. Nevertheless, we were able to find a feature with the same feature evaluations as n. This new feature has complexity five instead of four for Drexlers feature. Therefore, we ran the experiment with a maximum feature complexity of five.

With this equivalent feature, we could reconstruct the width-2 sketch for delivery, together with four other sketches.

Two of the other sketches that we found are:

$$\{\} \rightarrow \{n_0 \uparrow\}$$

with n_0 the amount of objects (i.e. packages) at their goal location and

$$\{\} \rightarrow \{n_1 \downarrow\}$$

with n_1 the amount of packages that are not at their goal location. Interestingly, n_0 and n_1 are each others complements. Therefore, both sketches have an analogous meaning.

Gripper Drexler et al. constructed instances with one to five balls for the gripper domain. Since the initial state is always the same (all balls and the robot start in room A), there is only one instance for each number of objects. Therefore we used five instances in total.

The width-2 sketch for the Gripper domain found by Drexler et al. has one rule and one feature:

$$\{\} \rightarrow \{n \uparrow\}$$

with n counting the number of well-placed balls. This sketch expresses that the number of well-placed balls should always increase.

We were able to find and verify this sketch, together with four other good sketches.

Miconic As in Drexler et al., we used instances with two, three, and four floors, and for each number of floors, two, three, or four passengers. Drexler et al. generated 100 instances for each configuration, resulting in 900 instances in total. After removing all duplicates, 503 instances remained.

We only found one good sketch with one rule and feature for Miconic, which was the same as Drexler's width-2 sketch:

$$\{\} \rightarrow \{n \uparrow\}$$

with n the number of passergers that are *served*. Passengers are *served* when they arrive on their target floor. The sketch thus says the number of passengers on their desired floor should increase.

Reward Drexler et al. generated 50 instances with a grid of 2x2 and 50 instances with a grid of 3x3. Each of the instances has a random number of accessible cells and a random number of rewards. After removing duplicate instances, we ended with 21 instances of 2x2 and 50 instances of 3x3, which gave us 71 instances.

For the Reward domain, the width-1 and width-2 sketches are the same. The sketch has one rule that states that the number of rewards on the floor should decrease. The sketch is the following:

$$\{\} \rightarrow \{n \downarrow\}$$

with n the number of rewards that are lying on the floor. The sketch says that one should always decrease the number of rewards on the floor. The only way to do this is by picking up rewards.

We were able to verify this sketch, together with four other sketches.

Spanner For the Spanner domain, we used instances with a corridor of three to five tiles, for each corridor length three to five nuts, and the number of spanners equal to or bigger than the number of nuts with a maximum of five spanners. For each configuration of variables, Drexler et al. provided 20 instances. After removing duplicates, 346 instances remained.

In Drexler et al., the width-1 and width-2 sketches found for the Spanner domain are the same:

$$\{\} \rightarrow \{n \downarrow\}$$

With n counting the number of unpicked spanners plus the number of loose nuts. Thus, the sketch says picking up a spanner or tightening a nut is good.

Our algorithm generates this sketch but is not considered as good. The reason for this lies in constraint two; no rules can lead to a dead state. One can decrease n by picking up a spanner, walking to the gate, and tightening a nut. If there are more nuts in the instance, a dead state is reached since the man cannot go back to pick up more spanners.

On the contrary, Drexler et al. require that if the effect of a rule holds in a dead state, there must be an alive state closer to the current state that can be reached by applying one of the rules. Therefore, this sketch is valid according to their definition. Their definition is based on the SIW_R algorithm, which will only search for the closest state that satisfies an effect. Since we do not assume the algorithm the sketches will be used in, our definition of a good sketch is different.

We did find six good sketches that can be found in the appendix. The first of these sketches is the following:

$$\{\} \rightarrow \{n_0 \uparrow\}$$

With n_0 counting the locations where no loose nuts are. The sketch encodes that tightening all nuts at a location is good.

Another working sketch we find is

$$\{n_0 > 0\} \rightarrow \{n_0 \uparrow\}$$

with the same feature n_0. Noticeable here is that n_0 is always positive, as there are only nuts at the gate and not at the other locations in the hallway. Therefore, both these sketches have the same meaning.

Visitall For the Visitall domain, we had to remove some ill-defined instances with cells disconnected from the other cells such that reaching the goal became impossible. Drexler et al. created 50 instances per feature setting, with grids of size 2x2 and 3x3, goal cells 50 percent of the grid, and 100 percent of the grid, which gave 400 files. After removing duplicate and ill-defined instances, we had 177 instances left.

The sketch Drexler et al. found for both width one and width two are the same; increase the number of visited cells:

$$\{\} \rightarrow \{n \uparrow\}$$

with n counting the number of cells that are visited. The sketch expresses that it is good to visit new cells.

We found two good sketches for one rule and one feature, one of which is the above sketch of Drexler et al. The other sketch is the following:

$$\{n > 0\} \rightarrow \{n \uparrow\}$$

for n the same feature. Since the agent has to start on a cell, there is always one cell already visited. Therefore the condition $n > 0$ always holds, and both found sketches are equivalent.

6.3.2 Experiment 2

Blocksworld-On For the second experiment, we only used the instances with three blocks, which are 13 instances.

Drexler et al.'s width-1 sketch for the Blocksworld-On domain is the following:

$$\begin{aligned} r_1 :& \{\} \rightarrow \{b, n \uparrow\} \\ r_2 :& \{\neg b\} \rightarrow \{b\} \end{aligned}$$

With n a feature counting the number of blocks that are correctly stacked upon each other and b expressing whether the robot arm is empty. The sketch says that it is always good to stack blocks on top of each other that should be stacked in the goal, and when you are still holding a block, it is good to put it down.

We found 6209 good sketches with two rules and features, including Drexlers width-1 sketch.

Delivery For the second experiment, we use the same instances as we did for the first experiment, which gives us ten instances in total.

Drexler's width-1 sketch is the following:

$$r_1 : \{\} \rightarrow \{n_1{=}, n_0 \downarrow\}$$
$$r_2 : \{n_0 > 0\} \rightarrow \{n_1 \uparrow\}$$

with n_0 the number of objects that are at any location, and n_1 the number of packages that are at their goal location. The first rule states that it is always good to decrease the number of objects at a location while keeping the number of packages at their goal location equal. In other words, picking up a package that is not at its goal location is always good. The second rule states that whenever there are some objects at a location (which is always the case because the truck is also an object that cannot be picked up), it is desired to increase the number of items at their goal location.

Our feature pool did not include the exact n_0 used in the Drexler sketches for the same reason as in experiment 1 but did include a feature with the same feature valuations in all states.

We found 9188 good sketches with two rules and features for the Delivery domain, including the width-1 sketch of Drexler with the equivalent feature.

Since Delivery is the only domain in which we used the same set of instances for both experiments, we can compare the time results. We see that both the average time to model-check a bad sketch and the average time to check a good sketch doubled.

Gripper We used the instances with up to two balls for the experiment with two rules and features.

Drexlers width-1 sketch for the Gripper domain is the following:

$$r_1 : \{\} \rightarrow \{n_1 \downarrow\}$$
$$r_2 : \{\} \rightarrow \{n_0 \uparrow, n_1{=}\}$$

with n_0 the number of balls that are lying in one of the rooms, and n_1 the number of balls in room A.

Analogous to the features in the Delivery domain, n_0 is not in our feature pool, but we can substitute it with an equivalent feature. With this alternative feature, Drexler's sketch was present in our set of good sketches. In total we found 37228 good sketches with two rules and maximum two features.

Another interesting sketch we found is the following:

$$r_1 : \{\neg b\} \rightarrow \{n \uparrow\}$$
$$r_2 : \{b\} \rightarrow \{n \uparrow\}$$

In which n the same feature as the Drexler width-2 sketch; counting the number of balls in room A, and b expressing whether the robot is not carrying any balls.

The sketch says that if the robot is not carrying any balls, the number of balls in room A should increase, and if the robot is carrying a ball, the number of balls in room A should increase. What we see is that this sketch is equivalent to Drexlers width-2 sketch since in any state we should try to increase n.

Analogous we found a sketch:

$$\begin{aligned} r_1 :&\{n_2 = 0\} \rightarrow \{n \uparrow\} \\ r_2&\{n_2 > 0\} \rightarrow \{n \uparrow\} \end{aligned}$$

in which n_2 is the number of locations at which balls are present.

In total, we found 544 sketches in which the effect of both rules were $\{n \uparrow\}$.

Miconic For the second experiment, we used instances with two floors and two or three passengers and instances with three or four floors and two passengers. For each parameter setting, we used up to five instances if available, which gave us a total of 19 instances.

The width-1 sketch found by Drexler is the following:

$$\begin{aligned} r_1 :&\{\} \rightarrow \{n_1 \uparrow\} \\ r_2 :&\{\} \rightarrow \{n_0 \uparrow\} \end{aligned}$$

with n_0 the amount of passengers in the elevator and n_1 the number of passengers who arrived at their goal location. r_1 says it is always good to bring a passenger to their goal destination, and r_2 says it is always good to let a passenger board the elevator.

We found 2914 good sketches with two rules and features. This set included the above width-1 sketch.

An interesting sketch in our set of good sketches is the following

$$\begin{aligned} r_1 :&\{\} \rightarrow \{n_1 \uparrow\} \\ r_2 :&\{\} \rightarrow \{n_1 \downarrow\} \end{aligned}$$

where n_1 has the same definition as in the width-1 Drexler sketch; the number of passengers that are *served*, i.e., at their goal destination. What we see here is that the rules contradict each other. Still, this is a good sketch. Due to the nature of the Miconic domain, passengers only board the elevator when they are not at their goal position and only step out of the elevator when they arrive at their desired floor, which results in being served. Therefore, once a passenger is served, they will not board the elevator again, and it is impossible not to be served anymore. Therefore, the number of served passengers can never decrease, and r_2 can never be followed. Despite this rule, the sketch still adheres to our definition of a good sketch since it should only be possible to follow one rule from a state, and we never mentioned that it should be possible to use each rule. 414 other good sketches contain $n_1 \downarrow$ in one of their effects.

Another interesting sketch we found had the same pattern as the sketch shown in the results of experiment two of the Gripper domain:

$$\begin{gathered}\{b\} \rightarrow \{n_1 \uparrow\} \\ \{\neg b\} \rightarrow \{n_1 \uparrow\}\end{gathered}$$

with b true only if no passengers are served yet, and n_1 again the number of passengers that are at their target floor. In total, we found 93 sketches in which both effects were $\{n_1 \uparrow\}$.

6.4 Width Zero sketches

We did not find any of the width zero sketches. In this section, we will discuss why this was the case for each domain.

Blocksworld-Clear and Spanner The width zero sketch of the blocks-clear domain is the following:

$$\begin{aligned} r_1 :& \{\} \rightarrow \{b, n_=\} \\ r_2 :& \{n > 0\} \rightarrow \{\neg b, n \downarrow\}\end{aligned}$$

In which n is the number of blocks that are stacked upon another block, and b is true when the robot isn't holding any blocks. The sketch thus says that one always wants to reach a state where the robot is not holding a block anymore while not adjusting the number of stacked blocks. If blocks are still stacked, one wants to go to a state where the robot is holding a block, and the number of stacked blocks decreases. This is not a good sketch in our definition. Consider a blocksworld with two blocks, A and B. Define s_1 as the state where block A is stacked upon block B, and s_2 as the state where the robot is holding block A and block B lies upon the table. The pair (s_1, s_1) then follows r_1. The transition from s_1 to itself is possible in the path (s_1, s_2, s_1). Following r_1 can thus cause an eternal loop and is not a good sketch according to our definition. Drexler et al. consider this sketch a valid one since their search algorithm only looks at the closest state satisfying a rule. Therefore, from s_1, they will use r_2 to get to state s_2, and from s_2 a new rule will be applied.

The width zero sketch of Spanner is the following:

$$\begin{aligned} r_1 :& \{\} \rightarrow \{n_0 \downarrow\} \\ r_2 :& \{n_1 > 0\} \rightarrow \{\}\end{aligned}$$

with n_0 the number of nuts that are not tightened plus the number of spanners that are not picked up yet.

To satisfy the first rule, the man can walk all the way to the last spanner and pick it up. This decreases the number of spanners that are not picked up, and therefore n_0. Because the man cannot walk back, this rule leads to a dead state. Again, this sketch is valid according to Drexler et al. because they only consider the closest state in which an effect will hold.

Blocksworld-On and Gripper The width zero sketch found by Drexler et al. for Blocksworld-On contains three rules and three features. The one for Gripper contains two features and three rules. Since we only ran our program with a maximum of two rules and two features, we did not attempt to find these sketches.

Reward and Visitall Drexler et al. added a distance feature to find a sketch of width zero for the Reward and Visitall domains. When trying to calculate the feature values of these distance features in states from larger instances, our program crashed. Therefore, we did not attempt to find these sketches.

7 Discussion and Future work

7.1 Discussion

We reconstructed the width-1 and width-2 sketches of Drexler et al. of seven of the nine domains they used to test their method. We did not try to reconstruct the sketches of the Childsnack domain, and the sketch for the Spanner domain wasn't model-checked as good. We think this is due to the difference between our definition and Drexler et al.'s definition of a good sketch.

Besides the sketches of Drexler et al., we also found other good sketches. When generating sketches with only one rule and one feature, one to sixteen sketches per domain were model-checked as good out of the 39 to 535 candidate sketches we generated. We noticed that we often get pairs of good sketches with the same meaning. Two patterns we see are

1. $\{\} \rightarrow \{x\} \equiv \{y\} \rightarrow \{x\}$ with y true in every alive state (except the goal states)
2. $\{x\} \rightarrow \{n\_count(C) \uparrow\} \equiv \{x\} \rightarrow \{n\_count(not(C)) \downarrow\}$ in which $not(C)$ is the complement of C.

We showed examples of this in the Blocksworld-Clear domain, the Delivery domain, the Spanner domain, and the Visitall domain. If we would remove these equivalent sketches in Blocksworld-Clear, we have nine of the sixteen sketches remaining. We would have three out of five remaining for Delivery and Gripper and three out of six for the Spanner domain.

As expected, the candidate sketch pool for two rules and features is much bigger than the one in the first experiment. The candidate sketch pools contained two-hundred thousand to two million candidate sketches. For each domain, two thousand to thirty thousand sketches were verified as good. Due to this large number of good sketches, we were not able to look at all of them. Nevertheless, we found some interesting patterns.

In, e.g., the Miconic domain, we found a sketch in which one rule would be a good sketch on its own, and the second rule had an effect that could never be reached. Four hundred thirteen other good sketches contained this effect in one of their rules, which is 14% of the good sketches we found. The effect can be combined with any possible condition to construct an inapplicable rule. Additionally, since we use two features, the effect can be combined with other feature value changes in effects that contain more feature value changes. Rules that contain this effect can be combined with any other rule that is a sketch on its own with two features, significantly enlarging the set of good sketches.

Another pattern that was present in different domains was a sketch rule:

$$\begin{aligned} r_1 &: \{x\} \rightarrow \{z\} \\ r_2 &: \{y\} \rightarrow \{z\} \end{aligned}$$

in which, in every alive state, either x or y is true (which is always the case in good sketches with two rules). This sketch is equivalent to a sketch with one rule of the form $\{\} \rightarrow \{z\}$. We showed an example of this in the Gripper domain, where we found 544 sketches with this pattern for z being $n \uparrow$, the effect of the Drexler width-2 sketch, which is 1.5% of the total amount of sketches we found. In the Miconic domain, we found 93 sketches in which both rules had the same effect as the Drexler width-2 sketch, which is only 0.03 % of the total amount of good sketches we found. Nevertheless, we hypothesize that for any other good sketch of the form $\{\} \rightarrow \{z\}$, we will encounter equivalent sketches with two rules such that in each state, one of their conditions hold, and both effects are $\{z\}$.

As we can see in tables 2 and 3, it takes between 1 and 78 milliseconds to find out that a sketch is not good. Verifying a good sketch takes longer. The time it takes depends on the number of expanded sketch rules and the number and size of the instances.
We expect it will take longer to check a sketch in experiment two than in experiment one. This is because increased features and rules result in longer logical formulas. In figure 3, we see this is true for domains Delivery and Gripper but not for Blocksworld-On and Miconic. We have no apparent reason for this phenomenon. One hypothesis was that since for both Blocksworld-On and Miconic, we only found one good sketch in experiment 1, we couldn't take the average over many sketches, and maybe these were coincidental sketches with many expanded sketch rules. But this theory was disproven by figure 5, where we can compare the time it takes to model-check a sketch given the number of expanded sketch rules. In these figures, we see that, even for only sketches with the same amount of expanded sketch rules, it still takes longer to model-check the sketch of experiment 1 than the sketches for experiment 2 for the Blocksworld-On and Miconic domains. Another theory is that it is coincidental by a bad run, and since we cannot take the average over other sketches, it creates an outlier, but this is merely speculation.

We expected that the time to model-check over an instance would increase for instances with a larger state space. Figure 4 confirms this theory. The increase in time has two reasons; first, for larger state spaces, the model-checker needs to consider more paths, resulting in a longer model-checking time. Secondly, for larger instances, features can have more different values, which results in more extensive logical formulas and, thus, an increase in model checking time. Figure 5 indeed shows that an increase in the number of expanded sketch rules increases the time it takes to model-check a sketch. We also see that the number of expanded sketch rules can increase fast. For sketches with only two rules and two features, the number of expanded sketch rules goes up to 18 for Delivery, Gripper, and Miconic and even up to 30 for a Blocksworld-On instance with three balls.

7.2 Limitations

One of the limitations of our method is that model-checking a good sketch can take a long time for large instances, especially when the sketch has many rules and features. When expanding a sketch, the number of rules blows up when the features can have many values. The more expanded sketch rules, the longer the LTL and CTL formulas become and the longer the model-checking process takes.

Another limitation is the combinatorial blowup of our sketch pool. The number of sketches in our candidate sketch pool grows tremendously with the size of our feature pool, the maximum number of features, and the maximum number of rules. To find that a sketch is not good takes several milliseconds, but depending on the amount and size of the instances, checking whether a sketch is good can take minutes for sketches with one rule and one feature. For sketches with two rules and two features, this can take hours for big instances. For a large number of sketch rules and features, it can be infeasible to find all possible good sketches.

Interestingly, our method finds all good sketches for a given domain. We saw that the number of good sketches can be very big when we allow more than one rule and feature in a sketch. For two rules and two features, thousands of sketches were found. Increasing the number of rules and features will result in even more good sketches. This large number of good sketches is impractical and suggests the need of additional conditions to reduce this number, or a metric to have an ordering on the good sketches.

Lastly, we don't use a complexity measure for our sketches. Therefore, we cannot know which of our good sketches will speed up the search for a plan faster than others.

7.3 Future work

We will discuss six directions of future work that we find interesting.

Reduce pool of candidate sketches For future work, we would like to reduce the pool of candidate sketches to speed up the search for a good sketch. We have four different ideas to address this. First, we can require the set of features in a sketch to distinguish the goals as was in the original definition of sketches [4]. By eliminating feature sets that fail to distinguish the goals, we can significantly decrease the number of feature sets and, consequently, the size of the candidate sketch pool. In the Miconic domain, we saw that sometimes it is impossible to follow one of the rules in a sketch. We might be able to find these rules already in this step and delete them before we combine them with other rules in sketches. A problem here is that a sketch rule that is impossible to follow in one instance can be necessary in another. Therefore this method would need some care. Another idea is to find a CTL_f^* formula that we can use to check whether a rule would always result in an inadequate sketch. An example is the first constraint we check; a rule can never lead to a dead state. If we can already filter out rules that do not adhere to this constraint, our final pool of sketches will be smaller.

Moreover, we can identify and eliminate equivalent sketch rules and sketches as they exhibit redundancy. Examples of equivalent sketch rules were seen in the result section of Blocksworld-Clear, Delivery, Spanner, and Visitall.

Other ways to generate candidate sketch pool We described a way to verify whether a sketch is good. We want to explore more efficient ways to generate candidate sketches that can be verified using our method. For example, given a feature set that distinguishes the goal, we can check the feature valuations in goal states to know in which direction the feature values should be manipulated eventually. This information might be used to generate sketches that are more feasible than any randomly generated sketch, and the search for good sketches can be more directed. We can use our verification method to check whether these sketches are good.

Speed up the verification process Another relevant direction of future work is to speed up the model-checking process. We observed that the number of expanded rules can grow rapidly, resulting in longer model-checking times. One solution might be to merge expanded rules where possible. Consider, for example, the Drexler width-1 sketch from the Gripper domain:

$$\begin{aligned} r_1 &: \{\} \rightarrow \{n_1 \downarrow\} \\ r_2 &: \{\} \rightarrow \{n_0 \uparrow, n_1 =\} \end{aligned}$$

In an instance where n_0 and n_1 can be equal to 1 or 2, the expanded sketch will look like:

$$\begin{aligned} r_{1_1} &: \{n_1 = 1\} \rightarrow \{n_1 < 1\} \\ r_{1_2} &: \{n_1 = 2\} \rightarrow \{n_1 < 2\} \\ r_{2_1} &: \{n_0 = 1, n_1 = 1\} \rightarrow \{n_0 > 1, n_1 = 1\} \\ r_{2_2} &: \{n_0 = 2, n_1 = 1\} \rightarrow \{n_0 > 2, n_1 = 1\} \\ r_{2_3} &: \{n_0 = 1, n_1 = 2\} \rightarrow \{n_0 > 1, n_1 = 2\} \\ r_{2_4} &: \{n_0 = 2, n_1 = 2\} \rightarrow \{n_0 > 2, n_1 = 2\} \end{aligned}$$

First of all we can delete rules r_{2_2} and r_{2_4}, since $n_0 > 2$ will never be true. Allowing disjunctions in the effects of expanded sketch rules, we can reduce these rules to:

$$\begin{aligned} r_{1_1 2_1} &: \{n_0 = 1, n_1 = 1\} \rightarrow \{(n_1 < 1) \lor (n_0 > 1 \land n_1 = 1)\} \\ r_{1_1 2_2} &: \{n_0 = 2, n_1 = 1\} \rightarrow \{(n_1 < 1) \lor (n_0 > 2 \land n_1 = 1)\} \\ r_{1_2 2_3} &: \{n_0 = 1, n_1 = 2\} \rightarrow \{(n_1 < 2) \lor (n_0 > 1 \land n_1 = 2)\} \\ r_{1_2 2_4} &: \{n_0 = 2, n_1 = 2\} \rightarrow \{(n_1 < 2) \lor (n_0 > 2 \land n_1 = 2)\} \end{aligned}$$

In this example, we reduced six expanded sketch rules to four. Having less extended rules leads to shorter *LTL* and *CTL* formulas, which can speed up the model checking. Additionally, we could simplify the effects of rules $r_{1_1 2_2}$ and $r_{1_2 2_4}$, since $n_0 > 2$ is never possible. This would result in the following expanded sketch rules:

$$\begin{aligned}
r_{1_1 2_1} :& \{n_0 = 1, n_1 = 1\} \rightarrow \{(n_1 < 1) \vee (n_0 > 1 \wedge n_1 = 1)\} \\
r'_{1_1 2_2} :& \{n_0 = 2, n_1 = 1\} \rightarrow \{(n_1 < 1)\} \\
r_{1_2 2_3} :& \{n_0 = 1, n_1 = 2\} \rightarrow \{(n_1 < 2) \vee (n_0 > 1 \wedge n_1 = 2)\} \\
r'_{1_2 2_4} :& \{n_0 = 2, n_1 = 2\} \rightarrow \{(n_1 < 2)\}
\end{aligned}$$

Another way to speed up the model-checking would be to find shorter logical formulas that express the constraints of a good sketch. Our formula for the third contains three times a disjunction over all expanded sketch rules. If we find a formula with fewer of these disjunctions, we can speed up the model-checking process. This also brings us to the next point.

Try out different logical formulas We believe there are different logical formulas that express the three constraints of a good sketch. We would love to find and experiment with them. We think it might be possible to find equivalent formulas that are shorter and, thus, faster to check.

Add complexity measure and use good sketches in planning algorithms Drexler et al. [10] used width as a complexity measurement for sketches. Future work is to also add a complexity measure to our method. A relevant complexity measure depends on the planning algorithm the sketches will be used in. An example could be the number of transitions that are allowed between a condition and its effect when following a rule. To verify a sketch given a maximum number of steps can be achieved by substituting the future (F) operators (resp. once (O)) in our CTL_f^* formulas by nested next (X) operators (resp. previous (Y)). This complexity measure suits a method like depth-first search, where one searches for the effect of applicable sketch rules instead of the goal.

For further testing, we would like to use our generated sketches in a planning algorithm to show that they indeed work.

Formal proofs An important future step is to provide formal proofs that establish the soundness and completeness of our sketch generation and verification method. Formal proofs could validate the reliability and correctness of our approach, strengthening its theoretical foundations.

8 Conclusion

We introduced a new, modular method to generate and verify sketches using automata-based model checking and implemented this method in Python. We made several contributions during this process. First, we provided a new definition of a good sketch, consisting of three constraints a sketch needs to adhere to. We provided a way to express sketch rules in logical languages. We translated the constraints for a good sketch into CTL_f^* formulas and showed that we can split them into separate CTL and LTL formulas, languages supported by more popular model checkers. Additionally, we provided a method to generate a pool of all possible sketches, which we used to construct sketches to verify lazily. To test our method, we tried to reconstruct the sketches learned by Drexler et al. [10], for which we were able to reconstruct the width-1 and width-2 sketches for seven out of eight domains. We did not find the Drexler sketches for one domain due to a difference in definition. While looking at our set of good sketches, we could identify patterns of equivalent sketches and sketch rules. We saw that the time to model-check a good sketch increases fast when the number of expanded sketch rules increases, which can be addressed in future work.

Our sketch generation and verification method allows us to find all possible good sketches for a domain and can be used as a baseline for new sketch generation methods.

We discussed many directions for future work, together with concrete steps to pursue them. First of all, we would like to add a complexity measure to our formulas such that we can use our sketches in planning algorithms. Next, it is an important future step to prove our method's soundness and completeness formally. Another direction of future work is reducing the pool of candidate sketches by exploiting the patterns of equivalent sketches and sketch rules we found. Alternatively, the modular nature of our approach allows for experimenting with more clever ways to generate sketches.

Lastly, future work is to find methods to speed up the model-checking by, for example, merging expanded sketch rules such that the logical formulas become shorter or finding other formulas that express the three constraints of a good sketch.

Acknowledgements

I want to thank my supervisors, Natacha Alechina and Brian Logan, for their guidance, support, insightful feedback, and enthusiasm regarding my topic throughout the entire journey of this thesis. I am also deeply thankful to Adam Vandervorst for his continuous encouragement and stimulating discussions while also giving me feedback on my Python code and hosting the experiments.

References

[1] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial intelligence*, 116(1-2):123–191, 2000.

[2] Blai Bonet, Guillem Francès, and Hector Geffner. Learning features and abstract actions for computing generalized plans. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):2703–2710, Jul. 2019.

[3] Blai Bonet and Hector Geffner. Features, projections, and representation change for generalized planning. *arXiv preprint arXiv:1801.10055*, 2018.

[4] Blai Bonet and Hector Geffner. General policies, representations, and planning width. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11764–11773, 2021.

[5] Simon Busard and Charles Pecheur. Pynusmv: Nusmv as a python library. volume 7871 of *LNCS*, pages 453–458. Springer-Verlag, 2013.

[6] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*, pages 359–364. Springer, 2002.

[7] Tomás de la Rosa and Sheila McIlraith. Learning domain control knowledge for tlplan and beyond. In *ICAPS 2011 Workshop on Planning and Learning*, pages 36–43. Citeseer, 2011.

[8] Dominik Drexler, Guillem Francès, and Jendrik Seipp. DLPlan, 2022.

[9] Dominik Drexler, Jendrik Seipp, and Hector Geffner. Expressing and exploiting the common subgoal structure of classical planning domains using sketches: Extended version. *arXiv preprint arXiv:2105.04250*, 2021.

[10] Dominik Drexler, Jendrik Seipp, and Hector Geffner. Learning sketches for decomposing planning problems into subproblems of bounded width: Extended version. *arXiv preprint arXiv:2203.14852*, 2022.

[11] Dominik Drexler, Jendrik Seipp, and Hector Geffner. Proofs, code, and data for the icaps 2022 paper, March 2022. Additional Acknowledgements: - This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. - The computations were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC), partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

[12] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time (preliminary report). In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, page 127–140, New York, NY, USA, 1983. Association for Computing Machinery.

[13] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[14] Guillem Francés, Miquel Ramirez, and Collaborators. Tarski: An AI planning modeling framework, 2018.

[15] Guillem Francès, Blai Bonet, and Hector Geffner. Learning general policies from small examples without supervision, 2021.

[16] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl—the planning domain definition language. 1998.

[17] Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[18] Chad Hogg, Héctor Munoz-Avila, and Ugur Kuter. Htn-maker: Learning htns with minimal additional knowledge engineering required. In *AAAI*, pages 950–956, 2008.

[19] Yuxiao Hu and Giuseppe De Giacomo. Generalized planning: Synthesizing plans that work for multiple environments. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 918, 2011.

[20] Wojciech Jamroga. *Logical methods for specification and verification of multi-agent systems*. Institute of Computer Science, Polish Academy of Sciences, 2015.

[21] Jonas Kvarnström and Patrick Doherty. Talplanner: A temporal logic based forward chaining planner. *Annals of mathematics and Artificial Intelligence*, 30(1):119–169, 2000.

[22] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. *The glory of the past*. Springer, 1985.

[23] Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In *ECAI 2012*, pages 540–545. IOS Press, 2012.

[24] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 968–973, 1999.

[25] Negin Nejati, Pat Langley, and Tolga Konik. Learning hierarchical task networks by observation. In *Proceedings of the 23rd international conference on Machine learning*, pages 665–672, 2006.

[26] Julie Porteous and Laura Sebastia. Extracting and ordering landmarks for planning. In *Proceedings UK Planning and Scheduling SIG Workshop*. Citeseer, 2000.

[27] Julie Porteous, Laura Sebastia, and Jörg Hoffmann. On the extraction, ordering, and usage of landmarks in planning. In *Sixth European Conference on Planning*, 2014.

[28] Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In *AAAI*, volume 8, pages 975–982, 2008.

[29] Simon Ståhlberg, Blai Bonet, and Hector Geffner. Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, pages 629–637, 2022.

[30] Simon Ståhlberg, Blai Bonet, and Hector Geffner. Learning generalized policies without supervision using gnns. *arXiv preprint arXiv:2205.06002*, 2022.

[31] Hankz Hankui Zhuo, Derek Hao Hu, Chad Hogg, Qiang Yang, and Hector Munoz-Avila. Learning htn method preconditions and action models from partial observations. In *Twenty-First International Joint Conference on Artificial Intelligence*. Citeseer, 2009.

[32] Hankz Hankui Zhuo, Héctor Munoz-Avila, and Qiang Yang. Learning hierarchical task network domains from partially observed plan traces. *Artificial intelligence*, 212:134–157, 2014.

A Example PDDL files

A.1 Blocksworld Domain

```
(define (domain blocksworld)
  (:requirements :strips)
(:types )
(:constants )
(:predicates (clear ?x)
             (on-table ?x)
             (arm-empty)
             (holding ?x)
             (on ?x ?y))

(:action pickup
  :parameters (?ob)
  :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
  :effect (and (holding ?ob) (not (clear ?ob))
               (not (on-table ?ob)) (not (arm-empty))))

(:action putdown
  :parameters  (?ob)
  :precondition (holding ?ob)
  :effect (and (clear ?ob) (arm-empty) (on-table ?ob)
               (not (holding ?ob))))

(:action stack
  :parameters  (?ob ?underob)
  :precondition (and (clear ?underob) (holding ?ob))
  :effect (and (arm-empty) (clear ?ob) (on ?ob ?underob)
               (not (clear ?underob)) (not (holding ?ob))))

(:action unstack
  :parameters  (?ob ?underob)
  :precondition (and (on ?ob ?underob) (clear ?ob) (arm-empty))
  :effect (and (holding ?ob) (clear ?underob)
               (not (on ?ob ?underob)) (not (clear ?ob))
               (not (arm-empty)))))
```

Figure 6: *PDDL description of the blocksworld domain*

A.2 Blocksworld Instance

```
(define (problem BW-rand-3)
(:domain blocksworld)
(:objects b1 b2 b3 )
(:init
(arm-empty)
(on b1 b3)
(on b2 b1)
(on-table b3)
(clear b2)
)
(:goal
(and
(on b1 b2))
)
)
```

Figure 7: *PDDL description of an instance of the blocksworld*

B Sketches

B.1 Blocksworld Clear

$\{\} \rightarrow \{\neg b_0\}$	b_0 := b_empty(c_and(c_primitive(clear,0),c_primitive(clear_g,0)))
$\{b_0\} \rightarrow \{\neg b_0\}$	b_0 := b_empty(c_and(c_primitive(clear,0),c_primitive(clear_g,0)))
$\{\} \rightarrow \{n_0 \uparrow\}$	n_0 := n_count(c_and(c_primitive(clear,0),c_primitive(clear_g,0)))
$\{n_0 = 0\} \rightarrow \{n_0 \uparrow\}$	n_0 := n_count(c_and(c_primitive(clear,0),c_primitive(clear_g,0)))
$\{\} \rightarrow \{\neg b_0\}$	b_0 := b_empty(c_and(c_primitive(clear_g,0),c_primitive(holding,0)))
$\{\} \rightarrow \{n_0 \downarrow\}$	n_0 := n_count(c_not(c_primitive(clear,0)))
$\{n_0 > 0\} \rightarrow \{n_0 \downarrow\}$	n_0 := n_count(c_not(c_primitive(clear,0)))
$\{\} \rightarrow \{n_0 \uparrow\}$	n_0 := n_count(c_primitive(clear,0))
$\{n_0 > 0\} \rightarrow \{n_0 \uparrow\}$	n_0 := n_count(c_primitive(clear,0))
$\{\} \rightarrow \{n_0 \downarrow\}$	n_0 := n_count(c_all(r_primitive(on,0,1),c_primitive(clear_g,0)))
$\{n_0 > 0\} \rightarrow \{n_0 \downarrow\}$	n_0 := n_count(c_all(r_primitive(on,0,1),c_primitive(clear_g,0)))
$\{\} \rightarrow \{b_0\}$	b_0 := b_empty(c_not(c_primitive(clear,0)))
$\{\neg b_0\} \rightarrow \{b_0\}$	b_0 := b_empty(c_not(c_primitive(clear,0)))
$\{\} \rightarrow \{b_0\}$	b_0 := b_empty(r_primitive(on,0,1))
$\{\} \rightarrow \{b_0\}$	b_0 := b_empty(c_some(r_primitive(on,0,1),c_primitive(clear_g,0)))
$\{\} \rightarrow \{n_0 \uparrow\}$	n_0 := n_count(c_and(c_primitive(clear,0),c_primitive(on-table,0)))

B.2 Blocksworld On

$\{\} \rightarrow \{n_0 \uparrow\}$	$n_0 :=$ n_count(c_equal(r_primitive(on,0,1),r_primitive(on_g,0,1)))

B.3 Childsnack

$\{\} \rightarrow \{b_0\}$	$b_0 :=$ b_empty(c_and(c_not(c_primitive(served,0)),c_primitive(served_g,0)))
$\{\neg b_0\} \rightarrow \{b_0\}$	$b_0 :=$ b_empty(c_and(c_not(c_primitive(served,0)),c_primitive(served_g,0)))

B.4 Delivery

$\{\} \rightarrow \{n_0 \uparrow\}$	$n_0 :=$ n_count(c_some(r_primitive(at,0,1),c_projection(r_primitive(at_g,0,1),1)))
$\{\} \rightarrow \{n_0 \uparrow\}$	$n_0 :=$ n_count(c_equal(r_primitive(at,0,1),r_primitive(at_g,0,1)))
$\{n_0 > 0\} \rightarrow \{n_0 \uparrow\}$	$n_0 :=$ n_count(c_equal(r_primitive(at,0,1),r_primitive(at_g,0,1)))
$\{\} \rightarrow \{n_0 \downarrow\}$	$n_0 :=$ n_count(c_not(c_equal(r_primitive(at,0,1),r_primitive(at_g,0,1))))
$\{n_0 > 0\} \rightarrow \{n_0 \downarrow\}$	$n_0 :=$ n_count(c_not(c_equal(r_primitive(at,0,1),r_primitive(at_g,0,1))))

B.5 Gripper

$\{\} \rightarrow \{n_0 \downarrow\}$	$n_0 :=$ n_count(c_all(r_primitive(at,0,1),c_one_of(rooma)))
$\{n_0 > 0\} \rightarrow \{n_0 \downarrow\}$	$n_0 :=$ n_count(c_all(r_primitive(at,0,1),c_one_of(rooma)))
$\{\} \rightarrow \{n_0 \uparrow\}$	$n_0 :=$ n_count(c_some(r_primitive(at,0,1),c_one_of(roomb)))
$\{\} \rightarrow \{n_0 \uparrow\}$	$n_0 :=$ n_count(c_equal(r_primitive(at,0,1),r_primitive(at_g,0,1)))
$\{n_0 > 0\} \rightarrow \{n_0 \uparrow\}$	$n_0 :=$ n_count(c_equal(r_primitive(at,0,1),r_primitive(at_g,0,1)))

B.6 Miconic

$\{\} \rightarrow \{n_0 \uparrow\}$	$n_0 :=$ n_count(c_primitive(served,0))

B.7 Reward

$\{\} \rightarrow \{b_0\}$	$b_0 :=$ b_empty(c_primitive(reward,0))
$\{\neg b_0\} \rightarrow \{b_0\}$	$b_0 :=$ b_empty(c_primitive(reward,0))
$\{\} \rightarrow \{n_0 \uparrow\}$	$n_0 :=$ n_count(c_primitive(picked,0))
$\{\} \rightarrow \{n_0 \downarrow\}$	$n_0 :=$ n_count(c_primitive(reward,0))
$\{n_0 > 0\} \rightarrow \{n_0 \downarrow\}$	$n_0 :=$ n_count(c_primitive(reward,0))

B.8 Spanner

$\{\} \rightarrow \{n_0 \uparrow\}$	$n_0 :=$ n_count(c_not(c_some(r_inverse(r_primitive(at,0,1)),c_primitive(loose,0))))
$\{n_0 > 0\} \rightarrow \{n_0 \uparrow\}$	$n_0 :=$ n_count(c_not(c_some(r_inverse(r_primitive(at,0,1)),c_primitive(loose,0))))
$\{\} \rightarrow \{n_0 \downarrow\}$	$n_0 :=$ n_count(c_some(r_inverse(r_primitive(at,0,1)),c_primitive(loose,0)))
$\{n_0 > 0\} \rightarrow \{n_0 \downarrow\}$	$n_0 :=$ n_count(c_some(r_inverse(r_primitive(at,0,1)),c_primitive(loose,0)))
$\{\} \rightarrow \{b_0\}$	$b_0 :=$ b_empty(c_primitive(loose,0))
$\{\neg b_0\} \rightarrow \{b_0\}$	$b_0 :=$ b_empty(c_primitive(loose,0))

B.9 Visit All

$\{\} \rightarrow \{n_0 \uparrow\}$	$n_0 :=$ n_count(c_primitive(visited,0))
$\{n_0 > 0\} \rightarrow \{n_0 \uparrow\}$	$n_0 :=$ n_count(c_primitive(visited,0))

C Implementation

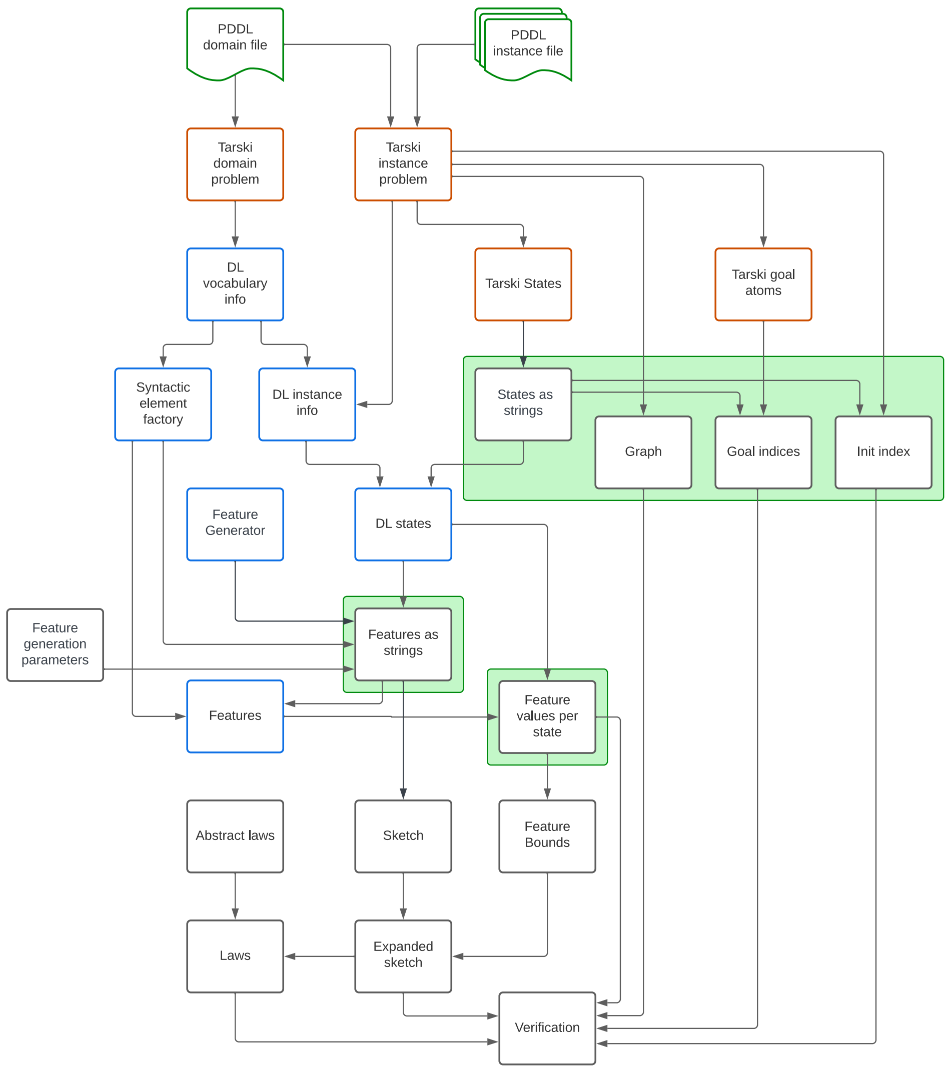

Figure 8: *Flowchart of the dependencies in our implementation. The orange outlined boxes represent objects from the Tarski library. The blue outlined boxes represent objects from the DLPlan library. The green boxes group the objects we cached together in files.*