Bachelorarbeit

Ein Framework für die Testautomatisierung in einer Fußballroboter-Simulation

Nico Holsten

Matrikelnummer: 6113546

13.09.2024

1. Gutachter: Dr. Tim Laue
2. Gutachter: Prof. Dr. Rolf Drechsler

Fachbereich 3: Mathematik und Informatik
Studiengang Informatik

Nico Holsten

Ein Framework für die Testautomatisierung in einer Fußballroboter-Simulation

Bachelorarbeit, Fachbereich 3: Mathematik und Informatik
Universität Bremen, 13.09.2024

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig angefertigt und nicht anderweitig zu Prüfungszwecken vorgelegt habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel verwendet. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 13.09.2024

.....

(Nico Holsten)

Inhaltsverzeichnis

1 Einleitung

1.1 Hintergrund

Diese Arbeit befasst sich mit der Entwicklung eines Frameworks zur Testautomatisierung in einer Fußballroboter-Simulation für das Team *B-Human*. *B-Human* setzt sich überwiegend aus Studierenden der Universität Bremen sowie Forschern dieser Universität und des Deutschen Forschungszentrums für Künstliche Intelligenz (DFKI) zusammen. Das Team hat sich das Ziel gesetzt, Software für die Teilnahme am *RoboCup* zu entwickeln. Der *RoboCup* ist ein internationaler Wettbewerb, welcher 1997 ins Leben gerufen wurde und sich zum Ziel gesetzt hat, die Entwicklung von Robotern und künstlicher Intelligenz durch das Spiel von Fußball zu fördern (vgl. RoboCup, 2024a). *B-Human* zählt derzeit zu den erfolgreichsten Teams des Wettbewerbs und hat bereits elf Weltmeistertitel sowie 15 regionale Meisterschaften gewonnen (vgl. B-Human, 2024b). Zuletzt konnte das Team beide Titel im Jahr 2024 für sich entscheiden. Abbildung 1.1 zeigt einen *B-Human* Roboter während des *RoboCups* 2024.

1.2 Motivation

Das Team *B-Human* nimmt seit 2009 an *RoboCup*-Turnieren teil. Jedes Jahr findet abwechselnd ein Bachelor- oder Masterprojekt statt, in dessen Rahmen oft unerfahrene Studierende die Aufgabe haben, die *B-Human*-Software zu erweitern, zu verbessern und auf den nächsten *RoboCup* vorzubereiten. Dadurch wächst die Software seit vielen Jahren kontinuierlich, was zu einer zunehmenden Komplexität führt. Viele Klassen und Funktionen wurden in dieser Zeit mehrfach überarbeitet und neu konzipiert. Dies führt dazu, dass nicht immer sichergestellt werden kann, dass keine neuen Fehler eingeführt wurden und Verbesserungen tatsächlich wirksam sind.

B-Human nutzt zum schnellen und unaufwendigem Testen die Robotersimulation *SimRobot*. In der Anwendung können einzelne Roboter, aber auch ganze Teams in verschiedenen Szenarien, wie unterschiedliche Spielfeldgrößen, gegeneinander spielen (siehe Abbildung 1.2). Eine ausführliche Erklärung zu *SimRobot* ist in Kapitel 2.3 zu finden. Aktuell ist es im Simulator nicht möglich Tests automatisiert mehrmals abfolgen

Abbildung 1.1: Ein *B-Human* Roboter führt während des *RoboCups* 2024 in Eindhoven einen Schuss aus. Im Hintergrund sind zwei gegnerische Roboter des Hamburger Teams *HULKs* zu sehen. (Foto: Ayleen Lührsen)

zu lassen. Der Simulator muss jedes Mal händisch zurückgesetzt und konfiguriert werden, was in jedem Testdurchlauf Zeit und die Aufmerksamkeit des Testenden beansprucht. Eine weitere Möglichkeit ist das Schreiben eines externen Skripts, welches dies übernimmt. Allerdings muss auch dies zunächst entwickelt werden.

Will man in der Realität testen, bleiben dem Testenden lediglich teilweise eingeschränkte Tests im Projektraum oder aufwendige Testspiele. So ist im Projektraum nur Platz für ein Tor und ein Bruchteil des Spielfeldes. Einige Aspekte lassen sich in dieser Form testen, während andere aufgrund des Bedarfs an mehr Spielfeldfläche oder zwei Toren nicht getestet werden können. Für die Testspiele ist es notwendig, dass die aufgerollten Spielfeldteile, Tore und Roboter zu einem geeigneten Raum transportiert und dort aufgebaut werden. Dies benötigt mehrere Studierende und viel Koordination. Des Weiteren ist ein beträchtlicher Zeitaufwand für den Aufbau und die Initialisierung der Infrastruktur zu verzeichnen. Außerdem muss der Raum dafür ausreichend groß und gut belichtet sein. Ein weiterer Aspekt ist, dass selbst in den ersten Spielen während der Meisterschaften häufig noch kleinere Ungenauigkeiten oder Fehler entdeckt werden, die durch automatisierte Tests im Vorfeld hätten identifiziert werden können. Dies zeigt, dass mehr Tests notwendig sind. Die Bereitstellung eines einfach zu konfigurierenden Test-Frameworks kann die Anzahl der im Vorhinein absolvierten Tests fördern.

Ziel dieser Arbeit ist es daher, durch die Einführung automatisierter Tests dieses

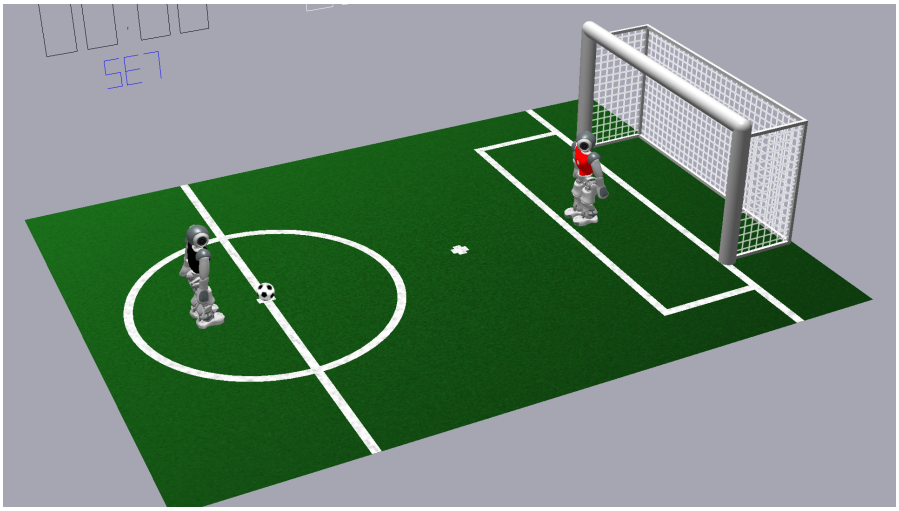


Abbildung 1.2: Zwei Roboter spielen in *SimRobot* auf einem kleinen Spielfeld gegeneinander.

Problem zu lösen und Zeit zu sparen, die anderweitig genutzt werden kann. Zudem sollen Fehler durch das automatisierte Testen frühzeitig erkannt, lokalisiert und korrigiert werden.

1.3 Zielsetzung

Das Ziel dieser Arbeit ist es, die Automatisierung von Tests in der *B-Human*-Software zu ermöglichen. Dabei soll das Roboterverhalten, welches bei *B-Human* durch die Nutzung von endlichen Zustandsautomaten realisiert ist, aber auch die Robotersoftware allgemein automatisiert testbar sein. Beim Verhalten wäre ein beispielhaftes Testszenario das Erzielen eines Tores in einer bestimmten Zeit. In anderen Bereichen der Informatik, beispielsweise der Bildverarbeitung mittels neuronaler Netze, findet häufig die Verwendung von Testdatensätzen Anwendung. Beim Verhalten ist dies allerdings nicht der Fall, da die Roboter in einer physikalischen Umgebung agieren müssen. Aber auch für automatisierte Tests der allgemeinen Robotersoftware, wie Sensorverarbeitung oder die Modellierung, soll das Test-Framework geeignet sein.

Da dies ein umfangreiches Thema ist, wurde eine Umfrage unter Entwicklern im *B-Human*-Team, welche Erfahrungen in der Programmierung des Verhaltens haben, durchgeführt. Das Ziel der Umfrage war es den größtmöglichen Nutzen für *B-Human* zu ermitteln. Da man nur Entwickler, welche im Gegensatz zu neuen Teammitgliedern bereits viele Erfahrungen beim Testen gesammelt haben, als Befragte der Umfrage aufgenommen hat, liegt die Teilnehmergröße der Umfrage lediglich bei vier Personen.

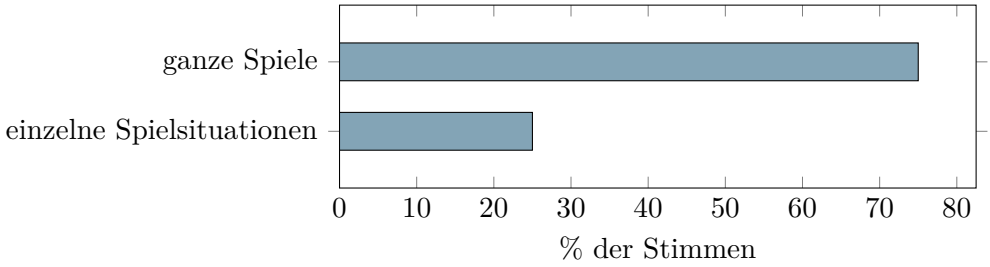


Abbildung 1.3: Das Ergebnis der Frage: Was soll testbar sein?

Durch diesen Schritt wurde allerdings ein fundierteres Ergebnis erhofft.

Die Umfrage umfasste vier Fragen mit Auswahlmöglichkeiten, welche alle beantworten werden mussten, sowie eine zusätzliche Freitextfrage, in der weitere Wünsche oder Anregungen geäußert werden konnten.

1.3.1 Umfrage

Zunächst wurde gefragt, welche Aspekte testbar sein sollen. Die erste Option betraf einzelne Spielsituationen mit einem spezifischen Ziel, wie etwa das Erzielen eines Tores oder das Betreten einer bestimmten Zone durch einen Roboter. Die zweite Option umfasste komplette Spiele zwischen zwei Teams, die beispielsweise unterschiedliche Verhaltensimplementierungen aufweisen. Auf diese Weise könnte überprüft werden, ob eine neue Verhaltensimplementierung tatsächlich zu einer Verbesserung führt. Es konnte jeweils nur eine Auswahl getroffen werden. Abbildung 1.3 zeigt, dass 75% der Befragten ganze Spiele bevorzugen, während die verbleibenden 25% einzelne Spielsituationen testen möchten.

Die zweite Frage bezog sich auf die Auswahl der Szenenarten, mit denen die Teilnehmer ihr Verhalten überwiegend testen. Zu den Szenenarten gehören die *Fast*-Szenen und die *PerceptOracle*-Szenen. In *PerceptOracle*-Szenen werden keine Kamerabilder gerendert und die Wahrnehmung der Roboter wird vom Simulator zur Verfügung gestellt. In *Fast*-Szenen ist dies ebenfalls der Fall, allerdings müssen hier auch keine Modelle, beispielsweise für den Ball, berechnet werden (vgl. Röfer, Laue, Baude u. a., 2019). Auch hier war nur eine Auswahlmöglichkeit zulässig. 75% der Befragten gaben an, hauptsächlich mit den *Fast*-Szenen zu testen, während 25% ihr Verhalten vorrangig in *PerceptOracle*-Szenen überprüfen. Diese Ergebnisse sind in Abbildung 1.4 dargestellt. Eine ausführliche Erläuterung der verschiedenen Szenenarten ist in Kapitel 2.3.1 zu finden.

Die nächste Frage, bei welcher nur eine Auswahl getroffen werden konnte, beschäftigte sich mit der Geschwindigkeit der Testdurchführung. Dabei wurde ermittelt, ob es den

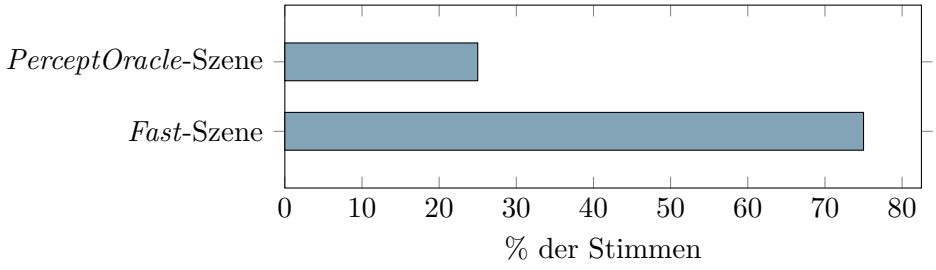


Abbildung 1.4: Das Ergebnis der Frage: Mit welchen Szenen testest du meistens?

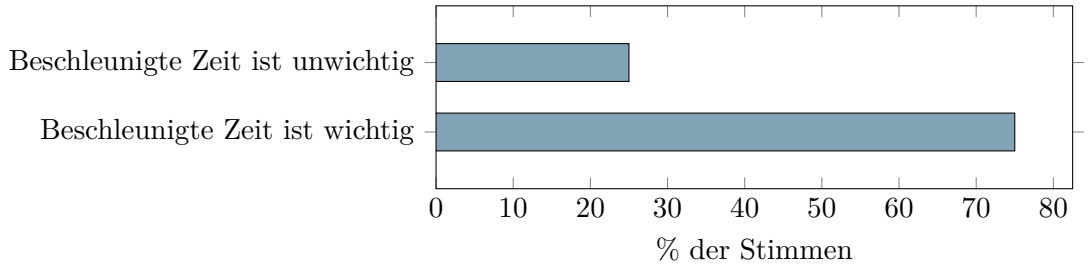


Abbildung 1.5: Das Ergebnis der Frage: Inwieweit ist die Geschwindigkeit der Testdurchführung wichtig?

Teilnehmern der Umfrage wichtig ist, dass die Tests schneller als in Echtzeit durchgeführt werden können. Wie in Abbildung 1.5 dargestellt, gaben 75% der Befragten an, dass eine Testdurchführung mit einer höheren Geschwindigkeit als Echtzeit wichtig ist, während 25% dies als unwesentlich betrachten.

Die letzte Auswahlfrage mit mehreren Antwortmöglichkeiten bezog sich auf die automatische Durchführung mehrerer Spiele und die Parameter, die anschließend ausgegeben werden sollen. Alle Befragten hielten die Anzahl der Tore der Teams für wichtig. Die Hälfte der Teilnehmer erachtete den prozentualen Anteil des Ballbesitzes als relevant. Die übrigen Auswahlmöglichkeiten, darunter Anzahl der Siege, Schüsse, Passquote, Dribblings, Klärungen und Freistöße, wurden von lediglich 25% der Befragten als wichtig eingestuft (siehe Abbildung 1.6).

Bei der letzten Frage handelte es sich um eine Freitextfrage zu weiteren Wünschen und Anregungen. Ein Teilnehmer äußerte, dass ihm sowohl ganze Spiele als auch einzelne Spielsituationen wichtig seien, er dies jedoch aufgrund der Beschränkung auf eine Auswahlmöglichkeit in der ersten Frage nicht angeben konnte. Ein weiterer Teilnehmer wünschte sich Heatmaps der Ballposition, Statistiken zum Spielstand vor einem Tor sowie spezifische Tests für besondere Anwendungsfälle. Ein anderer Teilnehmer gab an, dass für ihn Metriken wie Pässe oder Ballbesitz weniger relevant seien als eine binäre Entscheidung darüber, ob etwas funktioniert hat oder nicht. Beispielsweise, ob der Ball

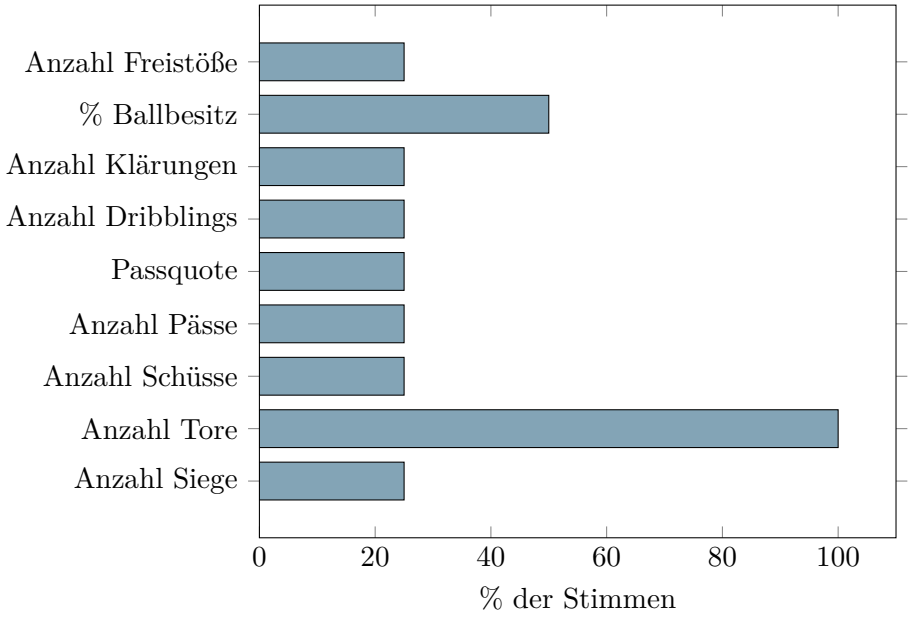


Abbildung 1.6: Das Ergebnis der Frage: Was ist dir bei der Auswertung im Anschluss der Tests wichtig?

ins Tor geschossen wurde.

1.3.2 Auswertung Umfrage

Aus der Umfrage geht eindeutig hervor, dass die Möglichkeit, zwei Teams automatisiert mehrere Spiele gegeneinander austragen zu lassen, von den Teilnehmern stark gewünscht wird. Für die Auswertung dieser sollen die Ergebnisse sowohl kompakt als auch in einer *CSV*-Datei ausgegeben werden. Ebenso besteht ein Bedarf an der automatisierten Durchführung einzelner Spielsituationen. In diesem Zusammenhang wurden folgende Ziele für das Testen von Situationen festgelegt:

- Das linke Team erzielt ein Tor
- Das rechte Team erzielt ein Tor
- Der Ball betritt eine angegebene Zone
- Ein bestimmter Roboter betritt eine angegebene Zone
- Ein bestimmter Roboter berührt den Ball
- Ein bestimmter Roboter wird penalisiert
- Ein bestimmtes Team kommt in Ballbesitz
- Ein bestimmter Roboter betritt einen angegebenen *Skill*

Ein *Skill* ist eine autonome Komponente, die Befehle einer höheren Strategie in-

terpretiert. Dies können Aktionen wie das Schießen oder Passen sein (vgl. B-Human, 2024a). Darüber hinaus wurde beschlossen, dass sowohl die jeweilige Negation als auch die Konjunktion mehrerer Ziele überprüfbar sein sollen. Da die Teilnehmer sowohl in der *Fast*-Szene als auch in der *PerceptOracle*-Szene testen, soll die Testdurchführung in beiden Szenen möglich sein, wobei für *Fast*-Szenen zusätzlich auch der 2D-Modus unterstützt werden soll.

Die Umfrage zeigt zudem, dass das Testen mit beschleunigter Zeit als wichtig erachtet wird, weshalb diese Funktionalität ebenfalls als Ziel aufgenommen wurde. Für die Auswertung ganzer Spiele sollen lediglich die Anzahl der Tore und die Anzahl der Siege berücksichtigt werden. Letzteres aufgrund der einfachen Implementierung und der engen Verknüpfung mit der Toranzahl.

Für das Testen von Situationen wurde weiterhin festgelegt, dass die Erzeugung von modifizierten Positionen und Rotationen der Roboter und des Balls mindestens in *Fast*-Szenen möglich sein soll. Diese modifizierten Positionen und Rotationen sollen am Ende der Tests für fehlgeschlagene Testdurchläufe gespeichert werden.

Zudem sollen die Tests nicht nur lokal, sondern auch in einer *GitLab-Pipeline* durchgeführt werden können. Eine Erklärung zu dem Begriff *Gitlab-Pipeline* ist in Kapitel 2.5 zu finden. Außerdem soll das Test-Framework innerhalb des Simulators steuerbar sein und keinen externen Ansatz, beispielsweise durch ein Skript, benutzen.

Heatmaps der Ballposition, Statistiken zum Spielstand vor einem Tor und Ausgabe des prozentualen Ballbesitzes wurden zunächst nicht als Ziele aufgenommen, da dies den Umfang dieser Arbeit überschreiten würde.

1.4 Verwandte Arbeiten

1.4.1 Verwandte Arbeiten im RoboCup Umfeld

Eine Inspiration für diese Arbeit stammt von Lienhoop (2022). Er simulierte für die Evaluation seines neu entwickelten Passverhaltens mehrere Halbzeiten in *SimRobot* mittels eines Python-Programms, das die simulierten Spiele im Simulator konfiguriert und startet. Lienhoop (2022) setzte den von Hasselbring (2020) konzipierten Trainer ein, welcher mittlerweile obsolet geworden ist und aus dem Quellcode entfernt wurde, um ganze Halbzeiten zu simulieren und die Ergebnisse extern als Konsolenausgabe zur Verfügung zu stellen. Diese Ergebnisse wurden ebenfalls vom Python-Programm ausgelesen, ausgewertet und archiviert. Lienhoop (2022) verfolgte für die Evaluation also einen externen Ansatz, um *SimRobot* initial zu starten, nach jeder Halbzeit neu zu starten und die Ergebnisse dem Nutzer zur Verfügung zu stellen. Da der Trainer

nicht mehr vorhanden ist, ist dieser Ansatz aktuell nicht mehr möglich. Zudem soll diese Arbeit den Testaufruf und die Testdurchführung innerhalb des Simulators ermöglichen, sodass kein externes Programm oder Skript dafür notwendig ist.

Eine weitere verwandte Arbeit stammt von der Universität Hamburg. Sell (2021) entwickelte eine Lösung für das *RoboCup* Team *Hamburg Bit-Bots*, die es ermöglicht Tests zu erstellen und ausführen. Der Fokus von Sell (2021) lag dabei eher darauf die Möglichkeit zu schaffen, dass Tests ausgeführt werden, wenn neuer Code zu dem bestehenden hinzugefügt wird. Die Tests für den C++-Code, also die Programmiersprache, welche auch *B-Human* zum Großteil einsetzt, wurden dabei mit dem *Google Test Framework* geschrieben. Unabhängig von der Implementierung ergab die Evaluation von Sell (2021), dass die entwickelten Software-Test-Praktiken wertvoll für das Team sind. Trotz einiger grundlegender Unterschiede, wie die Nutzung des *Google Test Frameworks* und die Erstellung von *Unit-Tests*, sind zwischen der Arbeit von Sell (2021) und dieser Gemeinsamkeiten zu erkennen. So beispielsweise die Verwendung der *Pipeline* und die automatisierte Testausführung beim Hinzufügen von neuem Code.

Klute (2024) vom *RoboCup* SPL Team *Nao Devils* entwickelte eine *Continuous Integration (CI) Pipeline* zur Automatisierung von Tests, um Leistungsmessungen durchzuführen und Qualitätsmetriken zu erheben. Getestet wird dabei in *SimRobot* mittels Kommandos und *Event Listenern*. Die Kommandos erzeugen verschiedene Ereignisse, wie beispielsweise eine Ballbewegung, wobei Geschwindigkeit und Ziel variabel sind. Klute (2024) hebt hervor, dass ein solches Kommando gegenüber einem Roboter, welcher den Ball schießt, den Vorteil hat, dass keine Ungenauigkeiten beim Schuss entstehen. Die *Event Listener* für bestimmte Aktionen, wie erzielte Tore, erlauben das Beobachten und Verfolgen von diesen Events während der Tests. Zudem werden in jedem Test verschiedene Metriken erhoben, wie beispielsweise verhinderte oder erzielte Tore. Das von Klute (2024) entwickelte Tool ermöglicht es außerdem vordefinierte Tests vor dem Veröffentlichen von Code lokal zu testen, aber auch den Fortschritt mit jeder Veröffentlichung zu erkennen. Die Arbeit macht es also möglich einzelne Testszenen sehr detailliert zu testen. Ganze Spiele sind allerdings nicht möglich. Klute (2024) gibt zudem an, dass Experten von verschiedenen *RoboCup* SPL Teams die Arbeit als geeignet und beeindruckend bewerteten. Diese Arbeit hat sich allerdings zum Ziel gemacht auch die Verhaltenssoftware der Roboter zu testbar zu machen. Dass man hierfür lediglich Roboter und keine Kommandos benutzt, ergibt durchaus Sinn, da so eine gesamte Kette von Verhaltenssoftware in einer Testszene geprüft wird.

Gabel und Godehardt (2024) betonen in ihrer Arbeit an, dass die Evaluation der Leistung eines *RoboCup* Teams anhand von wenigen Spielen unzureichend ist. Godehardt u. a. (2023) bestätigen dieses und geben an, dass ungefähr 1000 Spiele notwendig sind

um ein fundiertes Resultat, ob die jüngsten Änderungen des Codes nützlich waren, zu erhalten. Dies ist darauf zurückzuführen, dass die Leistung einer Mannschaft nach nur wenigen Spielen nur schwer beurteilt werden kann. Godehardt u. a. (2023) lassen die Spiele nach der Veröffentlichung von neuem Programmcode automatisch ausführen. Die dazugehörigen Ergebnisse werden dabei von den Testcomputern auf einem Webserver abgelegt. Zudem werden nächtliche Testspiele durchgeführt. Lienhoop (2022) gab in seiner Arbeit eine Messung der Simulationsgeschwindigkeit des 2D- und 3D-Modus für *B-Human* an. Für 1000 Spiele benötigt der 2D-Modus 6 Stunden und der 3D-Modus 116 Stunden. Eine Simulation von 1000 Spielen pro Nacht wäre also höchstes im 2D-Modus sinnvoll und überhaupt möglich, wenn man nur einen Testcomputer nutzt. Gabel und Godehardt (2024) nutzen für die Ausführung der Spiele 20 Rechner, was die benötigte Zeit drastisch verringert. Außerdem speichern sie die Log-Dateien von Ausreißer-Spielen, wobei diese außergewöhnlich gut oder schlecht sein können. Das Team analysiert diese dann, um den Programmcode zu optimieren. Dies stellt eine äußerst sinnvolle Mechanik zur kontinuierlichen Verbesserung des Programmcodes dar und wird auch als Ziel in dieser Arbeit verfolgt.

1.4.2 Weitere verwandte Arbeiten

Es gibt außerdem noch Arbeiten außerhalb des *RoboCup* Umfelds, welche einen Zusammenhang zu dieser Arbeit haben. So entwickelten Park und Kang (2012) ein Framework zum automatischen Testen von Roboter-Software in der Simulation. Während des Testens werden die unterschiedlichen Testfälle, welche in einer Datei formuliert sind, eingelesen und die Simulationsumgebung aufgesetzt. Dies ist vergleichbar mit dem Positionieren der Roboter auf den gewünschten Positionen des Spielfelds in *SimRobot*. Nach Beendigung der Testfälle werden die Ergebnisse in eine Datei geschrieben. Diese sind durch ein XML-Schema strukturiert, was die Lesbarkeit durch andere Programme sicherstellt. Dies ist zu vergleichen mit der Ausgabe als *CSV*-Datei in dieser Arbeit.

Robert, Guiochet und Waeselynck (2020) argumentieren, dass durch den Nichtdeterminismus von Robotern und deren Simulation Testfälle, welche exakt dieselben Eingabeparameter und Umgebungen haben, in aufeinanderfolgenden Testdurchläufen sowohl fehlschlagen als auch erfolgreich sein können. Robert, Guiochet und Waeselynck (2020) kamen in ihrer Studie zu dem Resultat, dass man effizienter testet, indem man eine höhere Vielfalt von Testfällen testet, als denselben Testfall repetitiv auszuführen. Dies legt nahe, dass verschiedene, sich stark ähnelnde Testfälle gut eignen, um bestimmte Komponenten oder Funktionen eines Roboters zu testen. Das Unterziel dieser Arbeit Positionen und Rotationen der Objekte automatisch modifizieren zu können, ähnelt sich

stark mit der Argumentation von Robert, Guiochet und Waeselynck (2020).

1.5 Aufbau der Arbeit

In Kapitel 2 werden zunächst die grundlegenden Konzepte und Fachbegriffe eingeführt, die für das Verständnis der Arbeit notwendig sind. Hierbei werden wichtige Definitionen erläutert, die als Basis für die nachfolgenden Kapitel dienen. Außerdem wird ein Einblick ins Testen in der Robotik gegeben. Das Kapitel 3 beschäftigt sich mit der detaillierten Erklärung der Implementierung des Test-Frameworks. Es werden die technischen Details und Schritte beschrieben, die zur Entwicklung und zum Aufbau des Frameworks erforderlich waren. Kapitel 4 bewertet die Ergebnisse, die durch die Implementierung des Test-Frameworks erzielt wurden. Dabei werden die Ergebnisse analysiert und interpretiert. Kapitel 5 fasst die wichtigsten Erkenntnisse der Arbeit zusammen. Zudem wird ein Ausblick auf mögliche zukünftige Entwicklungen gegeben.

2 Grundlagen

2.1 RoboCup SPL

Beim *RoboCup* treten in der *Standard Platform League* (SPL) zwei Teams mit je sieben baugleichen, humanoiden Robotern gegeneinander an. Da sich die Roboter hardwareseitig bis auf den Verschleiß nicht unterscheiden, gewinnt in der Regel die beste Software das Spiel und am Ende auch das gesamte Turnier. Das Spielfeld hat dabei eine Länge von neun Metern und eine Breite von sechs Metern. In jedem Spiel werden zwei Halbzeiten mit einer jeweiligen Länge von zehn Minuten gespielt. Zu Beginn einer Halbzeit führt eines der Teams einmalig einen initialen Anstoß aus. Dieses Recht wechselt mit dem Ende der ersten Halbzeit zu dem Team, welches diesen in der ersten Halbzeit nicht ausgeführt hat. Die Roboter in der SPL werden von niemandem gesteuert, sondern agieren autonom. Sie verfügen über zwei Kameras und Mikrofone sowie weiteren Sensoren und treffen beispielsweise anhand dieser Entscheidungen für den nächsten Zyklus. Roboter können penalisiert werden. Dies kann beispielsweise eintreten, wenn sie sich beim Start eines Spiels falsch positionieren.

Das allgemeine Ziel des *RoboCups* ist es, dass bis zur Mitte des 21. Jahrhunderts humanoide Roboter ein Spiel unter den offiziellen Regeln der FIFA gegen den amtierenden Fußball-Weltmeister gewinnen (vgl. RoboCup, 2024b).

2.2 Testen in der Robotik

Laut Belli (1998) stellt eine Prüfung „i.allg. einen Vergleich der erwarteten Werte (‚Referenzwerte') entsprechend einer Aufgabenstellung (‚Spezifikation, Problem') mit den tatsächlich gemessenen oder beobachteten Werten eines Systems bzw. eines Prozesses dar". Des Weiteren ermöglicht eine Prüfung die Identifikation, Lokalisierung und Korrektur von Fehlern. Wenn man nicht den ganzen Eingabebereich E, sondern eine endliche Teilmenge T davon betrachtet, so spricht man von einem Test. Ein Test dient also zur Erkennung, Lokalisierung und Korrektur von Fehlern (vgl. Belli, 1998).

Softwaretests prüfen nach Weißhardt (2024) „einen Softwarebaustein, durch einen jederzeit wiederholbaren Nachweis, auf die Erfüllung von zuvor festgelegten Anforderun-

gen“ und „misst dessen Qualität“. Es werden somit nicht nur Fehler erkannt, lokalisiert und korrigiert, sondern dazu wird auch noch die Qualität der geschriebenen Software gemessen. Die Qualität von Software kann dabei zum Beispiel durch Faktoren wie Effizienz, Performanz, Sicherheit und Korrektheit dargestellt werden (vgl. McCall, 1977).

In der Robotik können manuelle Tests in realen Szenarien, sogenannte Feldtests, zu gefährlichen und kostspieligen Fehlern führen. Darüber hinaus sind Feldtests durch den Umfang der Szenarien und Umgebungen, auf die sie angewendet werden können, stark eingeschränkt (vgl. Afzal, 2021). Beispielsweise können extrem helle Lichtverhältnisse, welche die Kameras von Robotern vor großen Herausforderungen stellen, nur schwer nachgestellt werden. Man müsste entweder darauf warten, dass diese Lichtverhältnisse natürlich eintreten. Alternativ könnte man diese künstlich selber nachstellen, was hohe Kosten verursachen könnte. Aus diesem Grund wird häufig mit *Software-in-the-loop* Simulatoren getestet (vgl. Afzal, 2021). Hierbei werden realisierte Software-Komponenten in einer simulierten Umgebung ausgeführt (vgl. Schäuffele und Zurawka, 2010). Diese sind in der Lage selbst mit geringer Wiedergabetreue ein großes Maß an Softwarefehlern zu entdecken (vgl. Afzal, 2021). Witte (2016) schreibt dazu, dass Tests, die wiederholt ablaufen, automatisiert werden können. Dies führt zu einer höheren Testabdeckung und einer verlässlichen Reproduzierbarkeit und spart Zeit und Kosten.

2.3 SimRobot

2.3.1 Allgemeines

SimRobot ist eine Robotersimulation für einen einzelnen Benutzer an einem einzelnen Arbeitsplatz (vgl. Laue und Röfer, 2008). Der Simulator ist dabei in der Lage mehrere Roboter zu simulieren. Die Roboter durchlaufen verschiedene Zustände und *Skills*, welche über den *BehaviourView* angezeigt werden können. Diese Verhaltensansicht zeigt alle aktuell aktiven Zustände und *Skills* des jeweiligen Roboters an. Der *SceneGraph* zeigt alle Objekte der aktuellen Szene an und ermöglicht es, beispielsweise die *Console* zu öffnen (vgl. Röfer, Laue, Baude u. a., 2019). Zu den *Skills* gehören unter anderem *stand*, *walk*, *shoot* und *observe*. In *SimRobot* gibt es zudem die Möglichkeit verschiedene Szenenarten zu nutzen um möglicherweise Ressourcen sparen zu können. In den *PerceptOracle*-Szenen werden keine Kamerabilder gerendert und die Wahrnehmung aller Roboter werden direkt vom Simulator zur Verfügung gestellt. In dieser Szene müssen sich die Roboter allerdings noch selbständig auf dem Spielfeld orientieren. In den *Fast*-Szenen werden noch mehr Ressourcen gespart, indem sogar die Modelle durch Daten aus dem Simulator ersetzt werden. Hier ist beispielsweise jedem Roboter stets bewusst, wo er und der

Ball sich aktuell befinden (vgl. Röfer, Laue, Baude u. a., 2019). Zu jeder Szenendatei gibt es eine dazugehörige Skriptdatei, welche benötigte Kommandos beinhaltet und beim Laden der Szene aufgerufen wird. Zudem können eigene Skriptdateien erstellt und ausgeführt werden, welche Kommandos vom Nutzer beinhalten. Für die Unterscheidung dieser Skriptdateien, werden die händisch erzeugten Skriptdateien in dieser Arbeit Nutzerskriptdateien genannt.

Wie auch bei einem *RoboCup* SPL Spiel, gibt es im Simulator verschiedene zu durchlaufende Spielzustände, die in dieser Arbeit eine Rolle spielen. Im Zustand *initial* können sich die Roboter frei bewegen, stehen im Simulator dabei aber zunächst am Seitenrand. Sobald der Zustand *ready* erreicht ist, begeben sich die Roboter für den Anstoß auf ihre Positionen. Während *set* stoppen die Roboter und warten auf den Anstoß. Im Zustand *playing* wird das eigentliche Fußballspiel der Roboter durchgeführt. Der Zustand *finished* tritt ein, wenn eine Halbzeit beendet wurde (vgl. RoboCup Technical Committee, 2022).

SimRobot unterstützt neben dem 3D-Modus (siehe Abbildung 2.1) auch einen 2D-Modus. Der 2D-Modus wurde nachträglich von Hasselbring (2020) implementiert, da dieser unter anderem eine höhere Simulationsgeschwindigkeit als der 3D-Modus bietet. Dies ist möglich, da die Kollisions- und Dynamikberechnung hier schneller ist und auf die Simulation von Sensoren und Ausführung der Bewegungssteuerung verzichtet wird. In Abbildung 2.2 ist der 2D-Modus zu sehen.

2.3.2 Konsole

Über die Konsole ist es in *SimRobot* möglich verschiedene Kommandos einzugeben. Diese Kommandos unterscheiden sich in *Initialisierungs Kommandos*, *Globale Kommandos* und *Roboter Kommandos*. *Initialisierungs Kommandos* werden hauptsächlich in Skriptdateien genutzt, welche ausgeführt werden, wenn die Simulation startet. *Globale Kommandos* ändern den Status der gesamten Simulation. *Roboter Kommandos* beeinflussen nur aktuell ausgewählte Roboter. Für diese Arbeit wird eine Auswahl von *Globalen Kommandos* und *Roboter Kommandos* nach Röfer, Laue, Baude u. a. (2019) näher betrachtet, da diese in der Implementierung des Test-Frameworks eingesetzt werden:

Globale Kommandos

call Führt die angegebene Skriptdatei aus.

dt Mit dem Paramter *off* ist es möglich, dass die Simulation schneller als Echtzeit, so schnell wie aktuell möglich, ausgeführt wird. Bei Übergabe von einer Taktfrequenz wird der Simulator mit dieser ausgeführt. Mit dem Standardwert *on*, wird der Simulator in Echtzeit ausgeführt.

Abbildung 2.1: Eine laufende 3D-Szene in *SimRobot*. In dieser spielen zwei Teams mit jeweils sieben Roboter gegeneinander. Am linken Rand sieht man den *SceneGraph*, am rechten Rand die *Konsole* zur Eingabe von Kommandos.

gc Setzt den aktuellen Status des Spiels. Diese können unter anderem *initial*, *ready*, *set*, *playing* oder *finished* sein.

mvo Bewegt das angegebene Objekt zu der mitgegebenen Position und rotiert dieses anhand der mitgegebenen Rotation.

sv Speichert die aktuelle Position der Roboter und des Balls in eine Skriptdatei. Für *Fast*-Szenen wird eine Reihe von *mvo*-Befehle gespeichert. Für *PerceptOracle*-Szenen werden die Positionen über den *StaticInitialPoseProvider* geladen. Eine Klasse, welche sicherstellt, dass die Roboter nach der Positionierung wissen, wo sie sich auf dem Feld befinden.

dr Aktiviert *Debug Requests* der angegebenen Klasse, welche beispielsweise zur Fehlersuche dienen können.

Roboter Kommandos

mv Bewegt das aktuell ausgewählte Objekt zu der angegebenen Position und rotiert dieses anhand der mitgegebenen Rotation.

Alternativ können die Befehle auch in eine Skriptdatei geschrieben und ausgeführt werden. Hierbei wird jede Zeile der Skriptdatei, welche nicht leer ist, nacheinander ausgeführt.

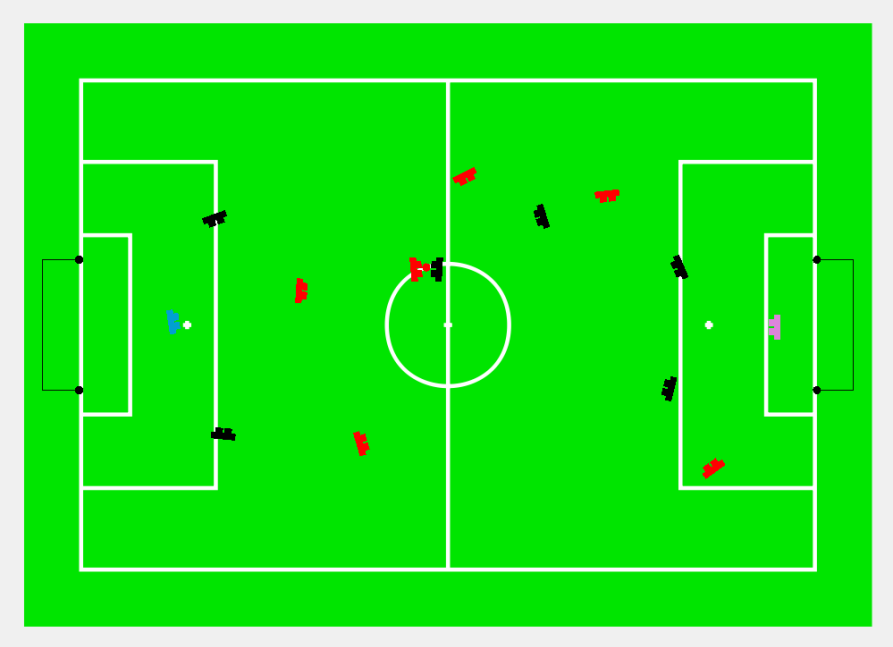


Abbildung 2.2: Eine laufende 2D-Szene in *SimRobot*. In dieser spielen zwei Teams mit jeweils sieben Robotern gegeneinander. Der Ball befindet sich aktuell zwischen den beiden Robotern innerhalb des Mittelkreises.

2.3.3 Räumliche Dimensionen

Den Ursprung des Koordinatensystems in *SimRobot* stellt der Mittelpunkt des Spielfeldes dar (siehe Abbildung 2.3). Die X-Achse erstreckt sich in Richtung des Tores des linken Teams, während die Y-Achse vom Tor des rechten Teams aus gesehen nach links verläuft. Die Z-Achse verläuft vom Spielfeld aus nach oben. Der Abstand zwischen dem linken und dem rechten Tor beträgt dabei 9000 mm. Die Distanz zwischen der rechten und linken Seitenauslinie beträgt 6000 mm. Der in Abbildung 2.3 oben rechts erkennbare Eckpunkt hat somit die Koordinaten X = -4500 und Y = -3000.

2.3.4 Architektur

Wie in Abbildung 2.4 zu sehen, besteht *SimRobot* grundsätzlich aus vier Komponenten. Zu diesen gehören *Controller*, *SimRobotCore2*, *SimRobotCore2D* und die *GUI*. Der *Controller* wird in jedem Simulationsschritt von der Simulation aufgerufen, liest die verfügbaren Sensoren aus, plant die nächste Aktion und stellt die Aktoren auf die gewünschten Zustände ein. *SimRobotCore2* und *SimRobotCore2D* sind die Simulationskerne. Sie modellieren die Roboter und die Umgebung, simulieren Sensorwerte und führen Befehle aus, die von der Steuerung oder dem Benutzer gegeben werden. Die *GUI*

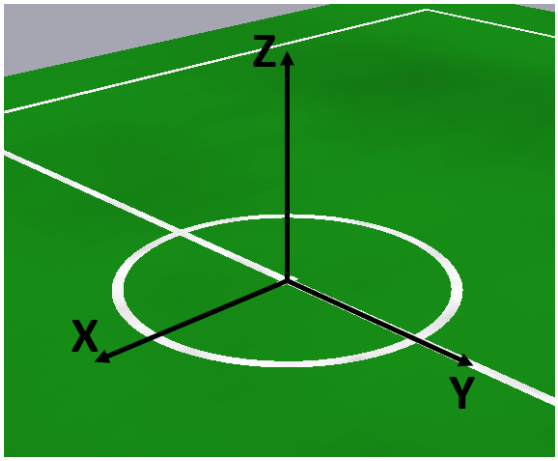


Abbildung 2.3: Koordinatensystem im Simulator. Die X-Achse erstreckt sich in Richtung des Tores des linken Teams, während die Y-Achse vom Tor des rechten Teams aus gesehen nach links verläuft. Die Z-Achse verläuft vom Spielfeld aus nach oben. (Bild: *B-Human*)

stellt Informationen dar und erlaubt die Interaktion mit dem Nutzer. *SimRobotCore2D* wurde zudem von Hasselbring (2020) für die 2D-Simulation entwickelt. Die *Scene* ist eine XML-Beschreibung und spezifiziert die Roboter und die Umgebung und wird zur Laufzeit geladen (vgl. Laue, Spiess und Röfer (2006)). In Summe ergibt sich daraus dann das *SimRobot Executable*.

2.3.5 Controller und Automatic Referee

Der *Controller* ist bei *B-Human* in der Klasse *RoboCupCtrl* implementiert. Diese Klasse enthält eine Liste von Objekten der Klasse *SimulatedRobot*, die als Schnittstelle zu den simulierten Robotern dient. Die Verarbeitung von Konsoleneingaben des Simulators erfolgt über die Klasse *ConsoleRoboCupCtrl*, die gleichzeitig von *RoboCupCtrl* erbt. Die Klasse *RobotConsole* verwaltet die Textkonsole eines Roboters und greift dabei auf *ConsoleRoboCupCtrl* zu. Sowohl *ConsoleRoboCupCtrl* als auch *RoboCupCtrl* greifen auf den *GameController* zu (siehe Abbildung 2.5).

In der *RoboCup* SPL ist der *GameController* die Schiedsrichter-Anwendung für die Spielverwaltung. Er kontrolliert und steuert also das Spiel (vgl. Röfer, Laue, Lienhoop u. a., 2024). Der *GameController* in *SimRobot* ist das Äquivalent der Simulation dazu. Da es in der Simulation keinen menschlichen Schiedsrichter gibt, dient der *GameController* als *Automatic Referee*. *RoboCupCtrl* ruft zyklisch die Methode `update()` vom *GameController* und der simulierten Roboter auf. Falls nun beispielsweise das linke Team ein Tor erzielt hat, überprüft der *GameController*, ob in dem aktuellen Zustand

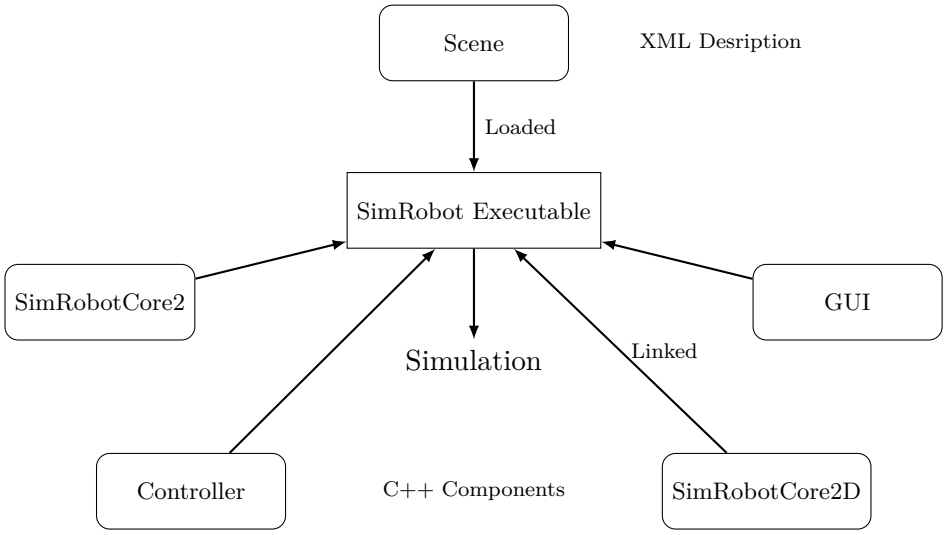


Abbildung 2.4: Die Module von *SimRobot* und ihre Abhängigkeiten. *Controller*, *SimRobotCore2*, *SimRobotCore2D* und die *GUI* sind C++-Komponenten. Bei der *Scene* handelt es sich um eine *XML Description*. Die vier C++ Komponenten ergeben verknüpft das *SimRobot Executable*. Die *Scene* wird zur Laufzeit geladen.

des Spiels überhaupt ein Tor erzielt werden durfte. Falls ja, zählt das Tor, falls nein, zählt es nicht. Der *Automatic Referee* lässt sich über die Konsole auch abschalten, für diese Arbeit hat er allerdings eine essenzielle Wichtigkeit.

2.4 Testarten und Techniken

Es gibt verschiedene Testarten und Techniken, welche in der Softwareentwicklung eingesetzt werden, um die Qualität der Software sicherzustellen und bestenfalls zu erhöhen. Im Folgenden werden einige für diese Arbeit relevante Testarten und Techniken erläutert. Daneben existieren noch weitere wie *Unit-Tests* oder *Integrationstests*, welche im Rahmen dieser Arbeit jedoch keine übergeordnete Rolle einnehmen.

2.4.1 Smoke-Test

Das regelmäßige Erstellen des Softwarestands und die Durchführung einfacher, automatisierter Tests sind gängige Praktiken in der Softwareentwicklung. Diese Tests überprüfen, ob die Software nach den vorgenommenen Änderungen grundsätzlich noch funktionsfähig ist. Solche einfachen Tests werden *Smoke-Tests* genannt und bieten einige Vorteile. In großen Teams arbeiten die einzelnen Teammitglieder oft separat an eigenen Kompo-

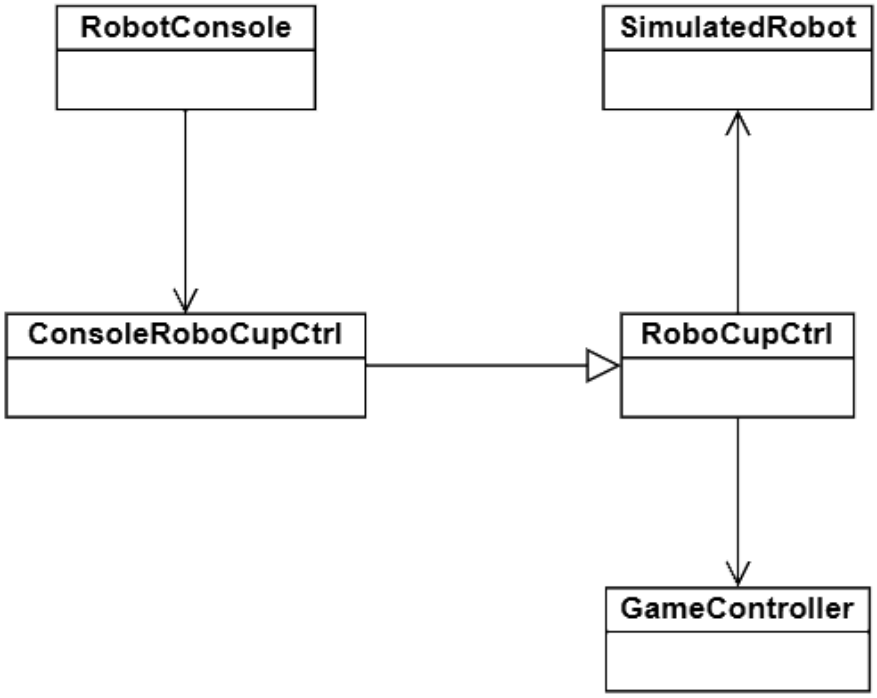


Abbildung 2.5: Klassendiagramm des Controllers.

nenten und integrieren diese schrittweise in das Gesamtsystem. Es besteht immer die Möglichkeit, dass der zusammengesetzte Code nicht optimal funktioniert. Regelmäßige *Smoke-Tests* können solche Schwachstellen schnell aufdecken und helfen, Fehler in einem beherrschbaren Rahmen zu halten. Darüber hinaus verringert diese Testart das Risiko von Qualitätsverlusten. Zu Beginn eines Projekts bringt man die Software auf einen stabilen Stand und stellt durch *Smoke-Tests* sicher, dass die Qualität nicht unter dieses Niveau sinkt. Zudem wirkt sich die erfolgreiche Durchführung der Tests positiv auf die Moral der Mitarbeiter aus, da es motivierend ist, wenn ein Produkt funktioniert, unabhängig davon, was es am Ende leisten soll (vgl. McConnell, 1996).

2.4.2 Regressionstest

Regressionstests dienen zur Verhinderung der Verminderung der Qualität des erstellten Produktes. Mit jedem neuen veröffentlichtem Code ist eine solche Regression möglich. Mit steigender Komplexität des Codes, steigt auch die Anzahl der notwendigen *Regressionstests* um eine Regression zu entdecken (vgl. *How To Do Good Regression Testing* 2017). Im Vergleich zu *Smoke-Tests* sind *Regressionstests* teurer, da sie wesentlich detaillierter sein müssen und eine regelmäßige Pflege erfordern, um sicherzustellen, dass

alle Funktionen abgedeckt sind. Sie bieten aber auch erhebliche Vorteile, darunter eine offensichtliche Verbesserung der Softwarequalität und die Sicherheit, dass Modifikationen oder Fehlerbehebungen keine unbeabsichtigten Änderungen an bestehenden Funktionen bewirken. Insgesamt tragen *Regressionstests* entscheidend zur Stabilität der Software bei (vgl. *What is Regression Testing?* 2024).

2.4.3 Fuzzing

Fuzzing beschreibt eine Testtechnik, bei welcher eine große Anzahl von Testfällen erzeugt wird, um so beispielsweise Fehler oder Abstürze aufzudecken. Dazu werden gezielt modifizierte Eingaben erstellt und an das zu testende Programm übergeben. Die Eingaben können dabei stochastischer Natur sein oder bestimmten Regeln folgen. Hierbei gibt es verschiedene Kategorien von *Fuzzing*, je nach dem Grad des Wissens über das zu testende Programm. Beim *Black-Box-Fuzzing* hat man kein Wissen über den Quellcode. Das *White-Box-Fuzzing* hingegen nutzt den Quellcode für die Eingabengenerierung. Im Anschluss an die Tests werden die identifizierten Fehler oder Abstürze untersucht, um diese zu beheben. Hierbei hängt der Erfolg des *Fuzzings* allerdings stark von der Qualität und Vielfalt der erzeugten Eingaben ab (vgl. Zhao u. a., 2024).

2.5 GitLab CI/CD Pipelines

B-Human nutzt das Tool *GitLab* für die Zusammenarbeit am Programmcode und die Versionsverwaltung. Neben diesen Punkten bietet *GitLab* die Möglichkeit von *CI/CD Pipelines*. Eine Pipeline definiert eine Reihe von Schritten, welche auf die Projektdatei angewendet werden. *CI* steht für *Continuous Integration* und soll sicherstellen, dass durch neue Arbeiten am Code keine neuen Probleme entstehen. Falls doch Probleme auftreten, sollen diese frühzeitig erkannt werden. *CD* steht für *Continuous Delivery* und *Continuous Deployment* und stellt sicher, dass Änderungen am Code in die Produktionsumgebung übertragen werden können (vgl. Cowell, Lotz und Timberlake, 2023).

Bei *B-Human* wird die Pipeline aktuell nur sehr rudimentär eingesetzt. Nach jedem Commit eines Nutzers werden *Pre-Commits* ausgeführt, die den Programmcode beispielsweise auf Rechtschreibfehler überprüfen oder sicherstellen, dass festgelegte Coderichtlinien eingehalten werden. Darüber hinaus wird der Code wöchentlich einer detaillierteren Prüfung unterzogen, bei der unter anderem ungenutzte Konfigurationsdateien identifiziert werden. Beide Beispiele gehören zur *Statischen Codeanalyse*, bei der der Programmcode auf Fehler untersucht wird, ohne ihn auszuführen (vgl. Louridas, 2006).

Die detaillierteren Prüfungen werden in *Docker-Containern* ausgeführt, welche leichtgewichtige Softwarepaketierungsmechanismen sind und alle externen Abhängigkeiten der Anwendung, wie Bibliotheken und Konfigurationen, beinhalten (vgl. Schenker, 2023). In diesen Containern laufen die *GitLab Runner*, welche die Kommandos ausführen, sofern *Docker* als Ausführungsumgebung für die *Runner* konfiguriert ist (vgl. Cowell, Lotz und Timberlake, 2023).

2.5.1 Konfiguration

Konfiguriert wird die *CI/CD Pipeline* über die projekteigene *.gitlab-ci.yml*-Datei. In dieser können für jeden *Job*, welcher für die Ausführung einer Aufgabe steht, per Parameter festgelegt werden, welche Aktion durchgeführt werden soll. Im Folgenden werden die für diese Arbeit relevanten Parameter erläutert.

Parameter

allow_failure Gibt an, ob der *Job* fehlschlagen darf, ohne dass die gesamte *Pipeline* fehlschlägt.

before_script Eine Anzahl von Befehlen, die vor dem *Job* ausgeführt werden.

image Gibt das gewünschte *Docker*-Software-Abbild an.

script Das Skript, welches vom *Runner* ausgeführt wird.

stage Definiert in welcher Phase der Job laufen soll. Dies können unter anderem *build*, *test* oder *deploy* sein.

tags Liste der Begriffe, die für die Auswahl eines *Runners* verwendet werden.

variables Definiert Variablen für die *Jobs*.

(vgl. GitLab, o. D.)

3 Implementierung

Der erstellte Programmcode ist im *B-Human*-Repository zu finden.

3.1 Architektur

Im Rahmen dieser Bachelorarbeit wurde die Architektur von *SimRobot* durch gezielte Anpassungen erweitert. Es wurde eine neue Klasse namens *TestController* erstellt, die innerhalb der Klasse *GameController* verwendet wird, wie in Abbildung 3.1 dargestellt. Des Weiteren verwenden die Klassen *GameController* und *TestController* nun die Klasse *RoboCupCtrl*. Dies ist notwendig, da so beispielsweise direkt in den Klassen auf die Schnittstelle zur *SimRobot GUI* und dem *SimRobot Controller* zugegriffen werden kann. Das bietet unter anderem die Vorteile, dass mittels des Namens eines Roboters auf das dazugehörige Objekt der Simulation zugegriffen und zudem zwischen 2D- und 3D-Simulation unterschieden werden kann.

3.2 Testarten

Es wurden zwei verschiedene Testarten implementiert, die über Parameter konfiguriert werden können. Diese Parameter werden dem *TestController* vor dem Start der Tests übermittelt. Es können sowohl komplette Spiele als auch einzelne Situationen getestet werden. Die Wahl der Szenenart, ob in einer *Fast*-Szene oder einer *PerceptOracle*-Szene getestet wird, erfolgt durch das Laden der entsprechenden Szene in *SimRobot*.

3.3 Benutzerschnittstelle

Als Benutzerschnittstelle wurde die Konsole von *SimRobot* gewählt, da die bisherige Steuerung des Simulators durch den Nutzer ebenfalls über die Konsole stattfindet. Für die Bedienung des Test-Frameworks wurde ein neues Kommando `test` in die Konsole implementiert. Dieses erwartet weitere Parameter, welche genau definieren, was für ein Test gestartet werden soll oder ob ein Test gestoppt werden soll. Die Auswertung dieser Parameter findet weiterhin in der Methode `executeConsoleCommand()` der

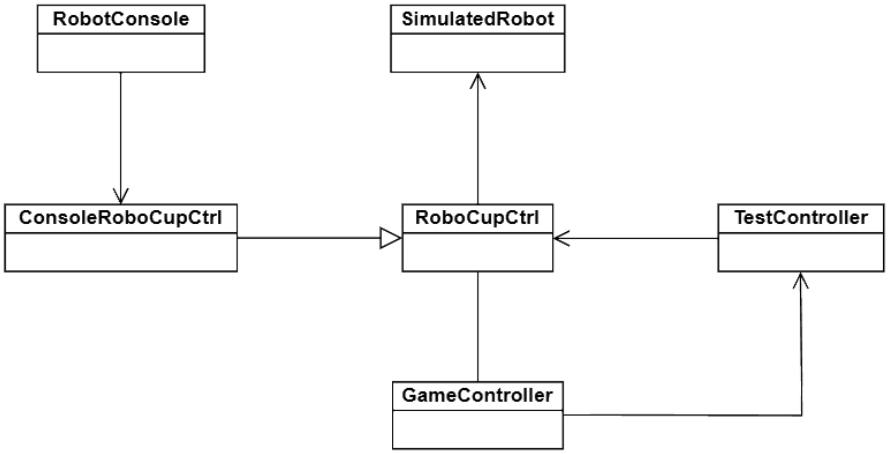


Abbildung 3.1: Klassendiagramm des Controllers nach der Implementierung. Eine neue Klasse *TestController* wurde erstellt und wird im *GameController* verwendet. Außerdem verwenden die Klassen *GameController* und *TestController* nun die Klasse *RoboCupCtrl*.

Klasse *ConsoleRoboCupCtrl* statt. Hier wird das eingegebene Kommando des Nutzers, welches vom Typen *String* ist, in einen Eingabestrom gewandelt. So kann das Kommando Wort für Wort interpretiert werden. Diese bereits bestehende Funktion wurde um die Interpretation des `test`-Kommandos erweitert, welche im Folgenden erklärt wird.

Den Start einer Testkonfiguration kennzeichnet das Wort `test`. Dann folgen die Möglichkeiten `help`, `stop`, `game` und `situation`. Durch Eingabe von `help` wird der Link zu einer Wiki-Seite ausgegeben, auf der die Bedienung des Test-Frameworks mit Beispielen erklärt wird. `stop` signalisiert dem *TestController* mittels eines Funktionsaufrufs über den *GameController*, dass ein laufender Test gestoppt werden soll, was in Kapitel 3.7 näher beschrieben wird. Nach diesen beiden Eingaben wird der Eingabestrom nicht weiter ausgelesen.

Falls das zweite Wort `game` lautet, werden als Nächstes die Anzahl der zu spielenden Spiele erwartet. Zudem kann optional eine Nutzerskriptdatei, sowie die Parameter *Realtime* und *Quit* angegeben werden. Wenn das zweite Wort `situation` lautet, werden als Nächstes die Anzahl der zu spielenden Situationen, sowie die Anzahl der maximal zu spielenden Sekunden pro Situation erwartet. Zudem kann optional eine Nutzerskriptdatei und der *Fuzzing*-Parameter, sowie die Parameter *Realtime* und *Quit* angegeben werden. Abschließend kann eine beliebig lange Liste von Zielen angegeben werden. Im Folgenden werden die einzelnen Parameter erläutert.

Allgemeine Parameter

Nutzerskriptdatei Optionaler Parameter, mit welchem zum Start jeder Halbzeit eine Skriptdatei ausgeführt werden kann.

Realtime Optionaler Parameter, welcher angibt, ob die Tests in Echtzeit durchlaufen werden sollen. Falls dieser nicht angegeben ist, laufen die Tests in beschleunigter Zeit ab.

Quit Optionaler Parameter, welcher angibt, ob nach Beendigung der Tests das Programm geschlossen werden soll. Bei Beendigung des Programms wird bei Erfolg `EXIT_SUCCESS` und bei Misserfolg `EXIT_FAILURE` zurückgegeben.

Spezifische Parameter für ganze Spiele

Anzahl Spiele Parameter, welcher angibt, wie viele Spiele gespielt werden sollen.

Spezifische Parameter für Situationen

Anzahl Situationen Parameter, welcher angibt, wie viele Situationen gespielt werden sollen.

Sekunden Parameter, welcher angibt, wie lange ein einzelner Testdurchlauf maximal dauern darf.

Fuzzing Optionaler Parameter, welcher angibt, ob die in der Nutzerskriptdatei übergebenen Positionen und Rotationen für die Tests leicht verändert werden sollen.

Ziele Parameter, mit welchem eine Liste von Zielen des Tests festgelegt werden. Diese werden konjunktioniert und können auch jeweils mittels eines Parameters negiert werden. Folgende Ziele wurden implementiert:

Tor linkes Team	Das linke Team muss ein Tor erzielen.
Tor rechtes Team	Das rechte Team muss ein Tor erzielen.
Roboter in Zone	Der angegebene Roboter muss die angegebene Zone betreten.
Roboter berührt Ball	Der angegebene Roboter muss den Ball berühren.
Ball in Zone	Der Ball muss die angegebene Zone betreten.
Team in Ballbesitz	Das angegebene Team muss in Ballbesitz sein.
Roboter penalisiert	Der *Automatic Referee* muss den Roboter penalisieren.
Roboter in Skill	Der angegebene Roboter muss den angegebenen *Skill* betreten.

3.4 Testinitialisierung

In *TestController* wurde eine Datenstruktur `TestData` definiert. *ConsoleRoboCupCtrl* initialisiert nach Auslesen der Parameter des Nutzers diese Struktur, in der die Parameter des Nutzers gespeichert werden. Diese Parameter werden in den entsprechenden Variablen der Struktur abgelegt.

Außerdem beinhaltet die Struktur noch eine Angabe über die Anzahl der bereits absolvierten Testdurchläufe, wie viele davon erfolgreich waren, das aktuelle Ergebnis nach Toren und die Ergebnisse aller bereits absolvierten Testdurchläufe des Tests. Die Ziele der Situations-Tests werden mittels eines Vektors vom Typen `TestTargetData` gespeichert. Dies ist eine weitere Struktur, welche die Testziele speichert. Für jedes angegebene Ziel wird ein Objekt der Struktur `TestTargetData` erstellt. Dies beinhaltet außerdem ein Flag, welches aussagt ob das Ziel in dem aktuellen Testdurchlauf schon erfüllt wurde und ob das Testziel negiert vom Nutzer eingegeben wurde. Beide implementierten Datenstrukturen wurden als *streamable* definiert, was den Vorteil bietet, dass sie in einen Datenstrom gewandelt werden können.

Zum Abschluss der Testparametrisierung wird die Methode `startTest()` des *GameControllers* aufgerufen. Die Methode besitzt als Übergabeparameter die erstellte `TestData`-Datenstruktur. Wenn der aktuelle Spielzustand *initial* ist, wird die `init()`-Methode des *TestControllers* auferufen. Diese erhält als Übergabeparameter den Pfad zur aktuellen Szenedatei und die erstellte `TestData`-Datenstruktur.

In der `init()`-Methode werden die Szenedatei und die dazugehörige Skriptdatei unter neuem Namen kopiert, sodass benötigte Anpassungen an diesen keine Auswirkungen auf das Nutzererlebnis außerhalb des Testbetriebs haben. Dazu werden beide Dateien mit der Namenserweiterung `AUTOTEST` vor der Dateiendung versehen. Falls der *Realtime*-Parameter nicht gesetzt ist, wird `dt off` ans Ende der Skriptdatei ergänzt, um die Tests in beschleunigter Zeit auszuführen. Zudem wird mit dem `call`-Kommando eine weitere Nutzerskriptdatei aufgerufen, falls in den Parametern des Nutzers eine solche angegeben wurde. In dieser wird eine Kaskadierung von `test`-Kommandos verhindert, indem sie nach weiteren `test`-Kommandos durchsucht wird und diese entfernt werden. Der Inhalt dieser Nutzerskriptdatei wird zudem noch für das *Fuzzing* als *String* in eine entsprechende Variable von `TestData` gespeichert. Außerdem werden die *Debug Requests* der Klasse *SkillRequest* durch das `dr`-Kommando aktiviert, um Informationen zu erhalten, in welchen *Skills* sich die Roboter befinden.

3.4.1 Spezifische Anpassungen für ganze Spiele

Das Kommando `gc ready` wird ergänzt, damit zum Start jeder Halbzeit der normale *ready*, *set*, *playing* Ablauf durch den *Automatic Referee* automatisch durchlaufen wird. So ist keine Nutzerinteraktion zum Start eines Testdurchlaufs notwendig und die Tests starten automatisiert.

3.4.2 Spezifische Anpassungen für Situationen

Da die Situation sofort starten soll, wird das Kommando `gc playing` ergänzt. So wird nicht der Ablauf *ready*, *set*, *playing* wie in den Spielen durchlaufen, sondern die Roboter fangen sofort an zu spielen. Auch dies dient zur Automatisierung des Teststarts. Außerdem wird die *Debug Request* der Klasse *SetupPoses* entfernt, falls vorhanden, sodass die Roboter an einem beliebigen Punkt starten können und nicht an die initialen Positionen an den Außenlinien gebunden sind. Da Roboter penalisiert werden, wenn diese innerhalb der ersten zehn Sekunden eine falsche Position eingenommen haben, werden diese Sekunden übersprungen. Eine Situation kann somit maximal 590 Sekunden dauern und nicht eine komplette Halbzeit, welche 600 Sekunden lang ist. Ohne diese beiden Anpassungen würde der *Automatic Referee* alle bewegten Roboter penalisieren.

3.5 Persistente Datenhaltung

Da der Simulator nach der Testinitialisierung mit der neu erstellten Szene neu gestartet wird, ist es erforderlich, die `TestData`-Struktur persistent zu speichern. Dies ist auch notwendig, da der *TestController* mit jedem Neustart von *SimRobot* neu initialisiert wird und die Testdaten sonst verloren gehen. Daher speichert der *TestController* regelmäßig alle Testdaten in der Datei *test.running*. Dazu werden die Daten in einen Datenstrom umgewandelt, der sie in die Datei schreibt. In dieser sind die Daten mittels Schlüssel-Werte-Paare organisiert (vgl. B-Human, 2024c). Beim Start eines Testdurchlaufs werden die Daten, ebenfalls durch einen Datenstrom, aus dieser Datei erneut eingelesen und am Ende des Testdurchlaufs wieder gespeichert.

3.6 Testdurchführung

Zu Beginn des Tests wird *SimRobot* mit der neu erzeugten Szene, dessen Pfad die `init()`-Methode zurückgegeben hat, neu gestartet. Durch die Änderungen an der Skriptdatei beginnt sofort das Spiel. Für die Kontrolle der Testziele und der Speicherung

von Ereignissen, beispielsweise Toren, wurden in *TestController* entsprechende Methoden implementiert, welche durch den *GameController* zyklisch bzw. bei Eintritt eines Ereignisses aufgerufen werden. Der Zeitpunkt des jeweiligen Aufrufs wird im Folgenden erklärt:

Tor linkes Team In *GameController* wird die Methode `goal(int side)` aufgerufen, wenn ein Tor erzielt wurde. `side` gibt dabei an, welches Team ein Tor erzielt hat. Falls das Tor durch den *Automatic Referee* gezählt wird, wird im *TestController* ausgewertet, ob das linke Team für die Erfüllung des Ziels ein Tor erzielen musste.

Tor rechtes Team Die Auswertung erfolgt analog zu *Tor linkes Team*.

Roboter in Zone Dem *GameController* sind alle Positionen der Roboter in der `update()`-Methode bekannt. Zur Auswertung wird beim *TestController* die Liste aller Robotername angefragt, welche für Ziele relevant sind. Von diesen Roboternnamen wird die aktuelle Position gespeichert. Der *TestController* wertet im nächsten Schritt für alle Robotername aus, ob diese ein *Roboter in Zone*-Ziel besitzen und ob der Roboter sich aktuell innerhalb des gewünschten Bereichs befindet.

Roboter berührt Ball Der *GameController* speichert den letzten Ballkontakt in einer Variable und welcher Roboter diesen ausgeführt hat. So wird zyklisch im *TestController* ausgewertet, ob der Roboter mit dem letzten Ballkontakt in einem Ziel gespeichert war.

Ball in Zone Der *GameController* hat zu jedem Zeitpunkt Zugriff auf die aktuelle Position des Balls. Der *TestController* prüft dann, ob die aktuelle Position innerhalb des gewünschten Bereichs liegt.

Team in Ballbesitz Der Einfachheit halber wurde angenommen, dass ein Team in Ballbesitz ist, wenn ein Roboter des Teams den Ball als letztes berührt hat. Die Auswertung erfolgt dann analog zu *Roboter berührt Ball*, wobei hier nur wichtig ist, zu welchem Team der Roboter mit dem letzten Kontakt gehört.

Roboter penalisiert Da der *GameController* den *Automatic Referee* implementiert, penalisiert er auch die Roboter. Wenn dies passiert, wird im *TestController* überprüft, ob der penalisierte Roboter für die Erfüllung penalisiert werden musste.

Roboter in Skill Durch die Aktivierung der *Debug Requests* der Klasse *SkillRequest* wird der aktuelle *Skill* des aktuellen Roboters als *String* in die *RobotConsole* gesendet. Dieser *String* wird zyklisch an den *GameController* weitergegeben. Der *TestController* überprüft dann, ob der aktuelle *Skill* vom aktuellen Roboter genutzt werden musste.

Beschreibung der Zielauswertung

Jedes Ziel hat zur Überprüfung eine eigene Methode in *TestController*, wessen Aufruf und Funktion hier beschrieben wird. Für die Kontrolle der maximalen Dauer eines Testdurchlaufs wurde ebenfalls eine eigene Methode implementiert, welche die Beendigung des aktuellen Testdurchlaufs bei Ablauf der Zeit einleitet. Die Auswertung der Ziele erfolgt dabei immer nach dem gleichen Muster, wie beispielsweise in Abbildung 3.2 für das Ziel *Roboter berührt Ball*. Als Übergabeparameter erhält die Funktion den Namen des Roboters, der den Ball als letztes berührt hat. Zuerst wird eine Zählvariable mit dem Wert 0 initialisiert, was die Menge der erfolgreichen Ziele darstellt. Nun wird über alle Elemente des Vektors des Typen `TestTargetData`, welcher alle Ziele beinhaltet, iteriert. Wenn das jeweilige Element das Ziel `robotTouch`, was für *Roboter berührt Ball* steht, hat, werden zwei Punkte geprüft. Zunächst, ob die Roboternamen übereinstimmen und das Ziel nicht negiert ist. Dann ist das Ziel erfolgreich und das `successful`-Flag wird gesetzt. Falls dies nicht der Fall ist, wird geprüft, ob die Roboternamen übereinstimmen und das Ziel negiert ist. Dann wird ein Abbruch des Testdurchlaufs eingeleitet, da ein Ereignis eingetreten ist, was nicht hätte eintreten dürfen und somit alle Ziele nicht mehr erreicht werden können. In diesem Fall wird die Funktion verlassen. Falls sie nicht verlassen wurde, wird die Zählvariable inkrementiert, falls das `successful`-Flag des Ziels gesetzt ist. Nachdem über alle Elemente iteriert wurde, wird geprüft, ob alle Ziele und mindestens eins erreicht wurden. Wenn dies der Fall ist, wird das Ende des Testdurchlaufs eingeleitet und dieser als erfolgreich gekennzeichnet, indem die Anzahl der erfolgreichen Testdurchläufe inkrementiert wird. Dann wird die Funktion mit Rückgabewert `true` verlassen, was dem *GameController* signalisiert, dass der Testdurchlauf beendet ist. Ansonsten wird die Funktion mit dem Rückgabewert `false` verlassen und der Testdurchlauf läuft weiter. Die Auswertung der weiteren Ziele erfolgt analog zu diesem Beispiel.

Neben der Auswertung der Ziele wird auch eine Kontrolle der maximalen Sekunden zyklisch durch den *GameController* im *TestController* aufgerufen. Wenn mindestens ein Ziel eine Negation beinhaltet, kann der Testdurchlauf erst erfolgreich sein, wenn die maximalen Sekunden abgelaufen sind. Beispielsweise kann erst kontrolliert werden, ob das rechte Team kein Tor erzielt hat, wenn die maximalen Sekunden des Tests abgelaufen sind. Dazu wird, wenn die maximalen Sekunden abgelaufen sind, nochmal über alle Elemente des Vektors iteriert und die Ziele mit Negationen überprüft. Falls im Anschluss dieser Überprüfung alle Ziele das `successful`-Flag gesetzt haben, ist der Testdurchlauf auch dann noch erfolgreich. Wenn der Testdurchlauf ausgewertet wurde, wird die Szene neu geladen und alle Ziele als nicht erreicht markiert um so den nächsten Testdurchlauf

```
1  Function checkRobotTouch(robotName):
2
3      successfulItems = 0
4
5      For each item in testData.targetData:
6
7          If item.target == robotTouch:
8
9              If robotName == item.robotName AND not negated:
10                 Set item.successful to true
11
12             Else if robotName == item.robotName AND negated:
13                 Set testEnded to true
14                 Return true
15
16         If item.successful:
17             Increment successfulItems
18
19     If successfulItems > 0 AND successfulItems == the size of
           testData.targetData:
20         Increment testData.numOfSuccess by 1
21         Set testPositive to true
22         Set testEnded to true
23         Return true
24
25     Return false
```

Abbildung 3.2: Pseudocode der Zielauswertung.

vorzubereiten. Dies passiert so lange bis die Anzahl der gespielten Testdurchläufe gleich die Anzahl der zu spielenden Testdurchläufe ist.

Bei ganzen Spielen werden der Einfachheit halber, wie schon durch Lienhoop (2022), nicht ganze Spiele gespielt, sondern einzelne Halbzeiten, wobei der Anstoß jeweils wechselt. Wenn der Nutzer also beispielsweise 5 ganze Spiele spielen lassen möchte, wird diese Zahl verdoppelt. Es werden also 10 Halbzeiten gespielt. Am Ende einer Halbzeit wird das Ergebnis gespeichert und der Simulator neu gestartet, sowie der Anstoß getauscht. Die Festlegung, welches Team Anstoß hat, erfolgt über die Bildung des Modulos 2 der Anzahl der gespielten Testdurchläufe. Ist das Ergebnis 0, dann hat das linke Team Anstoß, ansonsten hat das rechte Team Anstoß.

3.6.1 Verschiedene Szenentypen

Es sind sowohl *Fast*-Szenen im 3D- und 2D-Modus, als auch *PerceptOracle*-Szenen möglich. Diese Auswahl ist über das Laden der Szene möglich, wie in Kapitel 3.2 beschrieben. Falls die Roboter bei Situationen benutzerdefinierte Positionen einnehmen sollen, können diese Postitionen in der Vorbereitung der Tests mit dem `sv`-Kommando exportiert werden und die erzeugte Skriptdatei als Parameter beim Teststart übergeben werden. Für *PerceptOracle*-Szenen ist dies sogar zwingend notwendig, da die Roboter sonst kein Wissen über die eigene Position haben.

Fuzzing

Fuzzing wurde im Test-Framework durch eine Modifizierung der Positionen und Rotationen der Objekte implementiert. Bei Start der Situations-Tests kann ein *Fuzzing*-Parameter übergeben werden. Dieser Parameter gibt an, wie sehr die Position der Roboter auf der X- und Y-Achse maximal verändert werden soll. Die Festlegung der neuen Position zum Start eines Testdurchlaufs erfolgt dann durch einen Zufallsgenerator. Beispielsweise wird die neue X-Koordinate des Roboters durch einen Zufallsgenerator bestimmt. Der minimale Wert wird berechnet, indem der *Fuzzing*-Parameter von der aktuellen Position auf der X-Achse subtrahiert wird, während der maximale Wert durch Addition des *Fuzzing*-Parameters zur aktuellen Position auf der X-Achse ermittelt wird. Der durch Zufall bestimmte Wert liegt dann innerhalb dieser Grenzen. Ein Platzieren außerhalb des Spielfeldes wird dabei verhindert. Die Z-Achse bleibt zudem unverändert. Das *Fuzzing* erfolgt somit durch bestimmte Regeln, welche zum Teststart festgelegt werden. Da der Quellcode vorliegt, handelt es sich um *White-Box-Fuzzing*. Eine visuelle Darstellung der Positionierung ist Abbildung 3.3 zu entnehmen. Hierbei ist a der übergebene *Fuzzing*-Parameter. Die Kanten des Quadrats haben die Länge $2a$. Der Flä-

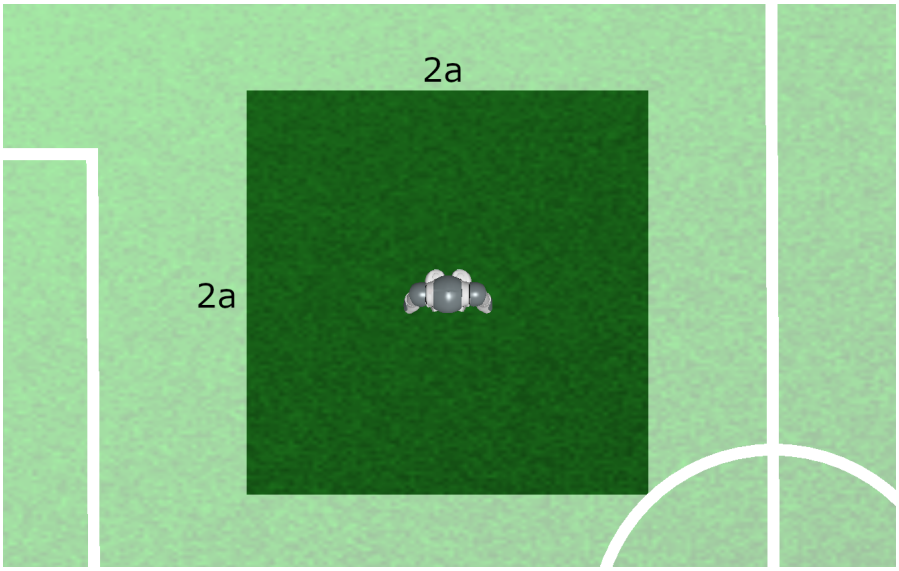


Abbildung 3.3: Visualisierung des *Fuzzings*. Der *Fuzzing*-Parameter ist a. Somit haben alle Kanten des Quadrats die Länge 2a. Der Flächeninhalt des Quadrats gibt den Bereich an, in welchem der Roboter platziert werden könnte. Der aktuelle Standort des Roboters kennzeichnet die Ausgangsposition.

cheninhalt des Quadrats gibt den Bereich an, in welchem der Roboter platziert werden könnte, wobei der aktuelle Standort des Roboters die Ausgangsposition kennzeichnet. Zudem werden die Roboter zufällig rotiert.

Implementiert wurde dies durch eine Verarbeitung des Inhalts der eingelesenen Nutzerskriptdatei, welcher während der Testinitialisierung in `TestData` gespeichert wurde. Diese wird nach `mvo`-Kommandos durchsucht und die Parameter dieser Kommandos wird durch die oben genannten Regeln verändert. Das Resultat wird wieder in der Nutzerskriptdatei gespeichert. Dies wird vor dem Start jedes Testdurchlaufs durchgeführt. Durch diese Implementierung ist *Fuzzing* nur in den *Fast*-Szenen möglich, da für *PerceptOracle*-Szenen der *StaticInitialPoseProvider* genutzt werden müsste. Dies hat den Grund, dass der *TestController* zur Laufzeit kein Wissen über die Szenenart hat und so nicht entscheiden kann, ob die Roboter mittels `mvo`-Kommandos oder durch Nutzung des *StaticInitialPoseProviders* platziert werden müssen, damit sie eine Orientierung auf dem Feld haben.

3.7 Testende

Ein Testende kann entweder durch das Beenden des letzten Testdurchlaufs erreicht werden oder direkt durch den Benutzer hervorgerufen werden. Dazu wurde das Kommando `test`

`stop` hinzugefügt. Um das Verzeichnis nicht mit temporären Dateien, die zur Laufzeit des Test-Frameworks benötigt werden, zu überfluten, wurde implementiert, dass alle Dateien, die für die Testauswertung nicht relevant sind, gelöscht werden. Dadurch wird sichergestellt, dass das Verzeichnis nach den Tests denselben Zustand wie vor Beginn der Tests hat. Dazu gehören die beiden kopierten Szenedateien, sowie die Datei *test.running*.

3.8 Ausgabe der Ergebnisse

Die Ausgabe der Ergebnisse unterscheidet sich darin, ob ganze Spiele oder Situationen getestet wurden. Für beide Fälle wurde eine Ausgabe in die Datei *test.results* implementiert. Bei Situationen wird in diese Datei lediglich geschrieben, wie viele Testdurchläufe durchgeführt wurden und wie viele davon erfolgreich waren. Beide Werte sind in der Struktur `TestData` zu finden und müssen nur ausgelesen werden. Falls *Fuzzing* aktiviert ist, werden nach jedem nicht erfolgreichem Testdurchlauf die verwendete Nutzerskriptdatei mit den modifizierten Positionen gespeichert. Da es sich dabei um `mvo`-Kommandos handelt, kann die Skriptdatei zur Fehleranalyse in Anschluss an die Tests ausgeführt werden. Bei ganzen Spielen ist die Ausgabe komplexer. Hier wurde implementiert, dass das Gesamtergebnis über alle Spiele, die Anzahl der Siege für das jeweilige Team, die Anzahl der Unentschieden und das durchschnittliche Ergebnis pro Spiel gespeichert wird. Des Weiteren werden die Ergebnisse der einzelnen Spiele, sowie der Gewinner mittels einer *CSV*-Datei dem Nutzer zur Verfügung gestellt. Dies soll eine einfache statistische Auswertung der Spiele ermöglichen. Alle notwendigen Informationen zu den Spielen stehen dabei in der `TestData`-Struktur. Das Gesamtergebnis wird aus den beiden Variablen, welche diese speichern, ausgelesen. Für die Anzahl der Siege sowie Unentschieden, werden alle erzielten Tore der Teams pro Spiel verglichen und so bestimmt, welches Team gewonnen hat. Das durchschnittliche Ergebnis wird mittels des Mittelwert der erzielten Tore bestimmt.

Zudem wird mit jedem gestarteten Testdurchlauf in der Konsole ausgegeben, wie viele Testdurchläufe schon absolviert wurden und wie der aktuelle Stand der Ergebnisse ist. Für diese Funktion wurde in den *TestController* und dem *GameController* eine Methode `getMessages()` implementiert, welche jeweils einen *String* zurückgibt. Der vom *TestController* zurückgegebene *String* beinhaltet dabei alle Informationen, welche in der Konsole ausgegeben werden sollen. Dieser *String* wird zum Start eines Testdurchlaufs beispielsweise mit dem aktuellen Stand der Ergebnisse befüllt. Die `getMessages()`-Methode des *GameControllers* gibt den *String* direkt als Rückgabeparameter weiter. In *ConsoleRoboCupCtrl* wird diese Methode zyklisch aufgerufen und überprüft, ob der Rückgabewert nicht leer ist und dann in die Konsole ausgegeben.

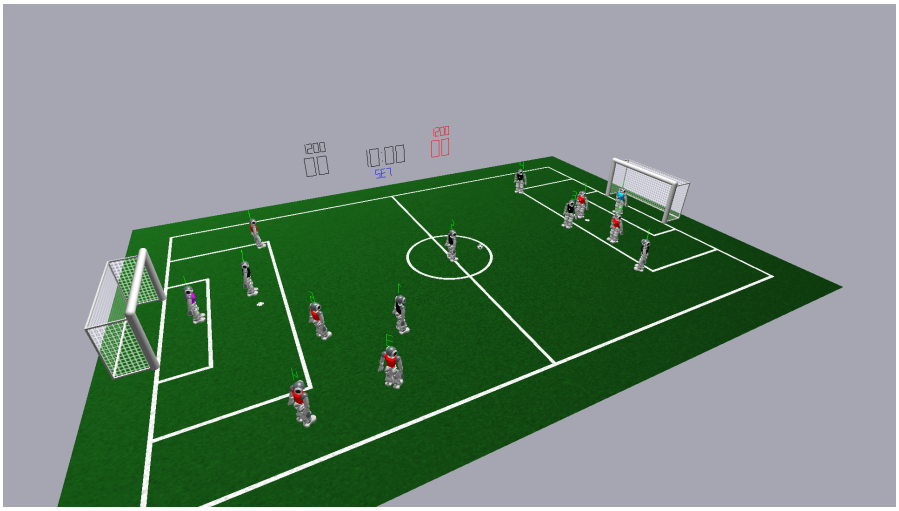


Abbildung 3.4: Beispielhafte Spielsituation: Das schwarze Team ist im Ballbesitz und in einer offensiven vier gegen zwei Situation.

3.9 Beispielkonfiguration

Man nehme an, dass man den Ausgang der Spielsituation aus Abbildung 3.4 testen möchte. Das schwarze Team ist im Ballbesitz und in einer offensiven vier gegen zwei Situation. Man erwartet nun, dass das schwarze Team innerhalb von 30 Sekunden ein Tor erzielt und das rote Team nicht in Ballbesitz kommt. Dies soll durch 10 Testdurchläufe getestet werden. Zudem sind die Postionen der Roboter in der Skriptdatei *positions.con* gespeichert. Das daraus resultierende Kommando ist dann folgendes.

```
1 test situation 10 30 positions.con goalLeftTeam neg
    teamPossession right
```

Listing 3.1: Test-Kommando der Beispielkonfiguration.

`test situation` gibt an, dass eine Situation getestet wird. Die Zahl `10` steht für 10 Testdurchläufe und die Angabe `30` für maximal 30 Sekunden pro Testdurchlauf. `positions.con` ist die Skriptdatei, welche die Positionen des Balls und der Roboter beinhaltet. Diese wurde zuvor mit dem `sv`-Kommando erzeugt. `goalLeftTeam` sagt aus, dass das linke Team ein Tor erzielen muss, `neg teamPossession right` gibt an, dass das rechte Team nicht den Ball berühren darf.

3.10 Integration eines Smoke-Tests in GitLab-Pipeline

3.10.1 Vorbereitungen für die Integration

Da der *Docker-Container*, auf welchem die Tests laufen sollen, keine physikalische Bildschirmausgabe besitzt, muss es möglich sein, *SimRobot* ohne Fenster zu starten. Hierzu wurde der *SimRobot*-Startparameter `-noWindow` ergänzt, welcher dafür sorgt, dass das Programm läuft, aber kein Fenster sichtbar ist.

Der *Smoke-Test* soll einen Situations-Test in einer *Fast*-Szene beinhalten, in welcher sieben Roboter auf der linken Seite gegen fünf Dummies auf der rechten Seite spielen. Die Dummies sind dabei Roboter, welche sich nicht bewegen, aber als solche für spielende Roboter erkennbar sind. Hierzu wurde eine bestehende Szene inklusive Skriptdatei händisch unter neuem Namen kopiert und in einem neuen Ordner *Tests* im *B-Human*-Verzeichnis gespeichert. Am Ende der Skriptdatei der Szene wurde das Test-Kommando gespeichert. Somit wird dieses beim Laden der Szene direkt aufgerufen. Dabei soll ein Testdurchlauf von maximal 120 Sekunden stattfinden. In dieser Zeit muss das linke Team ein Tor erzielen und im Anschluss wird das Programm beendet. Außerdem werden dem Test über eine Nutzerskriptdatei die Positionen der spielenden Roboter und des Balls mitgeteilt. Die sieben spielenden Roboter stehen in der Aufstellung, welche sie vor der Ausführung eines Anstoßes einnehmen würden und der Ball liegt auf dem Mittelpunkt (siehe Abbildung 3.5). Die Dummies werden über die ursprüngliche Szenendatei konfiguriert. Somit ergibt sich folgendes Test-Kommando.

```
1 test situation 1 120 Tests/TestOneTeamFastFormation.con q
      goalLeftTeam
```

Listing 3.2: Test-Kommando des *Smoke-Tests*.

Falls dieser Test nicht bestanden wird, kann davon ausgegangen werden, dass ein grundlegendes Problem vorliegt.

3.10.2 Anpassungen in der Skriptdatei

Die Konfiguration des neu erstellten Jobs ist in Abbildung 3.6 zu sehen. Für die Integration der Tests in die *GitLab-Pipeline* wurde ein neuer *Job* namens *smoke-test* zur *.gitlab-ci.yml* hinzugefügt. Für die Auswahl der *Runner* wurden die Tags *docker* und *ssse3* ausgewählt. So wird sichergestellt, dass der *Job* in einem *Docker-Container* auf dem *B-Human* eigenen Rechner *B-East* läuft, welcher die Befehlssatzerweiterung *SSSE3* unterstützt. Diese wird für die Kompilierung des Programmcodes benötigt. Die *stage* wurde auf *test* gesetzt, da es sich um einen Test handelt. In *variables* wurde die Variable

Abbildung 3.5: Formation der Roboter zu Beginn des *Smoke-Tests*. Es spielen sieben schwarze Roboter gegen vier rote Dummies.

`GIT_SUBMODULE_STRATEGY` auf *recursive* gesetzt, da die *Git-Submodule*, welche für die Kompilierung benötigt werden, rekursiv geholt werden sollen. Als *Docker-Software-Abbild* wurde das *kunitoki/clang-17* ausgewählt, welches eine vorkonfigurierte Umgebung mit *Clang 17*, einem *C++-Compiler*, enthält. In der *before_script*-Phase werden die Paketlisten aktualisiert und die benötigten Bibliotheken und Tools installiert. Des Weiteren wird durch das Setzen der Umgebungsvariable `QT_QPA_PLATFORM` sichergestellt, dass die Anwendung im *Offscreen-Modus*, also ohne Fenster, lauffähig ist. Außerdem wird das Arbeitsverzeichnis vorbereitet und der Programmcode kompiliert.

Die darauffolgende *script*-Phase beinhaltet dann die Durchführung des einfachen *Smoke-Tests*. Dies geschieht durch die Verwendung von `xvfb-run` ohne physikalische Ausgabe. Als Übergabeparameter erhält *SimRobot* den `-noWindow`-Parameter sowie den Pfad zu der erstellten Szene (siehe Kapitel 3.10.1). Durch die Benutzung des `quit`-Parameters beim Teststart wird das Programm nach Beendigung des Tests geschlossen und die *Pipeline* erhält das Ergebnis als Rückgabeparameter. Der *allow_failure*-Parameter ist auf `false` gesetzt. Sobald es bei der Kompilierung keine Fehler gab, hängt der Erfolg der *Pipeline* somit maßgeblich von dem Rückgabewert des Tests ab.

```yaml
smoke-test:
  tags:
    - docker
    - ssse3
  stage: test
  variables:
    GIT_SUBMODULE_STRATEGY:
      recursive
  image: kunitoki/clang-17
  before_script:
    - apt-get update
    - apt-get -y install libasound2-dev libbox2d-dev libgl-dev
      libqt6opengl6-dev libqt6svg6-dev qt6-base-dev xvfb
    - export QT_QPA_PLATFORM=offscreen
    - ln -s ld /usr/bin/ld.mold && cd Make/Linux/ && cmake -
      DCMAKE_BUILD_TYPE=Release -G Ninja -S . -B ../../Build/
      Linux/CMake/Release && ./compile Release SimRobot && cd
      ../../
  script:
    - echo "Running smoke tests"
    - xvfb-run -a ./Build/Linux/SimRobot/Release/SimRobot -
      noWindow Config/Scenes/Tests/TestOneTeamFast.ros2
    - cat Config/test.results
  allow_failure: false
```

Abbildung 3.6: Die erstellte Konfiguration des Jobs *smoke-test*.

4 Evaluation

In den nachfolgenden Unterkapiteln wird das entwickelte Test-Framework evaluiert. Dazu werden sowohl ganze Spiele als auch bestimmte Situationen lokal händisch gestartet und durch das Test-Framework automatisiert ausgeführt. Die Ergebnisse, welche durch das Test-Framework in Dateien geschrieben wurden, werden im Anschluss mit dem Erwartungswert der Testergebnisse verglichen.

4.1 Simulierte Spiele

Für die Evaluation von ganzen Spielen wurden zwei verschiedene Szenarien erstellt und über 50 Spiele, also 100 Halbzeiten, durch das Test-Framework simuliert. Die Ergebnisse werden im Nachhinein miteinander verglichen.

4.1.1 Ausgeglichene Spiele

Es spielen zwei Teams 50 Spiele gegeneinander. Beide Teams nutzen die aktuelle *B-Human*-Software. Es ist also zu erwarten, dass ein Großteil der Spiele ausgeglichen verläuft und am Ende beide Teams in etwa die gleiche Anzahl an Siege zu verbuchen haben. Die Anzahl der gewonnenen Spiele ist Abbildung 4.1 zu entnehmen. Die Anzahl der erzielten Tore pro Team ist in Abbildung 4.2 visuell dargestellt.

4.1.2 Einseitige Spiele

Zwei Teams spielen 50 Spiele gegeneinander. Das rechte Team nutzt die aktuelle *B-Human*-Software. Das linke Team benutzt ebenfalls die aktuelle *B-Human*-Software, allerdings mit der Einschränkung, dass die Laufgeschwindigkeit stark reduziert ist. Dazu werden mittels Konfigurationsparametern die entsprechenden Variablen im Konstruktor der Klasse *WalkingEngine*, welche Laufbewegungen erzeugt, gesetzt. Standardmäßig bewegen sich die Roboter mit maximal 300 mm/s vorwärts und 230 mm/s seitwärts. Das linke Team hingegen bewegt sich nun mit maximal 200 mm/s vorwärts und 150 mm/s seitwärts. Dies stellt einen erheblichen Vorteil für das rechte Team dar und somit ist zu erwarten, dass das rechte Team mehr Tore schießt und mehr Spiele gewinnt. Das

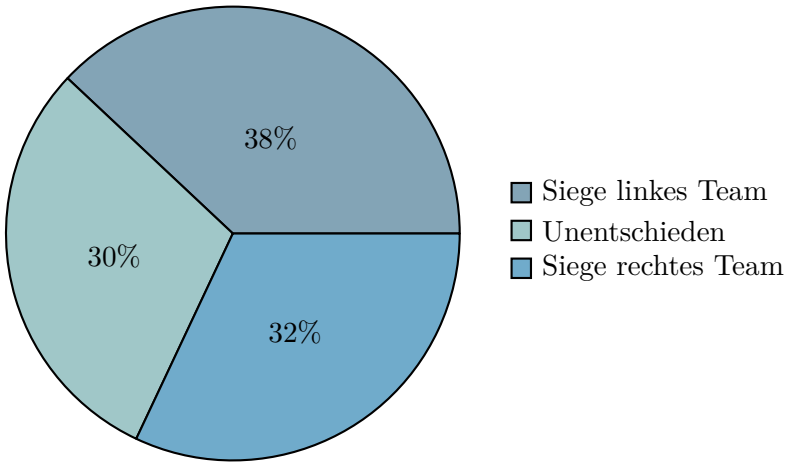


Abbildung 4.1: Ein Tortendiagramm, welches die Spielergebnisse der ausgeglichenen Spiele darstellt. Das linke Team gewann 38 Halbzeiten. Das rechte Team 30 Halbzeiten. 32 Halbzeiten gingen unentschieden aus.

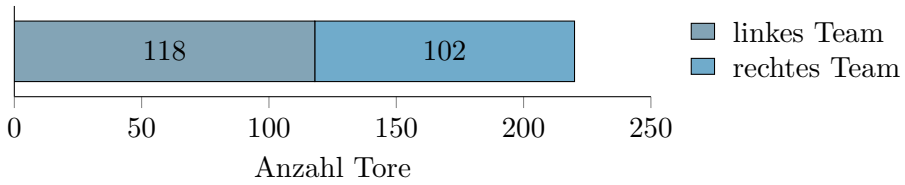


Abbildung 4.2: Ein gestapeltes Balkendiagramm, welches die Anzahl der erzielten Tore pro Team darstellt. Das linke Team erzielte 118 Tore. Das rechte Team 102 Tore.

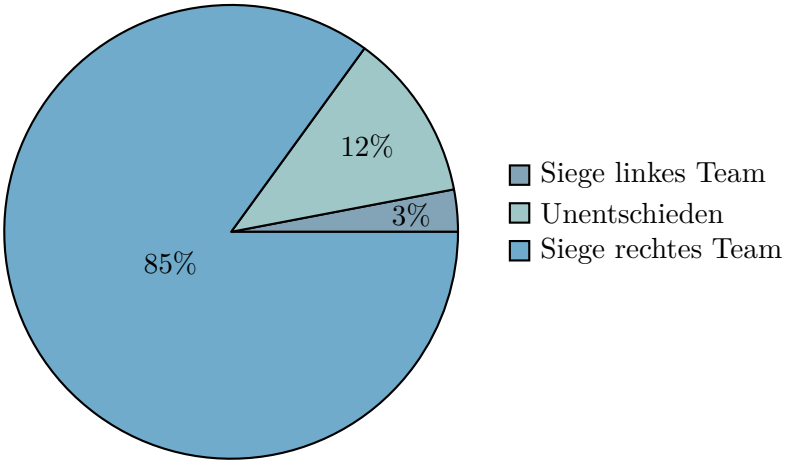


Abbildung 4.3: Ein Tortendiagramm, welches die Spielergebnisse der einseitigen Spiele darstellt. Das linke Team gewann drei Halbzeiten. Das rechte Team 85 Halbzeiten. Zwölf Halbzeiten gingen unentschieden aus.

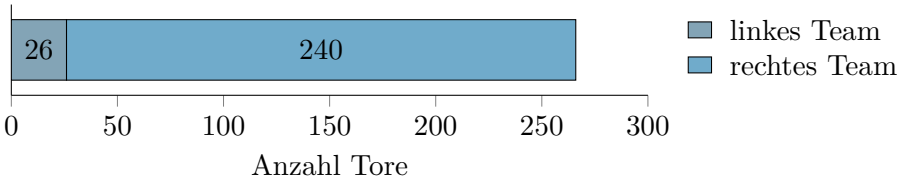


Abbildung 4.4: Ein gestapeltes Balkendiagramm, welches die Anzahl der erzielten Tore pro Team darstellt. Das linke Team erzielte 26 Tore. Das rechte Team 240 Tore.

Ergebnis der Spiele ist Abbildung 4.3 zu entnehmen. Die Anzahl der erzielten Tore pro Team ist in Abbildung 4.4 visuell dargestellt.

4.1.3 Analyse

Aus den beiden unterschiedlichen Szenarien geht eindeutig hervor, dass das Test-Framework die Spiele korrekt durchgeführt und ausgewertet hat. In Kapitel 4.1.1 ist zu erkennen, dass die Verteilung der Siege ausgeglichen ist. Das linke Team konnte 38 Spiele gewinnen, das rechte Team 32. Der Unterschied von 6 Spielen macht über 100 Spiele lediglich einen marginalen Unterschied aus. Zudem wurde 30 Mal unentschieden gespielt, was die Ausgeglichenheit der Partien nochmals betont. Auch die erzielten Tore geben dies wieder. Insgesamt wurden 220 Tore erzielt, wobei diese sich mit 118 für das linke und 102 für das rechte Team aufteilen. Auch hier ist das linke Team leicht im

Vorteil, wobei auch dieser Unterschied gering ist. Möglicherweise wäre hier eine größere Anzahl an Spielen besser gewesen, damit eine ausgeglichenere Bilanz resultiert wäre (siehe 1.4.1).

In Kapitel 4.1.2 kommt man zu einem anderen Ergebnis. Das rechte Team, welches die normale Laufgeschwindigkeit benutzt, konnte 85 von 100 Spielen gewinnen. Das linke Team, welches die langsame Laufgeschwindigkeit genutzt hat, konnte nur 3 Spiele für sich entscheiden und auch nur 12 Mal ein Unentschieden erzielen. Die erzielten Tore geben dies auch wieder. Das rechte Team hat mit 240 Toren fast das Zehnfache des linken Teams, welches lediglich 26 Tore erzielt hat, geschossen. Es lässt sich hier also eindeutig erkennen, dass das rechte Team über alle 100 Spiele einen Vorteil hatte.

4.2 Simulierte Situationen

Für die Evaluation von Situationen wurden verschiedene Szenarien mit unterschiedlicher Trivialität erstellt und mehrfach automatisiert ausgeführt.

4.2.1 Triviale Situation

Es spielen jeweils sieben Roboter gegeneinander. Der Ball liegt im Torraum des linken Teams, wie in Abbildung 4.5 zu sehen. Ziel ist es, dass der Torwart des linken Teams den Ball berührt und der Ball den Strafraum nach vorne verlässt. Es finden 100 Testdurchläufe mit einer maximalen Dauer von 60 Sekunden statt. Die Erwartung ist, dass die Situation mit einer hohen Wahrscheinlichkeit erfolgreich abgeschlossen wird. Somit ergibt sich folgendes Test-Kommando.

```
1 test situation 100 60 positions.con robotTouch robot1 ballInZone
    2850 -3000 -2850 3000
```

Listing 4.1: Test-Kommando des trivialen Tests.

In allen 100 Testdurchläufen konnte der Ball durch den Torwart aus dem Strafraum befördert werden. Vom Test-Framework wurde dies in allen Fällen richtig erkannt.

4.2.2 Nicht triviale Situation

Es spielen jeweils sieben Roboter gegeneinander. Der Ball liegt in der Hälfte des rechten Teams. Ziel ist es, dass die schwarzen Roboter mit der Nummer 4 und 7 den Ball berühren. Außerdem soll die schwarze Nummer 4 den *Skill pass* betreten und die schwarze Nummer 7 den *Skill shoot*. Zudem darf das linke Team kein Tor erzielen. Beispielsweise soll ein Roboter des rechten Teams den Ball blocken, was jedoch nicht explizit als Ziel angegeben

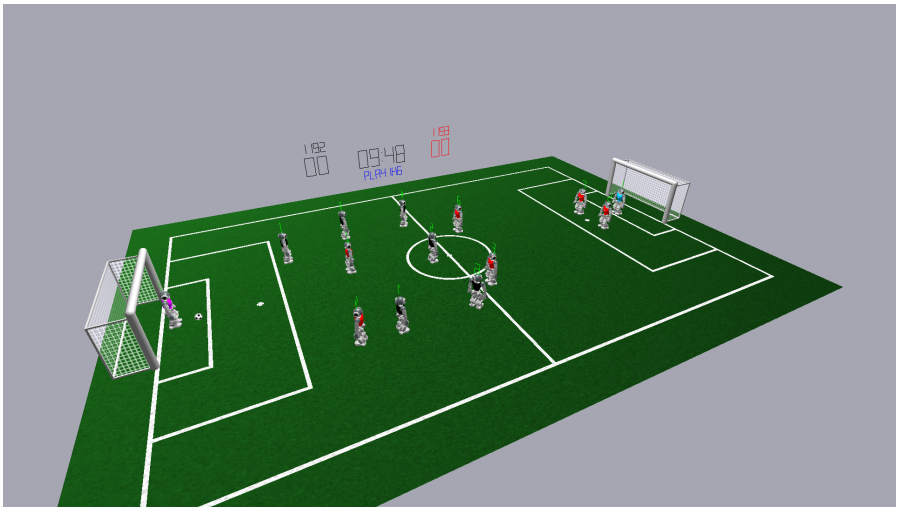


Abbildung 4.5: Anordnung der Roboter für den trivialen Test. Der Ball liegt im Torraum des linken Teams.

ist. Es finden 100 Testdurchläufe mit einer maximalen Dauer von 15 Sekunden statt. Die Erwartung ist, dass die Situation mit einer hohen Wahrscheinlichkeit erfolgreich abgeschlossen wird. Die Formation der Roboter ist in Abbildung 4.6 dargestellt. Das Test-Kommando ist im Folgenden angegeben.

```
1 test situation 100 15 positions.con robotTouch robot4 robotTouch
    robot7 neg goalLeftTeam robotInSkill pass robot4 robotInSkill
    shoot robot7
```

Listing 4.2: Test-Kommando des nicht trivialen Tests.

In 90 von 100 Testdurchläufen war der Test erfolgreich. In den zehn Fällen, in denen der Test nicht erfolgreich war, konnte das schwarze Team ein Tor erzielen. Dies war möglich, da der Ball zügig von der schwarzen Nummer 4 zur schwarzen Nummer 7 gespielt wurde und so genug Zeit für einen gezielten Torabschluss vorhanden war. Die restlichen Situationen wurden korrekt vom Framework als erfolgreich erkannt, da hier kein Tor erzielt wurde.

4.2.3 Mehrere Situationen mit steigender Schwierigkeit

Es werden insgesamt sieben Situationen gespielt. In der ersten Situation spielen sieben Roboter des linken Teams gegen sieben Roboter des rechten Teams, wobei alle Roboter des rechten Teams penalisiert sind. In der zweiten Situation sind nur noch sechs Roboter des rechten Teams penalisiert. In der dritten sind nur noch fünf Roboter des rechten Teams penalisiert. Die Situationen drei, vier, fünf, sechs und sieben erfolgen dazu

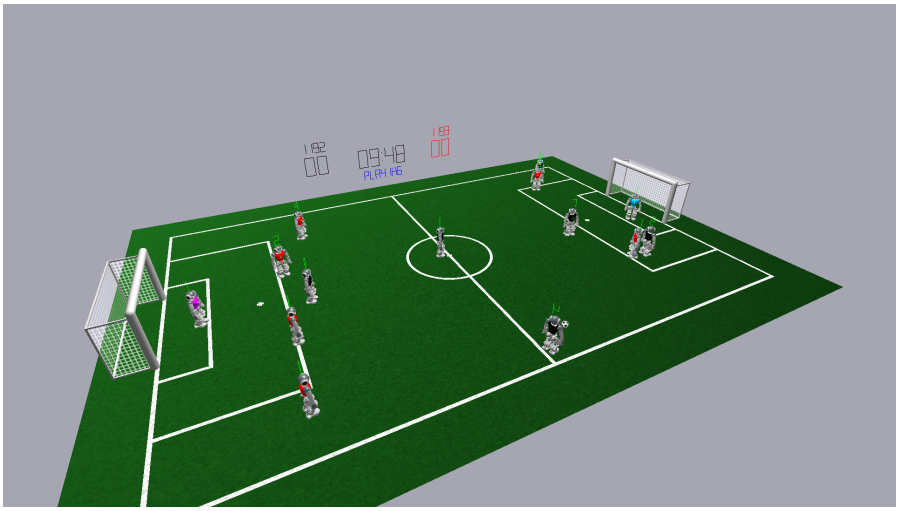
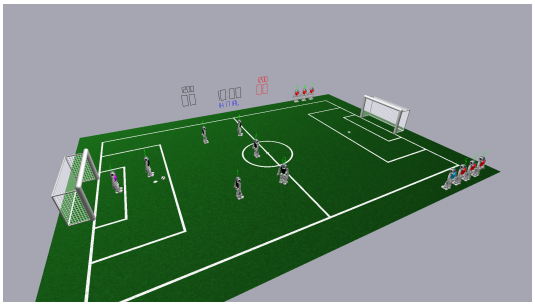


Abbildung 4.6: Anordnung der Roboter für den nicht trivialen Test. Der Ball liegt vor dem schwarzen Roboter mit der Nummer 4 nahe der rechten Torauslinie.

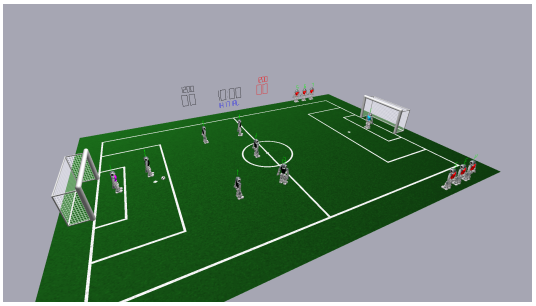
analog. Der Ball liegt jeweils in der Hälfte des linken Teams. Es werden je Situation 100 Testdurchläufe mit einer maximalen Dauer von 60 Sekunden gespielt. Das Ziel ist es, dass das linke Team ein Tor erzielt. Es wird erwartet, dass die Anzahl der erzielten Tore mit dem Hinzufügen eines Roboters sinkt oder gleichbleibend ist. Die einzelnen Szenen der Tests sind in Abbildung 4.7 zu sehen. In der ersten Situation, in welcher kein Gegenspieler auf dem Feld war, konnte das linke Team in nahezu allen Testdurchläufen ein Tor erzielen. Wie in Abbildung 4.8 zu sehen, konnte in lediglich fünf Testdurchläufen das Ziel nicht erreicht werden. Mit dem Hinzufügen des Torwarts konnten nur noch 54 Tore erzielt werden. Das Hinzufügen eines Verteidigers senkte diese Zahl auf 20. Das Hinzufügen der gegnerischen Roboter 3-7 sorgte dafür, dass in keinem Test mehr als zehn Tore erzielt werden konnten. Dies unterstreicht die korrekte Ausführung und Auswertung des Tests.

4.2.4 Situation in einer 2D-Szene

Es spielen jeweils sieben Roboter gegeneinander. Der Ball liegt im Mittelkreis. Die Teams sind in ihrer üblichen Formation aufgestellt, wenn ein Spiel beginnt (siehe Abbildung 4.9). Das Ziel ist es, dass beide Teams in Ballbesitz kommen und kein Tor fällt. Es finden 100 Testdurchläufe mit einer maximalen Dauer von zehn Sekunden statt. Die Erwartung ist, dass die Situation mit einer hohen Wahrscheinlichkeit erfolgreich abgeschlossen wird. Als Test-Kommando ergibt sich somit das folgende Kommando.



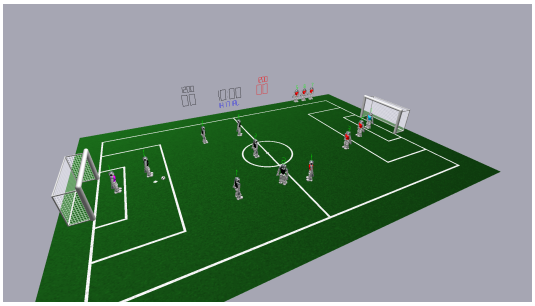
Situation mit keinem Gegenspieler



Situation mit einem Gegenspieler

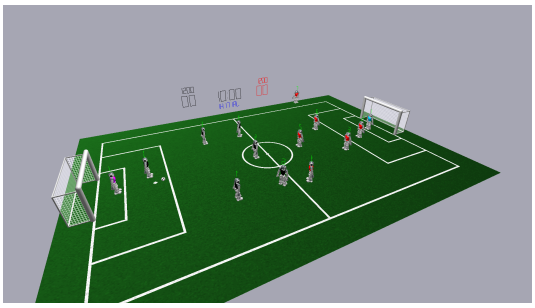
Situation mit 2 Gegenspielern

Situation mit 3 Gegenspielern

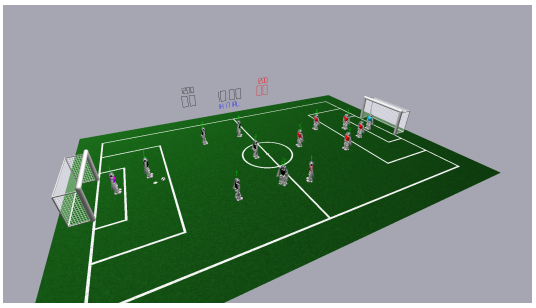


Situation mit 4 Gegenspielern

Situation mit 5 Gegenspielern



Situation mit 6 Gegenspielern



Situation mit 7 Gegenspielern

Abbildung 4.7: Alle gespielten Situationen mit steigender Schwierigkeit. Der Ball ist jeweils vor dem schwarzen Roboter im linken Strafraum. Das Ziel ist die Erzielung eines Tores durch das linke Team.

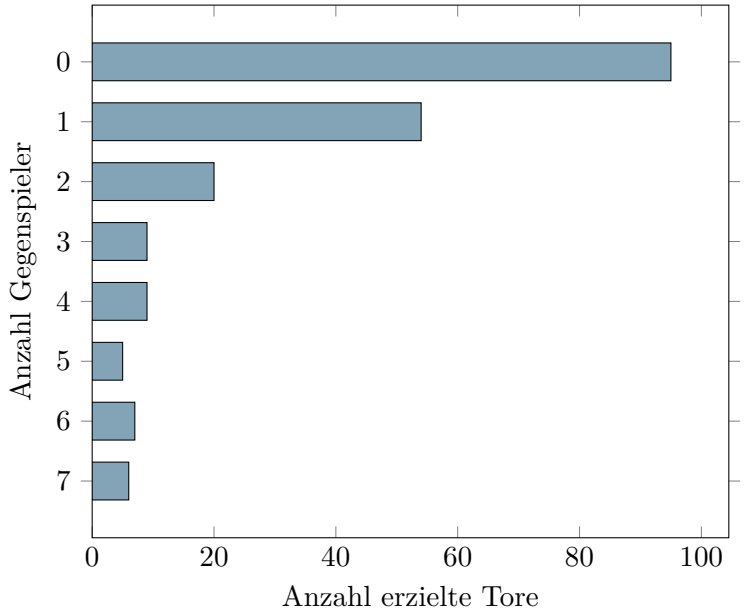


Abbildung 4.8: Ein Balkendiagramm, welches die Anzahl der erzielten Tore pro Test darstellt.

```
1 test situation 100 10 positions.con teamPossession left
    teamPossession right neg goalLeftTeam neg goalRightTeam
```

Listing 4.3: Test-Kommando des 2D-Tests.

In allen 100 Testdurchläufen wurden die Ziele erreicht. Der Roboter, welcher den Ball als erstes berührt, passt den Ball zunächst nach vorne zu einem Gegenspieler. Beide Spieler sind dann in einem Kampf um den Ball, was dazu führt, dass die angegebenen maximalen zehn Sekunden nicht für ein Tor ausreichen. Dies zeigt, dass das Test-Framework auch im 2D-Modus funktioniert.

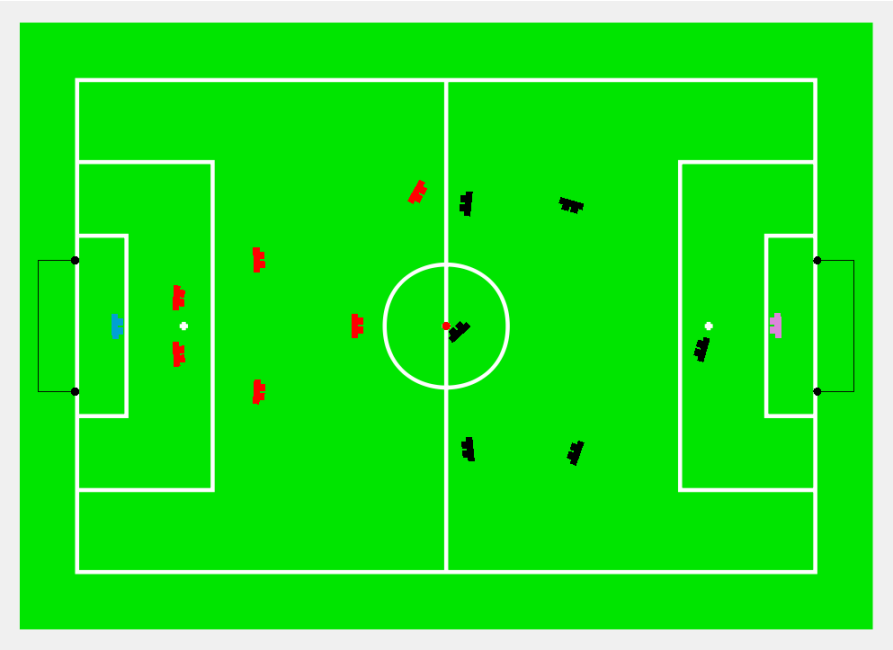


Abbildung 4.9: Anordnung der Roboter für den 2D-Test. Der Ball befindet sich auf dem Mittelpunkt des Spielfelds.

5 Abschluss

5.1 Fazit

In der vorliegenden Arbeit wurde ein Test-Framework entwickelt, welches darauf abzielt, das automatisierte Testen für das Team *B-Human* möglich zu machen. Die in Kapitel 1.3 definierten Ziele wurden dabei vollständig erreicht und aus Kapitel 4 geht hervor, dass das Test-Framework die Tests korrekt durchführt und auswertet.

Das Framework ermöglicht das automatisierte Testen von vollständigen Spielen zwischen Teams mit unterschiedlichen Verhaltensweisen. Die Resultate dieser Tests können anschließend statistisch mittels einer erzeugten *CSV*-Datei ausgewertet werden, um beispielsweise festzustellen, ob die Anpassungen im Verhalten eine Verbesserung darstellen.

Darüber hinaus erlaubt das Framework die Durchführung automatisierter Tests von Spielsituationen mit individuell festlegbaren Zielen, wie zum Beispiel erzielten Toren, dem Eintritt von Objekten in bestimmte Zonen oder dem Erzielen von Ballbesitz. Die Konjunktion und Negation dieser Ziele wurde dabei ebenfalls implementiert. Die Startpositionen der Roboter für jeden Testdurchlauf können zudem durch eine *Fuzzing*-Technik variiert und bei Misserfolg gespeichert werden. Beide Testarten sind in Echtzeit, beschleunigter Zeit, in den Szenarientypen 3D und 2D, sowie in verschiedenen Szenenarten durchführbar, was die Vielseitigkeit des Frameworks unterstreicht. Neben der lokalen Ausführung durch den Nutzer können die Tests auch über eine *GitLab-Pipeline* durchgeführt werden, etwa für *Smoke-Tests*.

Während der Entwicklung des Frameworks wurde auf Erkenntnisse im Bereich des *RoboCups* und allgemein der Softwareentwicklung in der Robotik zurückgegriffen. Zusammenfassend lässt sich sagen, dass das entwickelte Framework das Testen im Team *B-Human* signifikant vereinfacht und eine erhebliche Zeitersparnis bietet.

5.2 Ausblick

Das entwickelte Framework bietet viel Potenzial für eine Weiterentwicklung und eine umfangreichere Nutzung in der Zukunft. Eine vielversprechende Erweiterung wäre die Implementierung der Möglichkeit einen alten Codestand gegen den aktuellen Codestand

spielen zu lassen. So könnte man beispielsweise den aktuellen Codestand wöchentlich gegen den Codestand vom letzten *RoboCup* spielen lassen, um so nachzuvollziehen, ob im Laufe des Jahres Verbesserungen in den Code eingeflossen sind. Diese Funktionalität würde allerdings einige tiefgreifende Änderungen am Simulator erfordern.

Aktuell befindet sich der Simulator zum Start einer zu testenden Situation immer im Zustand *playing*. Hier wäre für die Zukunft auch interessant, wenn nicht nur das laufende Spiel, sondern auch Standardsituationen wie Elfmeter, Freistöße oder Eckbälle möglich wären. Eine solche Erweiterung würde die Vielseitigkeit des Frameworks erheblich steigern und zusätzliche Testmöglichkeiten bieten.

Die Eingabe der Zonen für die Ziele *Roboter in Zone* und *Ball in Zone* könnten zudem noch intuitiver gestaltet werden. Möglicherweise wären vorgefertigte Zonen wie *linker Torraum*, *rechte Spielfeldhälfte* oder *Mittelkreis* eine Möglichkeit. Allerdings sollte auch das Erstellen von individuellen Zonen überarbeitet werden, da die erstellte Zone beispielsweise im Simulator nicht visuell dargestellt wird und man diese somit nicht überprüfen kann.

Außerdem könnte man das Framework auch noch umfassender in der *GitLab-Pipeline* nutzen. Aktuell wird nur ein einzelner *Smoke-Test* durchlaufen. So könnte man die Anzahl der *Smoke-Tests* erhöhen und zusätzlich *Regressionstests* erstellen. Dies wäre aber auch mit einem erheblichen Aufwand verbunden, da sehr viele komplexe Testfälle erstellt und permanent erweitert werden müssten. Dennoch könnte eine solche Maßnahme die Testqualität erheblich verbessern und die Fehleranfälligkeit reduzieren.

Für Situationen wäre es sinnvoll, wenn man den Zielen eine gewisse Reihenfolge geben könnte. Eine solche Reihenfolge könnte zum Beispiel sein, dass zwei Roboter nacheinander den Ball zueinander passen sollen, um einen Doppelpass in die Testabfolge zu integrieren. Mit der aktuellen Implementierung ist dies nicht möglich, es würde allerdings noch Spielraum für bessere und umfangreichere Tests bieten.

In der *RoboCup* SPL gibt es eine Regel, welche besagt, dass Roboter beispielsweise nach dem Anstoß nicht direkt ein Tor erzielen dürfen. Zuvor muss ein anderer Roboter des eigenen Teams den Ball berühren. Falls dies doch passiert, erhält das gegnerische Team einen Abstoß und das Tor zählt nicht (vgl. RoboCup Technical Committee, 2022). Da in Situations-Tests direkt in den *playing*-Zustand gewechselt wird, beachten die Roboter diese Regel nicht und erzielen mitunter ein direktes Tor, welches dann vom *Automatic Referee* korrekterweise nicht gegeben wird. Dieses Problem sollte zukünftig beseitigt werden.

Während der Ausführung des *Smoke-Tests* kommt es teilweise zu ungefangenen Ausnahmen in Komponenten außerhalb des Test-Frameworks, was zum Fehlschlagen der *Pipeline* führt. Diese Ausnahmen müssen weiter eingegrenzt werden, um die Aussagekraft

einer fehlgeschlagenen *Pipeline* zu erhöhen.

Schließlich ist es aktuell noch nicht möglich *Fuzzing* in *PerceptOracle*-Szenen zu nutzen. Hier muss ein Mechanismus implementiert werden, womit dem *TestController* bewusst ist, auf welche Weise er die Positionen anpassen und speichern muss. Dies sollte so erfolgen, dass der Nutzer dies nicht angeben muss, da ein weiterer Parameter für das *Fuzzing* den Teststart komplizierter und unübersichtlicher machen würde. Bei dem Start eines Testdurchlaufs müssen die Positionen dann mittels des *StaticInitialPoseProviders* gesetzt werden. Für den Ball wäre ein `mvo`-Kommando ausreichend.

Insgesamt bieten diese potenziellen Weiterentwicklungen die Möglichkeit, das Framework weiter zu optimieren, was letztlich zu einer noch höheren Effizienz, Funktionalität und Qualität der gesamten Software führen könnte.

Literaturverzeichnis

Literatur

Afzal, Afsoon (2021). „Automated Testing of Robotic and Cyberphysical Systems“. Diss. Carnegie Mellon University, S. 2. DOI: 10.1184/R1/16645639.v1 (siehe S. 12).

Belli, Fevzi (1998). „Methoden und Hilfsmittel für die systematische Prüfung komplexer Software“. deutsch. In: *Informatik-Spektrum* 21, S. 337–346. DOI: https://doi.org/10.1007/s002870050114 (siehe S. 11).

Cowell, Christopher, Nicholas Lotz und Chris Timberlake (2023). *Automating DevOps with GitLab CI/CD Pipelines: Build efficient CI/CD pipelines to verify, secure, and deploy your code using real-life examples*. Packt Publishing Ltd (siehe S. 19, 20).

Gabel, Thomas und Eicke Godehardt (2024). „The Glee Didn't Last Long: How Continuous Integration Helps Revealing Fundamental Flaws in Team-Play“. In: Frankfurt University of Applied Sciences. RoboCup Symposium 2024. Eindhoven, Netherlands (siehe S. 8, 9).

Godehardt, Eicke, Mohamed Amine Allani, Alexander Julian Vieth und Thomas Gabel (2023). „1001 Games a Night—Continuous Evaluation of an Intelligent Multi-agent-Based System“. In: *Proceedings of Eighth International Congress on Information and Communication Technology*. Hrsg. von Xin-She Yang, R. Simon Sherratt, Nilanjan Dey und Amit Joshi. Singapore: Springer Nature Singapore, S. 715–722. DOI: 10.1007/978-981-99-3091-3_59 (siehe S. 8, 9).

Hasselbring, Arne (2020). „Optimierung des Verhaltens von Fußballrobotern mittels simulierter Spiele und Gegnerverhaltensmodellen“. Masterarbeit. Universität Bremen (siehe S. 7, 13, 16).

Klute, Thomas (2024). „Introduction of automated testing and continuous metrics and KPI monitoring in student-driven projects: a use case“. In: TU Dortmund. RoboCup Symposium 2024. Eindhoven, Netherlands (siehe S. 8).

Laue, Tim und Thomas Röfer (2008). „SimRobot - Development and Applications“. In: *The Universe of RoboCup Simulators - Implementations, Challenges and Strategies for Collaboration. Workshop Proceedings of the International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2008)*. Hrsg. von Heni Ben Amor, Joschka Boedecker und Oliver Obst. Venice, Italy, S. 1 (siehe S. 12).

Laue, Tim, Kai Spiess und Thomas Röfer (2006). „SimRobot–a general physical robot simulator and its application in robocup“. In: *RoboCup 2005: Robot Soccer World Cup IX 9*. Springer, S. 173–183. DOI: 10.1007/11780 (siehe S. 16).

Lienhoop, Jo (2022). „Dynamisches Passspiel im Roboterfußball“. Bachelorarbeit. Universität Bremen, S. 30–31 (siehe S. 7, 9, 29).

Louridas, Panagiotis (2006). „Static code analysis“. In: *Ieee Software* 23.4, S. 58–61. DOI: 10.1109/MS.2006.114 (siehe S. 19).

McCall, Jim A (1977). „Factors in software quality“. In: *US Rome Air development center reports* (siehe S. 12).

McConnell, Steve (1996). „Daily build and smoke test“. In: *IEEE software* 13.4, S. 144. DOI: 10.1109/MS.1996.10017 (siehe S. 18).

Park, Hong Seong und Jeong Seok Kang (2012). „SITAF: simulation-based interface testing automation framework for robot software component“. In: *Automation*. IntechOpen, S. 456–462. DOI: 10.5772/36995 (siehe S. 9).

Robert, Clément, Jérémie Guiochet und Hélène Waeselynck (2020). „Testing a non-deterministic robot in simulation - How many repeated runs ?“ In: *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, S. 263–270. DOI: 10.1109/IRC.2020.00048 (siehe S. 9, 10).

RoboCup Technical Committee (2022). *RoboCup Standard Platform League (NAO) Rule Book*. Abgerufen am 16.07.2024 (siehe S. 13, 46).

Schäuffele, Jörg und Thomas Zurawka (2010). *Automotive software engineering*. Springer, S. 334. DOI: 10.1007/978-3-658-43543-1 (siehe S. 12).

Schenker, Gabriel N (2023). *The Ultimate Docker Container Book: Build, Test, Ship, and Run Containers with Docker and Kubernetes*. Packt Publishing Ltd. Kap. 4. DOI: 10.0000/9781804613184-001 (siehe S. 20).

Sell, Finn-Thorben (2021). „Integration of Software Development Testing Practices in Robotics“. Bachelorarbeit. Universität Hamburg, S. 17–24 (siehe S. 8).

Weißhardt, Florian (2024). „Automatisiertes Test- und Evaluierungsframework für die Servicerobotik“. In: (Siehe S. 11).

Witte, Frank (2016). „Testautomatisierung“. In: *Testmanagement und Softwaretest: Theoretische Grundlagen und praktische Umsetzung*. Wiesbaden: Springer Fachmedien Wiesbaden, S. 223–233. DOI: 10.1007/978-3-658-09964-0_26 (siehe S. 12).

Zhao, Xiaoqi, Haipeng Qu, Jianliang Xu, Xiaohui Li, Wenjie Lv und Gai-Ge Wang (2024). „A systematic review of fuzzing“. In: *Soft Computing* 28.6, S. 5493–5522. DOI: 10.1007/s00500-023-09306-2 (siehe S. 19).

Internetquellen

B-Human (2024a). *B-Human - At a Glance*. Abgerufen am 10.09.2024. URL: https://docs.b-human.de/master/b-human-at-a-glance/#behavior-control (siehe S. 7).

B-Human (2024b). *Homepage*. Abgerufen am 16.07.2024. URL: https://www.b-human.de/ (siehe S. 1).

B-Human (2024c). *Serialization*. Abgerufen am 10.09.2024. URL: https://docs.b-human.de/master/architecture/serialization/#configuration-maps (siehe S. 25).

GitLab (o. D.). *CI/CD YAML syntax reference* (siehe S. 20).

How To Do Good Regression Testing (2017). Abgerufen am 01.08.2024. URL: https://www.ministryoftesting.com/articles/how-to-do-good-regression-testing (siehe S. 18).

RoboCup (2024a). *A Brief History of RoboCup*. Abgerufen am 28.08.2024. URL: https://robocup.org/a_brief_history_of_robocup (siehe S. 1).

RoboCup (2024b). *RoboCup Objective*. Abgerufen am 16.07.2024. URL: https://www.robocup.org/objective (siehe S. 11).

Röfer, Thomas, Tim Laue, Andreas Baude, Jan Blumenkamp, Gerrit Felsch, Jan Fiedler, Arne Hasselbring, Tim Haß, Jan Oppermann, Philip Reichenberg, Nicole Schrader und Dennis Weiß (2019). *B-Human Team Report and Code Release 2019*. Abgerufen am 02.08.2024. URL: https://www.b-human.de/downloads/publications/2019/CodeRelease2019.pdf (siehe S. 4, 12, 13).

Röfer, Thomas, Tim Laue, Jo Lienhoop, Ayleen Lührsen, Yannik Meinken und Sina Schreiber (2024). *Team Description for RoboCup 2023*. Abgerufen am 09.08.2024. URL: https://spl.robocup.org/wp-content/uploads/SPL2023_TDP_B_Human.pdf (siehe S. 16).

What is Regression Testing? (2024). Abgerufen am 01.08.2024. URL: https://www.guru99.com/regression-testing.html (siehe S. 19).

Abbildungsverzeichnis