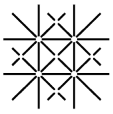

University of Basel

Schematic Invariant Synthesis Algorithm with Limited Grounding

Bachelor's Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Tanja Schindler

Mario Tachikawa
mario.tachikawa@unibas.ch
20-062-501

31.05.2024

Acknowledgments

First of all, I want to thank Prof. Dr. Malte Helmert for giving me the opportunity to write my Bachelor's thesis in the Artificial Intelligence Research Group and for offering me an interesting selection of thesis topics.
I am exceptionally thankful to my supervisor Tanja Schindler for supporting me through the whole journey of this work, for helping me understand the different topics, and for guiding me through the implementation of the algorithm. I also want to thank Florian Pommerening for the support of setting up everything in order for me to use the sciCORE cluster for the evaluation.

Abstract

The schematic invariant synthesis algorithm using limited grounding propsed by Rintanen was implemented and integrated into the classical planning system Fast Downward by replacing the existing invariant synthesis algorithm by Helmert. The invariant synthesis identifies find mutex groups which are used for the translation of the PDDL task into a finite domain representation task.
A comparison between the algorithms was evaluated by running experiments with the planning system on various propositional STRIPS benchmark tasks. The goal was to implement the schematic invariant synthesis algorithm that correctly finds ground invariants and leading the planning task to find a valid plan, where the efficiency was not of high priority.
The evaluation results show that using the implemented invariant synthesis correct ground invariants are found and that the search duration of the planner stays similar compared to the search with the already existing invariant synthesis. In comparison to the current invariant synthesis, the newly implemented schematic invariant synthesis algorithm is very inefficient and can therefore not compete with the current invariant synthesis by Helmert.

Contents

1 Introduction

Most humans are able to systematically plan and execute tasks, as long as these tasks are simple and do not have a large variety of options to achieve the goals. However, as soon as the tasks become more complex, where many different objects, locations or other problem-related attributes come into play, classical planning systems can be used to find plans to solve the tasks efficiently.
The goal of a classical planning system is to find a sequence of actions, also called a plan, that will find a path to a goal state from a given initial state. Such systems can make several preparations to find this sequence of actions more efficiently. One of these possible preparations is the translation of the input task into a more compact representation, called the finite domain representation, which helps the planning system to find the plan more efficiently. Such a translation can simplify a task, by combining propositions from the input task into a group, that can be translated into a single finite-domain variable. This is possible when only at most one proposition in the group is true in any reachable state from the initial state. In other words, different propositions in a group can never be true at the same time. Such groups of propositions, that can be represented by a single finite-domain variable are groups of mutually exclusive propositions called mutex groups. Invariants are formulas over propositional variables, that are true in every reachable state and can express such mutex groups [1].
Finding invariants for a planning task can be a challenging process, leading to the development of many different approaches for invariant synthesis[1]. Such algorithms can be divided into those that work with schematic formulas and those, that instantiate/ground the formulas.
Jussi Rintanen proposed in the paper "Schematic Invariants by Reduction to Ground Invariants"[2] an invariant synthesis algorithm with a novel idea of a hybrid algorithm, that uses both the grounded and ungrounded formulas, where the schematic formulas are only grounded with a small number of objects, defined by a limited grounding function. The number of objects should be as small as possible, but not too small, for the algorithm to still identify correct invariants. This invariant synthesis algorithm uses a different approach to other earlier works, which is the reason why implementing this algorithm in an existing planning system and comparing its performance is an interesting task.
In this thesis, the implementation of an algorithm for invariant synthesis based on Rintanen's algorithm [2] is documented. The algorithm is implemented in the domain-independent classical planning system "Fast Downward" [3]. This planning system is suitable for the implementation of Rintanen's algorithm, because the planner already uses an invariant synthesis algorithm during the translation process to identify mutex groups. This means that the algorithm can be embedded into an already efficiently working planner and therefore its performance can be evaluated.
In addition to the replacement of the existing invariant synthesis by Helmert, a maximal clique enumeration algorithm proposed by Tomita [4] is implemented to generate mutex groups from all invariants found by the algorithm. For the evaluation, the new algorithm is compared to the already existing algorithm by considering the performance and the quality of the synthesized invariants, specifically through the structure of the generated mutex groups.

2 Background

In the background section of this thesis all relevant definitions are presented, to have the theoretical framework for understanding all concepts described in later chapters. To have a clearer understanding of these often abstract definitions, a simple transport problem, that is similar to the the transport-opt08-strips problem contained in the Downward benchmarks[5]. will be used as an illustrative example throughout all definitions. The used transport problem is about delivering packages from an initial location to a goal location using vehicles, that can only move from location A to location B, if these two locations are connected by a road.

2.1 Planning Definitions

Definition 1 *Types and Objects [2]*

Let T be a finite set of types and O a set of objects.
A domain function $D : T \mapsto \mathcal{P}(O) \setminus \{\}$ defines for every type $t \in T$ the corresponding non-empty set of objects $D(t) \subseteq O$ that are of that type.

Since some types can be subtypes of other types, the sets of objects of different types do not need to be disjoint, but if $D(t) \cap D(t') \neq \emptyset$, either $D(t) \subseteq D(t')$ or $D(t') \subseteq D(t)$. In such a case, t is either a subtype of t' or t' a subtype of t.

Example:
In the transport problem, we have the following types, objects, and the domain function:

$$\begin{aligned} O = \{&\text{package-1, package-2, truck-1, truck-2,} \\ &\text{location-1, location-2, location-3,} \\ &\text{capacity-0, capacity-1, capacity-2}\} \\ T = \{&\text{locatable, vehicle, package, location, capacity-number}\} \\ D(\text{package}) = \{&\text{package-1, package-2}\} \\ D(\text{vehicle}) = \{&\text{truck-1, truck-2}\} \\ D(\text{locatable}) = \{&\text{package-1, package-2, truck-1, truck-2}\} \\ D(\text{location}) = \{&\text{location-1, location-2, location-3}\} \\ D(\text{capacity-number}) = \{&\text{capacity-0, capacity-1, capacity-2}\} \end{aligned}$$

As we can see here $D(\text{package}) \subseteq D(\text{locatable})$ and $D(\text{vehicle}) \subseteq D(\text{locatable})$. This is because package and vehicle are both subtypes of locatable.

Definition 2 *Schematic Variables[2]*

Let T be a set of types and V a set of schematic variables. Each schematic variable $v \in V$ has a type defined by $t_{var}(v) \in T$ and is therefore a placeholder for objects that are in $D(t_{var}(v)) \subseteq O$.

Schematic variables can have any name that is not contained in O, for example, x, y, z with the following types:

$$t_{var}(x) = \text{location}$$
$$t_{var}(y) = \text{truck}$$
$$t_{var}(z) = \text{package}$$

Definition 3 *Predicates[2]*

Let T be a set of types, P a set of predicate symbols. Each predicate $p \in P$ has an arity $ar(p) \in \mathbb{N}$ and an associated type, defined by the typing function $\tau_{pre}(p) \in T^{ar(p)}$.

For the transport problem, we have the following predicates:

$$P = \{\text{at, in, road, capacity, capacity-predecessor}\}$$

$$\begin{aligned}
\text{at} &: ar(\text{at}) = 2, \tau_{pre}(\text{at}) = \langle \text{locatable, location} \rangle \\
\text{in} &: ar(\text{in}) = 2, \tau_{pre}(\text{in}) = \langle \text{package, vehicle} \rangle \\
\text{road} &: ar(\text{road}) = 2, \tau_{pre}(\text{road}) = \langle \text{location, location} \rangle \\
\text{capacity} &: ar(\text{capacity}) = 2, \\
&\quad \tau_{pre}(\text{capacity}) = \langle \text{vehicle, capacity-number} \rangle \\
\text{capacity-predecessor} &: ar(\text{capacity-predecessor}) = 2, \\
&\quad \tau_{pre}(\text{capacity-predecessor}) = \\
&\quad \langle \text{capacity-number, capacity-number} \rangle
\end{aligned}$$

Definition 4 *Literals and Atoms[2]*

Let T be a set of types, P a set of predicate symbols, O a set of objects, V a set of schematic variables, and D a domain function. Let $p \in P$ be a predicate of arity $ar(p) = n$ and of type $\tau_{pre}(p) = \langle t_1, t_2, .., t_n \rangle$. An atom is of the form: $p(s_1, s_2, .., s_n)$, where s_i is either an object $o_i \in D(t_i) \subseteq O$ or a schematic variable $v_i \in V$ with $t_{var}(v_i) = t_i$.

Atoms can be divided into schematic atoms and ground atoms. For schematic atoms, all s_i are schematic variables $v_i \in V$ with $t_{var}(v_i) = t_i$ and for ground atoms all s_i are objects $o_i \in D(t_i)$.
Let a be an atom, then a and $\neg a$ are literals, where a is a positive literal and $\neg a$ is a negative literal.

The set $gnf(P, \tau_{pre}, D)$ consists of all possible ground atoms $p(o_1, o_2, .., o_n)$ such that $p \in P$, $n = ar(p)$, $\tau_{pre}(p) = \langle t_1, t_2, .., t_n \rangle$ and $o_i = D(t_i)$.
The set $sa(P, \tau_{pre}, V, t_{var})$ consist of all possible schematic atoms $p(v_1, v_2, .., v_n)$ such that $p \in P$, $n = ar(p)$, $v_i \in V$, $t_{var}(v_i) = t_i$ and $\tau_{pre} = \langle t_1, t_2, \ldots, t_n \rangle$

Schematic Literal Examples:

- $\neg at(x, y)$
- $road(l_1, l_2)$
- $in(p, v)$

These schematic literals have schematic variables with the following types:

- $t_{var}(x) =$ locatable
- $t_{var}(y), t_{var}(l_1), t_{var}(l_2) =$ location
- $t_{var}(p) =$ package
- $t_{var}(v) =$ vehicle

Ground Literal Examples:

- $\neg at(\text{truck-1}, \text{location-1})$
- $road(\text{location-1}, \text{location-2})$
- $in(\text{package-2}, \text{truck-2})$

These ground literals have objects with the following types:

- $D(\text{truck-1}) = D(\text{truck-2}) =$ vehicle
- $D(\text{location-1}) = D(\text{location-2}) =$ location
- $D(\text{package-1}) = D(\text{package-2}) =$ package

Definition 5 *State[2]*

Let P be a set of predicates p with type $\tau_{pre}(p)$ for $p \in P$, D a domain function. A state s is defined by the set of all positive ground literals over O and V, that are true in state s:
$s = \{a_1, a_2, \ldots, a_n\}$, *with* $a_i \in gnf(P, \tau_{pre}, D)$

An example of a state s in the transport example, where both packages are in truck-1 would be:

$$\begin{aligned} s = \{&in(\text{package-1}, \text{truck-1}),\\ &in(\text{package-2}, \text{truck-1}),\\ &at(\text{truck-2}, \text{location-1}),\\ &at(\text{truck-1}, \text{location-3}),\\ &road(\text{location-1}, \text{location-2}),\\ &road(\text{location-2}, \text{location-3}),\\ &capacity(\text{truck-1}, \text{capacity-0}),\\ &capacity(\text{truck-2}, \text{capacity-2}),\\ &capacity\text{-}predecessor(\text{capacity-0}, \text{capacity-1}),\\ &capacity\text{-}predecessor(\text{capacity-1}, \text{capacity-2})\} \end{aligned}$$

Definition 6 *Formula over O and V*[2]

Let V be a set of schematic variables, O be a set of objects, and P a set of predicates with arities $ar(p) = n$ for every $p \in P$.
The following are formulas over O and V:

1. *Atoms $p(s_1, s_2, \ldots, s_n)$ over V and O.*
2. *$\phi_1 \wedge \phi_2$, if ϕ_1 and ϕ_2 are formulas.*
3. *$\phi_1 \vee \phi_2$, if ϕ_1 and ϕ_2 are formulas.*
4. *$\neg\phi$, if ϕ is a formula.*

Note that similarly to atoms, formulas can also be divided into schematic formulas and ground formulas. A schematic formula consists only of atoms that are schematic atoms and a ground formula consists only of atoms that are ground atoms. The schematic formula is used to describe a set of ground formulas more compactly since the schematic variables are placeholders for potentially more than one object.

Definition 7 *Effects*[**2**]

Let $p(s_1, s_2, \ldots, s_n)$ be an atom, then $p(s_1, s_2, \ldots, s_n)$ and $\neg p(s_1, s_2, \ldots, s_n)$ are effects.

Similarly to atoms, effects can also be divided into schematic effects and ground effects. A schematic effect has only literals that are schematic literals and a ground effect has only literals that are ground literals.

Definition 8 *Action*

Let O be a set of objects, T be a finite set of types, V be a set of schematic variables and A be a finite set of actions. An action $a \in A$ over O and V is defined by a pair $\langle C, E\rangle$:

- *C is a set of atoms over O and V. This set describes a conjunction over all atoms in C forming a formula, called the precondition.*
- *E is a set of effects over O and V. This set describes a conjunction over all effects in E forming a formula.*

The effects can be divided into the add effects $add(a)$ and delete effects $del(a)$, which are defined as follows:

- *$add(a) = \{e_i | e_i \in E$ and e_i is a positive literal$\}$*
- *$del(a) = \{e_i | e_i \in E$ and e_i is a negative literal$\}$*

The precondition of an action can also be denoted as $pre(a) = C$.

Similarly to atoms, actions can be divided into schematic and ground actions.
A schematic action is defined by a pair $\langle C, E\rangle$ where C is a set of schematic atoms over V and E is a set of schematic effects over V. A ground action is also defined by a pair $\langle C, E\rangle$, where C is a set of ground atoms over O and E is a set of ground effects over O.

In this thesis, an action is often denoted as $a(v_1, v_2, ..., v_n)$, where a is the name of the action and v_i are all schematic variables occurring as schematic variables in the schematic atoms and schematic effects in C and E, considering that the conditions C and effects E are previously defined.

As an illustrative example, we use the drive action, which defines an action, where a vehicle drives from one location to another location.

Schematic action example:

- $A_{drive} = \langle \{at(v, l_1), road(l_1, l_2)\}, \{at(v, l_2), \neg at(v, l_1)\} \rangle$
- simple form: $drive(v, l_1, l_2)$
- $pre(drive) = \{at(v, l_1), road(l_1, l_2)\}$
- $add(drive) = \{at(v, l_2)\}$
- $del(drive) = \{at(v, l_1)\}$

Ground action example:

- $drive = \langle \{at(\text{truck-1}, \text{location-1}), road(\text{location-1}, \text{location-2})\},$
  $\{at(\text{truck-1}, \text{location-2}), \neg at(\text{truck-1}, \text{location-1})\} \rangle$
- simple form: $drive(\text{truck-1}, \text{location-1}, \text{location-2})$
- $pre(drive) = \{at(\text{truck-1}, \text{location-1}), road(\text{location-1}, \text{location-2})\}$
- $add(drive) = \{at(\text{truck-1}, \text{location-2})\}$
- $del(drive) = \{at(\text{truck-1}, \text{location-1})\}$

Definition 9 *Action application*

Let $a \in A$ be an action and s and s' be states.
An action a can be applied in a state s if $pre(a) \subseteq s$ holds. When the precondition of a is met in s, then state s' is reached after the application of action a. State s' is then: $s' = (s \setminus del(a)) \cup add(a)$.

The application of a in state s resulting in state s' can be denoted as $s \xrightarrow{a} s'$.

Definition 10 *Schematic Planning Problem Instance* **[2]**

A schematic planning problem instance is defined by a 6-tuple
$\Pi = \langle O, T, D, P, \tau_{pre}, A, I \rangle$ *with:*

- *O is a set of objects*
- *T is a finite set of types*
- *$D : T \mapsto \mathcal{P}(O) \setminus \{\}$ is a domain function*
- *P is a set of predicates*
- *τ_{pre} is a typing function*
- *A is a set of schematic actions over O and V*
- *I is the initial state described by a set of ground atoms, that are true in the initial state*

For the invariant synthesis, we are not interested in finding any sequence of actions. Because of this reason, no goal formula is defined here.
A schematic problem instance can be instantiated to a ground problem instance by instantiating all schematic actions in A to ground actions Note that the terms instantiating and grounding are used interchangeably to refer to the same concept.

For this thesis, we are only considering schematic planning problem instances that are restricted to STRIPS [6]. A schematic planning problem instance that is restricted to STRIPS adheres to the following constraints:

- The preconditions for all actions $a \in A$ are conjunctions of literals.
- The effects for all actions $a \in A$ are conjunctions of literals.

2.2 Invariant Synthesis related Definitions

Definition 11 *Reachable States*

Let I be the initial state, and A be a set of ground actions. Reachable states are defined as all states s_r, if there exists an action sequence $\langle a_1, a_2, ..., a_n \rangle$, with $a_i \in A$, such that $I \xrightarrow{a_1} s_1 \xrightarrow{a_2} ... \xrightarrow{a_n} s_r$.

Reachable states are all states that can be reached from the initial state through a sequence of actions.

Definition 12 *Invariant*[**1**]

Let $\Pi = \langle O, T, D, P, \tau_{pre}, A, I \rangle$ be a planning problem instance and S_r be the corresponding set of all reachable states from the initial state I. An invariant of Π is a formula ϕ over O and V, that is satisfied by all reachable states $s_r \in S_r$: $s_r \models \phi$.

Invariants can be divided into schematic invariants and ground invariants.
Schematic invariants are formulas only over V and ground invariants are formulas only over O.

A schematic invariant formula ϕ holds in a state $s_r \in S_r$, if $s_r \models \phi'$ for all possible instances ϕ' from ϕ, by replacing all schematic variables v_i by objects $o_i \in D(t_i), t_{var}(v_i) = t_i$.

Definition 13 *Invariant Candidates*[**2**]

Let χ be a (possibly empty) conjunction of inequalities $x \neq x'$, where x and x' are schematic variables and l_i are negative schematic literals. The following two expressions can express schematic invariant candidates:

$$\chi \implies (l_1 \vee l_2),$$
$$\chi \implies l_1$$

These implications are equivalent to the following two expressions:

$$\neg\chi \vee (l_1 \vee l_2),$$
$$\neg\chi \vee l_1$$

For this work, we are only interested in invariants that consist of a single negative atom or a disjunction of 2 negative atoms.

When such a schematic invariant candidate is instantiated the schematic variables are replaced by specific objects that satisfy the constraints given by the conjunction of inequalities χ, which means that the conjunction of inequalities χ is not needed anymore, because no more variables exist. Given this, ground invariant candidates are of the form:

$$(l_1 \vee l_2),$$
$$l_1$$

where l_i are negative ground literals.

As an example of the transport problem, the schematic invariant that states that a package is never contained in two different vehicles, where $p, v_1, v2$ are schematic variables with $t_{var}(p) = \text{package}$, $t_{var}(v_1) = t_{var}(v_2) = \text{vehicle}$:

$$(v_1 \neq v_2) \implies (\neg in(p, v_1) \vee \neg in(p, v_2))$$

Instantiating this schematic invariant results in these two ground invariants:

$$\neg in(\text{package-1}, \text{truck-1}) \vee \neg in(\text{package-1}, \text{truck-2}),$$
$$\neg in(\text{package-2}, \text{truck-1}) \vee \neg in(\text{package-2}, \text{truck-2})$$

Definition 14 *Substitutions*[**2**]

Let V be a set of schematic variables and O be a set of objects. A function $\sigma : V \mapsto V \cup O$ is a substitution.

Definition 15 *Application of Substitutions*[**2**]

Substitutions can be applied to a formula ϕ and an action a. A substitution defined by $\phi\sigma$ and $a\sigma$ is obtained from ϕ and a by replacing every occurrence of $v \in V$ by $\sigma(v)$.

Definition 15 *Regression*[**7**]

Let ϕ be a ground formula, representing a conjunction of ground literals and A be a set of actions. The regression of a ground formula ϕ with respect to an action $a \in A$ is defined by the following equation, where ϕ_i is a ground literal of ϕ and $e_i \in add(a)$:

$$regr_a(\phi) = pre(a) \wedge \bigwedge(\{\phi_1, \phi_2, ...\phi_n\} \setminus \{e_1, e_2, ..., e_k\}),$$

Definition 16 *Limited Instantiation*[**2**]

Let $\Pi = \langle O, T, D, P, \tau_{pre}, A, I\rangle$ be a planning problem instance, $prms_t(a)$ be the number of schematic variables of type $t \in T$ in action $a \in A$, $prms_t(p)$ be the number of schematic variables of type $t \in T$ in a schematic formula with predicate $p \in P$, For this invariant synthesis, a limited instantiation is used. This means that only a limited amount of objects is used when instantiating schematic formulas and schematic actions. To still find correct schematic invariants, the lower bound of objects that can replace schematic variables is defined by the following number:

$$L_t^N(A, P) = max(max_{a \in A} prms_t(a), max_{p \in P} prms_t(p)) + (N - 1) * (max_{p \in P} prms_t(p))$$

where N is the maximal number of literals in a disjunction for an invariant, which is 2 in our case, as mentioned in Definition 12.
Using that lower bound number, a new domain function can be generated, that defines how limited instantiation can be used.

Definition 17 *Limited Grounding Function*[**2**]

Let $\Pi = \langle O, T, D, P, \tau_{pre}, A, I \rangle$ *be a planning problem instance and* $L_t^N(A, P)$ *be the lower bound number.*
A limited grounding function D' *is a domain function, that needs to satisfy the following constraints for every* $t \in T$*:*
Either $D'(t) = D(t)$ *or*

1. $D'(t) \subset D(t)$,
2. $|D'(t)| \geq L_t^N(A, P)$,
3. $D'(t_0) \subset D'(t_1)$ *iff* $D(t_0) \subset D(t_1)$,
   for all $\{\{t_0, t_1\} |\ t_0, t_1 \in T\}$

Let C be a set of schematic formulas, C_D the ground instances of C using the domain function D and $C_{D'}$ the ground instances of C using the limited grounding function D' with the characteristics described above. Let ϕ be a schematic formula, that is a disjunction with at most N literals.
We assume, that $C_D \cup \{regr_{a\sigma}(\phi\sigma)\}$ is satisfiable, where $a \in A$, $a\sigma$ is a ground action and $\phi\sigma$ is a ground instance of ϕ, with respect to D and some substitution $\sigma : V \mapsto O$ and $a\sigma$ is relevant for $\phi\sigma$, meaning that $a\sigma$ shares some literals in the precondition or in the effects with $\phi\sigma$. Note that $C_D \cup \{regr_{a\sigma}(\phi\sigma)\}$ is a set of literals forming a conjunction.
Then $C_{D'} \cup \{regr_{a\sigma'}(\phi\sigma')\}$ is satisfiable for some σ' with range of σ' included in D'.

Definition 18 *Mutex Invariant*[**1**]

A mutual exclusion (mutex) invariant is a special kind of invariant. A mutex invariant states that certain literals can never be true at the same time.
$\Pi = \langle O, T, D, P, \tau_{pre}, A, I \rangle$ be a planning problem instance and
$p_1(x_1, y_1), p_2(x_2, y_2)$ be literals where $p_1, p_2 \in P$, $x_1, x_2, y_1, y_2 \in V$ and let χ is a (possibly empty) conjunction of inequalities $x \neq x'$, where x and x' are schematic variables.
For this work, only mutex invariants are relevant, that have one of the following forms:

$$\chi \implies \neg p_1(x_1, y_1)$$
$$\chi \implies \neg p_1(x_1, y_1) \vee \neg p_2(x_2, y_2)$$

When this example mutex invariant is an invariant of a planning task, then $p_1(x_1, y_1)$ and $p_2(x_2, y_2)$ can never be true at the same time if the two literals satisfy the inequality constraint. The above example is in schematic form, but in this work, both the schematic and ground mutex invariants are important. As described in Definition 13, for ground invariants the inequality constraint falls away. Ground mutex invariants therefore are of the following forms, with $o_1, o_2, o_3, o_4 \in O$.:

$$\neg p_1(o_1, o_2)$$
$$\neg p_1(o_1, o_2) \vee \neg p_2(o_3, o_4)$$

Definition 19 *Mutex Group*[**1**]

A mutex group is a set of literals, where at most one literal can be true in all reachable states from the initial state.
For this work only mutex groups with positive atoms are relevant. Mutex groups can be generated from a set of mutex invariants as stated in Definition 18. These mutex invariants can form a group when each mutex invariant shares at least one literal with every other mutex invariant of the group.

Example:

We have the following two ground mutex invariants:

$$\neg in(\text{package-1}, \text{truck-1}) \vee \neg in(\text{package-1}, \text{truck-2}),$$
$$\neg in(\text{package-1}, \text{truck-1}) \vee \neg at(\text{package-1}, \text{location-1})$$

Note that if $in(\text{package-1}, \text{truck-1})$ and $at(\text{vehicle-1}, \text{location-1})$ holds, then $at(\text{package-1}, \text{location-1})$ does not hold. Both mutex invariants share the literal $\neg in(\text{package-1}, \text{truck-1})$, which means that these two mutex invariants can form a mutex group:

$$\begin{aligned}\text{mutex group} = [&in(\text{package-1}, \text{truck-1}),\\ &in(\text{package-1}, \text{truck-2}),\\ &at(\text{package-1}, \text{location-1})]\end{aligned}$$

This mutex group states that the object "package-1" can only be in truck-1, truck-2 or at location-1 or in none of these locations, but never in two of these places simultaneously.

2.3 PDDL

The Planning Domain Definition Language (PDDL)[8] is a formal knowledge representation language designed to express planning models. This language has become a de-facto standard input language of many planning systems, including the planning system Fast Downward. A planning problem defined with PDDL is divided into two parts: the domain and the problem .
The domain part of a planning problem defines all relevant aspects of the "world" in which we are planning, that are the same for all problem instances of that planning problem, like the types, predicates and actions.
The problem part of a planning problem is a single instance of that planning problem, that defines the initial state, goal conditions and the objects. For one domain, there can be different problem instances, that take place in the same "world", but differ in various aspects like the size of the problem or the complexity[9].

The domain and problem definitions are usually placed in two separate PDDL files (.pddl). As an example of such a PDDL definition, some parts of a domain and a problem file for the transport problem used in the definitions are provided.

Domain Example [5]:

The following example is a part of the domain file including all types, predicates and one action. There are more actions to the original problem, but since all actions have the same structure, only one action is showcased.

```
(define (domain transport)
  (:requirements :typing :action-costs)
  (:types
        location target locatable - object
        vehicle package - locatable
        capacity-number - object
  )

  (:predicates
     (road ?l1 ?l2 - location)
     (at ?x - locatable ?v - location)
     (in ?x - package ?v - vehicle)
```

```
    (capacity ?v - vehicle ?s1 - capacity-number)
    (capacity-predecessor ?s1 ?s2 - capacity-number)
  )

  (:action drive
    :parameters (?v - vehicle ?l1 ?l2 - location)
    :precondition (and
        (at ?v ?l1)
        (road ?l1 ?l2)
    )
    :effect (and
        (not (at ?v ?l1))
        (at ?v ?l2)
        (increase (total-cost) (road-length ?l1 ?l2))
    )
  )
```

Problem Example:

The following example is a part of the problem file including the objects, the initial state, and the goal conditions.

```
(define (problem transport)
  (:domain transport)
  (:objects
    city-loc-1 - location
    city-loc-2 - location
    city-loc-3 - location
    truck-1 - vehicle
    truck-2 - vehicle
    package-1 - package
    package-2 - package
    capacity-0 - capacity-number
    capacity-1 - capacity-number
    capacity-2 - capacity-number
  )
  (:init
    (= (total-cost) 0)
    (capacity-predecessor capacity-0 capacity-1)
    (capacity-predecessor capacity-1 capacity-2)
    (at package-1 city-loc-3)
    (at package-2 city-loc-3)
    (at truck-1 city-loc-3)
    (capacity truck-1 capacity-4)
    (at truck-2 city-loc-1)
    (capacity truck-2 capacity-3)
  )
  (:goal (and
    (at package-1 city-loc-2)
    (at package-2 city-loc-2)
  ))
)
```

3 Schematic Invariant Synthesis

This chapter is dedicated to the invariant synthesis algorithm that was proposed by Jussi Rintanen in the paper "Schematic Invariants by Reduction to Ground Invariants" [2]. Many explanation rely on the paper of Rintanen [2]. More specifically we are looking at the schematic invariant synthesis algorithm, which grounds the initial schematic invariants candidates with the limited grounding function and in the end extracts the schematic invariants from the found ground invariants. Using the definitions described in the background chapter, the goal is to outline and explain the different components of the algorithm and why the algorithm finds correct invariants.

3.1 Motivation

Earlier invariant synthesis algorithms often use the ground representation of actions and formulas and do not work with the ungrounded, schematic representation. But when the grounded actions are induced by a small set of schematic actions, invariants can be represented in a compact form as a small number of schematic invariants. This led to the research of schematic invariant synthesis algorithms, that directly work with the schematic representation of actions and formulas, which find invariants in schematic form. The problem is that such algorithms are often complex and have a significantly worse performance than algorithms that use the ground representation, especially for problems, where the number of schematic actions is high and the number of ground instances is low. Jussi Rintanen therefore devised a hybrid algorithm that makes use of the benefits of both approaches[2].

3.2 Idea

The Idea of this algorithm is to use both, the schematic and ground representation of actions and formulas, in order to find schematic invariants. Starting with a set of initial schematic invariant candidates, that hold in the initial state, these candidates are not grounded with all objects, but only with a small number of objects using a limited grounding function as described in Definition 16 and Definition 17, to later perform the basic invariance tests with the grounded invariant candidates. This approach makes use of the structural symmetry of the state space generated by schematic actions, to correctly find invariants, even without considering all possible instantiated formulas.

3.3 Algorithm

This section provides a detailed explanation of the pseudo-code for the invariant synthesis algorithm proposed by Rintanen [2] outlined in Algorithm Algorithm 1.

Pseudo-Code[2]

Algorithm 1 Schematic Invariant Synthesis Algorithm

1: C_s := schematic formulas true in the initial state
2: A_s := schematic actions
3: C := all ground instances of C_s
by instantiating all schematic formulas in C_s using limited grounding
4: A := all grounded actions
by instantiating all schematic actions in A_s using limited grounding
5: **repeat**
6: $C_0 := C$
7: **for** each $a \in A$ and $c \in C$ **do**
8: **if** $C_0 \cup \{regr_a(\neg c)\} \in$ SAT for some c **then**
9: $C := (C \setminus \{c\}) \cup \text{weaken}(c)$
10: **end if**
11: **end for**
12: **until** $C = C_0$
13: I_s := all schematic invariants
extracted from the found ground invariants in C
14: **return** I_s

The algorithm assumes, that in the beginning, the following two sets are given:

1. The first given set is the set of schematic formulas C_s, that hold in the initial state. These formulas are the initial schematic invariant candidates of the problem. Each candidate $c_s \in C_s$ is a disjunction of at most N literals, where in our case $N = 2$.

2. The second given set is the set of all schematic actions A_s of the problem.

The first step of the algorithm (lines 3 and 4) is to instantiate these two sets, to have C, a set of all possible ground instances of C_s and a set of all grounded actions A. It is important to note that limited grounding is used here.

Limited Grounding of Schematic Invariant Candidates and Actions

The initial schematic invariant candidates and schematic actions are all instantiated using a limited grounding function $D'(t), t \in T$, where T is the set of all types. The limited grounding function satisfies the constraints defined in Definition 17.
This means that a schematic candidate $c_s \in C_s$ and a schematic action $a_s \in A_s$ are instantiated, by replacing every occurrence of a schematic variable x with $t = t_{var}(x) \in T$ by an object $o \in D'(t)$. Note that one schematic invariant candidate c_s can generate multiple ground invariant candidates when instantiated because the schematic variables of a candidate are replaced by all combinations of objects. However only those combinations are selected, that match the type-object mapping defined by $D'(t)$ and that do not violate the inequality constraints χ of c_s as described in Definition 13. Similarly, schematic actions can also generate multiple ground actions, but without having a restriction on inequalities.

Outer Loop

Now having the set of the initial ground invariant candidates C and the instantiated actions A, the loop, where the candidates are checked using an invariance test begins. At every outer iteration on line 6, the set of ground invariant candidates C is copied to C_0. This copy is later used, to be able to compare it with the possible modified candidate set C at the end of each outer iteration on line 12.

Inner Loop

Inside the outer loop, we have an inner loop, where we iterate over each combination of $a \in A$ and $c \in C$. For each of these combinations, we check on line 8 whether the following formula $C_0 \cup \{regr_a(\neg c)\}$ is satisfiable, denoted as $\in SAT$. When for some $c \in C$ this formula is satisfiable, the candidate is not an invariant, leading to line 9, the modification of the candidate set: $C := (C \setminus c) \cup \text{weaken}(c)$. The candidate set is modified by removing the candidate c because it is not an invariant and by adding weaker forms of c, defined by the function $weaken(c)$.

SAT-Check

The verification, of whether the formula $C_0 \cup \{regr_a(\neg c)\}$ can be satisfied is the heart of the algorithm, determining if a candidate is an invariant or not. In this subsection, on the one hand, the composition of the formula is discussed, and on the other hand, why the satisfiability of that formula leads to the verification of invariants is explained.

The formula is a set of formulas, forming a conjunction. The conjunction is built from a union of two sets. The first set is the ground invariant candidate set C_0, which was copied at the beginning of the outer loop. This formula C_0 forms a conjunction of all candidates inside C_0. The second set of the union is $\{regr_a(\neg c)\}$, the regression of the negated candidate c with respect to the ground action a. As stated in Definition 15 this results in the following conjunction:

$$regr_a(\neg c) = pre(a) \wedge \bigwedge(\{\neg c_1, \neg c_2, ...\neg c_n\} \setminus \{e_1, e_2, ..., e_k\})$$

where $pre(a)$ is the precondition of a, which is a conjunction of ground literals, c_i is a ground literal in the ground candidate c and $e_i \in add(a)$ is a ground literal from the add effects of a. Note that the ground candidate c is a disjunction of ground literals, meaning that the negation of c is a conjunction of the negated ground literals of c, because of the rule that: $\neg(l_1 \vee l_2 \vee ... \vee l_n) = \neg l_1 \wedge \neg l_2 \wedge ... \wedge \neg l_n$.
To understand the SAT check, it is important to understand the meaning of that regression. The regression $\{regr_a(\neg c)\}$ is a conjunction, that describes the condition that must hold, for a to be applicable and the application of a would lead to a formula where the disjunction of ground literals c is false. In other words, the regression describes what condition is needed for the application of a to make the ground candidate c false.
Given this information, the SAT-Check in line 8 checks, whether the conjunction of all ground candidates in that iteration and the condition for a to make the candidate c false is satisfiable.

This SAT check can have two outcomes, either the formula is satisfiable or not. But what does each outcome mean for the ground invariant candidate c?

When the SAT check returns true, it means an interpretation exists, so the formula is satisfied. As described earlier, the candidate set C_0 is a set of potential invariants, nevertheless, the candidates together as a conjunction describe formulas that hold in some states of the problem. When such a formula, combined with a condition for a to be applicable leading to a state where the candidate c does not hold (regression), is satisfiable, we can conclude that the ground invariant candidate c cannot be an invariant.

Conversely, when the SAT check returns false, we know that the regression cannot be satisfied at least for the states of the problem in which the conjunction of invariant candidates holds. If it is the case that the SAT check returns false for all actions $a \in A$, we can conclude that the candidate c remains a candidate, even though it is possible that the candidate c being an invariant can be proven wrong in later iterations.

Weakening

In the previous subsection, the SAT check was described. As mentioned there, when the SAT check returns true for a candidate c and an action a, the candidate cannot be an invariant. This leads to the removal of the candidate from the candidate set C_0 in line 9, but weaker forms of the candidate c are added to the set, where the weaker forms are defined in the function weaken(c). The strength of an invariant can be defined by the implications of the invariant formulas. When we have two invariant formulas ϕ_1 and ϕ_2 and it holds that $\phi_1 \implies \phi_2$, ϕ_2 is considered as a weaker invariant formula. This implication suggests that ϕ_1 is more restrictive since ϕ_1 is satisfied under fewer circumstances compared to ϕ_2.

Because in the loops only ground invariant candidates are used, weaken(c) adds a literal to the candidate c forming a disjunction, but only when the candidate consists of one literal, since we only consider invariants of at most $N = 2$ literals. For candidates with two literals, the weaken function is defined as weaken(c) $= \emptyset$.

It is important to note that with weakening the candidate c, the regression of the weaker candidate becomes a stronger condition, meaning that fewer states of the problem satisfy the regression. On the other hand, removing the candidate from the candidates set C_0 and adding weaker candidates to it, results in a formula that is satisfied by more states of the problem than before. Due to this fact, it is possible that candidates that were not removed from the candidates set in an earlier iteration can be removed in later iterations.

End the Loops

The algorithm continues to iterate over the candidates and actions until for all candidates and all actions the SAT check returns false. This is equivalent to the fact that the candidates set C_0 remained unchanged after all iterations leading to the equality of the set before the iterations as stated in line 12. The algorithm then successfully found the ground invariants contained in C.

Schematic Invariant Extraction

After the algorithm found a set of ground invariants, the schematic invariants need to be extracted. Since the ground invariants are generated using limited grounding, the following lemma can be applied when extracting the schematic invariants in line 13:

Lemma 1 *Invariance Modulo Renaming*[**2**]

For all rounds of the described algorithm, if for one substitution σ, $c\sigma \in C$, then $c\sigma' \in C$ for all other substitutions σ', if for all x and y , $\sigma(x) = \sigma(y) iff \sigma'(x) = \sigma'(y)$.

Example:
Assume there are 3 objects a, b and c which are of the same type and 3 schematic variables x, y and z.

- If $p(a, a, a)$ is a ground invariant, then $p(x, x, x)$ is a schematic invariant
- If $p(a, b, c)$ is a ground invariant, then $(x \neq y) \wedge (x \neq z) \wedge (y \neq z) \implies p(x, y, z)$ is a schematic invariant.

After the extraction of the schematic invariants leads to a set of all found schematic invariants I_s, which is the result of the algorithm.

4 Implementation

For this thesis, the invariant synthesis algorithm described in the previous chapter is implemented in the Fast Downward planning system by replacing the current invariant synthesis in the translation process. This chapter describes the details of how the algorithm was implemented and the difficulties involved in this process. Some additional implementations were necessary, for the algorithm to work in the planning system.

4.1 Fast Downward

Fast Downward is a domain-independent classical planning system based on heuristic forward search and hierarchical problem decomposition [3]. The planner can find plans for problems in the full range of PDDL2.2 [8], where STRIPS [6] is included. For this thesis, we only consider propositional STRIPS [6] tasks.

The Planner can be divided into 3 phases, where the first phase is the translation, the second phase is the knowledge compilation and the last phase is the search. The invariant synthesis is made during the translation process, which is implemented in Python.

The translation process converts a PDDL task into a task in finite domain representation. The translation process itself can again be divided into sub-steps:

1. Normalization
2. Invariant Synthesis
3. Grounding
4. Finite Domain Representation Task Generation

For this thesis, the Invariant Synthesis Algorithm proposed by Rintanen is implemented in Fast Downward, where the current Invariant Synthesis (step 2) is replaced.

4.2 Overview

This section provides an overview of the processes that are involved in the implementation of the algorithm. The translation of a PDDL [8] task into a task in finite domain representation can be divided into the following steps:

1. Normalization of the task.
2. Invariant Synthesis.
3. Generation of mutex groups.
4. Translation of Task into Finite Domain Representation.

Normalization is used to simplify the task, to have a task that has the following characteristics [1]:

- The goal formula is a conjunction of literals.
- All action preconditions are a conjunction of literals.
- All states are a conjunction of positive literals.

The normalization is already implemented in Fast Downward and is the starting point for the implementation. Since the structure of the found invariants is different than in the current implementation, building the mutex groups has also been implemented using a maximal clique enumeration algorithm. These found mutex groups can then be passed to the already implemented translation process. The relevant implementation for this thesis includes the invariant synthesis and generation of mutex groups.

4.3 Relevant Classes

For the implementation of the algorithm, several existing classes were used and new necessary classes were implemented. In this section, these classes are listed and explained. Since some parts of the classes are not relevant to the implementation of this thesis, only the necessary parts are listed and explained.

Class Type

```
class TypedObject:
    name: str
    basetype_name: str
```

The class Type represents a type as described in Definition 1. Each type is defined by the name of the type and by the name of the direct supertype. By direct supertype is meant, that the supertype can also have a supertype. For types that do not have any supertypes, the base type name is None. Given the base type name for each type of a task, the hierarchy of types of a task can be concluded.

Class TypedObject

```
class TypedObject:
    name: str
    type_name: str
```

The class TypedObject represents an object of a task as described in Definition 1. Each object is defined by the name of the object and the type of the object.

Class Predicate

```
class Predicate:
    name: str
    arguments: List[TypedObject]
```

The class Predicate represents a predicate as described in Definition 3. A predicate is defined by the name of the predicate and a list of typed objects that represent the arguments of the predicate.

Class Literal

```
class Literal:
    predicate: str
    args: List[str]
```

The class Literal represents a literal as described in Definition 4. A literal is defined by the name of the predicate, that matches with the name of an object of class Predicate and a list of strings that represent either a schematic variable or the name of an object. Note that schematic variables do not have their own class. Schematic variables can have any string that is not the name of any object of class TypedObject.

Class Atom and Class NegatedAtom

To distinguish between positive and negative literals, there are the classes Atom and NegatedAtom that inherit from the class Literal. These classes have an additional boolean field that defines if the literal is negated or not. These two classes also have a function to negate an object of its class.

```
class Atom(Literal):
    negated = False
    def negate(self):
        return self.NegatedAtom(self.predicate, self.args)

class NegatedAtom(Literal):
    negated = True
    def negate(self):
        return self.Atom(self.predicate, self.args)
```

Class InstantiatedAction

```
class InstantiatedAction:
    name: str
    objects: List[str]
    precondition: List[Literal]
    effects: List[Literal]
```

The InstantiatedAction class represents a grounded action as described in Definition 8. An object of that class has a name, a list of object names, where each of these object names matches with the name of a TypedObject class object, a list of literals that represents the precondition conjunction, and another list of literals that represents the effects.

Class Task

```
class Task:
    name: str
    objects: List[TypedObject]
    types: List[Type]
    predicates: List[Predicate]
    actions: List[Action]
    init: List[Atom]
```

The class Task holds all relevant objects of the PDDL task. Note that the actions are a list of Action class objects. The class Action has a similar structure to the InstantiatedAction class. Because we only need the instantiated actions for this implementation, the action class explained in this section. All actions are instantiated during the algorithm leading to a list of InstantiatedAction class objects.

Class SchematicInvariant

```
class SchematicInvariant:
    literals: List[Literal]
    inequalities: List[set[str]]
    var_mapping: dict[str, str]
```

The SchematicInvariant class represents a schematic invariant as described in Definition 13 and holds all necessary information for a schematic invariant formula. The field literals define the literals that form the disjunction of the formula. The inequalities is a list of sets, where each set represents one inequality of the two variable names it contains. The var_mapping field is a dictionary that defines a mapping from variable name to type name for all variables that are contained as arguments in all literals.

Class GroundInvariant

```
class GroundInvariant:
    literals = set[NegatedAtom]
```

The class GroundInvariant represents a ground invariant as described in Definition 13 by a set of NegatedAtom class objects. The ground invariant formula is the disjunction of the negative literals contained in the literals set.

4.4 Invariant Synthesis

The goal of the invariant synthesis is to find ground invariants that are of the form $\neg p(a, b)$ or $\neg p(a, b) \vee \neg q(c, d)$ because these invariants can then be used in the next step to build the mutex groups as described in Definition 19. The implementation of the invariant synthesis as described in chapter 3 can be divided into the following processes:

1. Creating the initial schematic invariant candidate set.
2. Generating the limited grounding function.
3. Grounding of the initial schematic invariant candidates using limited grounding.
4. Grounding of the schematic actions using limited grounding.
5. Finding ground invariants as described in chapter 3.
6. Extracting schematic invariants from the ground invariants.
7. Grounding the found schematic invariants using normal grounding resulting in a set of ground invariants.

Initial Schematic Invariant Candidate Set

The first step in the implementation is to generate an initial schematic invariant candidate set, that is later grounded using the limited grounding function. A schematic invariant is an initial candidate if all grounded candidates of that schematic invariant are true in the initial state. It is important that this candidate set consists of strong invariant candidates. because as described in algorithm explanation, the candidate set describes later in the SAT-Check a conjunction that holds in some states of the problem. By having an initial candidate set that contains strong candidates, the conjunction of all candidates describes a conjunction that holds in many states of the problem. Another point that is important for this initial candidate set is to have a high amount of different candidates.

The reason for this is that even if the candidates are later proven to not be an invariant, they are weakened generating other candidates that possibly are invariants. In other words, some invariants could potentially not be found if the initial candidate set is not strong enough and does not contain enough candidates.

The first step is to extract all schematic invariant candidates of the simplest form $\neg p(x, y)$, without any inequalities. schematic invariants candidates of this form are in the strongest possible form.
Finding these initial candidates can be achieved by iterating over all predicates of the task and checking whether all grounded negative literals having that predicate are true in the initial state Note that in this step the limited grounding function is not yet used. All candidates that are true in the initial state are added to the initial candidate set.

The next step is to create weaker invariant candidates that hold in the initial state using the predicates that are not part of any initial candidate contained in the initial candidate set after the first step. There are three different ways how schematic candidates can be made weaker:

1. Adding another literal to the candidate forming a disjunction
2. Setting two schematic variables equal
3. Adding an inequality of two schematic variables

First, for all these literals that are not true in the initial state and are in the simplest form, a literal is added to form a candidate with two literals that describe a disjunction. Only literals that share at least one variable of the same type are added together. These simple candidates and candidates describing a disjunction are added to a set "candidates_to_weaken" that needs to be further weakened.

The further expansion of the initial schematic invariant candidate set is outlined in the following pseudo code:

Algorithm 2 Expansion of initial schematic invariant candidate set

```
1: initial_candidates := all initial schematic invariant candidates
2: while size(candidates_to_weaken) not zero do
3:     candidate = candidates_to_weaken.pop()
4:     eq_candidates, non_eq_candidates = set_variables_equal(candidates)
5:     ineq_candidates, non_ineq_candidates = set_variables_unequal(candidates)
6:     candidates_to_weaken.update(non_eq_candidates)
7:     candidates_to_weaken.update(non_neq_candidates)
8:     new_schematic_candidates = eq_candidates + ineq_candidates
9:     initial_candidates.add(new_schematic_candidates)
10:    remove_unnecessary_weak_forms(candidates_to_weaken)
11: end while
```

In each iteration a schematic invariant candidate is processed, by setting the schematic variables equal and adding inequalities.
For both of those functions, all possible pairs of schematic variables are considered, that share the type or supertype, resulting in a set of new schematic candidates that are again checked if they are true in the initial set. Candidates that are true and those that are not true are seperated and returned by the corresponding function. The functions return empty sets if no further weakening is possible, which is the case when all schematic variables are either equal to other schematic variables or are part of an inequality. At the end of an iteration, candidates that describe a disjunction are removed, if they contain a literal, that itself is in the initial schematic candidate set with the same equalities and inequalities.

The implementation of the generation of the initial schematic invariant candidate set was one of the most challenging parts of the whole implementation. One of the difficulties was the weakening of schematic invariant candidates because several conditions for a pair of schematic variables had to be met for them to be set equal, to add an inequality, or to add another literal to the candidate. Especially in later iterations when several weakening forms are already combined in a candidate, making sure that correct weaker candidates are created was challenging.

Generation of the Limited Grounding Function

The limited grounding function is a mapping from a type to a set of objects. The first step for the generation of the limited grounding function as described in Definition 17 is the calculation of all lower bound numbers for each type, by using the formula in Definition 16. This lower bound number for type t defines how many objects need to be at least contained in the set of objects for type t.

As defined in point three in Definition 17, that $D'(t_0) \subset D'(t_1)$ iff $D(t_0) \subset D(t_1)$, for all $\{\{t_0, t_1\} |\ t_0, t_1 \in T\}$, means that all objects of type t_0 need to be contained in the set of objects for type t_1, if t_0 is a subtype of t_1. Because of this rule, in the implementation, we start at the bottom of the type hierarchy with choosing which objects of these types are contained in the corresponding set of objects for the limited grounding function. For the types at the bottom of the hierarchy, meaning all types that are not a supertype of any other type have exactly as many objects as defined by the lower bound number. For each higher level in the hierarchy, the types need to include all objects of their subtypes. After adding all subtype objects to the set of objects, other objects that were not yet included in any other object set need to be added, if the size of the object set is smaller than the lower bound. With this procedure, we made sure that the number of objects is as small as possible.

Grounding of the Initial Schematic Invariant Candidates and Schematic Actions using Limited Grounding

For this step of the invariant synthesis, the initial schematic invariant candidate set is instantiated using the limited grounding function described in the last subsection resulting in an initial ground invariant candidate set. The instantiation is processed by iterating over the set and instantiating every schematic candidate with limited grounding. Note that some schematic candidates can generate ground invariants that are also generated by other candidates, which is why a set is used, ensuring that no duplicates are contained in the set. The instantiation function generates all possible variable-to-object mappings enforcing the inequality constraints of the candidate and generates one object of the GroundInvariant class for each mapping. Each schematic variable with type t can only be replaced by objects defined by the limited grounding function.

Similarly to the grounding of the initial schematic invariant candidate set, the schematic actions are instantiated using the limited grounding function. The result is a set of objects of the InstantiatedAction class.

Finding Ground Invariants

The invariant synthesis algorithm proposed by Jussi Rintanen described in chapter 3 was implemented that finds the ground invariants given the initial ground invariant set and the ground actions set.

Algorithm 3 Implemented schematic invariant synthesis algorithm

```
1:  C := set of all initial ground invariant candidates
2:  A := set of all ground actions
3:  while True do
4:      C_weaken := empty set for storing candidates that need to be weakened
5:      for c in C do
6:          for a in A do r = generate_regression(a, c)
7:              if sat_check(C_0, r) is True then
8:                  C_weaken.add(c)
9:                  break
10:             end if
11:         end for
12:     end for
13:     if C_weaken empty then
14:         break
15:     else
16:         C = update_candidate_set(C, C_weaken)
17:     end if
18: end while
```

The ground invariants are found by iterating over all candidates in the ground invariant candidate set. For each candidate, an iteration over all ground actions is made. For each candidate c and action a, the regression is calculated and a satisfiability check (SAT-Check) with the candidate set C and the regression as described in chapter 3 is performed. When the SAT-check is true, the candidate needs to be weakened and is therefore added to C_{weaken}. Different than in the explanation in chapter 3, the candidates are removed from the candidate set C and the new weakened candidates are added to the candidate set C after the iteration over all candidates and not directly during the iteration. The iteration over the actions is stopped as soon as the SAT-check returns true for one of the actions. This process is repeated as long as at least one candidate is proven to not be an invariant, which can be checked, with the emptiness of C_{weaken}, because all

candidates that need to be weakened are stored in that set. When at the end of all iterations over C and A, C_{weaken} is empty, all candidates contained in C are ground invariants.

SAT-Check

The SAT-check was implemented by my supervisor Tanja Schindler.

The SAT check is used to check whether the conjunction of the union of all candidates in C and the regression of one candidate $c \in C$ with respect to an action a is satisfiable or not. This function takes the candidate set C and the regression of one candidate $c \in C$ with respect to an action a as an input and returns True, when the conjunction is satisfiable and false if the conjunction is not satisfiable. The function makes the following assumptions:

1. each candidate is a disjunction of at most two literals
2. the set of candidates is satisfiable
3. the set of candidates is closed under resolution

The first point holds, since we are only considering invariant candidates with at most two literals, that together form a disjunction, as mentioned in Definition 18. The second point also holds since we are considering only invariants with negative literals, meaning that the truth assignment where all literals are false and thus all negative literals are true will always result in the conjunction of all candidates being true. The third and last point assumes that the set of candidates is closed under resolution. A candidate set is closed under resolution when the resolution of any pair of candidates is contained in the set. A resolution of two disjunctions in general leads to a new combined formula, which is applicable when the two disjunctions share complementary literals, meaning that one disjunction contains the positive literal l and the other disjunction contains the negative literal $\neg l$. Then the combined disjunction without l and $\neg l$ is also contained in the set [9]. Because we only have negative literals in the ground candidate set, no pair of candidates can be combined with resolution, which means that the candidate set is closed under resolution.

Given these preconditions of the candidate set, the conjunction of the union of all candidates in C and the regression of one candidate $c \in C$ with respect to an action a can only be unsatisfiable when the literals of the regression falsify a candidate. The literals of the regression falsify a candidate when the negated literals of a candidate are contained in the regression.

The SAT-check therefore iterates over all candidates in C and checks whether the negated literals of the candidate are contained in the regression set. For candidates with one literal, the SAT-check returns false if that negated literal is found in the regression set and for candidates with two literals, the SAT-check returns false if both negated literals are found in the regression set. The SAT-check returns true only if for all candidates the negated literals are not contained in the regression set.

Candidate Set Update

When the iteration over all candidates is completed, the candidate set C is updated by removing all candidates in C_{weaken} and by adding all weakened forms of all candidates in C_{weaken}. A candidate c can only be weakened if it contains exactly one negated literal. The candidate c is weakened by creating ground invariant candidates with two negated literals with all negated literals that are not contained as single literal invariants in C and are not equal to c.

Schematic Invariant Extraction

After the ground invariants are synthesized, the schematic invariants need to be extracted from them. This can be achieved by applying the concept described in Lemma 1. The lemma describes how schematic invariants can be extracted from ground invariants that were instantiated using limited grounding.
The extraction is made by iterating over all ground candidates. For each ground invariant, a schematic invariant can be extracted. Such a schematic invariant has one schematic variable for each different object, by ensuring that the same objects are replaced by the same schematic variables and different objects are not replaced by the same schematic variables. For each pair of objects in the ground candidate, consisting of distinct objects, an inequality is added to the schematic invariant. This procedure is made by creating a list of all objects in the ground invariant and assigning each object to a schematic variable. For each pair of distinct schematic variables, an inequality set consisting of these two variables is created. After the iteration all extracted schematic invariants are added to the schematic invariant set, which is then instantiated again, by considering all objects, without the limited grounding function. These ground invariants are the result of the invariant synthesis algorithm.

4.5 Building Mutex Groups

The purpose of synthesizing ground invariants is to generate mutex groups that are later used for the translation. As described in Definition 19, invariants can form a mutex group when each invariant shares at least one literal with every other invariant of the group, resulting in a group of all literals contained in the disjunctions of the ground candidates.
In the implementation of this work, mutex groups are built by creating a graph from all ground invariants. Each vertex of the graph represents one literal. For each pair of positive literals that are contained together as a disjunction in negative form for a ground candidate in the ground invariant set, an edge exists in the graph between the two vertices representing the two literals. In such a graph, a mutex group of positive literals can be identified by a group of literals that are connected with every other literal of the group by an edge, meaning for every two distinct literals of the group a ground invariant with these two literals in negated form exists. Such a subset of vertices, with these properties, is called a clique. A clique in a graph is a subset of vertices, such that every vertex is connected to every other vertex in the subset by an edge.

For identifying such cliques in a graph, a maximal clique enumeration algorithm by Tomita [4] was implemented. It is important to note, that the algorithm is already implemented as a part of the Fast Downward planning system written in C++. For the implementation of this thesis, the algorithm from the planning system was translated from C++ to Python keeping the structure and logic as closely as possible of the already implemented algorithm, such that it is applicable in the context of the invariant synthesis, which is implemented in Python. Because of this reason, a detailed explanation of the algorithm is not provided, since this algorithm was not implemented as a part of this thesis.

Maximal Clique Enumeration Algorithm

The algorithm by Tomita [4] is a depth first search algorithm that identifies all maximal cliques in an undirected graph. The algorithm starts with one vertex of the graph and recursively expands the connected vertices and searches for larger subgraphs that are connected (cliques). For a more detailed explanation of the algorithm, we refer to the original Paper [4]. The algorithm takes a graph as an input and outputs all maximal cliques as a list of lists, where each inner list represents the vertices that form a clique. For building the mutex groups from the found ground invariants, a graph as described before is generated and given to the algorithm as an input and the algorithm returns a list of maximal cliques as an output.

Translation

The found mutex groups are the result of this thesis' implementation. Since distinct positive literals can be contained in multiple mutex groups, which is not allowed for finite domain translation, existing functions of the Fast Downward planner are used to prepare the found mutex groups for translation. These functions make sure that only literals where the truth value can change over time are contained in the mutex group and no literal is contained in more than one mutex group. In the translation, each of these mutex groups can be translated into one finite domain representation variable, where the finite domain representation variable has one possible value for every literal in the mutex group, plus an additional value for the case that all literals of that mutex group are false. All literals that are not contained in any mutex group and its truth value can change, are translated as one finite domain variable with two possible values, one indicating that the literal is true and the other values that the literal is false.

5 Evaluation

For the evaluation of the invariant synthesis algorithm implemented in the Fast Downward planning system, the algorithm is compared to the already existing invariant synthesis by Helmert. For the comparison of the algorithms the efficiency and the effectiveness are analyzed.

5.1 Data Sources and Experimental Methodology

For the evaluation the Fast Downward planning system a Fast Downward benchmark collection was used. This benchmark collection is an unofficial collection of international planning competition (IPC) benchmark instances [5]. The benchmark contains numerous different task domains defined in PDDL [8], where for each domain several problem instances exist. This collection also contains tasks that are not suitable for the invariant synthesis algorithm of this thesis, which is why only a subset of the tasks were used. Only tasks that have the following characteristics have been chosen for the evaluation:

1. Tasks that are restricted to the STRIPS format
2. Tasks, where the types and the type hierarchy are defined
3. Tasks that do not have actions with conditional effects

All computational experiments were performed on the sciCORE cluster of the University of Basel [10]. This environment ensures that all runs of the planning system are executed under consistent conditions. Each problem instance of all domains was solved once by Fast Downward with the already existing invariant synthesis by Helmert and once by Fast Downward with the newly implemented schematic invariant synthesis algorithm.

For the search, both algorithms used the same algorithms. As the search algorithm, the "A* search" algorithm with the "Landmark-Cut (LM-Cut) heuristic was chosen. This combination used with the Fast Downward planner finds optimal plans for STRIPS tasks and is very efficient [11]. For each run of the planner on the sciCORE cluster, an overall time limit of 10 minutes and an overall memory limit of 3584 MB are set. One run represents the planner to solve one specific problem instance.

The data from every run was analyzed using Python and the results are documented in this chapter.

5.2 Time and Memory Errors

In the first section, the number of occurrences of errors for each algorithm per domain is analyzed. There are three different errors that occurred for all runs. The first error is the "search out of time" error (s-o-t). This error occurs when the overall time limit is reached while the planner is searching for a plan. This means that the translation was completed successfully. When the time limit is reached during the translation step, then the "translation out of time" (t-o-t)error occurs. The third error that occurred is the "translation out of memory" error (t-o-m), which means that the overall memory limit was reached. In the following table, the number of errors per domain and per algorithm are displayed.

	Existing Inv. Syn.				New Schematic Inv. Syn.			
domain	t-o-m	t-o-t	s-o-t	total	t-o-m	t-o-t	s-o-t	total
barman-mco14	0	0	20	20	0	0	20	20
barman-opt11	0	0	16	16	0	0	16	16
barman-opt14	0	0	14	14	0	0	14	14
barman-sat11	0	0	20	20	0	0	20	20
barman-sat14	0	0	20	20	0	0	20	20
childsnack-opt14	0	0	20	20	0	0	20	20
childsnack-sat14	0	0	20	20	0	0	20	20
elevators-opt08	0	0	11	11	0	0	11	11
elevators-opt11	0	0	4	4	0	0	4	4
elevators-sat08	0	0	28	28	0	0	28	28
elevators-sat11	0	0	20	20	0	0	20	20
floortile-opt11	0	0	14	14	0	0	14	14
floortile-opt14	0	0	15	15	0	0	15	15
floortile-sat11	0	0	15	15	0	0	15	15
floortile-sat14	0	0	18	18	0	0	18	18
hiking-agl14	0	0	20	20	0	0	20	20
hiking-opt14	0	0	12	12	0	0	12	12
hiking-sat14	0	0	18	18	0	0	18	18
nomystery-opt11	0	0	6	6	0	0	6	6
nomystery-sat11	0	0	12	12	0	0	12	12
parking-opt11	0	0	19	19	0	8	11	19
parking-opt14	0	0	18	18	0	6	13	19
parking-sat11	0	0	20	20	0	20	0	20
parking-sat14	0	0	20	20	0	20	0	20
pegsol-08	0	0	3	3	0	0	3	3
pegsol-opt11	0	0	3	3	0	0	3	3
pegsol-sat11	0	0	3	3	0	0	3	3
scanalyzer-08	0	0	17	17	27	0	0	27
scanalyzer-opt11	0	0	10	10	19	0	0	19
scanalyzer-sat11	0	0	16	16	20	0	0	20
sokoban-opt08	0	0	2	2	0	0	2	2
sokoban-opt11	0	0	0	0	0	0	0	0
sokoban-sat08	0	0	7	7	0	0	7	7
sokoban-sat11	0	0	6	6	0	0	6	6
thoughtful-mco14	0	0	20	20	0	20	0	20
thoughtful-sat14	0	0	15	15	0	20	0	20
tidybot-opt11	0	0	7	7	0	20	0	20
tidybot-opt14	0	0	13	13	0	20	0	20
transport-opt08	0	0	19	19	0	0	19	19
transport-opt11	0	0	14	14	0	0	14	14
transport-opt14	0	0	14	14	0	0	14	14
visitall-opt11	0	0	10	10	0	0	10	10
visitall-opt14	0	0	15	15	0	0	15	15
woodworking-opt08	0	0	14	14	0	26	0	26
woodworking-opt11	0	0	9	9	0	20	0	20
woodworking-sat08	0	0	23	23	0	27	0	27
woodworking-sat11	0	0	19	19	0	19	0	19

Table 5.1: number of errors for each algorithm per domain

Note that for all domain names, the "-strips" ending is not included because of space reasons.

As we can see, the existing algorithm does not contain any "translation out of time" errors nor any "translation out of memory" errors, meaning that the invariant synthesis was successfully completed for all problem instances. For the schematic invariant synthesis algorithm, on the other hand, several problems caused these errors during the translation process. The "translation out of time" errors occurred in the parking, thoughtful, tidybot, and woodworking problems. For the parking domain, all problem instances have a very high number of objects compared to other problem instances from other domains. It could be the case that this high amount of objects causes the invariants synthesis to take too long to generate all initial schematic invariant candidates since every schematic invariant candidate needs to be tested if it is true in the initial state. For this step, the schematic invariant candidate is instantiated using all objects of the problem, meaning that one schematic candidate generates a large set of ground candidates that are checked.
For the other three domains, the number of objects is not higher compared to other domains, but the number of actions and predicates is very relatively high. From this, we can assume that similarly to the parking domain, the initial schematic invariant candidate set generation takes a long time, but other than in the parking domain, the invariants synthesis can possibly be the cause of the long duration, since for every candidate we need to iterate over a high amount of grounded actions.

The "translation out of memory" error was only caused by the scanalyzer problem instances. The structure of the domain does not indicate any reasons why this error occurs for the schematic invariant synthesis.

It is interesting to see, that in all cases where the planner using the schematic invariant synthesis had a translation error, a similar amount of "search out of time" errors occurred for the planner using the existing invariant synthesis. This means that even if the translation would be completed in time, the search would probably also take too long. For the following domains, the planner had more total errors using the schematic invariant synthesis algorithm:

- parking-opt14-strips
- scanalyzer-08-strips
- scanalyzer-opt11
- scanalyzer-sat11
- thoughtful-sat14
- tidybot-opt11
- tidybot-opt14
- woodworking-opt08
- woodworking-opt11
- woodworking-sat08

5.3 Time Performance for Finding Invariants

In this section of the evaluation, the efficiency of the invariant synthesis algorithms is analyzed and compared. Planning systems should work correctly and efficiently, and the time needed for the invariant synthesis is important for practical applicability. The time each algorithm needs to complete the invariant synthesis is considered, without taking the the time for the search into account. In both algorithms, the time was evaluated from the point where the function that finds the invariants is called right until the set of invariants is returned by the function.

The data is visualized by a scatter plot. The time needed for each problem instance within one domain is summed up, resulting in a scatterplot that visualizes the time needed for each algorithm per domain. For this evaluation, only problem instances were taken into account, if both algorithms successfully terminated the invariant synthesis. As we saw in the last section, translation out of time errors only occurred for the schematic invariant synthesis, meaning that the durations that exceeded the time limit by the schematic invariant synthesis were not displayed in the scatter plot, which would have led to higher total durations.

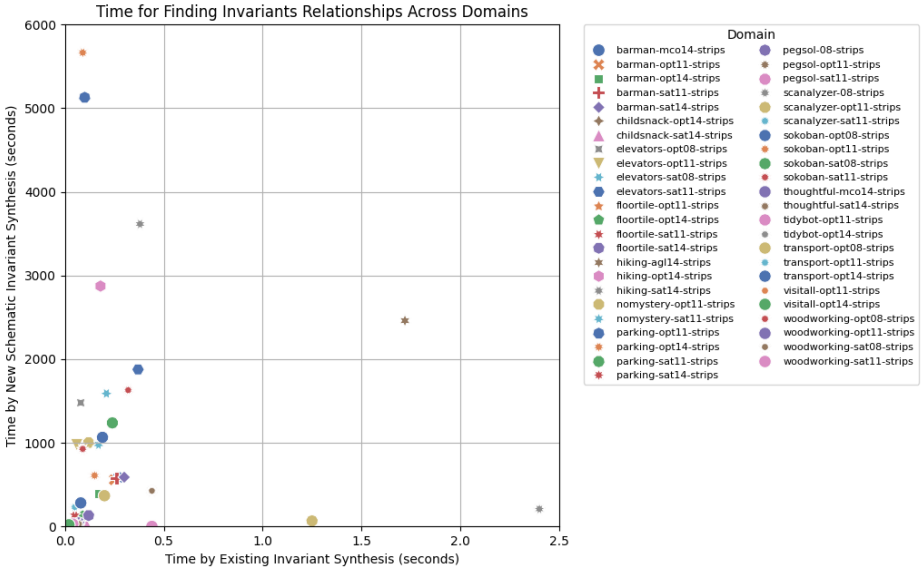

Figure 5.1: Relationship of the total time needed by each algorithm for finding invariants for all problem instances per domain

The scatterplot clearly displays the large time difference for the algorithms to find the invariants. While the times for the existing algorithm range from about 0 to 2.5 seconds, the newly implemented schematic invariant synthesis algorithm needs up to 6000 seconds to find invariants for all problems per domain. Since the time for some domains is very large, the points are not clearly visible. Therefore another scatterplot was created that displays the clustered points more clearly.

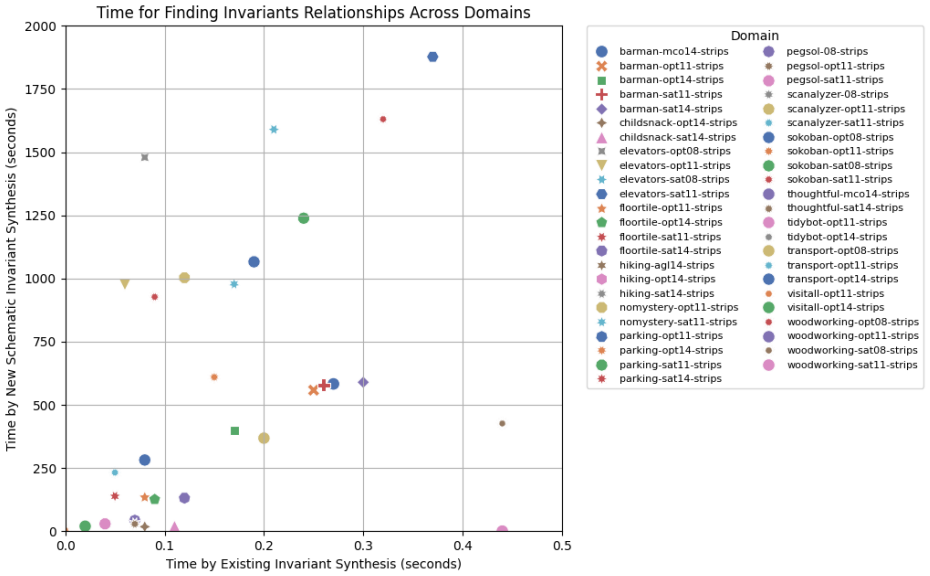

Figure 5.2: Relationship of the total time needed by each algorithm for finding invariants for all problem instances per domain (Zoomed in version

From this plot, we can conclude that the invariant synthesis by Helmert outperforms the implemented schematic algorithm by far. The reason for this large time difference is the inefficiency of the schematic invariant synthesis algorithm. The primary goal of the thesis was to implement a correctly working invariant synthesis, without the priority for efficiency. Optimized data structures and redundancy elimination would definitely lead to better performance.

5.4 Search Time Performance

After we analyzed the time for the translator, we now want to compare the time differences between the search times. From this comparison, we can conclude whether the invariant synthesis algorithm has an impact on how fast the planner can search for plans. For this comparison, a bar plot was created that shows the total search time for all problem instances grouped by the domains. For each domain, two bars are displayed, one for each algorithm.

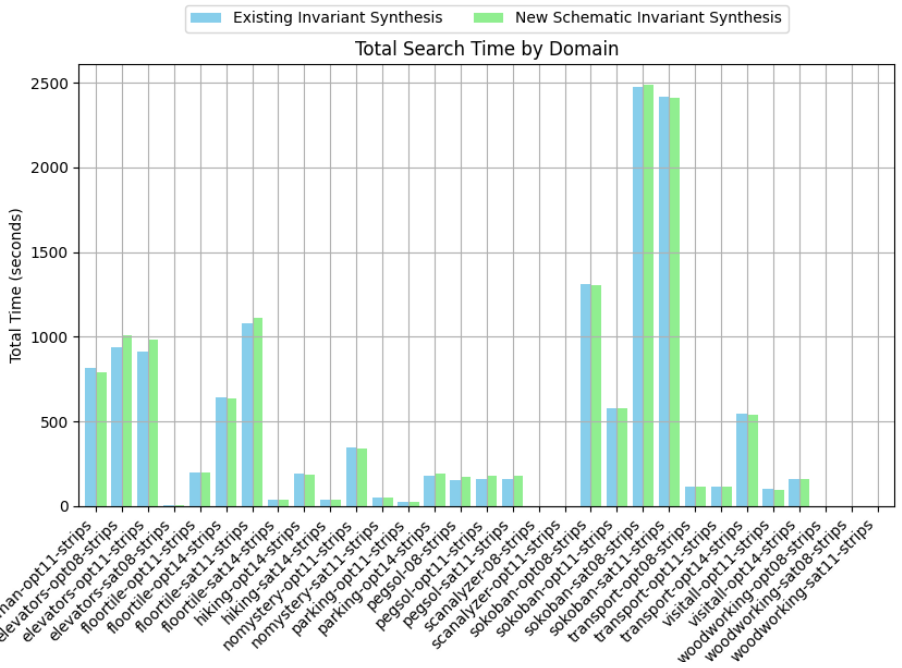

Figure 5.3: Search Time Comparison

We can see that the search times are very similar for all domains. There are no cases where one algorithm resulted in a very different search time compared to the other algorithm. From this, we can assume that both invariant synthesis algorithms find invariants and therefore build mutex groups that result in a similar finite domain representation task. Only by looking at that chart, we can not conclude whether the same amount of mutex groups or the same sizes of mutex groups are built.
Since some search times are relatively higher than others a second bar chart was created using log scales, so all times are visible.

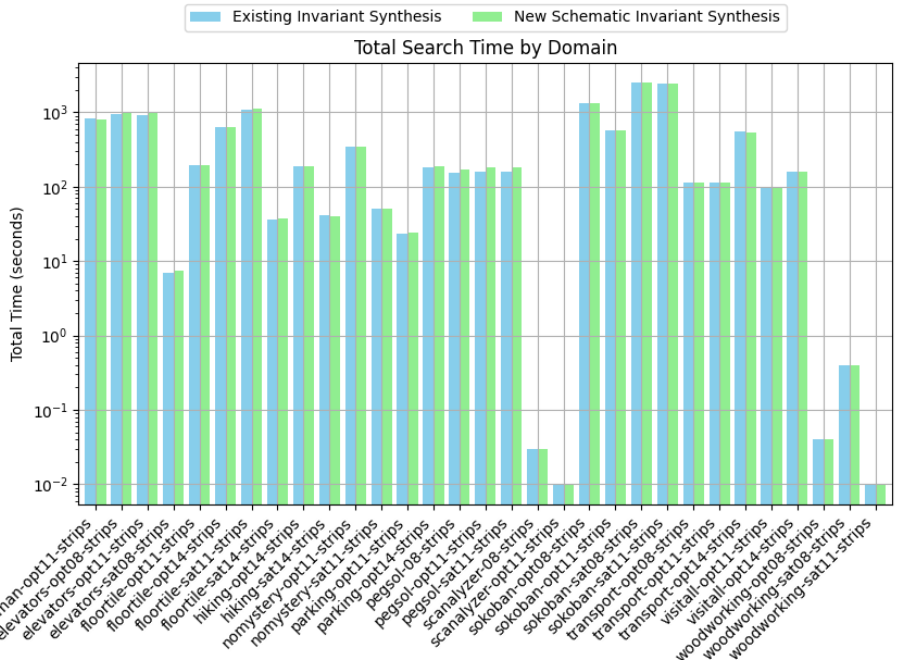

Figure 5.4: Search Time Comparison (log scale)

Even now, when domains that had relatively shorter search times are displayed, no drastic time differences can be found. In general, from these charts, we can not conclude which invariant synthesis is better for the search of the planner.

5.5 Plan Validity

When the search of the Fast Downward planning system successfully terminates, the output is a plan, describing a sequence of actions from the initial state to a state where the goal conditions are met. Such a plan can be validated using the plan validator from the downward lab [12]. The validator verifies whether the found plan is a valid plan, that describes a sequence of actions that correctly finds a path from the initial state to a state where the goal condition is met. The number of valid plans per domain and algorithm is displayed in the following scatterplot:

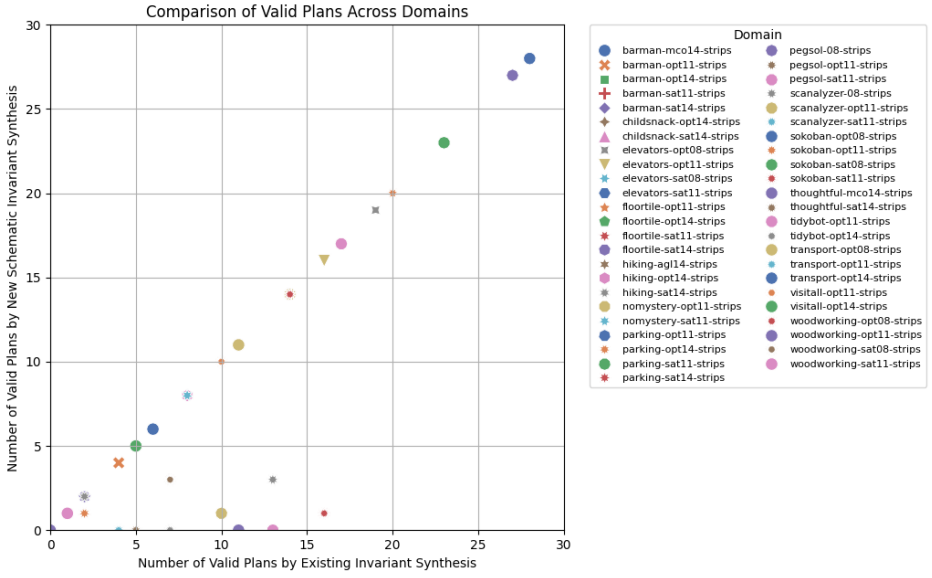

Figure 5.5: Number of valid plans per domain and algorithm

We can see that for most of the domain, the amount of found valid plans is equal for both algorithms. The domains where the planner using the schematic invariant synthesis did find less valid plans are exactly these domains, where more total errors occurred for the schematic invariant synthesis algorithm as analyzed in Time and Memory Errors section. From this, we can conclude that in all cases where the search found a valid plan with the existing invariant synthesis, the search also found a valid plan, when the search successfully terminated. Meaning that the planner does not find invalid plans using the schematic invariant synthesis algorithm.

5.6 Finite Domain Variable Structure

The last analysis that was made, is the structure of the finite domain representation variables. As described in Chapter 4, one mutex group is translated into one finite domain variable, that has one possible value for every literal in the mutex group, plus an additional value for the case that all literals of that mutex group are false. This means that when no mutex groups are used for the translation, one variable for every literal that can change in different states with two possible values would exist. This means that when mutex groups are found, the number of variables is reduced and the number of possible values is increased. In this section, we want to compare the structure of the finite domain variables by comparing the average number of possible values per variable and the number of variables.

Number of Variables

A table was created to display for both algorithms the number of variables per domain. By analyzing this table, we can assume which algorithm generates more mutex groups, which is indicated by fewer variables. Nevertheless, more mutex groups do not directly mean that the algorithm is better. It is possible that fewer mutex groups but with more literals per group can lead to a more efficient search.

domain	Existing Inv. Syn.	Schematic Inv. Syn.
barman-mco14-strips	7254	1731
barman-opt11-strips	2164	860
barman-opt14-strips	1781	664
barman-sat11-strips	5720	1500
barman-sat14-strips	7397	1756
childsnack-opt14-strips	1248	1410
childsnack-sat14-strips	1878	2082
elevators-opt08-strips	380	580
elevators-opt11-strips	245	363
elevators-sat08-strips	770	1535
elevators-sat11-strips	1006	2928
floortile-opt11-strips	624	624
floortile-opt14-strips	575	575
floortile-sat11-strips	728	728
floortile-sat14-strips	686	686
hiking-agl14-strips	494	494
hiking-opt14-strips	229	229
hiking-sat14-strips	293	293
nomystery-opt11-strips	190	190
nomystery-sat11-strips	250	250
parking-opt11-strips	656	724
parking-opt14-strips	735	810
pegsol-08-strips	994	1956
pegsol-opt11-strips	680	1340
pegsol-sat11-strips	680	1340
scanalyzer-08-strips	24	60
scanalyzer-opt11-strips	8	20
sokoban-opt08-strips	1518	1518
sokoban-opt11-strips	1066	1066
sokoban-sat08-strips	1652	1652
sokoban-sat11-strips	1244	1244
transport-opt08-strips	345	345
transport-opt11-strips	217	217
transport-opt14-strips	206	206
visitall-opt11-strips	773	773
visitall-opt14-strips	2258	2258
woodworking-opt08-strips	91	28
woodworking-sat08-strips	74	74
woodworking-sat11-strips	15	15
Total	47148	35124

Table 5.2: number of finite domain variables for each algorithm per domain

Similarly to the evaluation of the time for finding invariants, only problem instances are considered, where both algorithms successfully completed the translation. Problem instances where the schematic invariant synthesis algorithm had

a translation error were removed from this evaluation.
We can see that in this table in general, the number of variables is the same or lower for the existing invariant synthesis algorithm. Only for the barman domains and the woodworking-opt08-strips and domain, the schematic invariant synthesis algorithm has fewer variables. Especially the difference in the barman domains stands out, as the translation with the schematic invariant synthesis uses much fewer variables for the finite domain representation. This suggests, that for these barman domains, the schematic invariant synthesis generates mutex groups that are very different from the existing invariant synthesis. Nevertheless, as we saw in the last section, the search time for the barman domain, the planner is slightly faster using the existing invariant synthesis. It is also interesting to see that there are many cases where the exact same amount of variables are generated. This suggests that the found mutex groups are either the same or the number of found mutex groups is very similar.

Average Number of possible Values per Variable

Contrary to the last analysis where the number of variables was analyzed, which can be interpreted as the number of found mutex groups, in this part we analyze the average number of possible values per variable and the standard deviation of it. These values can be interpreted as the sizes of the mutex groups. For this comparison, a table is created that displays the mean of the number of possible values per variable and the standard deviation of that number per domain for both of the algorithms.

The mean was calculated for every individual problem instance run with one algorithm. These means are then averaged over the domain and algorithm using the weighted mean, by giving more weight to the means that were calculated over more variables. This is important since simply averaging over the means would not take into account how many variables this mean stands for. The standard deviation is also calculated for each individual problem instance run with one algorithm. The standard deviation over one domain is then calculated by the pooled standard deviation from each problem instance standard deviation, which similarly takes into account the number of variables for each standard deviation. Also for this table, only problem instances are considered, where both algorithms successfully completed the translation.

	Existing Inv. Syn.		New Schematic Inv. Syn.	
Domain	Mean	Std	Mean	Std
barman-mco14-strips	2.08	1.1	5.38	5.37
barman-opt11-strips	2.13	0.9	3.71	2.73
barman-opt14-strips	2.12	0.93	3.88	3.01
barman-sat11-strips	2.09	1.06	5.01	4.83
barman-sat14-strips	2.08	1.11	5.41	5.4
childsnack-opt14-strips	2.94	1.6	2.76	1.53
childsnack-sat14-strips	3.09	1.89	2.92	1.83
elevators-opt08-strips	8.84	5.54	6.4	5.09
elevators-opt11-strips	8.56	5.4	6.36	4.95
elevators-sat08-strips	17.95	8.32	9.84	9.69
elevators-sat11-strips	29.1	13.16	10.91	13.43
floortile-opt11-strips	5.73	7.19	5.73	7.19
floortile-opt14-strips	5.3	5.61	5.3	5.61
floortile-sat11-strips	5.84	8.2	5.84	8.2
floortile-sat14-strips	5.6	6.77	5.6	6.77
hiking-agl14-strips	5.2	1.56	5.2	1.56
hiking-opt14-strips	4.82	1.32	4.82	1.32
hiking-sat14-strips	5.94	1.74	5.94	1.74
nomystery-opt11-strips	23.34	43.8	23.34	43.8
nomystery-sat11-strips	25.09	47.58	25.09	47.58
parking-opt11-strips	8.29	10.16	7.86	8.92
parking-opt14-strips	8.05	9.81	7.65	8.64
pegsol-08-strips	2.94	5.35	2.0	0.0
pegsol-opt11-strips	2.94	5.41	2.0	0.0
pegsol-sat11-strips	2.94	5.41	2.0	0.0
scanalyzer-08-strips	3.0	1.0	2.0	0.0
scanalyzer-opt11-strips	3.0	1.0	2.0	0.0
sokoban-opt08-strips	5.61	11.45	5.61	11.45
sokoban-opt11-strips	4.98	10.9	4.98	10.9
sokoban-sat08-strips	5.99	12.33	5.99	12.33
sokoban-sat11-strips	6.09	12.98	6.09	12.98
transport-opt08-strips	19.71	8.43	19.71	8.43
transport-opt11-strips	13.3	4.47	13.3	4.47
transport-opt14-strips	26.91	14.88	26.91	14.88
visitall-opt11-strips	3.25	9.87	3.25	9.87
visitall-opt14-strips	3.06	14.05	3.06	14.05
woodworking-opt08-strips	2.59	0.97	2.46	0.9
woodworking-sat08-strips	2.45	0.87	2.45	0.87
woodworking-sat11-strips	2.4	0.8	2.4	0.8

Table 5.3: mean and standard deviation for possible values per variable for each domain and algorithm

Note that the values are rounded to two decimal places.
Looking at this table, the new schematic invariant synthesis algorithm tends to have a higher average of possible values per variable. Significant differences can be seen for the barman-mco14-strips domain, where the schematic invariant synthesis has a higher mean but also a higher variability.
Nevertheless, for elevators-sat08-strips and elevators-sat11-strips domain we have significantly higher means for the existing invariant synthesis. It is interesting to see that for both algorithms when the mean is high, the variablity seems also to be high. From this observation, only a small number of variables may have a

high number of possible values while the other variables still have a small number of possible values per variable. This would indicate that the algorithms both find some mutex groups with a large size while the other mutex groups have significantly fewer literals per group or no other mutex groups are found, leading to more variables with two possible values per variable. Even though we see that there are some differences in the structure of the finite domain representation of the tasks, the search times do not differ significantly for different algorithms, meaning that the quality of the found invariants and the corresponding mutex groups is similar for both algorithms.

6 Conclusion

The schematic invariant synthesis algorithm using limited grounding proposed by Rintanen [2] was implemented for this thesis as part of the Fast Downward planning system by replacing the already existing invariant synthesis by Helmert.

The implementation of the invariant synthesis algorithm and the integration of the algorithm into the planning system was a very challenging task and was the most time-consuming part of the entire project. The goal was to implement the proposed algorithm and ensuring that correct invariants are found that can be used for the generation of mutex groups, where the efficiency was not of high priority.

The implementation was evaluated by comparing the newly implemented invariant synthesis with the already existing invariant synthesis by Helmert. For the evaluation, the efficiency of the invariant synthesis, the structure of the translated task, and the impact of the found mutex groups on the search of the planning system were analyzed. The results of the evaluation showed that the implemented invariant synthesis correctly finds ground invariants and has a similar impact on the search of the planning system, compared to the existing invariant synthesis. Nevertheless, the inefficiency of the implemented algorithm, which can be observed in the evaluation, should not be ignored. Such a large difference in the time performance of the algorithms, the current implementation of the schematic invariant synthesis is cannot compete with the the invariant synthesis by Helmert.

In a future work, where the current implementation of this thesis focuses on the optimization of the algorithm by removing redundant computations and by using suitable data structures, could improve the downside of this algorithm drastically. Evaluating the result of such future work would be very interesting and could answer the question of whether the algorithm can compete with the existing invariant synthesis.

Starting this project with some knowledge of classical planning and no knowledge about invariant synthesis algorithms, the achievement of successfully implementing and integrating an algorithm described in a paper into a system that is completely new for me is a significant personal milestone. Through this project, my understanding of classical planning, invariant synthesis, propositional logic, and various other fields were improved.

All in all, I am very happy with the results of this thesis and hope that this contribution to classical planning has an impact on the research leading to future work in the field or in the optimization of the implemented algorithm.

Bibliography

[1] Helmert, M. Concise finite-domain representations for PDDL planning tasks. *Artif. Intell.*, 173(5-6):503–535 (2009).

[2] Rintanen, J. Schematic Invariants by Reduction to Ground Invariants. In *AAAI*, pages 3644–3650. AAAI Press (2017).

[3] Bacchus, F., Domshlak, C., Edelkamp, S., and Helmert, M., editors. *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011*. AAAI (2011).

[4] Tomita, E., Tanaka, A., and Takahashi, H. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42 (2006).

[5] AI Basel. downward-benchmarks. `https://github.com/aibasel/downward-benchmarks/tree/master`. Accessed: 2024-05-31.

[6] Fikes, R. and Nilsson, N. J. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *IJCAI*, pages 608–620. William Kaufmann (1971).

[7] Rintanen, J. Regression for Classical and Nondeterministic Planning. In *ECAI*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 568–572. IOS Press (2008).

[8] Edelkamp, S. and Hoffmann, J. PDDL2. 2: The language for the classical part of the 4th international planning competition. Technical report, Technical Report 195, University of Freiburg (2004).

[9] Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson (2020).

[10] University of Basel. sciCORE - Scientific Computing Core Facility. `https://scicore.unibas.ch` (n.d.). Accessed: 2024-05-31.

[11] Helmert, M. The Fast Downward Planning System. *CoRR*, abs/1109.6051 (2011).

[12] KCL Planning. VAL: Validation and Analysis of Plans. `https://github.com/KCL-Planning/VAL` (n.d.). Accessed: 2024-05-31.