POLITECNICO DI TORINO

Master of Science in Mathematical Engineering

Master's Degree Thesis

Study of Mutually Exclusive Invariants in Planning Processes

Supervisors
Prof. Fabio Fagnani
Prof. Sara Bernardini

Candidate
Joseph Stanton

Academic Year 2019-2020

Abstract

My initial task in this thesis was to understand what Automated Planners are, what Mutual Exclusive Invariants are and the sufficient conditions one could use to find Mutual Exclusive Invariants within Temporal Planners. Then using as input the Domains and Problem Instances taken from the International Competition on Automated Planning and Scheduling (ICAPS) in PDDL2.1 I implemented eight checks found within the Temporal Invariant Synthesiser (TIS) algorithm to find these Invariants. This was done using Object Oriented Programming in Python in which I constructed in an automated manner the Templates (Potential Invariants) and searched if any criteria of conditions could be met to prove the Templates Invariance. The main focus when searching for Invariants was the concept of Safety of Actions with respect to Templates. Furthermore, it was necessary to understand the concepts of Matching and Coverage in order to work on a Lifted Level. The results produced were positive and using the Invariants found by the synthesizer it is possible to generate fewer Multi-State Variables.

List of Tables

Contents

Chapter 1

Introduction

1.1 Overview of Planning Processes and PDDL

Artificial Intelligence is a broad field and only through many years of study and research can one truly have a relative grasp of what that spectrum entails. Research into this particular field began during world war two and really took off shortly after with the works of Alan Turing and John McCarthy. Before moving on to discussing classical planning a more general question that will be briefly discussed is what AI is. There are several definitions for AI however many of these only encompass certain aspects of it. A general description of AI is a machine or computer program that thinks and learns. AI divides itself into two schools of thought, these approaches lead to an agent behaving humanistically or rationally. In most cases an AI agent is often a hybrid of the two. Classical planning consists of thinking rationally and originates from logic, the purpose of which is the creation of an agent which operates according to the information it receives. This information is interprated according to the logical notation which has been used to construct the agent, the agent then operates accordingly to reach a certain goal if feasible or the best outcome that is possible. Humans do not need a plan when operating, only when dealing with long and complex tasks do humans need to explicitly plan. AI planning consists of the study and deliberation of this process.

The main objective of AI is the construction of *intelligent entities* which can develop in an automated manner a set of actions from which one can proceed to resolve a problem. This is done through the creation of a controller which can determine which actions to take in order to find a solution. As the agent is the one choosing the actions it must know the effects of the actions it takes and must be able to observe if only partially the "world" in which it functions, an unobservable process cannot be planned. A combination of three approaches can be taken in the development of the controller, using machine learning techniques, logical programming and the method which is the focus of this thesis which involves the use of planning processes. For the AI agent to be able to create a plan the "world" in which it operates must first be defined, a domain of the problem is created. A problem instance is then defined outlining the specific parameters (number of objects, state variables, etc...). Once the bot is aware of the specific problem it is then possible to attempt to solve it by searching through the state space. The set of actions available to it which are defined in the domain are used to reach the particular state the agent would like to move in to. Heuristic algorithms such as Dijkstra's or A^* algorithms are used to attempt to find a solution if one is present.

One of the faults with using planning processes involves scalability, the number of state variables created for even simple problems is astronomical, an example of this would be the Wumpus problem which even with a small grid and a limited number of holes and monsters can produce a surprising number of state variables.

Through the use of Invariants (a property of the environment which is always satisfied), the number of state variables can be reduced. The primary objective of this thesis will focus on theory related to Mutually Exclusive Invariants and the implementation of the algorithm TIS (Temporal Invariant Synthesis) which uses a set of checks and controls to find them. Since working from a grounded level is inefficient this algorithm will be implemented from a lifted level in which predicates are only partially grounded (not grounded with constants) in order to carry out a computationally efficient search. This will be discussed in further detail later on, beforehand a brief explanation of how planning processes are defined is provided along with a quick overview of the main programming language PDDL **(Planning Domain Definition Language)** used to create them.

Planning processes were originally written in STRIPS (Stanford Institue Research Problem Solver), a first order predicate language which defines a domain based on a set of atoms with variables or objects of a specified type, these atoms define the relationships between the objects. [2] A planning process can be defined as a *basic state model* consisting of:

- A finite and discrete space S
- A *known initial state* $s_0 \in S$
- A non-empty set $S_G \subseteq S$ of goal states
- Actions $A(s) \subseteq A$ applicable in each state $s \in S$
- $f(a, s)$ is the *deterministic transition function* where $s' = f(a, s)$ is the state that follows s after doing action $a \in A(s)$
- $c(a, s)$ is a *positive cost* for doing action a in a state s

A sequence of applicable actions $A = (a_0, a_1, ..., a_n)$ is known as a *plan*, these actions generate a state sequence $S = (s_0, s_1, ..., s_{n+1})$ where s_{n+1} is a goal state.
The formal definition for a planning instance is also provided below:

Definition 1.1.1. Simple Planning Instance A simple planning instance is defined as a pair

$$I = (D, P)$$

where $D = (F, R, A, \text{arity})$ is a tuple consisting of function symbols, relation symbols, actions, and an arity mapping of all these symbols onto their respective arities. $P = (O, Init, G)$ is a tuple consisting of the objects in the domain, the initial state specification and the goal state specification.

The Atm atoms of the planning instance are the expressions formed by applying the relation symbols in R onto the objects in O.

$Init$ consists of a set of literals formed from the atoms in Atm as well as a set of propositions asserting initial values for a subset of the primitive numeric expressions in the domain. These assertions each assign a single primitive numeric expression of the domain. These together form an initial state (or set of initial states) from which the particular problem instance must be resolved.

PDDL is standard language used to format planning processes. An action-centred language, PDDL was created based on the STRIPS formulation of planning problems, it has a syntax similar to Lisp. It can make use of numerical fluents as well as predicates when defining a domain and its problem instance. It is important to note that PDDL and planning processes make a distinction between the domain of a problem and its problem instance. This is important as it allows for testing planners with respect to other problem instances in order to measure their efficiency.
Planning processes did not take into account for time originally in PDDL and the sets of actions created were instantaneous A^i only. [1] An instantaneous action can be formulated with the following sets:

- $V_\alpha \subseteq V$, Schema Variables
- Pre_α^+, Positive Preconditions
- Pre_α^-, Negative Preconditions
- Eff_α^+, Add Effects
- Eff_α^-, Delete Effects

These preconditions and effects are sets of formulas l of the form: $(\forall v_1, ..., v_k : q)$ where:

- q is an atomic formula: $q = r(v'_1, ..., v'_n)$ with $r \in R$ and $arity(r) = n \geq k$
- $\{v_1, ..., v_n\} \subseteq \{v'_1, ..., v'_n\} \subseteq V$ are the quantified variables in l
- $\{v'_1, ..., v'_n\} \setminus \{v_1, ..., v_n\} \subseteq V_\alpha$ are the schemas variables in l

For notation we refer to the preconditions and effects of an action as $\text{Pre}_a = \text{Pre}_a^+ \cup \text{Pre}_a^-$ and $\text{Eff}_a = \text{Eff}_a^+ \cup \text{Eff}_a{}^-$. The action sets GA^i and GA^d refer to the set of instantaneous and durative ground actions. It is possible to execute an action a in state s if $\text{Pre}_a^+ \subseteq s$ and $\text{Pre}_a^- \cap s = \emptyset$ where a is an action that has been mapped using a grounding function to correpond with the problems objects.

With the introduction of PDDL2.1 further changes were made allowing for the insertion of durative actions, time and plan metrics. [5]

```

  (define (domain satellite)
  (:requirements :strips :equality :typing :durative-actions)
(:types satellite direction instrument mode)
 (:predicates
               (on_board ?i - instrument ?s - satellite)
         (supports ?i - instrument ?m - mode)
         (pointing ?s - satellite ?d - direction)
         (power_avail ?s - satellite)
         (power_on ?i - instrument)
         (calibrated ?i - instrument)
         (have_image ?d - direction ?m - mode)
         (calibration_target ?i - instrument ?d - direction))

```

Listing 1.1. Example of PDDL2.1 Domain Format.

Durative actions allowed for the introduction of problems which require sequential planning, durative actions can be sequential or continuous.

Definition 1.1.2. [6] (Plans). A plan P, with durative actions for a planning instance, I, consists of a finite collection of times actions which are pairs, each either of the form (t, a), where t is a rational-valued time and a is a simple action name - an action schema name together with the constants instantiating the arguments of the schema, or of the form $(t, a[t'])$, where t is a rational-valued time, a is a durative action name and t' is a non-negative rational-valued duration.

[1] The *simple plan* π induced by Π is the set of instantaneous timed actions such that:

1. $(t, a) \in \pi$ for each $(t, a) \in \Pi$ where a is an action.

2. $(t, a^{\text{st}}) \in \pi$ and $(t + t', a^{end}) \in \pi$ for all $(t, Da[t']) \in \Pi$, where Da is a durative action.

3. $((t_i + t_{i+1})/2, a^{\text{inv}}) \in \pi$ for each $(t, Da[t']) \in \Pi$ and for each i such that $t \leq t_i < (t + t')$, where t_i and t_{i+1} are in the time happening sequence of Π.

For each durative action $(t, Da[t']) \in \Pi$, the simple plan π contains the instantaneous timed actions (t, a^{st}), $(t + t', a^{end})$ and $((t_i + t_{i+1})/2, a^{inv})$. A plan Π and its corresponding induced plan π is admissible if concurrent instantaneous actions are non-interfering between each other and actions happening inside a durative action $Da = (a^{st}, a^{inv}, a^{end})$ are non interfering with the action a^{inv}. More precisely if

- $(t, a), (t, b) \in \pi$ imply that a and b are non-interfering.
- $(t, Da[t']) \in \Pi$ and $(s, b) \in \pi$ for some time $s \in (t, t + t')$ imply that a^{inv} and b are non-interfering.

In the PDDL2.1 domain instantaneous and durative actions are found in a set of actions A^a. The durative action can be broken down into a schema of three instantaneous actions $D\alpha = (\alpha^{st}, \alpha^{inv}, \alpha^{end})$, the start action, the invariant action and the end action. The schemas share a common set of variables with $V_{D\alpha} = V_{\alpha^{st}} = V_{\alpha^{inv}} = V_{\alpha^{end}}$. Depending on the annotation of the durative action the effects can either be immediate (in which case they are contained in the effects within the start action α^{st}) or delayed (in which case they can be found in the α^{end}) containing preconditions and effects. The invariant action α^{inv} never has any effects ($\text{Eff}_{\alpha^{inv}} = \emptyset$).

In order to obtain the sets of actions GA^i and GA^d we manipulate the action schemas in the set GA^a. The flattening operation allows for us to eliminate conditional effects and existentially quantified formulae. Once we have obtained a flattened schema α the formulas found in the conditions and effects are normalised. After applying these two operations we are left with a set of formulas l of the form $\forall v_1, ..., v_k : q$, q being the atomic formula. The notation Pre^+_α and Eff^+_α indicate the set of the positive formulas that are positive in α and Pre^-_α and Eff^-_α indicate the set of the positive formulas that are negative in α.

α^{str}	α^{inv}	α^{end}
$\text{Pre}^+_{\alpha^{\text{str}}} = \text{Pre}^{+\text{str}}_{D\alpha}$	$\text{Pre}^+_{\alpha^{\text{inv}}} = \text{Pre}^{+\text{inv}}_{D\alpha}$	$\text{Pre}^+_{\alpha^{\text{end}}} = \text{Pre}^{+\text{end}}_{D\alpha}$
$\text{Pre}^-_{\alpha^{\text{str}}} = \text{Pre}^{-\text{str}}_{D\alpha}$	$\text{Pre}^-_{\alpha^{\text{inv}}} = \text{Pre}^{-\text{inv}}_{D\alpha}$	$\text{Pre}^-_{\alpha^{\text{end}}} = \text{Pre}^{-\text{end}}_{D\alpha}$
$\text{Eff}^+_{\alpha^{\text{str}}} = \text{Eff}^{+\text{str}}_{D\alpha}$	$\text{Eff}^+_{\alpha^{\text{inv}}} = \emptyset$	$\text{Pre}^+_{\alpha^{\text{end}}} = \text{Eff}^{+\text{end}}_{D\alpha}$
$\text{Eff}^-_{\alpha^{\text{str}}} = \text{Eff}^{-\text{str}}_{D\alpha}$	$\text{Eff}^-_{\alpha^{\text{inv}}} = \emptyset$	$\text{Eff}^-_{\alpha^{\text{end}}} = \text{Eff}^{-\text{end}}_{D\alpha}$

Table 1.1: Durative Action Schema in its Instantaneous Action Schemas form.

Another particular characteristic of durative actions are their overall conditions, these conditions must be satisfied while the action is being carried out and sometimes must also be satisfied at the end of the action. This distinction must be held into account when looking at the feasibility of carrying out actions. The use of time in planning processes whether continuous or sequential leads to the introduction of concurrent planning. In layman terms it is possible to carry out actions simultaneuosly when feasible. This is of particular interest as when carrying out *Invariant Synthesis* the templates (or invariant candidates) must be checked with respect to the actions within the domain. These checks involve analysing the properties of an action, in the case of durative actions in particular instances it is necessary to carry out cross checking of action couples to see if they are "pairwise relevant non-overlapping" or "relevant right isolated". These last checks are relevant as not only do they allow us to search for invariants in which weaker conditions apply, they also be useful in the debugging of actions written within the domain.

Chapter 2

Overview of Invariant Synthesis

2.1 Invariants

[1] In the PDDL2.1 language, an *invariant* of a planning process is a property of the world states such that when it is satisfied in the initial state $Init$, it is satisfied in all the reachable states S_Γ.

Definition 2.1.1. [1] (*Mutual Exclusion Invariant*). A set of ground atoms $Z \in S$ is a mutual exclusion invariant set when, if at most one element of Z is true in the initial state, then at most one element of Z is true in any reachable state, namely:

$$|Z \cap \text{Init}| \ \leq 1 \ \Rightarrow \ |Z \cap S| \ \leq 1, \ \forall s \in S_\Gamma$$

A basic example of this can be shown looking at the Drivelog problem in which we have the template:

$$\{\text{empty}(v_1), \ \text{driving}(d, v_2)\}$$

where d (driver) is the counted variable and v_1 and v_2 are fixed, this invariant states that no more than one driver at any point can drive a particular truck.
As stated earlier in order to identify a given Invariant it is necessary to work using a template applying checks through the template onto a set of action schemas. In order to handle complexity quite often we use invariant templates to indicate and analyse several invariant sets. Before we can go into further detail about these templates it is necessary to introduce a few preliminary definitions which allow for their creation.

Definition 2.1.2. [1] (Template) Any Template τ can be referred to as a pair (C, F_C) where:

- C is a set of components with each *component* c being a tuple of the form $< r/k, p >$ where r is the relation symbol in R of arity k = arity(r), and $p \in 0, ..., k$ being the *counted* variable.

- F_C is an *admissible partition* of F_C.

When there is only one possible partition $F_C = \{\mathrm{F}_C\}$ this is known as a trivial partition, we simply write that $\tau = (C)$.

Definition 2.1.3. [1] (Admissible Partition). Given a set of components C and corresponding set of fixed variables F_C, an *admissible partition* of F_C is a partition $\mathcal{F}_C = G_1, ..., G_s$ such that $|G_j \cap F_c| = 1$ for each $c \in C$.

If two elements (c_1, i) and (c_2, j) of F_C belong to the same set of the partition $\mathcal{F}_C$, we use the notation $(c_1, i) \sim (c_2, j)$.

An example of a trivial partition can be found in the FloorTile problem, if we look at the Template:

$$\tau = \{(c_1{,}1), (c_2, 0), (c_3, 0)\}$$

where $c_1 = < RobotAt/2{,}0 >$, $c_2 = < Painted/2, 1 >$, $c_3 = < Clear/1{,}1 >$. The counted variable for c_1 is r (robot), for c_2 is c (colour) and for c_3 there is no fixed counted variable.

It is important to note that by defintion any admissible partition must be done such that every component contains the same number of fixed variables. If we look at the following components:

$$c_1 =< r/3, 0 > \quad c_2 =< l/3, 1 > \quad c_3 =< q/2, 2 >$$

with the corresponding variables in the relations $r(x, y, z)$, $l(a, b, c)$, $q(u, v)$ then we find that the following partitions can be made:

$$\{\{y, a, u\}, \{z, c, v\}\}, \{\{y, c, u\}, \{z, a, v\}\}, \{\{y, a, v\}, \{z, c, u\}\}, \{\{y, c, v\}, \{z, a, u\}\}$$

In order to carry out any checks between actions and any template it is necessary to introduce the concept of *Template Instance Weight*.

Definition 2.1.4. (Template Instance Weight). Let γ be an instance of template τ with instantiation $\gamma(\tau)$. Then the weight $w(\gamma, s)$ of γ in state s is the number of ground atoms of its instantation true in s:

$$w(\gamma, s) = |\gamma(\tau) \cap s|$$

With the introduction of Template Instance Weight it is possible to discuss the concept of safety (strong or otherwise). A candidate Template τ is an invariant if it meets certain necessary and sufficient conditions which can be checked using this weight, an example of a necessary condition being that all instantaneous actions A_i be strongly safe.

Definition 2.1.5. A set of Actions A is said to be strongly γ-safe if, for each $s \in S_A$ where $w(\gamma, s) \leq 1$, the successor state $s^{'} = \xi(s, A)$ also satisfies $w(\gamma, s^{'}) \leq 1$.

Definition 2.1.6. [1] For a template τ, a set of actions $A \subseteq GA$ is strongly safe if it is strongly γ-safe for every instance γ.

As a consequence of the previous definition we find the following:

Corollary 2.1.1. *[1] For a template τ, τ is an invariant if for each $a \in GA$, a is strongly safe.*

Before we proceed with the theory related to the classification of actions since this thesis mainly looks at checking for mutually exclusive invariants on a lifted level it important to discuss the creation of the classes via the *matching* operation. As stated before working on a grounded level is not practical when searching for invariants as the complexity of our problem would be astronomical as shown in the beginning due to the massive state space created. To avoid this we analyse our state space on a *lifted* level. Given an action schema, if one istantiation $a^* = gr^*(\alpha)$ satisfies P then all istantiations $a = gr(\alpha)$ satisfy P and the property P of ground actions is said to be *liftable*. By carrying out matching we can couple an action schema to a template allowing us to search if a ground formula is present in $\gamma(\tau)$.

Definition 2.1.7. [1] **(Matching)**. Consider a template $\tau = (C, F_C)$ and an action schema $\alpha \in A$. A formula l that appears in α is said to match τ via the template's component $c =< r/k, p >\in C$ if:

i. $\text{Rel}[l] = < r/k >$; and

ii. if l is universally quantified $\text{VarQ}[l]=\{p\}$

Given two formula's l and l' in α, we say that they are τ-coupled (and we write $l \sim_\tau l'$) if:

1. l and l' individually match τ via the components c and c'; and
2. if $(c, i) \sim_{F_C}$, $\text{Var}[l, i]=\text{Var}[l', j]$

In other words two components within a template τ are matching if their fixed variables are the same and are allocated to the same τ. - class L.

Proposition 2.1.1. For a template $\tau = (C, F_C)$ and an action schema $\alpha, \sim_\tau$ is an equivalence relation.

Definition 2.1.8. [1] (**τ-class**). For a template $\tau = (C, F_C)$ and an action schema α, an equivalence class of literals with respect to $\sim \tau$ is called a τ - class.

Remark 2.1.1. [1] Given a formula l in the action schema α that matches the template τ via component $c =< r/k, p >$. The potential structures of l are shown below:

$$p = k, l = r(v_0, ..., v_{k-1}),\ \forall i\ v_i \in V_\alpha$$

$$p < k, l = r(v_0, ..., v_{k-1}),\ \forall i\ v_i \in V_\alpha$$

$$p < k, l = (\forall v_p : r(v_0, ..., v_{k-1})),\ \forall i \neq p\ v_i \in V_\alpha$$

Given two formulas l_1 and l_2 in the action schema α that match the template τ via the components $c^1 =< r^1/k^1, p^1 >$ and $c^2 =< r^2/k^2, p^2 >$. The τ - coupling condition $l^1 \sim_\tau l^2$ is equivalent to having any pair of fixed vairables meet the following condition:

$$(c^1, j) \sim_{F_C} (c^2, h) \Rightarrow v^1_j = v^2_h$$

Definition 2.1.9. [1] (**Pure action schemas**). Considering a template τ, an action schema α and a τ-class L of formulas in α, we define α_L to be the action schema where we only consider formulas belonging to L. More precisely α_L is the action schema such that

$$\text{Pre}^{\pm}_{\alpha_L} = \text{Pre}^{\pm}_{\alpha} \cap L \quad \text{Eff}^{\pm}_{\alpha_L} = \text{Eff}^{\pm}_{\alpha} \cap L$$

α_L is referred to as an action schema.

One more concept we need to introduce before analysing and classifying pure action schemas α_L on a lifted level is the concept of *coverage*. This is only used in the case where we are dealing with *relevant weightless* actions however quite often this is found to be the case. When analysing coverage we must first fix an action schema α and a τ - class L of its formulas. We are introducing the concept of weight at a lifted level. This is at a level of formulas in L that lets us distinguish between simple and universally quantified formulas. In particular if $l \in L$, we define $w_l = 1$ if l is simple, and $w_l = w$ if l is universally quantified where $w = |O|$. For a subset $A \subseteq L$, we define $w(A) = \sum_{l \in A} w_l$. When all formulas in L are simple $w(\cdot)$ is just simple cardinality. Moreover if c is a component of τ, then w_c is equal to one if c does not have a counted variable and w in the instance that c does have a counted variable.

Definition 2.1.10. [1] (Coverage) Consider a component $c \in \tau$. We let L_c be a subset of formulas in L that match τ through the component c. A subset of formulas $M \subseteq L$ is said to *cover* the component c if $w(M \cap L_c) = w_c$. M is said to *cover* τ if M covers every component $c \in \tau$.

Remark 2.1.2. [1] Given a component $c \in \tau$, all the possible ground atoms generated by c are in $gr(M)$ if and only if M covers c. In particular, $\gamma(\tau) = gr(M)$ if and only if M covers τ.

In the majority of cases components do have counted variables and since most actions more often than not do not this leads to a lack of coverage but there will be more detail on this particular case later. Returning to the theory related safety looking at the previous corollary this is found to be a sufficient condition such that a template τ is an Invariant. Instantaneous actions must satisfy this condition such that an invariant exists however as we shall see later on a template can be an invariant even if not all actions are strongly safe. Before we move on to discuss other forms of safety we should first look at the characterisation of actions with respect to strong safety. Following the structure of preconditions and effects, instantaneous actions can be classified into four categories. After categorising them we will then show how each class is linked to strong safety. It is important to note that all theory related to safety and the checks within TIS shall be explained from a *lifted* level as this is how the checks were implemented. The following definitions are formally analogous to the definitions related to the classification of istantiated actions on a grounded level, preconditions and effects of a_γ are simply replaced with the respective ones found in α_L.

Definition 2.1.11. [1] (Classification of pure action schemas). A pure action schema α_L is:

- *unreachable* for τ if $w(\text{Pre}^+_{\alpha_L}) \geq 2$
- *heavy* for τ if $w(\text{Pre}^+_{\alpha_L}) \leq 1$ and $w(\text{Eff}^+_{\alpha_L}) \geq 2$
- *irrelevant* for τ if $w(\text{Pre}^+_{\alpha_L}) \leq 1$ and $w(\text{Eff}^+_{\alpha_L}) = 0$
- *relevant* for τ if $w(\text{Pre}^+_{\alpha_L}) \leq 1$ and $w(\text{Eff}^+_{\alpha_L}) = 1$

Definition 2.1.12. [1] (Classification of relevant action schemas). The pure relevant action schema α_L is *weighty* when it has a single relevant precondition: $w(Pre^+_{\alpha_L}) = 1$. A is *weightless* if $w(Pre^+_{\alpha_L}) = 0$.

A weighty action schemas α_L is either:

- *balanced* for τ if $Pre^+_{\alpha_L} \subseteq Eff^+_{\alpha_L} \cup Eff^-_{\alpha_L}$
- *unbalanced* for τ if $Pre^+_{\alpha_L} \cap (Eff^+_{\alpha_L} \cup Eff^-_{\alpha_L})$

A weightless action α_L is either:

- *bounded* for τ if L covers τ

- *unbounded* for τ if L does not cover τ

Following on from before in order to do be able to work on a lifted it is important to note the following corollary:

Corollary 2.1.2. *[1] Strong safety is a liftable property. Moreover an action schema α is strongly safe if and only if, for every τ - class of formulas L of α, α_L is unreachable, irrelevant, balanced or bounded.*

Two examples will now be introduced to help clarify the theory provided in the previous section, the first example illustrates the concept of strong safety in action sequences (on a grounded level). Strong safety in an action sequence does not mean that all the actions within that sequence are strongly safe. Usually in order to have any form of safety only the action $a^s t_{*L}$ need be (along with a few other conditions which may vary). The second example instead will look at matching and the classification of actions on a lifted level.

Example 2.1.1. (Strong Safety in Action Sequences) Consider a template τ obtained from the domain *CityCar*, in this particular case for simplicity we will look at this from a grounded level. Assume that we have an instance $\gamma(\text{AtGarage}(g, xy-final, \text{Starting}(m, g), \text{Clear}(xy-final), \text{AtCarJun}(m, xy-final), \text{Arrived}(m, xy-final))$ which corresponds with the action sequence A = (*carStart, carArrived*) where *carStart* and *carArrived* are actions with the following composition:

$$\text{Pre}^+_{\textit{carStart}} = \{\text{AtGarage}(g, xy - final), \text{Starting}(m, g), \text{Clear}(xy - final)\},$$

$$\text{Eff}^+_{\textit{carStart}} = \{\text{AtCarJun}(m, xy - final)\}, \text{Eff}^-_{\textit{carStart}} = \{\text{Clear}(xy - final), \text{Starting}(m, g)\}$$

and

$$\text{Pre}^+_{\textit{carArrived}} = \{\text{AtCarJun}(m, xy - final)\},$$

$$\text{Eff}^+_{\textit{carArrived}} = \{\text{Clear}(xy - final), \text{Arrived}(m, xy - final)\}$$

Note how *carStart* is γ - unreachable and therefore is not strongly γ - safe while *carArrived* is γ - heavy and not strongly γ - safe. The action sequence overall is γ - unreachable and as a result strongly γ - safe. It is important to understand these concepts when dealing with action sequences as when searching for safety as we shall see later on by changing the action sequence (by adding *irrelevant* actions) it is possible to render an action sequence unsafe.

Example 2.1.2. (Matching with L - classes). Looking at the floortile problem when looking at the template τ_{ft} with the following (and only) admissible partition $F_c = \{(c_1, 1), (c_2, 0), (c_3, 0)\}$ with the components having these respective relations:

$$c_1 =< robotAt/2, 0 >, \ c_2 =< painted/2, 1 >, \ c_3 =< clear/1, 1 >$$

Given the following when analysing the durative action *Up* we obtain the L - classes: $L_0 = \{robotAt(r, x), clear(x)\}$ and $L_1 = \{robotAt(r, y), clear(y)\}$. Since the component *painted* is not found amongst the action schema preconditions and effects it is excluded from the classes. The original action schema we have from the action is:

α	Up^{Str}	Up^{Inv}	Up^{End}
Pre^+_α	$\{RobotAt(r, x), Clear(y)\}$	$\{up(y, x)\}$	$\emptyset$
Pre^-_α	$\emptyset$	$\emptyset$	$\emptyset$
Eff^+_α	$\emptyset$	$\emptyset$	$\{RobotAt(r, y), Clear(x)\}$
Eff^-_α	$\{RobotAt(r, x), Clear(y)\}$	$\emptyset$	$\emptyset$

Table 2.1: Durative Action Schema **Up** seen as a triple of instantaneous action schemas.

After obtaining the classes and carrying out a filtering of the action with respect to each L - class we obtain the following action schemas:

α	$\text{Up}^{\text{St}}_{L0}$	$\text{Up}^{\text{Inv}}_{L0}$	$\text{Up}^{\text{End}}_{L0}$
Pre^+_α	$\{RobotAt(r, x)\}$	$\emptyset$	$\emptyset$
Pre^-_α	$\emptyset$	$\emptyset$	$\emptyset$
Eff^+_α	$\emptyset$	$\emptyset$	$\{Clear(x)\}$
Eff^-_α	$\{RobotAt(r, x)\}$	$\emptyset$	$\emptyset$

α	$\text{Up}^{\text{St}}_{L1}$	$\text{Up}^{\text{Inv}}_{L1}$	$\text{Up}^{\text{End}}_{L1}$
Pre^+_α	$\{Clear(y)\}$	$\emptyset$	$\emptyset$
Pre^-_α	$\emptyset$	$\emptyset$	$\emptyset$
Eff^+_α	$\emptyset$	$\emptyset$	$\{RobotAt(r, y)\}$
Eff^-_α	$\{Clear(y)\}$	$\emptyset$	$\emptyset$

Looking at the two starting pure action schemas Up^{st}_{L0} and Up^{st}_{L1} it is quite clear that they are both irrelevant. When analysing Up^{end}_{L0} and Up^{end}_{L1} instead both are found to be unbounded and are therefore not strongly safe with respect to τ however that does not mean they could not be *weakly* safe.

Going into further detail about the bounded case it can be shown that a bounded action set is even safer than the balanced case. This can illustrated by taking into account all the predicates present within the bounded actions. Given the relevant predicate p, such that $\text{Eff}^+_{A_\gamma} = \{p\}$, since A is bounded the rest of the istantiation $\gamma(\tau) \setminus \{p\}$ is accessed negatively such that $\gamma(\tau) = \text{Pre}_{A_\gamma} \cup \text{Eff}_{A_\gamma}$. Since A is weightless by definition $|\text{Pre}^+_{A_\gamma}| = 0$ and given $\text{Eff}^+_{A_\gamma} = \{p\}$ we find that $\gamma(\tau) \setminus \{p\} = \text{Pre}^-_{A_\gamma} \cup \text{Eff}^-_{A_\gamma}$ as a result this shows that the weight found after executing a bounded set will be exactly one since all the possible predicates other than p are contained within the negative parts of the action.

The focus of this thesis will focus mainly on analysing all forms of *safety* in durative actions. In order to do this it is necessary to understand what other forms of safety exist and lift the last remnants of notation that are necessary to search for safety. A less strong form of safety is individual safety, the formal definition is the following:

Definition 2.1.13. [1] **Individually Safe Actions**: A sequence of action sets $\mathbf{A} = (A^1, A^2, ..., A^n)$ is individually γ-safe if for every sequence of states $(s^0, ..., s^n) \in \mathbf{S_A}$ we have that

$$w(\gamma, s^0) \leq 1 \implies w(\gamma, s^i) \leq 1 \quad \forall i = 1, ..., n$$

The following proposition shows that it is possible for templates to be invariants when simple safety exists despite this being a relatively weak property.

Proposition 2.1.2. [1] Given a template τ, assuming that for every executable simple plan π its happening sequence A_π is individually γ - safe with respect to every instance γ. Then τ is said to be an invariant.

The consequences due to the subtle differences between the case of individual safety and strong safety are difficult to understand however using an example we can understand why individual safety is a weak property and not robust enough to prove the existence of invariants. Subsequences of individually safe sequences may not be individually safe, this is shown in the following example.

Example 2.1.3. [1] Consider an action set $A = (a^1, a^2)$ and a set of states $S_A = \{(s^0, s^1, s^2) | q \notin s^0 \; s^1 = s^0 \cup q', s^2 = s^1\}$ which are compatible with A. Since a^2 by hypothesis is applicable in s^1 and $s^1 = s^0 \cup q'$ therefore $q \notin s^0$ as a result. A is individually γ - safe since $w(\gamma, s^i) \leq 1$ for every state $s^i \in S_A$. Now looking at a subsequence of A we find that $A^1_1 = (a^1)$ is not individually γ - safe as a result of a^1 being unbounded and therefore not strongly γ - safe.
We will see how by adding a γ - irrelevant action it is possible to cause a failure in γ - safety of the sequence A. Now consider an action set $\tilde{A} = (a^1, b, a^2)$ with a set of states compatible with $\tilde{A} : S_{\tilde{A}} = \{(s^0, s^1, s^2, s^3) | s^1 = s^0 \cup \{q'\}, s^2 = s^1 \setminus \{q\}, s^3 = s^2\}$. Notice how q can be in s^0 in this case as the action b guarantees the applicability of a^2, if $q \in s^0$ as a^1 adds q' to s^0 and $w(\gamma, s^1) = 2$ meaning the new sequence is not individually γ - safe.

The difference is subtle and yet crucial, in the case of individual safety, only for a particular predefined sequence (of states) does safety apply where as in the case of strong safety no matter which order of actions occurs safety will always be present no matter the set of states. Individual γ-safety is a weak property as even if a sequence of actions is individually safe its subsequences may not be. Therefore alone it is not sufficient to prove the invariance of a template.

Definition 2.1.14. [1] **Executable and Reachable Actions**. The sequence $\mathbf{A} = (A^1, A^2, ..., A^n)$ is called:

- executable if $\mathbf{S_A} \neq \emptyset$
- γ-(un)reachable if $\mathbf{S_A}(\gamma) \neq \emptyset$ ($\mathbf{S_A}(\gamma) = \emptyset$)

The previous definition is useful when a template is istantiated and therefore only good for a grounded level. The following proposition is of far more significance as it is used in the algorithm when analysing the executability of durative actions $D\alpha$. [1] Looking at the *postconditions* Γ^+ and Γ^- of an *auxillary action* α_*, where:

$$\Gamma^+_\alpha = (\text{Pre}^+_\alpha \setminus \text{Eff}_\alpha{}^-) \cup \text{Eff}_\alpha{}^+, \quad \Gamma^-_\alpha = (\text{Pre}^-_\alpha \setminus \text{Eff}_\alpha{}^+) \cup \text{Eff}_\alpha{}^-$$

and the auxiliary action $D\alpha = (\alpha^{st}_*, \alpha^{end}_*)$ is a sequence of action schemas in which the action α^{inv} has been incorporated into the action schemas α^{st} and α^{end} in a manner such that we obtain the following:

$$\begin{aligned} \text{Eff}^\pm_{\alpha^{st}_*} &= \text{Eff}^\pm_{\alpha^{st}}, & \text{Pre}^\pm_{\alpha^{st}_*} &= \text{Pre}^\pm_{\alpha^{st}} \cup (\text{Pre}^\pm_{\alpha^{inv}} \setminus \text{Eff}^\pm_{\alpha^{st}}) \\ \text{Eff}^\pm_{\alpha^{end}_*} &= \text{Eff}^\pm_{\alpha^{end}}, & \text{Pre}^\pm_{\alpha^{end}_*} &= \text{Pre}^\pm_{\alpha^{end}} \cup \text{Pre}^\pm_{\alpha^{inv}} \end{aligned}$$

Proposition 2.1.3. [1] Executability of auxillary durative actions is a liftable property. Precisely, $D\alpha_*$ is executable if and only if:

$$\Gamma^+_{\alpha^{st}_*} \cap \text{Pre}^-_{\alpha^{\text{end}}_*} = \Gamma^-_{\alpha^{st}_*} \cap \text{Pre}^+_{\alpha^{\text{end}}_*} = \emptyset$$

where

Just as it is done on a grounded level on a lifted level proving that an action is executable is a sufficient condition to prove that an action $D\alpha_{*L}$ is reachable.

Definition 2.1.15. Reachable Action Schemas. $D\alpha_{*L}$ is said to be *reachable* if it is executable and

$$w(\text{Pre}^+_{\alpha^{st}_*} \cup (\text{Pre}^+_{\alpha^{end}_*} \setminus \text{Eff}^+_{\alpha^{st}_*}) \leq 1$$

Definition 2.1.16. [1] **Safe Actions**. A sequence of action sets $A = (A^1, A^2, ..., A^n)$ is γ - safe if it is executable and the subsequences of A^k_1 are individually γ - safe for every $k = 1, ..., n$.

Remark 2.1.3. [1] Note that if $A = (A^1, A^2, ..., A^n)$ is γ - safe, the first action set A^1 must necessarily be strongly γ - safe. In the other direction, note that if A is executable and every A^j for $j = 1, ..., n$ is strongly γ - safe then A is γ - safe.

This leads to the following definition in the classification of possible types of safety in relation to action sequences, there are only two types, strongly and weakly safe actions.

Definition 2.1.17. [1] **Strongly and Weakly Safe actions**. A sequence of action sets $A = (A^1, A^2, ..., A^n)$ is:

- Strongly γ - safe if it is executable and every A^j for $j = 1, ..., n$ is strongly γ - safe.
- Weakly γ - safe if it is γ - safe but not strongly γ - safe.

The following theorem ensures that the concept of safe sequences is robust to the insertion of irrelevant actions.

Theorem 2.1.1. *[1] Consider a γ - safe sequence $A = (A_1, A_2)$ and γ - irrelevant action sets $B^1, B^2, ..., B^n$. Then the sequence $\tilde{A} = (A^1, B^1, B^2, ..., B^n, A^2)$ is either non-executable or γ-safe.*

Definition 2.1.18. [1] **Safe Durative Action Schemas.** A durative action $D\alpha_{*L}$ is is said to be *weakly safe* of *type(x)* where $x \in \{a, b, c, d\}$ if the following conditions are satisfied:

1. $D\alpha_{*L}$ is reachable
2. α^{st}_{*L} is strongly safe
3. α^{end}_{*L} is unbounded
4. $D\alpha_{*L}$ satisfies one of the following conditions:
   (a) α^{st}_{*L} irrelevant, $w(\mathrm{Pre}^+_{\alpha^{st}_{*L}}) = 1, \mathrm{Pre}^+_{\alpha^{st}_{*L}} \subseteq \mathrm{Eff}^-_{\alpha^{st}_{*L}}$
   (b) α^{st}_{*L} irrelevant, $w(\mathrm{Pre}^+_{\alpha^{st}_{*L}}) = 1, \mathrm{Pre}^+_{\alpha^{st}_{*L}} \not\subseteq \mathrm{Eff}^-_{\alpha^{st}_{*L}}, \mathrm{Pre}^+_{\alpha^{st}_{*L}} \subseteq \mathrm{Eff}_{\alpha^{end}_{L}}$
   (c) α^{st}_{*L} irrelevant, $w(\mathrm{Pre}^+_{\alpha^{st}_{*L}}) = 0, \mathrm{Pre}^+_{\alpha^{st}_{*L}} \cup \mathrm{Eff}^-_{\alpha^{st}_{*L}} \cup \mathrm{Eff}_{\alpha^{end}_{L}}$ covers τ
   (d) α^{st}_{*L} relevant, $\mathrm{Eff}^+_{\alpha^{st}_{L}} \subseteq \mathrm{Eff}_{\alpha^{end}_{L}}$

Elaborating on the rational behind the previous definition, looking at condition (2) it is quite logical why the only case in which weak safety can be found is when α^{end}_{*L} is unbounded, in the event that:

- α^{end}_{*L} were irrelevant or bounded then due to α^{st}_{*L} also being strongly safe the durative action $D\alpha_*$ would be strongly safe.

- α_{*L}^{end} were heavy or unbalanced then $D\alpha_*$ could not be safe. This proposition

Corollary 2.1.3. *[1] Safety for durative auxillary actions is a liftable property. $Da_* = gr(D\alpha_{*L})$ is safe if and only if:*

- *$D\alpha_*$ is executable*
- *For every τ - class L of formulas in $D\alpha$, one of the following conditions hold:*
  - *(a) $D\alpha_{*L}$ is strongly safe*
  - *(b) α_{*L}^{st} is strongly safe and $D\alpha_{*L}$ is unreachable*
  - *(c) $D\alpha_{*L}$ is weakly safe of type(x) where $x \in \{a, b, c, d\}$*

Now that we have explained all of the necessary concepts related to safety it is finally possible to show a concrete sufficient condition such that a template τ could be an invariant. The following corollary was the basis for the majority of the checks that I implemented within the TIS algorithm, all the Invariants that were found.

Corollary 2.1.4. *[1] Given a template τ, if the set of instantaneous action schemas A^i, and the set of A^d durative action schemas satisfy the following properties:*

1. *For every $(D\alpha, L) \in A^dC(wk, \tau)$, then $D\alpha_{*L}$ is weakly safe of type(a)*
2. *For every $(\alpha, L) \in AC(\tau) \setminus (A^{st})C(wk, \tau) \cup A^{end}C(wk, \tau))$ then α_L is either irrelevant or balanced.*

Then τ is an invariant.

Chapter 3

Implementation

In order to implement the TIS algorithm it was necessary to make use of Object Oriented Programming to create the objects Template, Component and Predicate. A component as stated in the theory is an object containing a predicate and a counted variable, the predicate objects were obtained from the parser which translated the domain file from its format in PDDL into a readable format that Python would be able to read. Once a component object is created it is then possible to start to create Templates. Only templates containing more than one argument were considered for the initial templates (which contained only a single component since the synthesizer takes a bottom up approach when searching for templates). This is due to the fact that a single component would be able to have an admissible partition only in the instance that it has one fixed variable and one counted variable. It would be possible to create an admissible partition for a single component template with one argument however this would be equivalent to grounding the template which bears little relevance when analysing templates on a lifted level. At this point a set of candidate templates was created such that I could start testing for possible invariants, each candidate template was cycled through the method *checkBalance* which received in input the template itself as well as the *task*, a translated format of the domain obtained from the parser, this was done in order to have the relevant actions available to test against the template.

The first check implemented was to see if there were any actions to test against the template, in the event that were none then this resulted in a *Trivial Template* being found. The code was written in a manner such that in the instance that this occured the function would exit and return *None* resulting in the template being placed in the list of Trivial Templates. In the event that relevant actions were found the code would carry on with the next step of entering into an internal method *checkAction-Balance* which would carry out matching with respect to every action in order to obtain the *Pure Action Schemas* and then proceed with using the majority of the checks within the TIS algorithm to check every action schema. The creation of the pure action schemas was broken down into three phases. The first consisted of obtaining the relevant components within the template τ that matched the action predicates. Filtering through the action preconditions and effects only components in the template that matched predicates within action schemas were kept, predicates with the same name but different argument variables were different entities. A list of lists is created as a result with each list containg the predicate along with there respective arguments and their fixed argument in the last index. This list of lists was used to create the L-classes, running through the fixed variables list which was created from the list of lists, L-classes were then formed by grouping predicates which had the same fixed argument variable together. Orgininally the code was designed to create L - classes for each action segment however after looking through the theory again it was changed to create classes based on the predicates found within the durative action instead of splitting the action up. In the last step which followed on from the creation of the L-classes it was necessary to run through the action schema again and proceed with creating the pure action schema with respect to every L-class. The pure action schemas obtained would contain only predicates that matched up with predicates found within that particular L-class. This results in the creation of three dictionaries for the pure action schemas of every action, the three dictionaries represent a segment of the durative action with each dictionary containing the pure action schema of every L-class for that particular segment of the durative action. After some deliberation I decided to proceed with the creation of a further two dictionaries for the auxillary action schemas as well. These schemas had the same structure as the pure action schemas with the main difference being that the preconditions for each segment had been modified as seen in the theory.

In order to maintain some structure to my code and retain the information created from carrying out matching I decided to create a list of tuples with each tuple containing the action along with its pure and auxillary pure action schemas. Several for loops for the actions were carried out when running through the TIS algorithm, the main one was fundamental as it was necessary to carry out the instantaneous checks for each pure action schema. The results obtained for every schema of every action were saved within a dataframe to keep hold of the information for the checks that would follow as well as to allow for the creation and export of a csv file for the template being analysed. The results of one particular template can be seen below:

In the event that one of the schemas was found to be *Heavy* or *Unbalanced* the function would return a *False* boolean value and exit resulting in the template being scrapped in the false invariant list. This list can later be looked at to search for possible candidates by modifying them manually as part of the *guess, check and repair* approach which is quite common when carrying out Invariant Synthesis. In the instance that no heavy, unbalanced or unbounded action schemas were found

	Action	Class	Class_Id	Result	Section
0	left	['robot-at', ['?r', '?x'], 0]	L_0	Irrelevant	Start
1	left	['robot-at', ['?r', '?y'], 0]	L_0	Irrelevant	Start
0	left	['robot-at', ['?r', '?x'], 0]	L_0	Unbounded	End
1	left	['robot-at', ['?r', '?y'], 0]	L_0	Unbounded	End
0	up	['robot-at', ['?r', '?x'], 0]	L_0	Irrelevant	Start
1	up	['robot-at', ['?r', '?y'], 0]	L_0	Irrelevant	Start
0	up	['robot-at', ['?r', '?x'], 0]	L_0	Unbounded	End
1	up	['robot-at', ['?r', '?y'], 0]	L_0	Unbounded	End
0	right	['robot-at', ['?r', '?x'], 0]	L_0	Irrelevant	Start
1	right	['robot-at', ['?r', '?y'], 0]	L_0	Irrelevant	Start
0	right	['robot-at', ['?r', '?x'], 0]	L_0	Unbounded	End
1	right	['robot-at', ['?r', '?y'], 0]	L_0	Unbounded	End
0	down	['robot-at', ['?r', '?x'], 0]	L_0	Irrelevant	Start
1	down	['robot-at', ['?r', '?y'], 0]	L_0	Irrelevant	Start
0	down	['robot-at', ['?r', '?x'], 0]	L_0	Unbounded	End
1	down	['robot-at', ['?r', '?y'], 0]	L_0	Unbounded	End

then strong safety is present within all the action schemas and the function returns True indicating that it is an invariant. This is a vary strong property and rarely occured when looking at different domains. In the event that an unbounded action schema is found my code would skip the part related to checking if the action in question is a durative or instantaneous action under the presumption that my code is to be used solely for durative action schemas. The code therefore proceeds with the step of checking for executability for the actions containing unbounded schema. Since the unbounded schemas cannot be *strongly safe* the only other possible form of safety we can search for is weak safety. Therefore it is necessary to check for reachability and as a consequence search for executabilty within the action schemas that are unbounded. In the event that all the unbounded action schemas are reachable it is also necessary to check for strong safety with respect to the group of auxillary start actions GA^{st}_{*L}.

In the final few checks if all the unbounded actions are found to be executable and all the auxillary start actions are found to be strongly safe then we search for type(a) simple safety within all the durative actions that contain an unbounded schema. If all these respective durative action are found to be simply safe of type(a) and if in the last checks no action schemas are found to be unreachable or bounded then we have found an invariant.

Chapter 4

Experiments

In the domains that were tested all the action schemas were found to either be irrelevant or unbounded. As a result the majority of checks were carried out with respect to each action, what was of interest was seeing that most of the invariants found were as a result of Corollary 2.1.4. I tested my code on three domains in particular, the satellite domain, the drivelog domain and the floortile domain. Most of the checks were carried out on single component templates, in the event that a template was found not to be an invariant these templates could then be adjusted. If a template is found to have a heavy or unbalanced action schema then nothing can be done to rectify the template not being an invariant. If however a template was found to not be an invariant due to one of its action schemas being unbounded and not meeting the sufficient criteria to be an invariant then something can be done. Based on the particular unbounded action schemas other components which share common argument variables with the templates components within the particular unbounded action schemas can be added. One should analyse carefully the components that could be added, if a component is found to not add any further value to the search then it should not be considered. For example when looking at the floortile problem several trivial invariants were found due to their components irrelevance, the predicates *up, down, left, right* when analysed as singular component templates were all found to be trivial invariants as they were all contained within the *overall conditions* and therefore all the actions against which the action schemas would be tested would lead to an irrelevant classification. In order to speed up this process a *get_threats* method was used, this would retrieve the set of actions with respect to each predicate of each component that could potentially be a "threat" to the invariance of the template. If no actions were found (due to the predicates within the template only being within the overall conditions of each action) then that template is considered trivial. Returning to the adjusting of non invariant templates any predicates that share a common argument variable with the template components and are deemed relevant (are found not to be trivial) are possible candidates to adjust the template, these branch candidates can then be added to the template creating a new template object and then be ran through the checks again to see if a sufficient condition is met. An example of a template which was found following this procedure is the template τ with the admissible partition $\{(\text{robotAt}(r,x),0),(\text{clear(x)},1)\}$. After discovering that (robotAt(r,x), 0) as a template is not an invariant we search for possible components to add, we find the predicate (clear, 1) and after reapplying the checks we find that it is an invariant. Following in another direction we also find the template $\{(\text{robotAt}(r,x),0),(\text{painted},1),(\text{clear(y)},1)\}$ this also is an example of a more complex invariant.

Chapter 5

Related Work and Conclusion

The majority of the checks within the algorithm were implemented, it was necessary to carry out non - safe mutex in order to be able to use these checks when searching for invariants. In the event that the algorithm arrived at one of two points in which a check is not implemented and in the instance that the previous checks did not result in proving that the template is not an invariant then it is assumed that we are dealing with an invariant. This is a common procedure when constructing invariant synthesisers. The last two checks deal with finding invariants based on the corollaries which find that the *durative actions* A^d are either all *relevat right isolated* or are all *pairwise relevant non overlapping*. This theory has been omitted as these checks were not implemented. Another part of the code which could be added to fully complete the implementation of the algorithm is the use of the *guess, check, repair* approach. Currently the code only checks for singular component templates, other templates can be checked for however. In order to do this one must create the template within the *find_invariants* method within the file *new_invariant_finder.py*.
The finding of these invariants is useful as it allows for the invariant sets of boolean state variables found in the PDDL2.1 domains to be tranformed to multi-valued state variables in another language which allows for this.
Another type of invariant which could be searched for using the TIS algorithm are metric invariants however further research has yet been done in this field of study.
In the end the code successfully found several invariants within temporal planning domains, all of these invariants were found as a result of corollary 2.1.4.

Appendix A

Invariant Synthesiser Code

This appendix contains the code I wrote in order to implement the TIS algorithm, some of the intermediary code has been omitted however all the objects and principle methods used can be found here.

```
import pddl
import invariants
from template import Template
from component import Component


def generate_components(predicate):
  predicate_component_list = []

  for idx in range(len(predicate.arguments)+1):
    predicate_component_list.append(Component(predicate, idx))

  return predicate_component_list


def generate_initial_templates(components):
  initial_candidates = []

  for component in components:

    if len(component.predicate.arguments)!=component.
   counted_variable and len(component.predicate.arguments)!=1:
      list_component = component
      initial_candidates.append(Template(list_component))

  return initial_candidates

def find_invariants(task):

  components = set()
  for predicate in task.predicates:
```

```
    components.update(generate_components(predicate))

  components = sorted(frozenset(components))

  component1=None
  component2=None
  component3=None

  for component in components:

    if(component.predicate.name=='robot-at' and component.counted_variable==0):
      component1=component

    if(component.predicate.name=='painted' and component.counted_variable==1):
      component2=component

    if(component.predicate.name=='clear' and component.counted_variable==1):
      component3=component


  example_template = [component1, component2, component3]

  print "generating invariant candidates"
  candidates = generate_initial_templates(components)

  if not all(x is None for x in example_template):
    t = Template(example_template)
    t.__addFixedArgument__(component1, 1)
    t.__addFixedArgument__(component2, 0)
    t.__addFixedArgument__(component3, 0)
    candidates.insert(0,t)

  seen_candidates = set(candidates)

  def enqueue_func(invariant):
    if invariant not in seen_candidates:
      candidates.append(invariant)
          #List for invariant candidates to be seen

          #List of already examined invariant candidates
          seen_candidates.add(invariant)

balance_checker = Balance_Checker(task)

  list_trivial_templates = []
  list_templates_true = []
  list_not_template = []
```

```

  for possible_template in candidates:
    [component.predicate.name for component in possible_template.
   components]
    tuple_template_action_results , number = possible_template.
   check_balance(balance_checker , task , enqueue_func)

    if tuple_template_action_results == None:
      list_trivial_templates.append(possible_template)

    elif tuple_template_action_results:
      list_templates_true.append([possible_template,number])

    else:
      list_not_template.append([possible_template])

  print(tuple_template_action_results)

  return list_templates_true , list_trivial_templates
```

Listing A.1. main.py

```

class Component(object):

    def __init__(self, predicate, counted_variable):
        self.predicate = predicate
        self.counted_variable = counted_variable

    def __getNumberFixedVariables__(self):

        check = True
        if len(self.predicate.arguments)==self.counted_variable:
            check = False

        return (len(self.predicate.arguments) - check)

    def __getNumberArguments__(self):
        return len(self.predicate.arguments)

    def __getPredicate__(self):
        return self.predicate

    def __getPredicateName__(self):
        return self.predicate.name
```

Listing A.2. component.py

```

from component import Component
import collections as col
```

```python
import pddl
import pandas as pd
import csv

class Template(object):

    def __init__(self, components):
        self.fixed_variables = []


        if isinstance(components, Component):
            #This doesn't work exactly as the each predicate must have the same number of fixed variables as the others.

            #if len(components.predicate.arguments)!=components.counted_variable:
            if len(components.predicate.arguments)<=2:

                self.components = [components]

                for i in range(components.counted_variable):
                        self.fixed_variables.append((components, i))

                for i in range(components.counted_variable+1, len(components.predicate.arguments)):
                    self.fixed_variables.append((components, i))

  def __addFixedArgument__(self, component, fixedArgument):
        if(len(self.fixed_variables)<self.__getNumberFixedArguments__()*len(self.components)):
            if(isinstance(component, Component) and isinstance(fixedArgument, int)):
                if(component in self.components):
                    if(fixedArgument<=(component.__getNumberArguments__()-1)):
                        if((component, fixedArgument) not in self.fixed_variables):
                            self.fixed_variables.append((component, fixedArgument))
                    else:
                        raise ValueError('WARNING: Fixed argument value is bigger than the number of arguments.')
                else:
                    raise ValueError('WARNING: object inserted was NOT a Component!!')
```

Listing A.3. template.py (Template)

```python
def obtain_pure_action_schemas(task, action):

```

```

        def matching_components(task, predicate_set):

                predicates = predicate_set[:]
                #predicates2 = predicate_set[:]
                l_class_list = []

                fixed_argument_list = []
                for predicate in predicates:
                    fixed_argument_list.append(predicate[1][predicate[2]])

                unique_fixed_argument_list = []

                for argument in fixed_argument_list:
                    if argument not in unique_fixed_argument_list:
                        unique_fixed_argument_list.append(argument)

                for argument in unique_fixed_argument_list:
                    l_class = [predicate for predicate in predicates if predicate[1][predicate[2]]==argument]
                    l_class_list.append(l_class)

                sifted_l_class_list = [l_class for l_class in l_class_list if l_class]
                return sifted_l_class_list


        relevant_predicates_start = []
        relevant_predicates_mid = []
        relevant_predicates_end = []

        for component in self.fixed_variables:

            for effect in action.get_add_start_peffects():
                if(effect.predicate == component[0].predicate.name):
                    if([effect.predicate, [arg.name for arg in effect.args], component[1]] not in relevant_predicates_start):
                        relevant_predicates_start.append([effect.predicate, [arg.name for arg in effect.args], component[1]])

            for condition in action.get_pos_start_conds():
                if(condition.predicate == component[0].predicate.name):

```

```
                    if([condition.predicate,[arg.name for arg in condition.args], component[1]] not in relevant_predicates_start):
                        relevant_predicates_start.append([condition.predicate,[arg.name for arg in condition.args], component[1]])

            for effect in action.get_del_start_peffects():
                if(effect.predicate == component[0].predicate.name):
                    if([effect.predicate, [arg.name for arg in effect.args], component[1]] not in relevant_predicates_start):
                        relevant_predicates_start.append([effect.predicate, [arg.name for arg in effect.args], component[1]])

            for condition in action.get_neg_start_conds():
                if(condition.predicate == component[0].predicate.name):
                    if([condition.predicate,[arg.name for arg in condition.args], component[1]] not in relevant_predicates_start):
                        relevant_predicates_start.append([condition.predicate,[arg.name for arg in condition.args], component[1].counted_variable])

            for condition in action.get_pos_all_conds():
                if(condition.predicate == component[0].predicate.name):
                    if([condition.predicate,[arg.name for arg in condition.args], component[1]] not in relevant_predicates_start):
                        relevant_predicates_mid.append([condition.predicate,[arg.name for arg in condition.args], component[1]])

            for condition in action.get_neg_all_conds():
                if(condition.predicate == component[0].predicate.name):
                    if([condition.predicate,[arg.name for arg in condition.args], component[1]] not in relevant_predicates_start):
                        relevant_predicates_mid.append([condition.predicate,[arg.name for arg in condition.args], component[1]])

            for effect in action.get_add_end_peffects():
                if(effect.predicate == component[0].predicate.name):
```

```
                    if([effect.predicate, [arg.name for arg in effect.args], component[1]] not in relevant_predicates_start):
                        relevant_predicates_end.append([effect.predicate, [arg.name for arg in effect.args], component[1]])

            for condition in action.get_pos_end_conds():
                if(condition.predicate == component[0].predicate.name):
                    if([condition.predicate,[arg.name for arg in condition.args], component[1]] not in relevant_predicates_start):
                        relevant_predicates_end.append([condition.predicate,[arg.name for arg in condition.args], component[1]])

            for effect in action.get_del_end_peffects():
                if(effect.predicate == component[0].predicate.name):
                    if([effect.predicate, [arg.name for arg in effect.args], component[1]] not  in relevant_predicates_start):
                        relevant_predicates_end.append([effect.predicate, [arg.name for arg in  effect.args], component[1]])

            for condition in action.get_neg_end_conds():
                if(condition.predicate == component[0].predicate.name):
                    if([condition.predicate,[arg.name for arg in condition.args], component[1]] not in relevant_predicates_start):
                        relevant_predicates_end.append([condition.predicate,[arg.name for arg in    condition.args], component[1].counted_variable])

        predicates = relevant_predicates_start + relevant_predicates_mid + relevant_predicates_end

        aux_relevant_predicates_start = relevant_predicates_start[:]
        aux_relevant_predicates_end = relevant_predicates_end[:]

        for predicate in relevant_predicates_mid:
            if(predicate[0] not in [relevant_predicates[0] for relevant_predicates in relevant_predicates_start]):
                aux_relevant_predicates_start.append(predicate)

            if(predicate[0] not in [relevant_predicates[0] for relevant_predicates in relevant_predicates_end]):
                aux_relevant_predicates_end.append(predicate)

```

```python

        relevant_predicates = [relevant_predicates_start , relevant_predicates_mid , relevant_predicates_end]

        all_predicates = relevant_predicates_start + relevant_predicates_mid + relevant_predicates_end
        predicates = []

        for predicate in all_predicates:
            if predicate not in predicates:
                predicates.append(predicate)

        l_classes = matching_components(task, predicates)


        pure_action_schema_start = {"L_"+str(i) :l_classes[i] for i in range(len (l_classes))}
        pure_action_schema_mid = {"L_"+str(i) :l_classes[i] for i in range(len (l_classes))}
        pure_action_schema_end = {"L_"+str(i) :l_classes[i] for i in range(len (l_classes))}

        auxiliary_action_schema_start = {"L_"+str(i) :l_classes[i] for i in range(len(l_classes))}
        auxiliary_action_schema_end = {"L_"+str(i) :l_classes[i] for i in range(len(l_classes))}

        def create_pure_action_schemas_start(action, values):
                #Returns a list [positive_preconditions, negative_preconditions, positive_effects,  negative_effects]
                # for the pure action schema for a T_class
                list_effects_condtions_start = []

                intersecting_pos_preconds_start = []
                list_effects_condtions_start.append(intersecting_pos_preconds_start)
                intersecting_neg_preconds_start = []
                list_effects_condtions_start.append(intersecting_neg_preconds_start)
                intersecting_pos_effs_start = []
                list_effects_condtions_start.append(intersecting_pos_effs_start)
                intersecting_neg_effs_start = []
                list_effects_condtions_start.append(intersecting_neg_effs_start)
                list_effects_condtions_start.append(values)

                #Obtaining positive effects
                for effect in action.get_add_start_peffects():
```

```
                arguments = [arg.name for arg in effect.args]
                for predicate in values:
                    if effect.predicate == predicate[0]:
                        if arguments == predicate[1]:
                            intersecting_pos_effs_start.append(effect.predicate)

            #Obtaining negative effects
            for effect in action.get_del_start_peffects():
                arguments = [arg.name for arg in effect.args]
                for predicate in values:
                    if effect.predicate == predicate[0]:
                        if arguments == predicate[1]:
                            intersecting_neg_effs_start.append(effect.predicate)

            #Obtaining positive conditions for L schema
            for cond in action.get_pos_start_conds():
                arguments = [arg.name for arg in cond.args]
                for predicate in values:
                    if cond.predicate == predicate[0]:
                        if arguments == predicate[1]:
                            intersecting_pos_preconds_start.append(cond.predicate)

            #Obtaining negative conditions for L schema
            for cond in action.get_neg_start_conds():
                arguments = [arg.name for arg in cond.args]
                for predicate in values:
                    if cond.predicate == predicate[0]:
                        if arguments == predicate[1]:
                            intersecting_neg_preconds_start.append(cond.predicate)

            return list_effects_condtions_start

    def create_pure_action_schemas_end(action, values):

            list_effects_conditions_end = []

            intersecting_pos_preconds_end = []
            list_effects_conditions_end.append(intersecting_pos_preconds_end)
            intersecting_neg_preconds_end = []
            list_effects_conditions_end.append(intersecting_neg_preconds_end)
            intersecting_pos_effs_end = []
```

```
                list_effects_conditions_end.append(intersecting_pos_effs_end)
                intersecting_neg_effs_end = []
                list_effects_conditions_end.append(intersecting_neg_effs_end)
                list_effects_conditions_end.append(values)

                #Obtaining positive effects
                for effect in action.get_add_end_peffects():
                    arguments = [arg.name for arg in effect.args]
                    for predicate in values:
                        if effect.predicate == predicate[0]:
                            if arguments == predicate[1]:
                                intersecting_pos_effs_end.append(effect.predicate)

                #Obtaining negative effects
                for effect in action.get_del_end_peffects():
                    arguments = [arg.name for arg in effect.args]
                    for predicate in values:
                        if effect.predicate == predicate[0]:
                            if arguments == predicate[1]:
                                intersecting_neg_effs_end.append(effect.predicate)

                #Obtaining positive conditions for L schema
                for cond in action.get_pos_end_conds():
                    arguments = [arg.name for arg in cond.args]
                    for predicate in values:
                        if cond.predicate == predicate[0]:
                            if arguments == predicate[1]:
                                intersecting_pos_preconds_end.append(cond.predicate)

                #Obtaining negative conditions for L schema
                for cond in action.get_neg_end_conds():
                    arguments = [arg.name for arg in cond.args]
                    for predicate in values:
                        if cond in predicate[0]:
                            if arguments == predicate[1]:
                                intersecting_neg_preconds_end.append(cond.predicate)

                return list_effects_conditions_end

        def create_pure_action_schemas_mid(action, values):

                list_conditions_mid = []
```

```
            intersecting_pos_preconds_mid = []
            list_conditions_mid.append(intersecting_pos_preconds_mid)
            intersecting_neg_preconds_mid = []
            list_conditions_mid.append(intersecting_neg_preconds_mid)
            list_conditions_mid.append(values)

            #Obtaining positive conditions for L schema
            for cond in action.get_pos_all_conds():
                arguments = [arg.name for arg in cond.args]
                for predicate in values:
                    if cond.predicate == predicate[0]:
                        if arguments == predicate[1]:
                            intersecting_pos_preconds_mid.append(cond.predicate)

            #Obtaining negative conditions for L schema
            for cond in action.get_neg_all_conds():
                arguments = [arg.name for arg in cond.args]
                for predicate in values:
                    if cond.predicate == predicate[0]:
                        if arguments == predicate[1]:
                            intersecting_neg_preconds_mid.append(cond.predicate)

            return list_conditions_mid

    def create_aux_pure_action_schemas_start(action, values):
            #Returns a list [positive_preconditions, negative_preconditions, positive_effects,  negative_effects]
            # for the pure action schema for a L_class

            list_aux_action_schema_start = []

            intersecting_pos_preconds_start = []
            intersecting_neg_preconds_start = []
            intersecting_pos_effs_start = []
            intersecting_neg_effs_start = []
            intersecting_pos_preconds_mid = []
            intersecting_neg_preconds_mid = []

            #Obtaining positive effects
            for effect in action.get_add_start_peffects():
                arguments = [arg.name for arg in effect.args]
                for predicate in values:
                    if effect.predicate == predicate[0]:
                        if arguments == predicate[1]:
```

```
                                intersecting_pos_effs_start.append(effect.predicate)

                #Obtaining negative effects
                for effect in action.get_del_start_peffects():
                    arguments = [arg.name for arg in effect.args]
                    for predicate in values:
                        if effect.predicate == predicate[0]:
                            if arguments == predicate[1]:
                                intersecting_neg_effs_start.append(effect.predicate)

                #Obtaining positive conditions for L schema
                for cond in action.get_pos_start_conds():
                    arguments = [arg.name for arg in cond.args]
                    for predicate in values:
                        if cond.predicate == predicate[0]:
                            if arguments == predicate[1]:
                                intersecting_pos_preconds_start.append(cond.predicate)

                #Obtaining negative conditions for L schema
                for cond in action.get_neg_start_conds():
                    arguments = [arg.name for arg in cond.args]
                    for predicate in values:
                        if cond.predicate == predicate[0]:
                            if arguments == predicate[1]:
                                intersecting_neg_preconds_start.append(cond.predicate)

                #Obtaining positive conditions for L schema
                for cond in action.get_pos_all_conds():
                    arguments = [arg.name for arg in cond.args]
                    for predicate in values:
                        if cond.predicate == predicate[0]:
                            if arguments == predicate[1]:
                                intersecting_pos_preconds_mid.append(cond.predicate)


                #Obtaining negative conditions for L schema
                for cond in action.get_neg_all_conds():
                    arguments = [arg.name for arg in cond.args]
                    for predicate in values:
                        if cond.predicate == predicate[0]:
                            if arguments == predicate[1]:
                                intersecting_neg_preconds_mid.append(cond.predicate)

```

```

                #ADDING POSITIVE PRECONDITIONS
                set_pos_preconds_start = set(intersecting_pos_preconds_start)
                set_pos_preconds_mid = set(intersecting_pos_preconds_mid)
                set_pos_effects_start = set(intersecting_pos_effs_start)
                pos_preconds_start = set_pos_preconds_start.union(set_pos_preconds_mid-set_pos_effects_start)
                list_aux_action_schema_start.append(list(pos_preconds_start))

                #ADDING NEGATIVE PRECONDITIONS
                set_neg_preconds_start = set(intersecting_neg_preconds_start)
                set_neg_preconds_mid = set(intersecting_neg_preconds_mid)
                set_neg_effects_start = set(intersecting_neg_effs_start)
                neg_preconds_start = set_neg_preconds_start.union(set_neg_preconds_mid-set_neg_effects_start)
                list_aux_action_schema_start.append(list(neg_preconds_start))

                #ADDING START_EFFECTS (they remain the same as the normal actions)
                list_aux_action_schema_start.append(intersecting_pos_effs_start)
                list_aux_action_schema_start.append(intersecting_neg_effs_start)

                #ADDING THE L_CLASS
                list_aux_action_schema_start.append(values)

                return list_aux_action_schema_start

        def create_aux_pure_action_schemas_end(action, values):

                list_aux_action_schema_end = []

                intersecting_pos_preconds_end = []
                intersecting_neg_preconds_end = []
                intersecting_pos_effs_end = []
                intersecting_neg_effs_end = []
                intersecting_pos_preconds_mid = []
                intersecting_neg_preconds_mid = []

                #list_effects_conditions_end.append(values)

```

```
                    #Obtaining positive effects
                    for effect in action.get_add_end_peffects():
                        arguments = [arg.name for arg in effect.args]
                        for predicate in values:
                            if effect.predicate == predicate[0]:
                                if arguments == predicate[1]:
                                    intersecting_pos_effs_end.append(effect.predicate)

                    #Obtaining negative effects
                    for effect in action.get_del_end_peffects():
                        arguments = [arg.name for arg in effect.args]
                        for predicate in values:
                            if effect.predicate == predicate[0]:
                                if arguments == predicate[1]:
                                    intersecting_neg_effs_end.append(effect.predicate)

                    #Obtaining positive conditions for L schema
                    for cond in action.get_pos_end_conds():
                        arguments = [arg.name for arg in cond.args]
                        for predicate in values:
                            if cond.predicate == predicate[0]:
                                if arguments == predicate[1]:
                                    intersecting_pos_preconds_end.append(cond.predicate)

                    #Obtaining negative conditions for L schema
                    for cond in action.get_neg_end_conds():
                        arguments = [arg.name for arg in cond.args]
                        for predicate in values:
                            if cond.predicate == predicate[0]:
                                if arguments == predicate[1]:
                                    intersecting_neg_preconds_mid.append(cond.predicate)

                    #Obtaining positive conditions for L schema
                    for cond in action.get_pos_all_conds():
                        arguments = [arg.name for arg in cond.args]
                        for predicate in values:
                            if cond.predicate == predicate[0]:
                                if arguments == predicate[1]:
                                    intersecting_pos_preconds_mid.append(cond.predicate)

                    #Obtaining negative conditions for L schema
                    for cond in action.get_neg_all_conds():
                        arguments = [arg.name for arg in cond.args]
```

```
                     for predicate in values:
                         if cond.predicate == predicate[0]:
                             if arguments == predicate[1]:
                                 intersecting_neg_preconds_mid.append(cond.predicate)

                 #ADDING POSITIVE PRECONDITIONS
                 set_pos_preconds_end = set(intersecting_pos_preconds_end)
                 set_pos_preconds_mid = set(intersecting_pos_preconds_mid)
                 pos_preconds_end = set_pos_preconds_mid.union(set_pos_preconds_end)
                 list_aux_action_schema_end.append(list(pos_preconds_end))

                 #ADDING NEGATIVE PRECONDITIONS
                 set_neg_preconds_end = set(intersecting_neg_preconds_end)
                 set_neg_preconds_mid = set(intersecting_neg_preconds_mid)
                 neg_preconds_end = set_neg_preconds_mid.union(set_neg_preconds_end)
                 list_aux_action_schema_end.append(list(neg_preconds_end))

                 #ADDING END_EFFECTS (they remain the same as the normal actions)
                 list_aux_action_schema_end.append(intersecting_pos_effs_end)
                 list_aux_action_schema_end.append(intersecting_neg_effs_end)

                 #ADDING THE L_CLASS
                 list_aux_action_schema_end.append(values)

                 return list_aux_action_schema_end

 for key, value in pure_action_schema_start.iteritems():
             if value:
                 pure_action_schema_start[key] = create_pure_action_schemas_start(action, value)

         for key, value in pure_action_schema_mid.iteritems():
             if value:
                 pure_action_schema_mid[key] = create_pure_action_schemas_mid(action, value)

         for key, value in pure_action_schema_end.iteritems():
             if value:
```

```
                    pure_action_schema_end[key] = create_pure_action_schemas_end(action, value)

            for key, value in auxiliary_action_schema_start.iteritems():
                auxiliary_action_schema_start[key] = create_aux_pure_action_schemas_start(action, value)

            for key, value in auxiliary_action_schema_end.iteritems():
                auxiliary_action_schema_end[key] = create_aux_pure_action_schemas_end(action, value)

            pure_action_schemas = []
            pure_action_schemas.append(pure_action_schema_start)
            pure_action_schemas.append(pure_action_schema_mid)
            pure_action_schemas.append(pure_action_schema_end)

            auxiliary_pure_action_schemas = []
            auxiliary_pure_action_schemas.append(auxiliary_action_schema_start)
            auxiliary_pure_action_schemas.append(auxiliary_action_schema_end)

            return pure_action_schemas, auxiliary_pure_action_schemas
```

Listing A.4. Matching and Pure Action Schemas Method

The following code consists of the methods used for the checks when implementing the TIS algorithm.

```

def instantaneous_action_checks(action, key, schema, time_period):
                if(len(schema[0])>=2):
                    result = pd.DataFrame({'Action': action.name, 'Class_Id': key,  'Class': schema[4], 'Section': time_period, 'Result': 'Unreachable'})
                    return result

                if(len(schema[0])<=1):
                    if((len(schema[2])==0)):
                        result = pd.DataFrame({'Action': action.name, 'Class_Id': key,  'Class': schema[4], 'Section': time_period, 'Result': 'Irrelevant'})
                        return result

                    #Is the action heavy?
                    if(len(schema[2])>=2):
                        result = pd.DataFrame({'Action': action.name, 'Class_Id': key, 'Class': schema[4], 'Section': time_period, 'Result': 'Heavy'})
```

```
                    return result


                if(len(schema[2])==1):

                    if(len(schema[0])==1):

                        pos_effects = set(schema[2])
                        neg_effects = set(schema[3])
                        union_effects = pos_effects.union(neg_effects)
                        pos_preconditions = set(schema[0])
                        inter_effects_with_pos_preconditions = pos_preconditions.intersection(union_effects)

                        if pos_preconditions in union_effects:
                            result = pd.DataFrame({'Action': action.name, 'Class_Id': key, 'Class': schema[4], 'Section': time_period, 'Result':'Balanced'})
                            return result

                        elif not inter_effects_with_pos_preconditions:
                            result = pd.DataFrame({'Action': action.name, 'Class_Id': key, 'Class': schema[4], 'Section': time_period, 'Result':'Unbalanced'})
                            return result

                    if(len(schema[0])==0):

                        template_predicates = set([component.predicate.name for component in self.components])

                        #Class Predicates
                        l_class_predicates = set([predicate[0] for predicate in schema[4]])
                        if(l_class_predicates == template_predicates):

                            #Here I am checking that the predicates do not have a counted variable, if they do then I assume it is unbounded which it is in the majority of cases
                            for predicate in l_class_predicates:
                                for component in self.components:
                                    if(predicate == component.predicate.name):
                                        if(component.counted_variable < len(component.predicate.arguments)):
```

```
                                                result = pd.DataFrame({'Action': action.name, 'Class_Id': key, 'Class': schema[4], 'Section': time_period, 'Result': 'Unbounded'})
                                                return result

                                result = pd.DataFrame({'Action': action.name, 'Class_Id': key, 'Class': schema[4], 'Section': time_period, 'Result': 'Bounded'})
                                return result



                            else:
                                result = pd.DataFrame({'Action': action.name, 'Class_Id': key, 'Class': schema[4], 'Section': time_period, 'Result': 'Unbounded'})
                                return result
                    else:
                        print('ERROR IN INSTANTANEOUS CLASSIFICATION CODE!!')
                        result = pd.DataFrame({'Action': action.name, 'Class_Id': key, 'Class': schema[4], 'Section': time_period, 'Result': 'UNCLASSIFIED'})
                        return result
```

Listing A.5. Instantaneous Action Check Method

```
def check_type_a_simple_safety(aux_start_schema):

                type_a_bool = False

                if(len(aux_start_schema[0])==1):
                    pre_cond_pos_start = set(aux_start_schema[0])
                    effs_neg_start = set(aux_start_schema[3])

                    if pre_cond_pos_start.issubset(effs_neg_start):
                        print("Weakly safe of Type(A)")
                        type_a_bool = True

                return type_a_bool
```

Listing A.6. Type(a) Simple Safety Check Method

```
def check_simple_safety(action_schema_result, pure_schemas, aux_schemas):
                #Should I use proposition 75 (page 36) and definition 76 (Executability and Reachability)
                #I DON'T NEED TO CHECK FOR EXECUTABILITY SINCE THIS CHECK WAS DONE BEFORE
                bool_simple_safety = False
                aux_start_schema = aux_schemas[0]
```

```

            aux_pre_cond_pos_start = set(aux_start_schema[0])
            aux_effs_neg_start = set(aux_start_schema[3])

            pure_end_schema = pure_schemas[2]
            pos_end_effs = set(pure_end_schema[2])
            neg_end_effs = set(pure_end_schema[3])
            end_effects = pos_end_effs.union(neg_end_effs)

            if(action_schema_result == "Irrelevant"):

                if check_type_a_simple_safety(aux_schemas):
                    bool_simple_safety = True
                    return bool_simple_safety

                elif(len(aux_start_schema[0])==1):
                    if aux_pre_cond_pos_start not in aux_effs_neg_start and aux_pre_cond_pos_start in end_effects:
                        print("Weakly safe of Type(B)")
                        bool_simple_safety = True
                        return bool_simple_safety

                elif(len(aux_start_schema[0])==0):

                    aux_pre_cond_neg_start = set(aux_start_schema[1])
                    cover_check_union = aux_pre_cond_neg_start.union(end_effects, aux_effs_neg_start)
                    template_predicates = set([component.predicate.name for component in self.components])

                    l_class_predicates = [predicate[0] for predicate in aux_start_schema[4]]

                    if template_predicates == cover_check_union:
                        for predicate in l_class_predicates:
                            for component in self.components:
                                if(predicate == component.predicate.name):
                                    if(component.counted_variable == len(component.predicate.arguments)):
                                        print("Weakly safe of Type(C)")
                                        bool_simple_safety = True
                                        return bool_simple_safety

            elif(action_schema_result == "Relevant"):
```

```

                   pure_start_schema = pure_schemas[0]
                   pos_start_effs = set(pure_start_schema[2])

                   if pos_start_effs in end_effects:
                       print("Weakly safe of Type(D)")
                       bool_simple_safety = True
                       return bool_simple_safety


               return bool_simple_safety
```

Listing A.7. Simple Safety Check Method

```

def executability_check(action):
               #PROPOSITION 75 (pg36) EXECUTABILITY

               bool_result = True

               #POSITIVE START CONDITIONS
               pos_start_cond = set([predicate.predicate for predicate in action.get_pos_start_conds()])

               #POSITIVE INTERMEDIATE CONDITIONS
               pos_inv_cond = set([predicate.predicate for predicate in action.get_pos_all_conds()])

               #POSITIVE START EFFECTS
               pos_start_effect = set([predicate.predicate for predicate in action.get_add_start_peffects()])

               #NEGATIVE START EFFECTS
               neg_start_effect = set([predicate.predicate for predicate in action.get_del_start_peffects()])

               #NEGATIVE END CONDITIONS
               neg_end_cond = set([predicate.predicate for predicate in action.get_neg_end_conds()])

               #NEGATIVE INTERMEDIATE CONDITIONS
               neg_inv_cond = set([predicate.predicate for predicate in action.get_neg_all_conds()])

               #NEGATIVE START CONDITIONS
               neg_start_cond = set([predicate.predicate for predicate in action.get_neg_start_conds()])

               #POSITIVE END CONDITIONS
               pos_end_cond = set([predicate.predicate for predicate in action.get_pos_end_conds()])
```

```python

            AUX_pos_start_cond = pos_start_cond.union(pos_inv_cond - pos_start_effect)
            AUX_pos_postconditions_start = pos_start_effect.union((AUX_pos_start_cond - neg_start_effect))

            AUX_pre_end_cond = neg_end_cond.union(neg_inv_cond)

            check_executability_1 = AUX_pos_postconditions_start.intersection(AUX_pre_end_cond)

            #If empty then the action is executable
            AUX_neg_start_cond = neg_start_cond.union(neg_inv_cond - neg_start_effect)

            AUX_neg_start_postconditions = neg_start_effect.union(AUX_neg_start_cond - pos_start_effect)

            AUX_positive_end_conditions = pos_end_cond.union(pos_inv_cond)

            check_executability_2 = AUX_neg_start_postconditions.intersection(AUX_positive_end_conditions)

            if(check_executability_1 or check_executability_2):
                bool_result = False

            return bool_result
```

Listing A.8. Action Executability Check

```python
def check_auxilary_action_reachable(action, action_tuples):

    action_tuples_aux_schemas = [action_tuple[2] for action_tuple in action_tuples]

    for aux_action_schemas in action_tuples_aux_schemas:
        for key in aux_action_schemas[0]:

            cond_pos_start = set(aux_action_schemas[0][key][0])
            cond_pos_end = set(aux_action_schemas[1][key][0])
            eff_pos_start = set(aux_action_schemas[1][key][2])

            reachable_test_set = cond_pos_start.union(cond_pos_end - eff_pos_start)
            if len(reachable_test_set)>1:
                return False

    return True
```

Listing A.9. Auxiliary Action Reachable Check

```
def final_check(action, action_tuples, unbounded_pure_actions, aux_action_df):

                #(8) FINAL CHECK FOR SIMPLE SAFETY
                action_start_results_filter = action_df['Section']=='Start'
                action_start_results = action_df[action_start_results_filter]

                unbounded_actions_start_filter = action_start_results['Result']=='Unbounded'
                unbounded_actions_start = action_start_results[unbounded_actions_start_filter]

                end_unbounded_pure_actions_filter = unbounded_pure_actions["Section"]=='End'
                end_unbounded_pure_actions = unbounded_pure_actions[end_unbounded_pure_actions_filter]

                string_list_unbounded_actions = set(end_unbounded_pure_actions["Action"].astype(str).values.tolist())
                unbounded_end_action_tuples = [action_tuple for action_tuple in list_action_tuples if action_tuple[0].name in string_list_unbounded_actions]

                if not unbounded_actions_start.empty:
                    print unbounded_actions_start
                    return False

                elif not end_unbounded_pure_actions.empty:

                    start_aux_action_df_filter = aux_action_df["Section"]=='Start'
                    start_aux_action_df = aux_action_df[start_aux_action_df_filter]

                    for action_tuple in unbounded_end_action_tuples:
                        action_name_filter = end_unbounded_pure_actions["Action"] == action_tuple[0].name
                        action_name = end_unbounded_pure_actions[action_name_filter]

                        class_Ids = set(action_name["Class_Id"].astype(str).values.tolist())

```

```
                    for class_id in class_Ids:
                        aux_action_filter = start_aux_action_df['Action']==action_tuple[0].name
                        aux_action = start_aux_action_df[aux_action_filter]

                        class_id_filter = aux_action["Class_Id"]==class_id
                        action_class_id_result = aux_action[class_id_filter]

                        irrelevant_result_filter = action_class_id_result['Result']=='Irrelevant'
                        irrelevant_result = action_class_id_result[irrelevant_result_filter]


                        class_id_pure_action_schemas = [schema[class_id] for schema in action_tuple[1]]
                        class_id_aux_pure_action_schemas = [schema[class_id] for schema in action_tuple[2]]

                        if not irrelevant_result.empty:
                            if not check_simple_safety('Irrelevant', class_id_pure_action_schemas, class_id_aux_pure_action_schemas):

                                return False

                        balanced_result_filter = action_class_id_result['Result']=='Balanced'
                        balanced_result = action_class_id_result[balanced_result_filter]

                        bounded_result_filter = action_class_id_result['Result']=='Bounded'
                        bounded_result = action_class_id_result[bounded_result_filter]

                        if not balanced_result.empty or not bounded_result.empty:
                            if not check_simple_safety('Relevant', [schema[class_id] for schema in action_tuple[1]], [schema[class_id] for schema in action_tuple[2]]):

                                return False

                        else:

                            return False
```

```
                    return True
```

Listing A.10. Final Check

```

def algorithm_1(self, actions_to_check):
            action_schema_section = ['Start', 'Inv', 'End']

            #action_df contains all the results for the instantaneoues checks for every action relevant to the template.
            action_df = pd.DataFrame(None, columns=['Action', 'Class_Id', 'Class', 'Section','Result'])
            aux_action_df = pd.DataFrame(None, columns=['Action', 'Class_Id', 'Class', 'Section','Result'])

            list_action_tuples = []

            for action in actions_to_check:

                list_action_schemas = []
                pure_action_schemas, auxiliary_pure_action_schemas = obtain_pure_action_schemas(task, action)
                list_action_schemas.append(pure_action_schemas)

                for action_schema in pure_action_schemas:

                    #For every action I introduce a boolean, if any of the pure action schemas
                    #reveal that an invariant doesn't exist the bool_result returns FALSE

                    bool_result = True
                    for key, schema in action_schema.iteritems():
                        if(pure_action_schemas.index(action_schema)!=1):
                            time = action_schema_section[pure_action_schemas.index(action_schema)]
                            result = instantaneous_action_checks(action, key, schema, time)
                            action_df = pd.concat([action_df, result], sort = True)


                #Here I create a tuple containing all the corresponding action_schemas corresponding to a durative action's durative action schema
                action_tuple = tuple([action, pure_action_schemas, auxiliary_pure_action_schemas])
                list_action_tuples.append(action_tuple)

```

```
        template_list = [str(component.predicate) + "_" + "{" + str(component.counted_variable) + "}" for component in self.components]
        file_name = ""

        for component in template_list:
            file_name += " " + component

        file_name = file_name.replace(":", "")
        file_name = file_name.replace("?", "")
        file_name = file_name.replace(" ", "_") + ".csv"

        action_df.to_csv(path_or_buf= file_name)

        print(action_df)

        #(1) CHECK FOR HEAVY OR UNBALANCED ACTIONS
        unbalanced_pure_actions_filter = action_df['Result']=='Unbalanced'
        unbalanced_pure_actions = action_df[unbalanced_pure_actions_filter]

        heavy_pure_actions_filter = action_df['Result']=='Heavy'
        heavy_pure_actions = action_df[heavy_pure_actions_filter]

        if not unbalanced_pure_actions.empty or not heavy_pure_actions.empty:
            number = 1
            print("Possibly Not Invariant")
            return False, number

        #(2)CHECK FOR UNBOUNDED ACTIONS
        number = 2
        unbounded_pure_actions_filter = action_df['Result']=='Unbounded'
        unbounded_pure_actions = action_df[unbounded_pure_actions_filter]
        print(unbounded_pure_actions)

        if not unbounded_pure_actions.empty:

            #(4)CHECK FOR EXECUTABILITY OF UNBOUNDED ACTIONS
            number = 4

            unbounded_actions = set(unbounded_pure_actions["Action"].astype(str).values.tolist())
```

```
            unbounded_action_tuples = [action_tuple for action_tuple in list_action_tuples if action_tuple[0].name in unbounded_actions]

            for action_tuple in unbounded_action_tuples:
                action_to_check = action_tuple[0]
                result = executability_check(action_to_check)

                if not result:
                    print("POSSIBLY NOT INVARIANT")

            for action_tuple in list_action_tuples:
                aux_action_schemas = action_tuple[2]
                for action_schema in aux_action_schemas:
                    for key, schema in action_schema.iteritems():

                        aux_action_schema_section = ['Start', 'End']
                        time = aux_action_schemas.index(action_schema)

                        if(time==0):
                            result = instantaneous_action_checks(action_tuple[0], key, schema, aux_action_schema_section[time])
                            aux_action_df = pd.concat([aux_action_df, result], sort = True)


            unbounded_aux_pure_actions_filter = aux_action_df['Result']=='Unbounded'
            heavy_aux_pure_actions_filter = aux_action_df['Result'] == 'Heavy'
            unbalanced_aux_pure_actions_filter = aux_action_df['Result'] == 'Unbalanced'

            unbounded_aux_pure_actions = aux_action_df[unbounded_aux_pure_actions_filter]
            heavy_aux_pure_actions = aux_action_df[heavy_aux_pure_actions_filter]
            unbalanced_aux_pure_actions = aux_action_df[unbalanced_aux_pure_actions_filter]

            #(5) STRONGLY SAFE ACTIONS CHECK
            number = 5
            if not unbounded_aux_pure_actions.empty or not heavy_aux_pure_actions.empty or not unbalanced_aux_pure_actions.empty:
                print(unbounded_aux_pure_actions)
```

```
                print("POSSIBLY NOT INVARIANT: NOT STRONGLY SAFE")
                return False, number

            #OBTAINING THE auxiliary ACTION SCHEMA INSTANTANEOUS CHECKS
            for action_tuple in list_action_tuples:
                aux_action_schemas = action_tuple[2]
                for action_schema in aux_action_schemas:
                    for key, schema in action_schema.iteritems():
                        time = aux_action_schemas.index(action_schema)
                        if(time==1):
                            result = instantaneous_action_checks(action_tuple[0], key, schema, aux_action_schema_section[time])
                            aux_action_df = pd.concat([aux_action_df, result], sort = True)


            #(6) CHECKING FOR SIMPLE SAFETY FOR THE DURATIVE ACTIONS OF UNBOUNDED ACTION SCHEMAS

            number = 6
            for action_tuple in unbounded_action_tuples:
                aux_action_schemas = action_tuple[2]

                action_aux_filter = aux_action_df['Action'] == action_tuple[0].name
                action_aux = aux_action_df[action_aux_filter]
                #print(action_aux)

                action_filter = action_df['Action'] == action_tuple[0].name
                action_pure_results = action_df[action_filter]

                for action_schema in aux_action_schemas:

                    for key, schema in action_schema.iteritems():

                        id_aux_filter = action_aux['Class_Id'] == key
                        aux_action_schema_results = action_aux[id_aux_filter]

                        id_filter = action_pure_results['Class_Id'] == key
```

```
                        pure_action_schema = action_pure_results[id_filter]

                        print(pure_action_schema)

                        time = aux_action_schemas.index(action_schema)
                        action_schema_start_filter = aux_action_schema_results['Section'] == "Start"
                        aux_action_schema_start = aux_action_schema_results[action_schema_start_filter]

                        action_schema_end_filter = aux_action_schema_results['Section'] == "End"
                        aux_action_schema_end = aux_action_schema_results[action_schema_end_filter]

                        print(aux_action_schema_start)


                        if 'Irrelevant' in aux_action_schema_start.values and 'Unbounded' in aux_action_schema_end.values:

                            aux_pure_action_schemas_start_end = aux_action_schemas[0][key]


                            if not check_type_a_simple_safety(aux_pure_action_schemas_start_end):

                                print 'A pure action schema is not Type (a) simply safe (Definition 78)'

                                #STRONG SAFETY - IRRELEVANT, HEAVY, BALANCED, BOUNDED..."""
                                #SO IF UNBALANCED, HEAVY OR UNBOUNDED THE AUX ACTION IS NOT STRONGLY SAFE!!!!

                                #(7Y) CHECK AUX_END FOR STRONG SAFETY AND CHECK FOR REACHABILITY

                                aux_end_actions_filter = action_aux['Section']=='End'
                                aux_end_actions_results = action_aux[aux_end_actions_filter]


                                unbalanced_aux_end_actions_filter = aux_end_actions_results['Result']=='Unbalanced'
```

```
                                unbalanced_aux_end_actions = aux_end_actions_results[unbalanced_aux_end_actions_filter]

                                heavy_aux_end_actions_filter = aux_end_actions_results['Result']=='Heavy'
                                heavy_aux_end_actions = aux_end_actions_results[heavy_aux_end_actions_filter]

                                unbounded_aux_end_actions_filter = aux_end_actions_results['Result']=='Unbounded'
                                unbounded_aux_end_actions = aux_end_actions_results[unbounded_aux_end_actions_filter]


                                if check_auxiliary_action_reachable(action, list_action_tuples) and unbalanced_aux_end_actions.empty and unbounded_aux_end_actions.empty and heavy_aux_end_actions.empty:
                                    #Corollary 73 and Def 90
                                    print('INVARIANT FOUND!!')
                                    return True, number

                                else:
                                    number = 8
                                    if "robot-at" in [component.predicate.name for component in self.components]:
                                        print('analyse')
                                    #(8) Final Check


                                    final_check = final_check(action, list_action_tuples, unbounded_pure_actions, aux_action_df)

                                    if final_check:
                                        return True, number

                                    else:
                                        return False, number

            #(7N) CHECK FOR UNREACHABLE OR BOUNDED ACTION SCHEMAS
            bounded_pure_actions_filter = action_df['Result']=='Bounded'
            bounded_pure_actions = action_df[bounded_pure_actions_filter]
            unreachable_pure_actions_filter = action_df['Result']=='Unreachable'
```

```
                unreachable_pure_actions = action_df[
unreachable_pure_actions_filter]

                if bounded_pure_actions.empty and
unreachable_pure_actions.empty:
                    #DEFINITION 70/71
                    print('INVARIANT FOUND!!')
                    return True, number

                #(8) Final Check
                number = 8
                if final_check(action, list_action_tuples,
unbounded_pure_actions, aux_action_df):
                    return True, number

                else:
                    return False, number

            else:
                return True, number

            return bool_result
```

Listing A.11. TIS Algorithm Checks Method

Appendix B

Domains

The following appendix contains the domains which were used when testing the algorithms effectiveness when searching for invariants. [5]

```

(define (domain floor-tile)
 (:requirements :typing :durative-actions)
 (:types robot tile color - object)

(:predicates
    (robot-at ?r - robot ?x - tile)
    (up ?x - tile ?y - tile)
    (down ?x - tile ?y - tile)
    (right ?x - tile ?y - tile)
    (left ?x - tile ?y - tile)
    (clear ?x - tile)
                (painted ?x - tile ?c - color)
    (robot-has ?r - robot ?c - color)
                (available-color ?c - color)
                (free-color ?r - robot))

(:durative-action change-color
  :parameters (?r - robot ?c - color ?c2 - color)
  :duration (= ?duration 5)
  :condition (and (at start (robot-has ?r ?c))
        (over all (available-color ?c2)))
  :effect (and (at start (not (robot-has ?r ?c)))
           (at end (robot-has ?r ?c2))))

(:durative-action paint-up
  :parameters (?r - robot ?y - tile ?x - tile ?c - color)
  :duration (= ?duration 2)
  :condition (and (over all (robot-has ?r ?c))
        (at start (robot-at ?r ?x))
      (over all (up ?y ?x))
      (at start (clear ?y)))
```

```
  :effect (and (at start (not (clear ?y)))
          (at end (painted ?y ?c))))

(:durative-action paint-down
  :parameters (?r - robot ?y - tile ?x - tile ?c - color)
  :duration (= ?duration 2)
  :condition (and (over all (robot-has ?r ?c))
        (at start (robot-at ?r ?x))
      (over all (down ?y ?x))
      (at start (clear ?y)))
  :effect (and (at start (not (clear ?y)))
          (at end (painted ?y ?c))))


; Robot movements
(:durative-action up
  :parameters (?r - robot ?x - tile ?y - tile)
  :duration (= ?duration 3)
  :condition (and (at start (robot-at ?r ?x))
        (over all (up ?y ?x))
      (at start (clear ?y)))
  :effect (and
          (at start (not (robot-at ?r ?x)))
        (at end (robot-at ?r ?y))
        (at start (not (clear ?y)))
              (at end (clear ?x))))

(:durative-action down
  :parameters (?r - robot ?x - tile ?y - tile)
  :duration (= ?duration 1)
  :condition (and (at start (robot-at ?r ?x))
        (over all (down ?y ?x))
      (at start (clear ?y)))
  :effect (and (at start (not (robot-at ?r ?x)))
          (at end (robot-at ?r ?y))
        (at start (not (clear ?y)))
              (at end (clear ?x))))

(:durative-action right
  :parameters (?r - robot ?x - tile ?y - tile)
  :duration (= ?duration 1)
  :condition (and (at start (robot-at ?r ?x))
        (over all (right ?y ?x))
      (at start (clear ?y)))
  :effect (and (at start (not (robot-at ?r ?x)))
          (at end (robot-at ?r ?y))
        (at start (not (clear ?y)))
              (at end (clear ?x))))

(:durative-action left
```

```
  :parameters (?r - robot ?x - tile ?y - tile)
  :duration (= ?duration 1)
  :condition (and (at start (robot-at ?r ?x))
        (over all (left ?y ?x))
      (at start (clear ?y)))
  :effect (and (at start (not (robot-at ?r ?x)))
           (at end (robot-at ?r ?y))
         (at start (not (clear ?y)))
               (at end (clear ?x))))
)

```

Listing B.1. FloorTile Domain.

```

(define (domain satellite)
  (:requirements :strips :equality :typing :durative-actions)
(:types satellite direction instrument mode)
 (:predicates
               (on_board ?i - instrument ?s - satellite)
         (supports ?i - instrument ?m - mode)
         (pointing ?s - satellite ?d - direction)
         (power_avail ?s - satellite)
         (power_on ?i - instrument)
         (calibrated ?i - instrument)
         (have_image ?d - direction ?m - mode)
         (calibration_target ?i - instrument ?d - direction))




  (:durative-action turn_to
   :parameters (?s - satellite ?d_new - direction ?d_prev -
   direction)
   :duration (= ?duration 5)
   :condition (and (at start (pointing ?s ?d_prev))
                   (over all (not (= ?d_new ?d_prev)))
             )
   :effect (and  (at end (pointing ?s ?d_new))
                 (at start (not (pointing ?s ?d_prev)))
          )
  )


  (:durative-action switch_on
   :parameters (?i - instrument ?s - satellite)
   :duration (= ?duration 2)
   :condition (and (over all (on_board ?i ?s))
                      (at start (power_avail ?s)))
   :effect (and (at end (power_on ?i))
                (at start (not (calibrated ?i)))
```

```
                  (at start (not (power_avail ?s)))
            )

  )


  (:durative-action switch_off
   :parameters (?i - instrument ?s - satellite)
   :duration (= ?duration 1)
   :condition (and (over all (on_board ?i ?s))
                         (at start (power_on ?i))
                    )
   :effect (and (at start (not (power_on ?i)))
                  (at end (power_avail ?s))
            )
  )

  (:durative-action calibrate
   :parameters (?s - satellite ?i - instrument ?d - direction)
   :duration (= ?duration 5)
   :condition (and (over all (on_board ?i ?s))
          (over all (calibration_target ?i ?d))
                         (at start (pointing ?s ?d))
                         (over all (power_on ?i))
                         (at end (power_on ?i))
                    )
   :effect (at end (calibrated ?i))
  )


  (:durative-action take_image
   :parameters (?s - satellite ?d - direction ?i - instrument ?m -
   mode)
   :duration (= ?duration 7)
   :condition (and (over all (calibrated ?i))
                         (over all (on_board ?i ?s))
                         (over all (supports ?i ?m) )
                         (over all (power_on ?i))
                         (over all (pointing ?s ?d))
                         (at end (power_on ?i))
                 )
   :effect (at end (have_image ?d ?m))
  )
)
```

Listing B.2. Satelitte Domain.

```

(define (domain driverlog)
  (:requirements :typing :durative-actions)
```

```
  (:types            location locatable - object
    driver truck obj - locatable)

  (:predicates
    (at ?obj - locatable ?loc - location)
    (in ?obj1 - obj ?obj - truck)
    (driving ?d - driver ?v - truck)
    (link ?x ?y - location) (path ?x ?y - location)
    (empty ?v - truck)
)

(:durative-action LOAD-TRUCK
  :parameters
   (?obj - obj
    ?truck - truck
    ?loc - location)
  :duration (= ?duration 2)
  :condition
   (and
   (over all (at ?truck ?loc)) (at start (at ?obj ?loc)))
  :effect
   (and (at start (not (at ?obj ?loc))) (at end (in ?obj ?truck))))

(:durative-action UNLOAD-TRUCK
  :parameters
   (?obj - obj
    ?truck - truck
    ?loc - location)
  :duration (= ?duration 2)
  :condition
   (and
        (over all (at ?truck ?loc)) (at start (in ?obj ?truck)))
  :effect
   (and (at start (not (in ?obj ?truck))) (at end (at ?obj ?loc))))

(:durative-action BOARD-TRUCK
  :parameters
   (?driver - driver
    ?truck - truck
    ?loc - location)
  :duration (= ?duration 1)
  :condition
   (and
   (over all (at ?truck ?loc)) (at start (at ?driver ?loc))
  (at start (empty ?truck)))
  :effect
   (and (at start (not (at ?driver ?loc)))
  (at end (driving ?driver ?truck)) (at start (not (empty ?truck)))
   ))
```

```
(:durative-action DISEMBARK-TRUCK
  :parameters
   (?driver - driver
    ?truck - truck
    ?loc - location)
  :duration (= ?duration 1)
  :condition
   (and (over all (at ?truck ?loc)) (at start (driving ?driver ?truck)))
  :effect
   (and (at start (not (driving ?driver ?truck)))
  (at end (at ?driver ?loc)) (at end (empty ?truck))))

(:durative-action DRIVE-TRUCK
  :parameters
   (?truck - truck
    ?loc-from - location
    ?loc-to - location
    ?driver - driver)
  :duration (= ?duration 10)
  :condition
   (and (at start (at ?truck ?loc-from))
   (over all (driving ?driver ?truck)) (at start (link ?loc-from ?loc-to)))
  :effect
   (and (at start (not (at ?truck ?loc-from)))
  (at end (at ?truck ?loc-to))))

(:durative-action WALK
  :parameters
   (?driver - driver
    ?loc-from - location
    ?loc-to - location)
  :duration (= ?duration 20)
  :condition
   (and (at start (at ?driver ?loc-from))
  (at start (path ?loc-from ?loc-to)))
  :effect
   (and (at start (not (at ?driver ?loc-from)))
  (at end (at ?driver ?loc-to))))

)
```

Listing B.3. Driverlog Domain.

Bibliography

[1] S. Bernardini, F. Fagnani, D. Smith, *Extracting mutual exclusion invariants from lifted temporal planning domains*, 2018.

[2] H. Geffner, *A Concise Introduction to Models and Methods for Automated Planning*, 2013.

[3] S. Russell, P. Norvig, *Artificial Intelligence - A Modern Approach*, 2009.

[4] A. Coles, *Tutorial: Introduction to AI Planning*, https://www.youtube.com/watch?v=EeQcCs9SnhU , 2013

[5] ICAPS Competitions http://icaps-conference.org/index.php/Main/Competitions

[6] M. Fox, D. Long An Extension to PDDL for Expressing Temporal Planning Domains