# DYNAMIC PROGRAM STATE-BASED TESTING

by
GENCO COSGUN

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
May 2022

## ABSTRACT

DYNAMIC PROGRAM STATE-BASED TESTING

GENCO COSGUN

Computer Science and Engineering Master's Thesis, 2022

Thesis Supervisor: Assoc. Prof. Cemal Yilmaz

Keywords: Testing autonomous systems, Dynamic testing, Answer Set Programming

Autonomous systems are systems that can change their behavior in response to unanticipated events during operation. Driverless cars and house cleaning robots are two common examples of such systems. Testing these systems is, indeed, a difficult task due to their autonomous and unpredictable nature. For example, a static test case for testing the adaptive cruise control system (ACC) of an autonomous car in a quite specific scenario may be rendered useless, if the autonomous car makes an unexpected move during the execution (such as, changing the lane, rather than staying on the predetermined lane). In this thesis, to do a better job of testing autonomous systems, we propose a dynamic, program state-based testing approach. At a very high level, the proposed approach takes as input a set of test scenarios to be executed, continuously monitors the current state of the system under test, figures out whether the current state matches with some of the test scenarios or whether some of the test scenarios can be reachable from the current state with the help of a predefined set of actions, if so, takes the actions to dynamically steer the system into the scenario (by using AI planning, if necessary), runs the tests once the system is in the expected state, and validates the results. Our approach is, indeed, a generic approach, which can be applied not only for testing autonomous systems, but also for testing other types of systems. We, in particular, use a declarative logic programming language, namely Answer Set Programming (ASP), to model the state of the system under test, the test scenarios to be executed, the actions to be taken as well as the test oracles. To evaluate the proposed approach, we carried out a number of empirical studies in two different application domains: testing the ACC of autonomous cars and testing a computer game, namely Pacman. The results of our experiments strongly suggest that the proposed approach is flexible enough to address different testing scenarios in various domains.

# ÖZET

DURUM TABANLI DİNAMİK PROGRAM TESTİ

GENCO COSGUN

Bilgisayar Bilimi ve Mühendisliği Yüksek Lisans Tezi, 2022

Tez Danışmanı: Assoc. Prof. Dr. Cemal Yılmaz

Anahtar Kelimeler: Otonom sistemlerin test edilmesi, Dinamik test, Çözüm kümesi programlama

Otonom sistemler, işlem sırasında, beklenmedik olaylara karşılık olarak davranışlarını değiştirebilen sistemlerdir. Sürücüsüz araçlar ve ev temizleme robotları bu tarz sistemler için 2 yaygın örnektir. Otonom ve tahmin edilemez yapılarından dolayı bu sistemlerin test edilmesi bilhassa zor bir işlemdir. Örneğin, otonom aracın adaptif hız sabitleyici (ACC) modülünün test edilmesi için hazırlanmış statik bir test durumu, eğer araç uygulama sırasında beklenmedik bir hareket yaparsa (önceden karar verilen şeriti değiştirme gibi) boşa düşebilir. Bu tezde, otonom sistemlerin testi için, dinamik ve program durum tabanlı bir test yaklaşımı önermekteyiz. Önerilen yaklaşım, girdi olarak bir grup test senaryosu alır, test edilen sistemin durumunu sürekli monitor eder, geçerli durumun test senaryolarına uyup uymadığını ya da geçerli durumdan tanımlanmış bir grup aksiyon ile ulaşılabilen bir test senaryosu olup olmadığını anlar, şayet varsa, sistemi senaryoya yönlendirmek için dinamik olarak aksiyonları alır(gerekirse AI planlama ile), sistem istenen duruma gelince testi koşar ve sonuçları onaylar. Biz, sistemin durumunu, test senaryolarını, olası aksiyonları ve test beklentilerini modellemek için deklaratif mantıksal programlama, bilhassa, çözüm kümesi programlama kullanmaktayız. Önerilen yaklaşımı değerlendirmek için bir grup deneysel çalışma gerçekleştirmekteyiz: otonom araçların ACC modüllerinin test edilmesi ve bir bilgisayar oyunu, Pacman. Deneylerimizin sonuçları önerilen yaklaşımın değişik alanlardan farklı test senaryolarının ifade edilebilmesi için oldukça yeterli olduğu göstermektedir.

## ACKNOWLEDGEMENTS

I want to thanks to my thesis advisor Prof. Cemal Yilmaz for his support, trust and vision, further my loved ones since they were always with me during my years in Sabanci University.

*To my loved ones*

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF ABBREVIATONS

**ACC** Adaptive Cruise Control iv, vi, 1, 2, 5, 6, 26, 29, 31, 35, 38, 39, 40, 41, 42, 43

**AI** Artificial Intelligence . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iv, vi, 7, 19, 43

**ASP** Answer Set Programming . iv, 3, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 19, 30, 35, 40, 41, 43

**CCRM** Car-to-Car Rear Moving . . . . . . . . . . . . . . . . . . . . . . . . . . xii, 26, 29, 32, 38, 39

**CCRS** Car-to-Car Rear Stationary . . . . . . . . . . . . . . . . . . . . . . . . . xii, 26, 29, 32, 38, 39

**Euro NCAP** European New Car Assessment Programme . . . . . . 25, 29, 35, 36, 42

**GVT** Global Vehicle Target . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25, 26, 31

**STRIPS** STanford Research Institute Problem Solver . . . . . . . . . . . . . . . . . . . . . . . . 19

**SUT** System Under Test 2, 3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 27, 30, 32, 34, 35, 36, 40, 42, 43

**TTC** Time-to-collision . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35, 36

**VUT** Vehicle Under Test . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2, 25, 26, 29, 31, 32, 39

# 1. INTRODUCTION

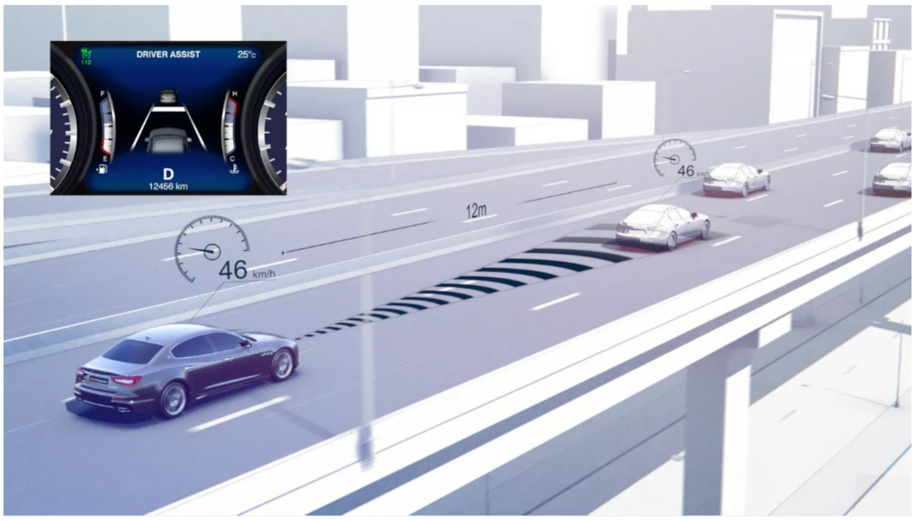
Autonomous systems are taking bigger parts in our lives year by year. Within different devices like house cleaning robots or autonomous vehicles, we face new automation each year. Parallelly, as long as the capabilities of software development increase, new areas for automation will be emerged since each piece of automation means less effort for humanity. Moreover, automated systems are being developed to guarantee safety for human life on different occasions like driver assistance systems.

The more important points automated systems take place, the more crucial their testing becomes. However, the applicability of testing autonomous systems is highly restricted due to their unpredictable natures. It is a difficult task to define test cases that are expecting autonomous systems to match an exact state since they may avoid the expected state due to their decisions. When the nature of autonomous systems is considered, reaching a targetted state can not be guaranteed. For instance, we may want to test the agility of a house robot close to wall corners. More specifically, we may want to observe how the robot will behave when it leads to a wall corner. Innately, the expected behaviour is a full return and avoiding the crash. This would be a reasonable test case since the maneuver ability of the robot is being evaluated under possible conditions. On the other hand, during executions, decisions that the robot makes may not lead to the mentioned wall crossing conditions which will be resulted in a way that we could not test what we aimed to test. This is indeed a waste of resources.

Since the case we are for using for the ongoing example is static, we are not aware of the current state of the system. In other words, we assume that the system is going to behave as we defined in our case. However, if an unexpected behaviour occurs, our case will be failed.

In order to give a more realistic example, we can consider the ACC modules of autonomous vehicles. This module provides the ability of adjust the speed of the autonomous vehicle depending on the leading vehicle in order to prevent any chances of collision. As can be seen from the Figure 1.1 , the vehicle with ACC module or vehicle under test(VUT) is adjusting its speed depending on the leading vehicle.

Figure 1.1 Adaptive Cruise Control, Maserati (2022)



Let's assume that we are aiming to test the mentioned feature with a static test case that expects from VUT to adjust its speed depending on the close by leading vehicle in the same lane. Additionally, our oracle from this case to both vehicles should result in the same speed. However, for the given example, during operation, if the vehicle with ACC module decides to change its lane for any reason, we will not be able to execute the aimed test case since this action is out of our expected behaviour. In this case, we will be wasting our resources.

In order to solve such problems, hereby with this thesis, we propose a dynamic testing approach that monitors the system state continuously while searching for possible test cases that are applicable to the current state. If there is a test case that it is ready to be executed for the current state our approach informs the SUT about the actions in order to apply the detected test case over the system under test, for short SUT. In the remaining of this paper, test cases that their conditions fit the current state will be referred as applicable test cases. Following, the expected result for the detected test case is recorded in order to check whether the state after action executions fits the expectations. In the remaining of this paper, these expectations from a test case execution will be referred as "oracle" or "test oracle"

For the ongoing robot in wall crossing example, a test case condition can be explained as "there should be walls on at least 2 sides of the robot". For the same example, a test case actiom can be defined an action that triggers the robot to calculate a new route and oracle can be defined as "there should be no collision inside the system" or "robot is following a calculated path with a speed higher than 0"

In this work, we present our approach for testing autonomous systems during runtime in a *constant* and *continuous* manner. Hereby, with *constant* we mean our approach is looking for applicable test cases, i.e, not yet executed test cases that fits the current state of SUT, out of the entire test suite as long as SUT is being executed. Moreover, with *continuous*, we mean that our approach does not require any update or interruption of SUT after a test case execution either the case failed or succeeded.

Although, test case examples mentioned in this chapter and explanations are all about autonomous systems, our approach is applicable for other types of systems as well. However, only the problems of testing autonomous systems is in our scope.

The remainder of the paper is organized as follows: in Section 2, the previous work about the testing of autonomous systems and usage examples of ASP related to this topic is discussed. Following, in Section 3, the details of our proposed approach are explained with an example. In Section 4, we shared the evaluation framework together with subject applications and experiment results. Thereinafter we discussed the threats to validity for our experiments, Section 5. Finally, in Section 6, we summarized our contribitions and shared a group of future work ideas.

# 2. RELATED WORK

Even though there are various definitions of *autonomous systems*, (Bradshaw, Hoffman, Woods & Johnson (2013)), in a high level, an autonomous system is different from static systems due to their independency and decision making abilities, (Koopman & Wagner (2016)). Even though there are various examples of autonomous systems like production lines, vehicles or house robots, the common attribute is an object that has the ability of behaving that depends on its own decisions. The philosophy and ethics of such autonomous systems are detailly mentioned in, Charisi, Dennis, Fisher, Lieck, Matthias, Slavkovik, Sombetzki, Winfield & Yampolskiy (2017).

Adversely to the increasing usage areas, testing of autonomous systems are difficult depending on various reasons like non-determinism or complex systems for testing as mentioned in Helle, Schamai & Strobel (2016), which we strongly recommend to the reader.

For instance, in the given work, Lindvall, Porter, Magnusson & Schulze (2017) proposed a model-based testing approach that the test scenarios are implemented with JUnit and the oracles of test cases are decided metamorphically. With other words, rather than declaring an oracle, expected results of a test case are decided after many executions of the similar scenarios. While results that occur after most of the executions are decided as oracle, minor differences are ignored due to the non-determinism of autonomous systems. Our approach includes disparities considering that implemented test case scenarios are not a solution for flexibility problem we mentioned in Section 1 and oracles are expressed from the output of test scenarios.

In the work of Lill & Saglietti (2012) , in order to model-based testing of autonomous systems, different modelling notations are compated and Coloured Petri Nets,(Jensen, Kristensen & Wells (2007)), is decided as the most promising one. Furthermore, an example autonomous system is modelled for evaluation with the selected notion. However, even if this modelling notation express every condition for state transmission, we believe that expressing any autonomous system with finite state based modelling is a difficult and contradictive to the nature of autonomous systems.

In order to overcome this non-deterministic nature, Fisher, Dennis & Webster (2013) proposed an approach depending on the fact that even if the system environment is unpredictable, decision making agent of the autonom object is predictable and applicable for fully modelling. Therefore, for detected criterias, we can assume that this agent will behave the same which means the autonom object in the environment will at least try a deterministic action even though the result is not guaranteed. However for such systems without a decision making agent or for systems that this agent is unavaliable for monitoring, this approach is impracticable.

For the mentioned problem in the previous paragraph, in the work of Eder, Huang & Peleska (2021), a framerwork that combines the module and the system based testing. In the proposed approach, while complete tests are executed over the module level, concurrent and applicable ones are tested on the system level and this decision is made by test agents. On the other hand, the problem we mentioned in previous paragraghp is still alive for testing modules, furthermore this approach has extra test agents for combination of multiple level tests.

We believe that the effort for either modelling the entire environment or the behaviour of an autonom object is unefficient and not flexible due to the non-determinism of such systems. Furthermore, the requirement of a static agent is not a solution for many testing scenarios like the ones requires black-box testing. For instance, one of the agents of a autonom vehicle can be ACC module that manages the acceleration. On the other hand, tester may not be able to monitor the decisions or the reasons of these decisions. Moreover, finally, what important is the behaviour of the autonom vehicle, not the efforts of the ACC module. For these reasons, rather than modelling the behaviours of system, we decide to monitor the system state and accept it as the criteria for test cases. This decision brings flexibility since we do not expect a defined behaviour.

In order to express the monitored state we used ASP, which is a solution finder for declared logic programs(Brewka, Eiter & Truszczyński (2011); Lifschitz (2019)). The detailed reason of this decision will be explanied in the following sections, however ASP has a wide range of usage areas as disccused by Erdem & Patoglu (2018). However the usage of ASP with software testing is relatively new. One of the examples in the literature,Erdem, Inoue, Oetsch, Pührer, Tompits & Yılmaz (2011), proposes that ASP can be used for computing sequence covering arrays which is an array of events that includes all combinations of $t$ events (Kuhn, Higdon, Lawrence, Kacker & Lei (2012)).

Following, we could not find any usage example of ASP with autonomous system testing. Therefore, our approach is different than mentioned previous works and a novel idea even more the first usage of ASP with autonomous system testing.

Recent years, autonomous vehicles or even autonomous driver assist systems like lane keeping or ACC has been in interest of society. The main goal of these systems are reducing the effort of the driver or even saving the driver from collisions. For instance Marsden, McDonald & Brackstone (2001) and Xiao & Gao (2010) defined ACC systems as control mechanism of the longitudal movement of the vehicle with acceleration adjustment depending on the other vehicles and obstacles in order to prevent collisions. One of the most famous usage example of the ACC module is the case of the adjusting the speed of vehicle depending on the closest leading vehicle even if the leading vehicle starts to slow in a second. Due to this interest, there has been various amount of publications for futuristic versions of ACC like (Milanés, Shladover, Spring, Nowakowski, Kawazoe & Nakamura (2013)).

Together with getting more common, testing requirement for autonomous vehicles is becoming a crucial topic. Since autonomous vehicles share the common nature of autonomous systems, similar problems as mentioned in Section 1 like infeasibility is valid for autonom vehicles as well. This validity can be seen from the work of Koopman & Wagner (2016). Many different testing strategies for autonomous vehicles can be seen from Huang, Wang, Lv & Zhu (2016).

# 3. APPROACH

By the proposed approach, we are aiming to obtain a testing tool that monitors the state of an autonomous system and detects applicable test cases for the current state, following to detection, executes them, and reports the results. Hereby with "applicable test case", we mean a test case that either its conditions directly fit the current state of the system or it is possible to steer the system under test to a state that fits.

In our approach, we take input as test scenarios that are aimed to be tested. Furthermore, we are given the states of the SUT in a continuous manner. The goal of the tool here is to *solve* the recent state with the conditions of test cases and detect an applicable test case if exists. Following, if such a case was detected, our tool aims to inform the SUT about the required actions. After the action executions, tool tries to solve the given expectations of the detected case with the current state after the action executions in order to decide whether the case is succeeded or failed.

Obviously, most of the test case detection processes are expected to end up with no applicable test cases due to the nature of autonomous systems. In this case, our approach looks to if it is possible to steer the SUT into a state that fits the conditions of a test case. In order to do so, we opted to use an AI planner that calculates a sequence of actions that the effects of these actions will carry a given initial state to a defined goal one.

This architecture brings the ability to interrupt the system as less as possible with minimum overhead. As mentioned in Section 1, minimum interruption is one of the key concepts of autonomous system testing.

In order to obtain such a tool, as mentioned in previous paragraphs, our tool should be given the current state and desired test suite (all the test cases that wanted to be executed). Therefore, we propose a modeling mechanism. By this mechanism, we aim to express the state and test cases in a flexible and generic way.

As a solution, in the proposed approach, for expressing state and test case information, we used a knowledge representation method called *declarative logic programming*. More specifically, we used ASP, an instance of *declarative logic programming* that aims to compute a set of answers to a given problem under declared constraints, Lifschitz (1999).

## 3.1 Expressing the System State

An integral part of our approach is expressing the system state. Even though we are aware that there are various methods to express the state of systems, in order to obtain a generic structure, we decided to use ASP. Detailed information about ASP can be seen in Section 2.

For our approach, expressing the state of a SUT is actually declaring a set of ASP rules which donates relationships of properties(Lifschitz (2019)). In our modeling strategy, we used these rules as the attribute names in their heads and value declarations in their bodies which indicate the current value of the relevant attribute. These definitions can also be considered as mapping of the current state.

For instance, we can consider the well-known 2D game Pacman as a usage example. In this game, the main object pacman is trying to collect pellets in a maze in order to survive levels while other monsters chase it in the maze. During this struggle, pacman either finishes the pellets and survives the current level or collides with a monster and loses a life. Game is over when pacman loses the last life. With our modeling mechanism, an example state for Pacman can be declared as ;

*pacmanPosition(17,15).*
*pacmanSpeed(normal).*
*pacmanRemainingLives(5).*
*pacmanDirection(right).*
*pacmanMode(normal).*

This is indeed an ASP program in which each row is an ASP rule that defines an example state from the Pacman game. For instance, the first rule with the head "pacmanPosition" means pacman is on (17,15) coordinates with normal speed and remaining lives as 5.

For sake of simplicity, we suppose that pacman may have stop, slow, normal or high speed. Similarly, we suppose that pacman may modes normal, respawn or freak which models usual mode, mode that occurs after a collision with a monster while returning to respawning point, and mode that after pacman eats a power pellet and convert a mode that monsters are vulnerable to pacman, respectively.

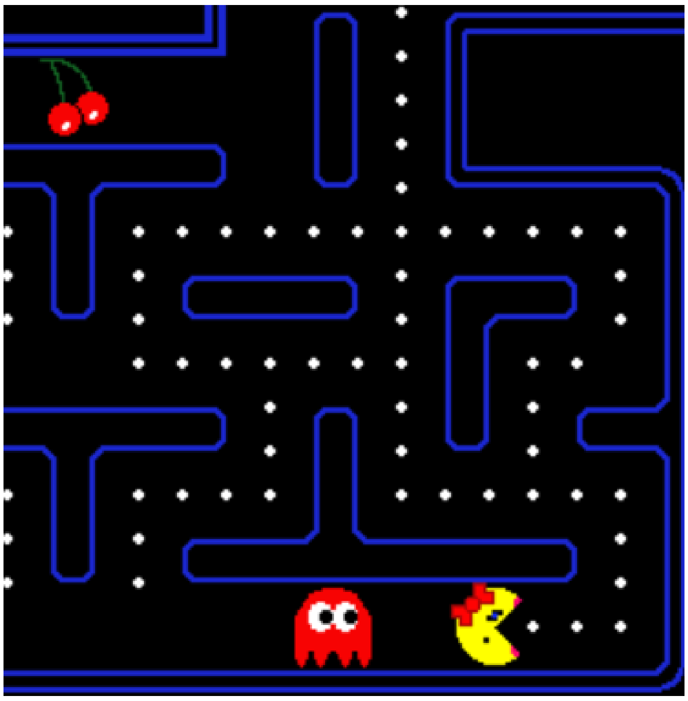
It is important to mention that we designed this state information to be collected from the SUT in a continuous manner. Therefore we can assume that after 3 collected states, pacman will be on (18,15) coordinates and the collected state will be updated accordingly.

The same state can be followed with the information of monsters. Since we have multiple monsters, we can append an identifier to the values;

*monsterPosition(red,14,15).*
*monsterDirection(red,right).*
*monsterPosition(pink,8,7).*
*monsterDirection(pink,down).*

which means red monster is on (14,15) and heading right while pink monster is on (8,7) and heading down. The given state example can be visualized as;

Figure 3.1 State Example For Pacman Game

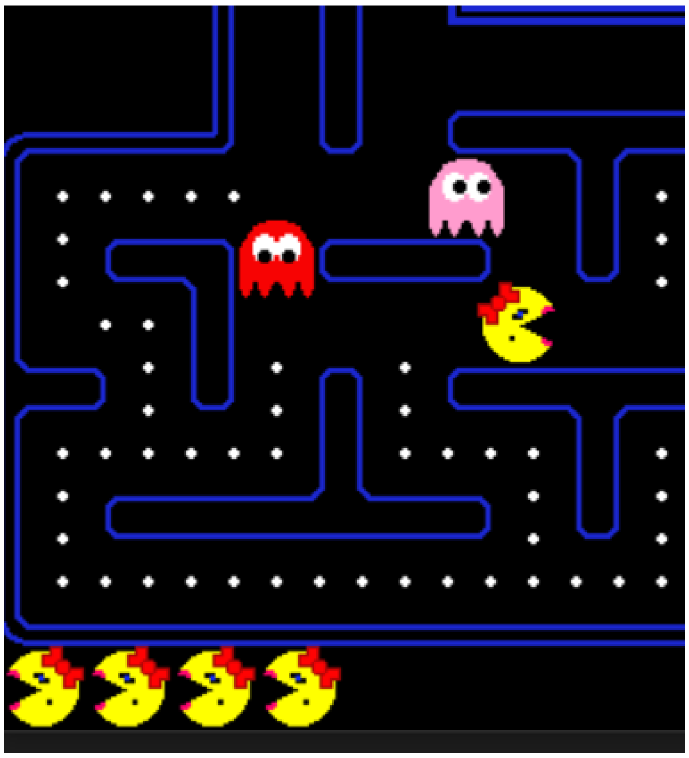


Additional to these basic information, state can include computational knowledge as well. We would like to remind that, as long as it fits with ASP, any information can be modeled depending on the user design. For example, tester may want to add information about the relative positions of monsters for pacman;

*relativeMonsterPosition(red,behind,equal).*
*relativeMonsterPosition(pink,behind,below).*

For one more example, instance state from Pacman game that can be seen from Figure 3.2;

Figure 3.2 Second State Example For Pacman Game



can be modeled with;

*pacmanPosition(8,10).*
*pacmanSpeed(normal).*
*pacmanRemainingLives(4).*
*pacmanDirection(right).*
*pacmanMode(normal).*
*monsterPosition(red,4,12).*
*monsterDirection(red,down).*
*monsterPosition(pink,7,13).*
*monsterDirection(pink,right).*
*relativeMonsterPosition(red,behind,above).*
*relativeMonsterPosition(pink,behind,above).*

Finally, with our modelling strategy, with an ASP program that consists of various number of ASP rules, we were able to express states from the ongoing example.

However, I would like to report that this is only a way to express the state from Figure 3.2. The same state can be expressed with different ways.

## 3.2 Expressing the Test Cases

In our modeling strategy, test cases are expressed as ASP programs as well. In our approach, test cases consist of 3 main components as conditions which defines the conditions that the current state should fit in order to apply the test case, actions that define the required actions in order to apply the detected case over the SUT and oracle which define the expectations from the execution of test case actions.

As mentioned before, we opted to express the test cases with explained components by using ASP. An ASP rule is consist of 2 components which are head and body separated by ":-". Furthermore the semantic of an ASP rule is simple ; *the head is true only if the body is true.*

As related to this semantic, we expressed the conditions of test cases that should be satisfied with the current state to declare that case is ready to be executed, in the body while the actions and oracle are expressed in the head part.

For instance, the following test case that is indeed an ASP rule is for validating the collision case when a monster is chasing pacman with a small distance between them. In case of a close monster coming through pacman, when pacman is stopped, due to the planned collision, we expect pacman to lose 1 live. Actually, with this example sentence, all the required information for a test case, conditions, actions and oracle are all mentioned. Conditions for this case is the existence of a monster that is close to pacman and heading to it, while the action is stopping pacman and the oracle is pacman losing 1 live compared to the beginning state.

*testCase(action(stopPacman()),
oracle(pacmanRemainingLives(ORL),pacmanMode(respawn)),
identifier(pacmanCollisionWithLife(right))) :- pacmanMode(normal),
pacmanPosition(PX,PY), monsterPosition(MID, PX-3, PY),
monsterDirection(MID,right), pacmanRemainingLives(PRL), ORL = PRL -1, not
covered_pacmanCollisionWithLife(right).*

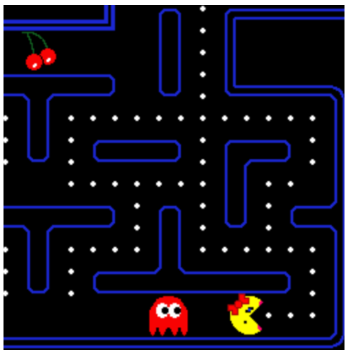
As can be seen from the test case, the head and body part of ASP rule is separated by ":-". Furthermore, the information given in *actions* keyword is the actions needed to be executed for experimenting test case. The information given in *oracle* keyword describes that pacman is expected to lose a life since the variable “ORL” is declared as current remaining lives minus 1 in the body part and pacman should be in respawn mode due to the expected collision. Defining such variables is flexibility that usage of ASP provides to our approach. Following, our approach is defined in a way that declared oracle of a test case is going to be recorded in order to check it whether these expectations satisfies the received state after the execution of actions.

Finally, the information given in *identifier* keyword describes a unique identifier for each case in order let us know which case had been executed successfully and which has not yet. If the recorded oracle and current state after action executions satisfy, the identifier of the applied case will be marked as covered by the appendix keyword *covered_* in order to prevent the detection of the same case again. For instance, for the given case, in case of pacman will have 1 less remaining lives and will be in respawn mode *covered_pacmanCollisionWithLife(right).* after “stopPacman()” execution, this case will be marked as covered. Therefore, our last condition is the non-existence of this rule.

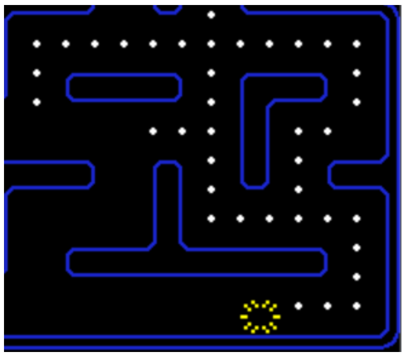
Since we expect the declared actions to be executed over the SUT, our approach assumes that implementation of these actions is given on the system side. For instance, our tool expects that there will be a corresponding call for “stopPacman()” action in the SUT. However, this implementation is out of the scope of the proposed approach.

Furthermore, the conditions written in the body part declare the suitable values for the relevant test case execution. Our first condition for this test is that pacman should be in normal mode which means vulnerable to colliding with monsters. The following condition is the occurrence of a monster in the same horizontal line, however 3 coordinates behind in vertical line than pacman since pacman is expected to be on (PX,PY) coordinates while a monster is expected to be on (PX-3,PY) coordinates. Moreover, this monster in behind should follow the right direction because this is where pacman is in 2D coordinate system. Furthermore, we need the current remaining lives of pacman in order to compute expected lives. Obviously, if pacman stops at this moment, just like declared in the test case actions, collision is inevitable.

The Figure 3.3 demonstrates the steps of a test case execution. As can be seen from the Figure 3.3a, SUT has a state that fits the conditions of the ongoing test case example. At this moment, our tool detects the applicable test case and informs the declared action that is responsible for stopping pacman. Foloowing, Figure 3.3b shows the state after the action execution, as can be seen, the collision has happened

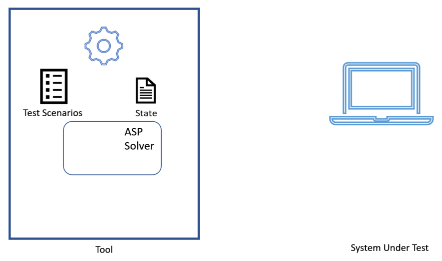


(a) Current State that Fits the Ongoing Test Case Example

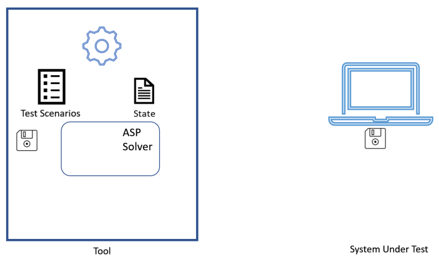


(b) Current State after the Action Execution

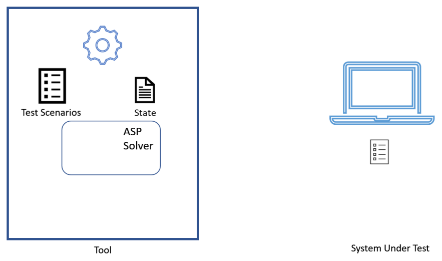
Figure 3.3 Test Case Detection



(a) Initial Test Case Detection Step



(b) Detected Test Case



(c) Steering the SUT into a Target State

Figure 3.4 General Flow of Our Application

as expected from the test case.

## 3.3 General Flow of the Proposed Approach

So far, we discussed how we expressed the required information for our approach. In this section, I will be explaining the general flow of the proposed approach. In the Figure 3.4, this general flow can be seen.

In the Figure 3.4, our tool is described as a rectangle with a ASP solver inside of it while SUT is described as a icon of a laptop. As mentioned, our tool takes the test scenarios as input and continuously receives current states from the SUT.

As can be seen from Figure 3.4a, each time our tool receives a current state, it looks if there is a test scenario that is ready to be executed for the current state. In case of a detected applicable test scenario, our tool sends the declared actions of the detected case to the SUT and records the test oracle in order to check if the test execution resulted as expected as can be seen from Figure 3.4b.

However, if there is no applicable test scenario inside of the entire set, our tool looks if it is possible to calculate a sequence of actions that will steer SUT into a state that fits the conditions of a test scenario as can be seen from Figure 3.4c. In this case, after the execution of each action inside of the calculated sequence, we expect to receive the current state and run the planner again since SUT is an autonomous system and may behave outside of our expectations.

If there is no applicable scenario inside of the entire set and it is not possible to calculate a sequence of actions to steer the system, our tool stays idle in order to avoid interrupting the system. In this case SUT will keep its execution and send current states to our tool for further test scenario detections.

Here, the mentioned efforts of our tool such as searching for an applicable test case or searching for a sequence of actions or oracle control with the current state after action executions are done by an ASP solver. Therefore, with a single knowledge representation method, we obtained a flexible structure. I would like to note that, since ASP is our only requirement *as long as test scenarios for an application and system state of it can be expressed with ASP, our approach is applicable.*

When we consider the ongoing state example, refer to the Figure 3.1, given test case in Section 3.2 is applicable for red monster since it is on coordinates (14,15) while normal pacman is on (17,15) and red monster is heading to right direction. In this case, our tool reports an array with a single action in it, "stopPacman()" and records the oracle, "pacmanRemainingLives(ORL),pacmanMode(respawn)" for following comparisons with current states.

Following we expect our tool to receive the state of the SUT after the execution of "stopPacman()" and check if this state satisfies the recorded oracle. Finally, at this point, our tool requires another information from the user which is the "system constraints". We expect from tester to model the constraints of attributes. For instance, without system constraints, the oracle "pacmanMode(respawn)." would be acceptable even if we have "pacmanMode(normal)" in the state. Therefore tester should give the constraint that "pacmanMode()" must have a single value with;

*:- #count{MODE:pacmanMode(MODE)} != 1.*
*:- #count{PRL:pacmanRemainingLives(PRL)} != 1.*

With such constraints, our tool is able to filter out states and oracles that do not fit which means our tool detects failed test case executions. In such a case, our tool removes the oracle and keeps its execution by looking for applicable test cases for the current state.

Inevitably, we may have attributes with may take multiple values, as a result we can not accept that all attributes should have a single value at a time. Therefore, we expect from user to define constraints in a way that tester wants again. As mentioned before, given constraint example is just a one way to do this, therefore other ways to declare such constraints are applicable as well.

As mentioned before, if the oracle satisfies the state, the applied test scenario is marked as covered. Therefore, in the following test case detection iterations, this case will not be detected again.

Following, our tool keeps the execution with the same state as the test case detection phase. This structure does not include any effect to the SUT depending on the result of a test case. As mentioned in Section 1, this is a solution that our tool proposes for the problems of testing autonomous systems.

When these system constraints are considered, oracle control phase of our tool can be explained as searching if the recorded oracle, current state and declared system constraints fit.

For now on, we discussed an initial version of our tool with a test case example. This case, tests the collision case while a monster chasing pacman from the left side. However, tester may want to test this case for all possible monster directions. With the help of ASP, rather than modeling 4 test cases only with a small change of "monsterDirection" attribute, for values right,left,up and down, tester can declare rules such as "crashable" for possible directions and use them like macros in test case conditions. For instance, the same test case can be modeled like;

*testCase(action(stopPacman()),*
*oracle(pacmanRemainingLives(ORL),pacmanMode(respawn)),*
*identifier(pacmanCollisionWithDirection(MDIR))) :- pacmanMode(normal),*
*crashable(MDIR), monsterDirection(MID,MDIR),*
*pacmanRemainingLives(PRL),*
*ORL = PRL -1, not covered_pacmanCollisionWithDirection(MDIR).*

Here with, "monsterDirection(MID,MDIR)", tester makes this test case applicable for all directions a monster can have. Similarly, the identifier and coverage criteria is updated as well. For crashable rule on the other hand, tester can define rules for all possible directions;

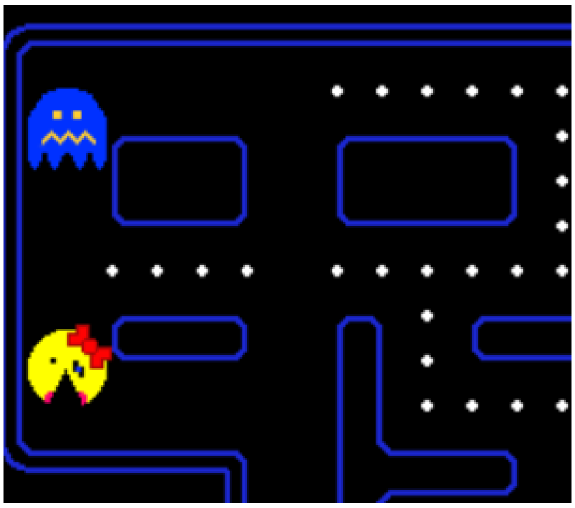
*crashable(right):-pacmanPosition(PX,PY),monsterPosition(MID,PX-3,PY), monsterDirection(MID,right).*
*crashable(left):-pacmanPosition(PX,PY),monsterPosition(MID,PX+3,PY), monsterDirection(MID,left).*
*...*

With these declarations, for all 4 possible directions a monster can have, tester was able to model a single test case. Morever, since this case can be detected for all possible directions once -if all detections end up with successful executions-, tester defined the coverage criteria as well. Moreover, in the future, tester may want to test the collision case while pacman is in freak mode, tester can use the same crashable rule as;

*testCase(action(reversePacman()), oracle(pacmanRemainingLives(PRL)), identifier(freakPacmanCollisionFrom(MDIR))) :- pacmanMode(freak), crashable(MDIR), monsterDirection(MID,MDIR), pacmanRemainingLives(PRL), not covered_freakPacmanCollisionFrom(MDIR).*

Expected conditions for this case can be described as there should be close monster in the reverse direction of freak Pacman as can be seen in Figure 3.5;

Figure 3.5 Freak Pacman Collision Case



## 3.4 Time-out Mechanism

So far, our tool was expecting the effect of the actions in the following state. As described in Section 3.3, our tool is checking the recorded oracle with state occured just after the action executions. However, this is not always the case for autonomous systems since there can be required duration for the effects of the actions to be appeared. In this chapter, our solution to this duration problem is explained with the ongoing freak pacman collision example. In order to be more explanatory, this case has conditions that pacman should be in freak mode while there is a crashable monster. When these conditions fit, pacman should reverse its direction and forwards while our expectation is that pacman will lose no lives after collision.

Previosuly, we assumed that in each iteration, pacman and monsters will proceed 3 coordinates depending on their direction with normal speed. However, this may not be case depending on the execution of the SUT. Therefore, we can not restrict that we expect this oracle in the following state. Rather than this, tester can define a time-out duration for the oracle control of a case. This time-out duration works as a time buffer. When declared, our tool expects the given oracle at least once during the declared period. In each iteration, our tool receives the current state and checks oracle with the current state. In case a satisfiable state, then the case is succeeded. However, unsatisfability inside the time-out period keeps our tool idle and makes it wait for the following state. As long as tool receives at least once satisfiable state with the oracle within the declared duration, then the case is succeeded. The detected case fails only if there has been no satisfable state during timeout.

The freak pacman test case with a timeout can be declared with the keyword "locked()" including the time-out parameter;

*testCase(action(locked(3), reversePacman(), setPacmanSpeed(high)),*
*oracle(pacmanRemainingLives(PRL)),*
*identifier(freakPacmanCollisionFrom(MDIR))) :- pacmanMode(freak),*
*crashable(MDIR), monsterDirection(MID,MDIR),*
*pacmanRemainingLives(PRL),*
*not covered_freakPacmanCollisionFrom(MDIR).*

Now, this case is expecting at least a single state with pacman losed no live in 3 seconds.

## 3.5 Global Rules for System Environment

Even though this time-out mechanism improved the practicibality of our tool, ongoing example case can be further imporoved. Still, only oracle the test has that pacman is expected to lose no lives. However, due to the collision with freak pacman, we are also expecting a dead monster even though we do not care which one.

In order to apply global constraints like expecting at least one dead monster inside the total system envrionment(entire maze for our example) or expecting no close monsters to pacman inside the total system envrionment, tester can define filters and *import* them into the oracles of test cases.

The goal with these constraints are not the expectations of single objects but expectations over the general state of the SUT. For instance, a tester can declare expectations of all the objects should have same speed or same mode.

For the ongoing freak pacman case example, in order to declare expectation of at least one dead monster which means in spawn mode, tester can define the global rule with indicating a group name and a pointer name of the rule;

*%@ monsterState atLeastOneDeadMonster*
*:-not monsterMode(_,spawn).*

This rule with pointer "atLeastOneDeadMonster" filters any state with no monster in spawn mode. Tester can append rule to the test case as ;

*testCase(action(locked(3), reversePacman(), setPacmanSpeed(high)),*
*oracle(import(monsterState,atLeastOneDeadMonster),pacmanRemainingLives(PRL)),*
*identifier(freakPacmanCollisionFrom(MDIR))) :- pacmanMode(freak),*
*crashable(MDIR), monsterDirection(MID,MDIR),*
*pacmanRemainingLives(PRL),*
*not covered_freakPacmanCollisionFrom(MDIR).*

Again, this is just a one way to declare such filter for the example. Therefore other ways for this declaration can be used.

## 3.6 Planner Feature

So far, our modeling strategy can express comprehensive test cases with clearly defined conditions, actions and oracle. This modeling ability will be evaluated in a detailed manner in Section 4. However, our tool still is not flexible for experimenting with autonomous system. As discussed in the Section 1, these systems are non-deterministic, therefore declaring conditions for test cases may not be efficient. For instance, in the ongoing example, we declared a condition such that pacman should be in freak mode. However, depending on the game dynamics, this condition is unlikely to happen. Obviously, this problem is a limitation for our test throughput performance.

As a solution, in this thesis, we propose using an AI planner implemented by ASP. We aim to compute necessary actions to steer the SUT from an initial state to a state that fits the conditions of a selected test case. Such planner can compute necessary sequence of actions to be taken for steering the system from an initial state to a goal one. Here, what planner takes as input are possible set of *actions*, their *effects* and *non-effects* over the attributes of state. While *effects* mean the modifications that actions make over the attributes of state, *non-effects* declares under what actions the attributes will remain same.

While there are many planners in the literature, we decided to use a planner type called STRIPS,Fikes & Nilsson (1971). These type of planners, assumes that given actions are deterministic and the world is static. Such planners try possible actions in an iterative manner until try reach either the goal state or the given threshold value for maximum actions,Haslum, Lipovetzky, Magazzeni & Muise (2019). By each iteration, these planners update their current state with the declared effects of actions and check if they fit the goal one. Unless fits, they keep trying the given actions until they reach maximum number of actions.

Declaring a threshold value for maximum number of actions is crucial for our tool since it is very common to receive a state from SUT that is non-convertible to a goal state of a test case. Therefore, there is still an opportunity that our tool stays idle since a test can not be detected with neither with a planner nor directly.

Since we aim to use this planner in autonomous systems, we designed our approach in a way thay after the execution of each action inside of the calculated sequence, the planner run again in order to calculate the plan again. With this design, we aimed to cover the possible changes that may happen due to autonomous behaviour.

The AI planner that implemented by ASP, capsulates each given attribute with a new rule name, "holds" and appends an external variable which declares the iteration number. For instance, the rule *pacmanMode(normal).* in the initial state is being converted to *holds(pacmanMode(normal),*0*).* and will go on as *holds(pacmanMode(normal),*1*).* to further iterations as long as planner tries new actions. Lets assume that in the following action, planner picked an action that effects the value of "pacmanMode" attribute, what we will obtain is *holds(pacmanMode(freak),*2*).* .

In this thesis, we designed the planner as an optional input even though we believe it is a powerful feature which boosts test throughput and applicability of our tool. We will be evaluating this belief in the Section 4.

Furthermore, the effects of our planner will be explained by an example with the test case ;

*testCase(action(locked(2), setSpeed(200), forwardPacman(2), setSpeed(50)),*
*oracle(import(monsterState,atLeastOneDeadMonster)),*
*test(pacmanChasesFreightMonFrom(PDIR))) :-*
*pacmanDirection(PDIR), pacmanMode(freak), freakPacmanCrashable(),*
*not covered_pacmanChasesFreightMonFrom(PDIR).*

This test case applies the scenario of a freak pacman detects a monster on its way and accelerates in order to catch it. Defined oracle is that we expect at least one dead monster in the environment.

For this example, we have the planner;

*{*
*setPacmanMode(freak,t);*
*setPacmanMode(respawn,t);*
*}* =1.

*holds(pacmanMode(freak),t) :- setPacmanMode(freak,t).*
*holds(pacmanMode(respawn),t) :- setPacmanMode(respawn,t).*

*holds(pacmanMode(MODE),t) :- holds(pacmanMode(MODE),t-*1*), not setPacmanMode(t).*

This planner has 2 actions that defined in the first rows for whether chancing the mode of pacman into freak or respawn. Following 2 rows however, defines the effect of such actions to the SUT as if "setPacmanMode(freak,t)" action is calculated for the iteration *t*, then the system should have "pacmanMode(freak)" in the state of

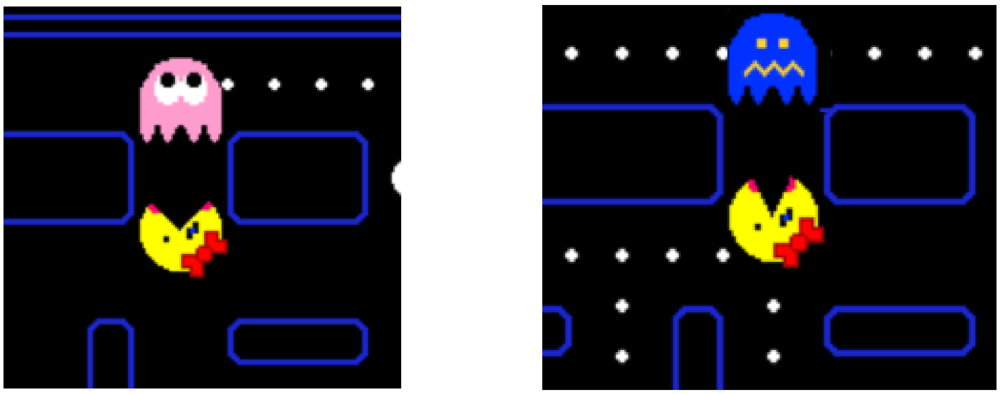

(a) Current State that Does Not Fit the Test Case (b) Current State after Planner Output that Fits the Test Case

Figure 3.6 General Flow of Planning

the SUT. The last rule describes the non-effects of such actions as “pacmanMode” attribute will not be changed unless a “setPacmanMode(freak,t)” exist.

Again, the described planner is just a way to do this and there can be various ways to define such planner.

The Figure 3.6 demonstrates the effects of planner in our tool with an example test case. In Figure 3.6a an example state that does not fit to the conditions of the ongoing test case example. Even though there is a monster on the way of pacman, pacman is not in the freak mode. Since the test case and current state do not satisfy, our tool checks if it possible to calculate a plan for the current state. As can be seen from the given planner has possible actions to change the mode of pacman and actually one of these possible actions is responsible for changing the mode of pacman to freak, as defined condition. Therefore, when the current state is send to the planner, it outputs the action which is responsible for chancing the mode of pacman to the freak,“setPacmanMode”, and sends this action to the SUT. Following, 3.6b shows the current state that fits the conditions of the test case after the exeuction of planned action.

However, usage of planner comes with an limitation which is using variables “MDIR” or “MID” in the test case conditions. Since planners are expecting an exact goal state which is the conditions for a test case in our approach, they can not calculate values of such variables like an usual asp solver. We overcome this problem with writing more generic rules, for instance rather than “crashable(MDIR)”, “crashableWithMonster(pink)” can be used. Naturally, the conditions for the this rule should be declared for the planner again. As we mentioned, we avoided using a variable for monster identification and used the identifier itself direclty. Although this issue seems to limit our modeling feature, we believe that we can still model any test case.

This belief will also be evaluated detailly in the following section.

If we return to the ongoing example, we can convert the test case avaliable for a planner with removing the variables in the body;

*testCase(action(locked(2), setSpeed(200), forwardPacman(2), setSpeed(50)),*
*oracle(import(monsterState,atLeastOneDeadMonster)),*
*test(pacmanChasesFreightMon(pink))) :-*
*pacmanMode(freak), crashableWithMonster(pink),*
*not covered_pacmanChasesFreightMonFrom(pink).*

Finally, with the help of a planner, we were able to reach a fitting state from an initial state that did not fit. We believe that the usage of planner will boost the test throughput performance of our approach. So far the planner we used a basic version of planner as mentioned in the beginning of this section. However since we run the planner after execution of each action in the calculated sequence by the planner, a basic version planner is also sufficient.

## 3.7 High Level Implementation

Following, the pseudo code of the explained approach can be seen;

**Algorithm 1:** Implemented Tool

```
currentState = self.systemUnderTest.getState();
if self.isThereRecordedOracle() then
    if self.isSystemLockedByOngoingTestCase() then
        if self.solveOracleWithCurrentState() then
            self.recordCurrentIndetifierAsExecuted(self.currentIdentifier);
            print("Current case hase executed successfully");
            self.updateToolAttributes();
        else
            if self.areWeStillInLockedDuration() then
                return None;
            else
                print("Current case hase failed");
                self.updateToolAttributes();
            end
        end
    else
        if self.solveOracleWithCurrentState(currentState) then
            self.recordCurrentIndetifierAsExecuted(self.currentIdentifier);
            print("Current case hase executed successfully");
            self.updateToolAttributes();
        else
    end
else

end
if self.isThereSuitableTestCase(currentState) then
    self.testIdentifier, self.testActions, self.testOracle =
     self.parseAvailableTestCase();
    return self.testActions;
else
    if self.ifAPlanCanBeCalculated(currentState) then
        self.testIdentifier, self.planActions, self.testActions, self.testOracle =
         self.calculatePlan();
        return self.planActions + self.testActions;
    else
end
return None;
```

If it is needed to be explained by words, just after receiving the current state, our tool checks if there is recorded oracle from the previous iteration in order to decide either detected case is failed or not. Furthermore, the tool checks if the system is locked. In case of locked status, even though the current state is not satisfiable with the recorded oracle, tool stays idle. It is sufficient to obtain a single state that is satisfiable with the oracle in order to decide the test case as succeeded. If there is no lock mechanism activated or it is timed out, obviously, tool tries to solve the oracle and current state to decide either the case has succeeded or failed.

Following tool solves the test suite with the received state and checks if there is an applicable test case. If detected, tool configures the lock mechanism, records the oracle and returns the declared actions to the system. However, if no applicable test case can be detected, tool asks to planner if the conditions of a selected test case may be satisfiable with the current state under some modifications. In this case, if planner is able to create a plan that modificates the received state to a final state that is satisfiable for the selected test case, it returns a series of required actions to the tool. Following tool appends the test case actions to received actions and returns to the system.

Only case that our tool stays idle is the case that there is no directly applicable test case and planner can not find a plan that steers the system under test from a current state to an applicable one. In this case, our tool waits for following states.

# 4. EXPERIMENTS

In this chapter, first the subject applications that we experimented our approach with are explained. Following, we explained our evaluation framework together with criterias of our evaluations. Finally, the results of our experiments and a brief discussion about these results are shared.

## 4.1 Subject Application

We evaluated the proposed approach with 2 different applications. The first application is an open source game that a main object is trying to avoid a group of monsters while eating pellets to gain score and survive levels, called Pacman. For this application, we used an open-source implementation by Richards (2022). The final subject application is however a realistic automobile simulation, Esmini (2022). This simulation, scenario player as they called themselves, is able to manage a designed environment that can include vehicles, road curves or pedestrians. For the simplicity of the evaluations, we used a road with 3 lanes with no obstacles like pedestrians and 2 vehicles as GVT and VUT.

While Pacman is developed with Python, Esmini is developed with C++. However, in order to obtain comphrensive evaluations, both applications had modifications. For example, we modificated the Pacman application as Pacman moves around the maze by itself depending on random decisions rather than user decisions. Moreover, we implemented necessary Python scripts to execute the source code of Esmini which was written with C++. Following, we implemented a switch based structure to process automobiles either with their own decisions or with given implementation.

For both applications, we designed effective test suites that includes reasonable test cases. As a matter of fact, we aimed to model the official Euro NCAP tests cases for ACC module of Esmini,(EuroNCAP (2020)). Out of the defined 5 types of test cases for ACC, 3 most well known ones as CCRS, CCRM and cut-in are selected. All of these test cases consist 2 vehicles as a leading one which will be called as global vehicle, for short GVT and a following autonomous one with ACC module which is the vehicle under test, for short VUT. In the following of this report, vehicle with ACC module will be referres as “vehicle under test”, VUT while the leading car will be referred as global vehicle or leading vehicle, GVT.

While CCRM test cases refer situations as the vehicle with ACC approaches a global car with a relatively higher speed while CCRS test cases refer to situations as the leading global car instantly stops. On the other hand cut-in cases refers to bit more complicated situation; global car which is on the adjacent lane of VUT, operates a lane change move through the VUT which creates a collision risk. For all test cases, we expect VUT to obtain the speed of the global car and avoid any collision.

For Pacman game, we designed a test suite that covers the features of the game like eating usual or power pellets or trying to return and run approach when a monster is detected from the previous direction. Following this test suite is accepted as our goal suite and used as a criteria for modeling evaluations.

## 4.2 Operational Model

We used clingo as the answer set solvers for our tool,see 3.4 and **??**, in order to solve the current state with given test suite or oracle controls, Gebser, Kaminski, Kaufmann & Schaub (2019). Clingo is tool which is a combination of a grounder that transforms user-defined logic programs into a variable-free format and a solver that computes possible answers for logic programs, see of Potsdam (2022).

In order to execute Esmini with Python scripts, which was written with C++, we used *ctypes* library of Python. Also for time calculations we used the built-in time library of Python.

All the experiments were carried out on a 2.1 GHz Intel Core i5 CPU computer with 16 GB of RAM running 64 bits Windows 10 as the operating system.

## 4.3 Evaluation Framework

Our experiments aims to evaluate if such a testing approach is able to model comprehensive test cases for autonomous systems, monitor the system while either detecting or planning applicable test cases and execute given test cases. Since our approach is an end-to-end testing strategy as mentioned in Section 3, it combines *modeling* and *execution* under a single roof. Therefore, we carried out distinct evaluations for these components.

Moreover, during these experiments, we evaluated the generalizability of our approach. As mentioned in Section 1, we aimed to obtain a tool that is useful for all autonomous systems. Even though we used 2 subject applications, when the features of these systems considered, the advantages and limitations of our approach is evaluated in a general manner.

### 4.3.1 Modeling Evaluations

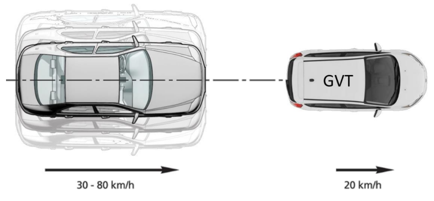
To experiment the modeling abilities of our approach, first, we aimed to model a test suite for Pacman game and model it with using our format, mentioned in Section 3.

From the Table 4.1, test case explanations and related condition, action, oracle and coverage criteria information can be seen. To be more descriptive, besides the 1-2 sentence explanation, table can be read as if the given conditions fit to recent state and the given actions are executed, we expect the SUT to obey the given conditions. Moreover, coverage criteria column depicts the required criteria of value combinations for covering a test case. For instance, by 4 directions, we mean that the relevant test cases should be successfully executed for 4 possible directions of Pacman as left, right, down or up.

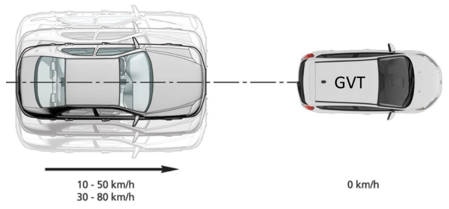
In order to challange our approach, additional to basic test cases like *when pacman collides with a monster in normal mode, pacman converts to spawn mode*, we modelled cases like *when pacman faces a monster coming from opposite direction, if pacman jumps just before the collision, pacman loses no lives.* By complex test cases, we mean cases with more spesific conditions that requires more comparisons. Following, we experiment how suitable our approach is to cover this suite by calculating the ratio of the modelled test cases.

Table 4.1 Test case explanations of goal test suite of Pacman

| test case explanation | conditions | coverage criteria |
|---|---|---|
| pacman crashes a wall while a monster chasing it, pacman stops and collision happens | conditions | actions |
| | there should be a monster chasing pacman while pacman has direction stop | |
| | oracle | coverage criteria |
| | pacman loses 1 live | 4 directions & 4 monsters |
| pacman being chased by a monster, pacman returns to opposite direction and collision happens | conditions | actions |
| | there should be a monster chasing pacman | pacman returns |
| | oracle | coverage criteria |
| | pacman loses 1 live | 4 directions |
| pacman being chased by a monster, pacman goes freak and returns to opposite direction and accelerates, collision happens | conditions | actions |
| | there should be a monster chasing pacman | pacman returns, goes freak, accelerates |
| | oracle | coverage criteria |
| | pacman loses no lives, there should be at least one dead monster | 4 directions |
| pacman eats power pellet | conditions | actions |
| | there should be a power pellet on the direction of pacman | |
| | oracle | coverage criteria |
| | there should be no monsters in normal mode | 4 directions |
| pacman eats fruit | conditions | actions |
| | pacman detects a fruit on its direction | |
| | oracle | coverage criteria |
| | there should be no fruit in the maze | 2 directions |
| pacman run away from a monster coming from opposite direction, pacman runs away, no collision happens | conditions | actions |
| | there should be a monster approaching to pacman from opposite direction | pacman returns & accelerates |
| | oracle | coverage criteria |
| | pacman loses no lives, no close by monsters | 4 directions |
| pacman crashes to a wall | conditions | actions |
| | there should be a close wall on the direction of pacman | |
| | oracle | coverage criteria |
| | pacman has direction stop | 4 directions |
| pacman detects a freight monster on the current direction, accelerates, collision happens | conditions | actions |
| | there should be a freight monster on the direction of pacman | pacman accelerates |
| | oracle | coverage criteria |
| | there should be at least one dead monster | 4 directions |
| pacman detects a freight monster close by, accelerates, collision happens | conditions | actions |
| | there should be a freight monster close to pacman | pacman accelerates |
| | oracle | coverage criteria |
| | there should be at least one dead monster | Any direction |
| pacman detects a freight monster on the other side of the maze portal, accelerates and collision happens | conditions | actions |
| | there should be a monster on the other side of a portal when pacman is also close to the same portal from the other side | pacman accelerates |
| | oracle | coverage criteria |
| | there should be at least one dead monster | 4 directions |
| pacman stops while a monster approaching from a diagonal direction, loses an live | conditions | actions |
| | there should be a close monster that is heading pacman from horizontal/vertical when pacman is heading vertical/horizontal | pacman stops |
| | oracle | coverage criteria |
| | pacman loses 1 live | verticalhorizontal |
| pacman detects a monster coming from opposite direction, jumps over it, loses no live | conditions | actions |
| | there should be a monster approaching to pacman from opposite direction | pacman jumps |
| | oracle | coverage criteria |
| | pacman loses no lives | 4 directions |
| pacman with 1 remanining live, detects a monster chasing it and stops, collision happens, pacman obtains 5 lives | conditions | actions |
| | there should be a monster chasing pacman with 1 remaining live | pacman stops |
| | oracle | coverage criteria |
| | pacman has 5 remaining lives | Any direction |
| pacman with 1 remanining live, detects last pellet in the maze on the current direction, collision happen with the last pellet, pacman obtains 5 lives | conditions | actions |
| | pacman detects the last pellet in the maze | |
| | oracle | coverage criteria |
| | pacman has 5 remaining lives | Any direction |



(a) Explanation of CCRM Test Cases



(b) Explanation of CCRS Test Cases

Figure 4.1 Visual explanation of CCRM and CCRS test cases, both taken from EuroNCAP (2017)

Following, for the same evaluation, we aimed to model a test suite and a planner for ACC module of autonomous vehicles. ACC module is a driving assistant module that adjusts the speed of the vehicle depending on the speed of other vehicles as described in Section 1. If the distance between the vehicle with ACC gets lower than a threshold value, ACC forces the vehicle to adjust its speed depending on the leading vehicle in order to prevent collisions.

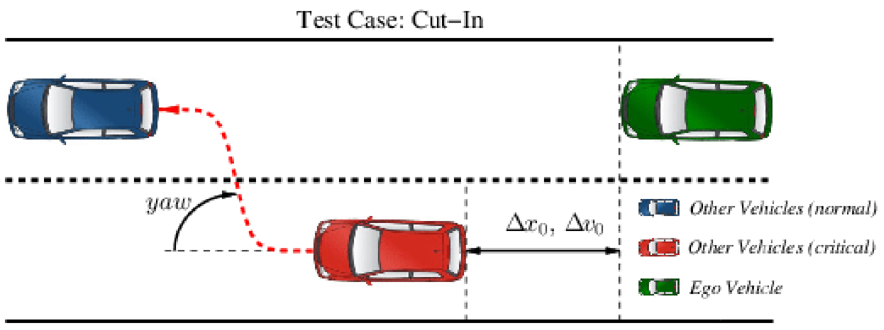
For testing of ACC module, we accepted the test cases from Euro NCAP documentation as the goal suite. Test cases and detailed information can be seen from EuroNCAP (2020). By this selection, we aimed to evaluate the modeling ability of our approach with real test cases defined for the assessment of actual vehicles. In order to provide detailed explanations of these test types;

**CCRM** : For this type of testing, the conditions are the existence of a leading vehicle within a close distance with a relatively lower speed than the approaching VUT. Visual explanation of this can be seen from Figure 4.1a.

**CCRS** For CCRS testing, the conditions are similar with the CCRM cases, however the condition for global vehicle here is a full stop rathen than a relatively slower speed than VUT. Visual explanation of this can be seen from Figure 4.1b.

**Cut-in** Cut-in test cases are relatively complex than the previous ones. Here, the global vehicle should be initialized in an adjacent lane of the VUT, later the global car should pass the lane of the VUT with a relatively lower speed. General expectations are the same with previous test types, VUT should adjust the speed of the global vehicle.

Figure 4.2 Visual Explanation of Cut-in Test Cases From Zhou & del Re (2018)



From the Figure 4.2, the red vehicle is chancing lane with a yaw angle and cutting the way of green vehicle which is the ego vehicle or vehicle under test.

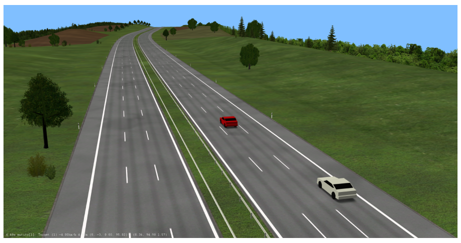
Detailed information about these test types can be found in EuroNCAP (2020).

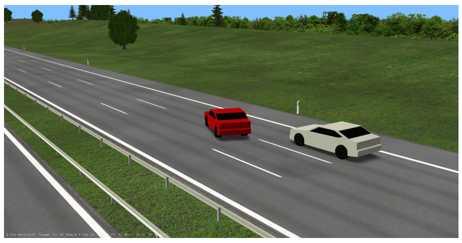
By the amount of test cases from goal suites we could and we could not model, we aimed to evaluate the modeling abilities of our approach. We calculated a metric as the ratio of modelled test cases to total number of test cases in goal suite. Moreover, by the amount of time and effort we spend while modeling these cases and planner, we aimed to evalute our efforts. Imperiously, we reported this metric as a duration. In order to evaluate this, I, who has been working with ASP for about 6 months now, reported the amount of time that it takes for me to model the goal suites and the planner. Even though this is a subjective evaluation since I am evaluating depending on my experience, we though this might helpful in order to provide a first impression for readers.

Note that, only for this evaluation, I attempted to modelled these from the most understandable level. For instance, for setting a speed to a vehicle with given ID, rather that using *setSpeed(gvt1,20)*, I used *pressAcceleration(gvt1)* which has the effect;
*holds(gvt_speed(GVTID,OLDSPEED+5),t) :- pressAcceleration(GVTID), holds(gvt_speed(GVTID,OLDSPEED),t-1).*
that increases the speed in the moment $t-1$ by 5.

### 4.3.2 Execution Evaluations

To experiment the execution abilities, we calculated the amount of overhead that is created by our tool to both SUT's. Briefly, we calculated the average amount


(a) Inital State For Cut-in Test Case


(b) Actions for Cut-in Test Case

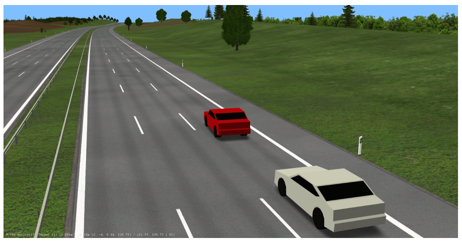

(c) Expected Final State For Cut-in Test Case

Figure 4.3 Steps of a Cut-in Test Case

of time spended by our tool for each executed case. Basically, we measured the time spend during test case detection, oracle comparison and plan calculation steps. Therefore, to accomplish this calculation, we implemented a version of our tool which the code space between the input -system state- and the output -sequence of actions- is surrounded by time calculations.

We carried out many executions of modelled test suites and reported the durations for each executed case. Obviously, we expect this average overhead to be minimum even though we are aware that it is not possible to reach 0 ms overhead.

For instance, the Figure 4.3 demonstrates the steps of a test case as initial state, actions and expected state. For inital state, the red vehicle, GVT is in the adjacent lane of a the VUT and the distance between them is sufficient to test the effects of ACC module, see Figure 4.3a. Following, red vehicle takes a turn and passes to the lane that vehicle under test is in with a relatively slower speed, see Figure 4.3b. Finally, parallely to our declared oracle, white vehicle adjusts the speed of the leading vehicle due to the ACC module, see Figure 4.3c.

Moreover, in order to discuss the effects of the planner, we calculated its overhead separately. This overhead created by the planner will be referred as *planner overhead* for seperating it from general overhead of the tool. We aimed to discuss planner's necessity depending on the extra overhead and its effect of test throughput. Since we expect our tool to detect test cases that would be applicable if a couple of actions could be executed, we are aiming an increase on test throughput.

The usage of a planner requires a maximum length for plans, maximum number of actions in our case. This threshold variable is selected as 6 during our experiments. Lowering this value would decrease the overhead of planner since the planner would not try longer plans, lower threshold value might miss possible plans.

In addition to extra overhead, we also evaluated the performance of the planner with initializing the SUT's with different states. To be more explanatory, with the planner, the tool has multiple states to calculate a plan to convert a test as applicable for a single current state. For example, CCRS and CCRM test cases expects at least a global car in the same lane with the VUT. However with a planner, a plan can be calculated for these cases when the global car is in one of the adjacent lanes. Therefore, multiple states should satisfiable for the planner. To experiment this, we initiated the vehicle simulator with different initial states and expected that tool is still capable to calculate plans.

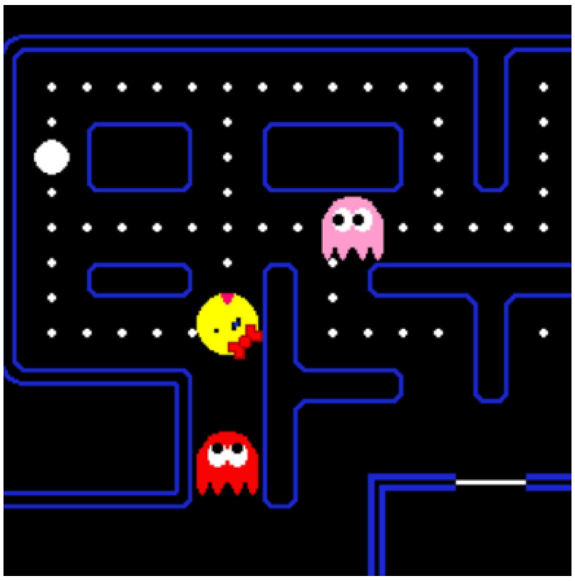
## 4.4 Data and Analysis

As mentioned in 4.3, the results of modeling and execution phases are reported separately.

### 4.4.1 modeling Evaluations

Initially, we designed model declarations for both SUT's. However during test case modelings, we realized a group of required attributes that we missed while designing the model. For instance, we realized the requirement of the distance between pacman and monsters. Otherwise, it was not applicable to declare "close monster" rule.

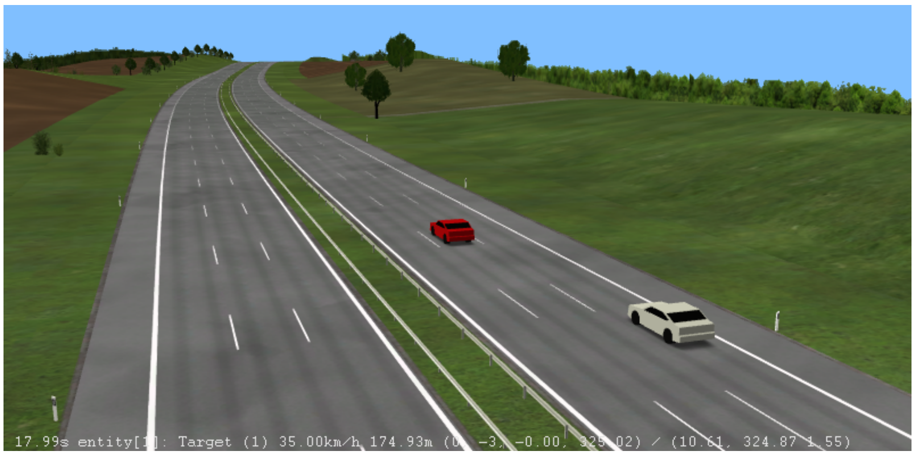
Figure 4.4 Final State Design For Pacman Game



Finally, for the moment visualized in Figure 4.4, designed model example can be seen as follows;

*pacmanMode(alive).*  
*pacmanPosition(6,16).*  
*pacmanDirection(up).*  
*pacmanRemainingLives(3).*  
*monsterPosition(red,6,12).*  
*monsterDirection(red,up).*  
*monsterMode(red,normal).*  
*monsterPosition(pink,11,18).*  
*monsterDirection(pink,left).*  
*monsterMode(pink,normal).*  
*pacmanMonsterDistance(red,63).*  
*pacmanMonsterDistance(pink,102).*

Figure 4.5 Final State Design For Esmini



```
17.99s entity[1]: Target (1) 35.00km/h 174.93m (0, -3, -0.00, 325.02) / (10.61, 324.87 1.55)
```

For the moment visualized in Figure 4.5, designed model example can be seen as follows;

*vutPosition(14,338).*
*vutLane(right).*
*vutSpeed(50).*
*accControllerActive(0).*
*gvtPosition(gvt1,11,352).*
*gvtLane(gvt1,middle).*
*gvtSpeed(gvt1,35).*
*relativeDistance(vut,gvt1,190).*
*relativePositionWithVut(gvt1,front,left).*

Again, these example states are just a one way to express states. Even though we preferred to express them as given in this report for the experiments, there are other ways to express such SUT's.

For some test cases with a coverage criteria of 4 directions, I preffed writing rules independenlty from test cases and append these rules into cases if needed as an "crashable" example explained in Section 3 . For instance, rather than writing 4 seperate cases for the test of pacman detects a monster chasing him and returns direction for collision, I wrote;

*testCase(*
*action(reverse,forwardPacman(2)),oracle(pacmanRemainingLives(ORL))*
*,test(pacmanChaseCrash(PDIR))):-*
*pacmanDirection(PDIR), pacmanRemainingLives(PRL), crashable(PDIR), ORL = PRL -1, not covered_pacmanChaseCrash(PDIR).*

while the rule *crashable* has conditions for directions of Pacman separately;

*crashable(right) :-...*
*crashable(left):- pacmanPosition(PX,PY),pacmanDirection(left),*
*monsterPosition(MID,PX+2,PY),monsterDirection(MID,left),*
*monsterMode(MID,normal).*
*crashable(down) :- ...*

Note that, I also used these rules for the conditions of other test cases.

As a result of our modeling efforts of test cases, we were able to model all of them given in Table 4.1. Therefore for the Pacman game, we did not face a case that can not be modelled and obtained 100% modeling success.

For me who is more or less experienced with ASP, this modeling for test cases and state took about half a day. During this effort, I used different example states to actually test if written cases are correct and I achieved all test case detections for applicable states.

For ACC module testing, out of 21 test cases defined in EuroNCAP (2020), we were able to model all of them, once more.

However, we faced the issue of TTC calculation. Depending on Hayward (1972); Hydén (1996); Minderhoud & Bovy (2001), TTC describes the duration before the collision of 2 vehicles under given conditions. Although there is a certain explanation, there are different approaches for the formulation of TTC, in the literature Hou, List & Guo (2014); Saffarzadeh, Nadimi, Naseralavi & Mamdoohi (2013). However what is restrictive for our approach is that, as mentioned in Hou et al. (2014), optimal calculation for TTC is simulation based algorithms which is what we are testing already. Due to this requirement, it is not possible to give a test action implementation that assume to obtain a exact TTC value for 2 vehicles. Therefore, we were not be able to append TTC values into our test case conditions, oracles and planner. Besides this uncertainy problem, we implemented a function that manages the speed of given vehicles and assume to obtain a reasonable distance between them to test ACC module.

Under these conditions, if we consider the test cases from Euro NCAP that includes TTC values as half modelled, the modeling succession we obtained is 95.23% .

To generalize the problem, as mentioned in Section 3, we assume that the user will provide action implementations that their results are deterministic. For instance, when our planner calculates a sequence of actions depending on their effects, we assume that execution of these actions over SUT will resulted as planned. However, since it is not possible to provide such implementations for nondeterministic metrics like TTC, our approach falls short.

For modeling the goal test suite from Euro NCAP, the amount of time I spend was about 1 day. The longest effort was to figure a way for solving the mentioned TTC problem. Following, for the high level planner modeling, I did not spend more than 2 hours. In order to obtain such planner, the number of modelled actions was just 7 as *setSpeed, increaseDistance, decreaseDistance, passRightLane, passLeftLane, assignAccController and assignExternalController*. While *increaseDistance or decreaseDistance* actions are arranging a reasonable distance between vehicles for test conditions, other action names are self-explanatory.

### 4.4.2 Execution Evaluations

As mentioned in Subsection 4.3.2, we calculated the average amount of overhead that over tool creates to SUT's in milliseconds by executing the cases.

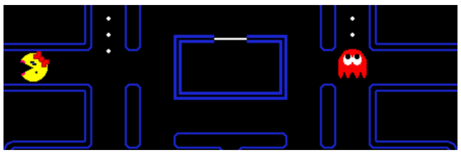
For Pacman game, in addition to this general overhead, we also calculated the overhead that is created by test action executions inside of it. Actually, during these action executions, our approach does not interrupt the system, on the contrary, it leaves the execution to the SUT in order to let it execute the given actions. For instance, we mean the amount of time to convert the monsters into freight mode while *freakPacman* action is being executed. Therefore, we reported 2 types of average overhead, *with-actions*, and *without-actions*, including the run-time duration of actions and not, respectively.

Table 4.2 presents the average overhead for both calculation types. The average overhead for 14 test cases *without-actions* is 19.808 ms. In other words, on average, for a successful test case, our approach interrupted the execution of the SUT for 19.808 ms. As expected, test cases with more complex conditions created longer overheads than relatively simple ones. For instance, test case looking for a monster approaching from a diagonal direction created 25.545 ms overhead on average while one looking for a wall in the direction of Pacman created 13.756 ms on average.

We observed that average overhead of *with-actions* are longer and widely distributed than *without-actions*; this is to be expected as the actions of test cases are various and specific to cases. However, these durations should not be considered as an overhead since during executions of actions, our tool is not interrupting the SUT since the system is on execution during this time.

Table 4.2 Test case explanations and measured overheads for both calculation types

| test case explanation | average run-time overhead | |
|---|---|---|
| | without-actions | with-actions |
| pacman crashes a wall while a monster chasing it, pacman stops and collision happens | 18.992 ms (128 executions) | 718.318 ms (88 executions) |
| pacman being chased by a monster, pacman returns to the opposite direction and collision happens | 20.883 ms (60 executions) | 381.973 ms (40 executions) |
| pacman being chased by a monster, pacman goes freak and returns to the opposite direction and accelerates, collision happens | 18.562 ms (16 executions) | 503.181 ms (12 executions) |
| pacman eats power pellet | 18.578 ms (76 executions) | 381.476 ms (40 executions) |
| pacman eats fruit | 18.5 ms (20 executions) | 507.384 ms (12 executions) |
| pacman run away from a monster coming from the opposite direction, pacman runs away, no collision happens | 19.681 ms (68 executions) | 383.883 ms (44 executions) |
| pacman crashes to a wall | 13.756 ms (76 executions) | 36.204 ms (44 executions) |
| pacman detects a freight monster on the current direction, accelerates, collision happens | 23.689 ms (60 executions) | 379.906 ms (32 executions) |
| pacman detects a freight monster close by, accelerates, collision happens | 17.375 ms (16 executions) | 373.571 ms (8 executions) |
| pacman detects a freight monster on the other side of the maze portal, accelerates and collision happens | 26.75 ms (12 executions) | 888.000 ms (12 executions) |
| pacman stops while a monster approaching from a diagonal direction, loses a live | 25.405 ms (36 executions) | 712.363 ms (20 executions) |
| pacman detects a monster coming from opposite direction, jumps over it, loses no live | 19.608 ms (20 executions) | 37.823 ms (20 executions) |
| pacman with 1 remaining live, detects a the monster chasing it and stops, collision happens, pacman obtains 5 lives | 15.035 ms (28 executions) | 704.5 ms (8 executions) |
| pacman with 1 remaining live, detects last pellet in the maze in the current direction, collision happen with the last pellet, pacman obtains 5 lives | 20.5 ms (8 executions) | 706.5 ms (8 executions) |


(a) Inital State For Portal Test Case

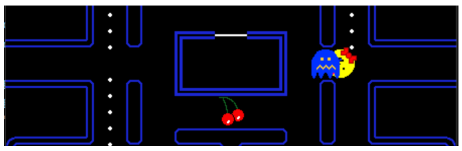

(b) Final State for Portal Test Case

Figure 4.6 Steps of Portal Test Case

Table 4.3 Test case types and measured overheads

| test case explanation | | | average run time overhead | |
|---|---|---|---|---|
| type | VUT speed | GVT Speed | planner overhead | total overhead |
| ccrs | 70 | 0 | 48.326 | 57.028 |
| | 80 | 0 | 38.449 | 47.681 |
| | 90 | 0 | 40.632 | 49.752 |
| | 100 | 0 | 37.343 | 45.789 |
| | 110 | 0 | 39.376 | 48.181 |
| | 120 | 0 | 35.396 | 43.339 |
| | 130 | 0 | 34.580 | 43.880 |
| ccrm | 80 | 20 | 15.229 | 27.650 |
| | 90 | 20 | 39.053 | 48.859 |
| | 100 | 20 | 43.283 | 52.510 |
| | 110 | 20 | 43.052 | 51.882 |
| | 120 | 20 | 37.726 | 45.225 |
| | 130 | 20 | 31.765 | 40.593 |
| | 80 | 60 | 35.443 | 43.263 |
| | 90 | 60 | 35.348 | 43.326 |
| | 100 | 60 | 38.886 | 46.512 |
| | 110 | 60 | 21.936 | 28.919 |
| | 120 | 60 | 32.267 | 39.580 |
| | 130 | 60 | 39.006 | 46.157 |
| cut-in | 50 | 10 | 46.447 | 54.643 |
| | 120 | 70 | 33.107 | 38.631 |

The maximum overhead obtained for a test case is 26.75 which belongs to the case of pacman detects a freight monster on the other side of the portal.. This case has the conditions that pacman and at least a monster should be inside of portal boundries that declared in system constraints, can be seen from Figure 4.6a. Mentioned conditions are the most complex ones inside the entire test suite. Furthermore, this case has actions for converting to freak pacman and accelerating it, following to actions, expected oracle is at least one dead monster, can be seen from Figure 4.6b.

For testing of ACC module, we calculated the average overhead of test cases and the planner seperately. Results can be seen from the Table 4.3 .

After 23 executions, we observed average 44.924 ms overhead that our tool creates. Inside of this overhead, we have average 36.510 ms overhead created by the planner which averagely 81.270% of the total overhead. Therefore, remaining, 8.414 ms overhead is the overhead of our tool without planner created by other steps like test case search or oracle solution, see Section 3.

Out of 3 test types, we have the maximum overhead average for CCRS cases with 47.950 ms while 46.637 ms for cut-in cases and 42.873 for CCRM cases. With 39.777

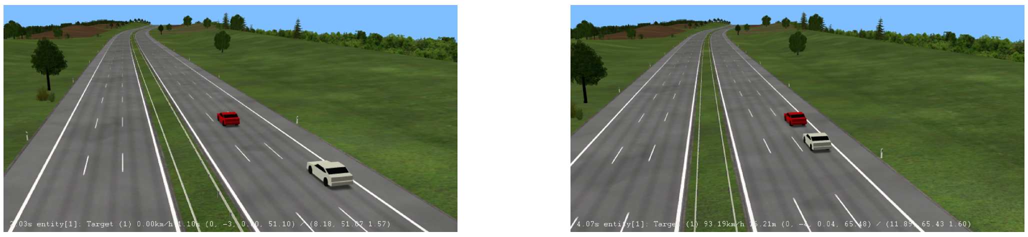

(a) An example initial state (b) Final state after plan execution

Figure 4.7 Effect of a planner for a unsatisfiable initial state

ms cut-in cases have the most planner overhead while CCRS cases have 39.165 ms and CCRM cases have 34.416 ms. After all executions, inside of these 7 possible actions in planner, only 5 of them have been used in plans and averagely, we obtained plans with 3.857 actions.

During executions, averagely, we were able to execute these 21 test cases in 99.42 seconds which lead to 4.734 seconds per test. Additionaly, we experimented these cases with 3 initial states with different speeds and initial lanes. As a result, planner were able to detect required actions to reach a state that satisfies a test case conditions, for instance, when we started the system as global vehicle is in the adjacent lane of VUT, see Figure 4.7a while searching for a CCRM case, modelled planner returned a sequence of actions that takes the global vehicle to the lane of VUT, see Figure 4.7b.

While the maximum average overhead which is 57.028 ms created by CCRS test case with speeds 70km/h and 0km/h , minimum one is the CCRM test case with speeds 80km/h and 20km/h . The reason of this minimum value is the fact that this test case mostly executed as first case, just after initial state. As a more detailed explanation; for all cases, we expect the external controller of VUT is enabled in order to activate ACC module when conditions are met and test its performance. Therefore for all cases, we are enabling ACC module as a test action. Thus, after each case, VUT vehicle has ACC controller activated which needs to disabled for satisfaying the conditions of following case. However this is not the case for the first test case since both vehicles were initialized with external controllers.

## 4.5 Discussion

During the experimented evaluations, we aimed to evaluate the proposed approach in a compherensive manner. Our goal was to evaluate the flexibility of our approach for different applications from different domains.

During the beginning of our work, our biggest doubt was the overhead that will be generated by our tool. Our experiments with Pacman and ACC module showed that for such autonomous systems our tool is efficient since our overhead is acceptable and not creating gaps enough to brake the system. For future applications as SUT's, the main question of the tester should be;

> Are 30-40 ms of execution gaps during test case executions are negligible for my system when the test automation and increase on test throughput considered ?

We believe that, for most systems, the answer of this questions would be yes. For instance, for a vehicle simulator, the amount of reported overhead did not brake the execution of ACC module or the simulation itself.

As will be mentioned in the following sections, experimenting this approach with different SUT's will show that our approach is applicable for autonomous systems from various domains.

We believe that modeling test cases with the proposed format is also a practical method. With the support of declarative logic programming, our format is easy to write and understandable even for non-technical stakeholders. For most test cases for ACC module, for all test types, I moduled a case and created the rest with copy-paste and modified necessary conditions.

Furthermore, with the usage of ASP our tool gained flexibility since the same system can be modeled with different ways. The single assumption of our tool is *as long as test scenarios for an application and system state of it can be expressed with ASP, our approach is applicable.*

When the amount of time to model the test suites is considered, obviously, this task requires at least basic ASP knowledge. Specially for oracle conditions, tester should be able to model constraints like "speed attribute of pacman object must not have more than 1 value" or "given 2 vehicles must have the same speed value".

Furthermore, we believe that modeling a planner requires even more knowledge for ASP. For instance, additional to the configuration of "speed value of a vehicle changes with either setSpeed or manageDistance actions", tester should also model "speed value of a vehicle does not change until there is either setSpeed or manageDistance action in the current iteration". These type of non-effect declarations become more complex for more complex actions.

On the other hand, even though I was able to manage this, defining a planner is not a simple task due to requirement of planners and domain spesific knowledge. While the planner is being modelled, additional to "what will change if this action happens?" for each actions, "what are the actions that does not change the value of this attribute?" for each attribute should also be considered. For the concept of declarative logic programming, I do not consider this effort as a simple task.

However, we believe that our approach requires a planner inevitably since autonomous systems are unpredictable and waiting for a state that directly fits the conditions of a test case is time consuming, and even non-deterministic. Actually, we faced this issue for ACC module testing. As seen from the test cases of EuroNCAP (2020) or given in Section 4.3.2, cases expect the vehicles to obtain exact speeds like 20km/h or 80km/h . As we mentioned in Section 1, manually assigning these speed values is what we are trying to avoid. However, waiting an autonomous system to reach such values by itself is not guaranteed. As a result, we were able to overcome this problem with the help of the planner and executed the cases no mather what was the initial speed.

To summarize, we strongly suggest that ASP provides a flexible way of modeling test cases and state information for autonomous systems. With such an approach that uses ASP for modeling, we require no further information. We believe that the features of ASP include direct solutions to the problems of testing autonomous systems, mentioned in Section 1. Moreover, our tool has shown an impactful performance during the experiments and resulted in acceptable overhead and coverage values.

# 5. THREATS TO VALIDITY

As mentioned in Section 4, we evaluated our approach with 2 different applications. Thus, this number for subject applications is our main external threat. In order to have further evaluations, we plan to evaluate our approach with other applications, in particular, more realistic and complex applications.

Experimenting our approach with a vehicle simulation using Euro NCAP test cases was the main goal at the beginning, furthermore, this subject application includes similarities with house cleaning robots such as a autonomous object that is aiming to avoid collisions with other objects.

Another related threat that should be mentioned is, the subject application for ACC testing, Esmini, is not a well-known and complex simulation. However, it is open source, it provides controller options like ACC which fits our case. Furthermore, it provides an environment to implement scripts to manage the behaviour of vehicles if relevent controller is enabled. Therefore, at this moment, Esmini was the optimal selection for a subject application.

During our evaluations, we experimented the approach with a single computer mentioned in Section 4.2. Since our evaluations are mostly based on overhead generation, another computer with a lower process power could be used. However, since our approach does not have any complex requirements, it is easy to reproduce the experiments on a different computer.

Our biggest internal threat is the designed test suite for Pacman game. As we mentioned in Section 4.3.1 , we designed our own test suite even though this may lead to a biased evaluation. On the other hand, I have been working on testing domain for sufficient time and believe that designed test cases are both inclusive enough to test the SUT and complex enough to experiment the modeling phase. In addition, the test suite we used for ACC testing is designed by a well-known community and their cases are accepted by the literature.

# 6. CONCLUSION

Our main motivation for this work was to come up with an approach that solves the problems of testing autonomous systems which occur due to their autonomous behavior. We presented and evaluated an approach that monitors the system continuously and searches for applicable test cases and executes them if there are any, with minimum overhead.

As a result, in this thesis, we came up with a novel and flexible testing approach. We would like to mention that even if we discussed the applicability of autonomous systems, our approach is applicable to other types of systems as well. However, only the autonomous systems were within the scope of our thesis.

The proposed approach uses ASP in order to model the information like state or test suite. The results of our case studies suggest that the proposed approach is flexible enough to model various applications from different domains. As discussed in Section 4, we were able to express all the test cases we wanted to and execute them with relevant SUT's. Therefore, our approach can be labeled as a generic one since as long as state and test case definitions can be expressed with ASP, our approach is applicable.

Our experiment results indicate that the proposed approach obtained low overhead together with high test throughput, especially with the planner. Even for the planner, we were able to use an AI planner that is implemented as an ASP program just like the states and test cases. Finally, we expressed all the required information for the planners of ACC tests and executed cases with planners. Even though the usage of planner increases the average overhead of our approach, the flexibility and test throughput increase it brings is irrefutable.

As a future work, first, we plan to increase the number of subject applications, this approach can be evaluated with applications from different domains. However, we believe that the most crucial future work for the discussed work is evaluating it with more realistic applications rather than computer games or simulations.

Finally, for the current version of the implementation, we handled test selection for the planner by starting from the first test case in the expressed suite and proceeding one by one when the previous one is planned for a suitable state. However we believe that more efficient test selection algorithms may increase test throughput performance of our tool since they will perform better compared to ordered selection.

# BIBLIOGRAPHY

Bradshaw, J. M., Hoffman, R. R., Woods, D. D., & Johnson, M. (2013). The seven deadly myths of" autonomous systems". *IEEE Intelligent Systems*, *28*(3), 54–61.

Brewka, G., Eiter, T., & Truszczyński, M. (2011). Answer set programming at a glance. *Communications of the ACM*, *54*(12), 92–103.

Charisi, V., Dennis, L., Fisher, M., Lieck, R., Matthias, A., Slavkovik, M., Sombetzki, J., Winfield, A. F., & Yampolskiy, R. (2017). Towards moral autonomous systems. *arXiv preprint arXiv:1703.04741*.

Eder, K. I., Huang, W.-l., & Peleska, J. (2021). Complete agent-driven model-based system testing for autonomous systems. *arXiv preprint arXiv:2110.12586*.

Erdem, E., Inoue, K., Oetsch, J., Pührer, J., Tompits, H., & Yılmaz, C. (2011). Answer-set programming as a new approach to event-sequence testing.

Erdem, E. & Patoglu, V. (2018). Applications of asp in robotics. *KI-Künstliche Intelligenz*, *32*(2), 143–149.

Esmini (2022). Esmini documentation. `https://github.com/esmini/esmini`.

EuroNCAP (2017). Test protocol – aeb systems. `https://cdn.euroncap.com/media/26996/euro-ncap-aeb-c2c-test-protocol-v20.pdf`.

EuroNCAP (2020). Assisted driving highway assist systems - test assesment protocol. `https://www.euroncap.com/en/for-engineers/protocols/general/`.

Fikes, R. E. & Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, *2*(3-4), 189–208.

Fisher, M., Dennis, L., & Webster, M. (2013). Verifying autonomous systems. *Communications of the ACM*, *56*(9), 84–93.

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2019). Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, *19*(1), 27–82.

Haslum, P., Lipovetzky, N., Magazzeni, D., & Muise, C. (2019). An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, *13*(2), 1–187.

Hayward, J. C. (1972). Near miss determination through use of a scale of danger. *Unknown*.

Helle, P., Schamai, W., & Strobel, C. (2016). Testing of autonomous systems–challenges and current state-of-the-art. In *INCOSE international symposium*, volume 26, (pp. 571–584). Wiley Online Library.

Hou, J., List, G. F., & Guo, X. (2014). New algorithms for computing the time-to-collision in freeway traffic simulation models. *Computational intelligence and neuroscience*, *2014*.

Huang, W., Wang, K., Lv, Y., & Zhu, F. (2016). Autonomous vehicles testing methods review. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, (pp. 163–168). IEEE.

Hydén, C. (1996). Traffic conflicts technique: state-of-the-art. *Traffic safety work with video processing*, *37*, 3–14.

Jensen, K., Kristensen, L. M., & Wells, L. (2007). Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, *9*(3), 213–254.

Koopman, P. & Wagner, M. (2016). Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety*, *4*(1), 15–24.

Kuhn, D. R., Higdon, J. M., Lawrence, J. F., Kacker, R. N., & Lei, Y. (2012). Combinatorial methods for event sequence testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, (pp. 601–609). IEEE.

Lifschitz, V. (1999). Action languages, answer sets, and planning. In *The Logic Programming Paradigm* (pp. 357–373). Springer.

Lifschitz, V. (2019). *Answer set programming*. Springer Heidelberg.

Lill, R. & Saglietti, F. (2012). Model-based testing of autonomous systems based on coloured petri nets. In *ARCS 2012*, (pp. 1–5). IEEE.

Lindvall, M., Porter, A., Magnusson, G., & Schulze, C. (2017). Metamorphic model-based testing of autonomous systems. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, (pp. 35–41). IEEE.

Marsden, G., McDonald, M., & Brackstone, M. (2001). Towards an understanding of adaptive cruise control. *Transportation Research Part C: Emerging Technologies*, *9*(1), 33–51.

Maserati (2022). Adaptive cruise control with stop & go. `https://www.maserati.com/global/en/ownership/maserati-manuals/safety/adaptive-cruise-control`.

Milanés, V., Shladover, S. E., Spring, J., Nowakowski, C., Kawazoe, H., & Nakamura, M. (2013). Cooperative adaptive cruise control in real traffic situations. *IEEE Transactions on intelligent transportation systems*, *15*(1), 296–305.

Minderhoud, M. M. & Bovy, P. H. (2001). Extended time-to-collision measures for road traffic safety assessment. *Accident Analysis & Prevention*, *33*(1), 89–97.

of Potsdam, U. (2022). Potasco guide. `https://github.com/potassco/guide/releases/tag/v2.2.0`.

Richards, J. (2022). Pacman game documentation. `https://pacmancode.com/`.

Saffarzadeh, M., Nadimi, N., Naseralavi, S., & Mamdoohi, A. R. (2013). A general formulation for time-to-collision safety indicator. In *Proceedings of the Institution of Civil Engineers-Transport*, volume 166, (pp. 294–304). Thomas Telford Ltd.

Xiao, L. & Gao, F. (2010). A comprehensive review of the development of adaptive cruise control systems. *Vehicle system dynamics*, *48*(10), 1167–1192.

Zhou, J. & del Re, L. (2018). Safety verification of adas by collision-free boundary searching of a parameterized catalog. In *2018 Annual American Control Conference (ACC)*, (pp. 4790–4795). IEEE.