



UPPAAL2JETRACER

Handbuch

Karlsruher Institut für Technologie (KIT)
KASTEL-Institut für Informationssicherheit und Verlässlichkeit
Modellierung und Analyse im Mobility Software Engineering (MASE)
Jun.-Prof. Dr. Maike Schwammberger
Am Fasanengarten 5
76131 Karlsruhe

Name	E-Mail-Adresse
Ethan David Hale	udgud@student.kit.edu
Felix Dold	unmsq@student.kit.edu
Julian Amadeus Kuhn	uewfh@student.kit.edu
Louis Kevin Fink	uutyc@student.kit.edu
Moritz Maas	utwlw@student.kit.edu

Inhaltsverzeichnis

1	Aufbau	5
1.1	Globale Dateien & UPPAAL-Modell	6
1.2	Parser & Declarations	7
1.3	Controller & JetRacerROS2	8
1.4	Versionsverwaltung	10
1.5	Webinterface	11
1.6	Verwendete Pakete	13
2	Verwendung	15
2.1	Installation und Start	15
2.1.1	Vorbereitung	15
2.1.2	Installation	16
2.1.3	Start	17
2.2	Bedienung	18
2.2.1	Kommandozeile	18
2.2.2	Webinterface	21
2.3	Eingabefomat	24
2.3.1	Unterstützte Automaten	24
3	Erweiterung	26
3.1	Declaration-Parser	26
3.1.1	Neuen Datentyp hinzufügen	26
3.1.2	Wertebereich eines Typs verändern	27
3.1.3	Neuen Typqualifizierer hinzufügen	29
3.1.4	Variablendeklarationen einschränken	30
3.2	Hardware-Commandsystem	31
3.2.1	Hardware-Commands	31
3.2.2	Andere Roboter	33
3.3	Logger hinzufügen	35
3.4	Tests schreiben	37
3.5	Dokumentation generieren	38

4 Fehlerbehandlung	39
4.1 Fehlermeldungen	39
4.2 Probleme am JetRacer	41
4.2.1 Fehlende Spurtreue	41

Auch wenn in diesem Handbuch der Lesbarkeit halber das generische Maskulinum verwendet wird, sind stets alle Geschlechter angesprochen.

Dieses Handbuch, inklusive der damit verbundenen Software, möchten wir Maike, Qais und Nils widmen.

Abbildungsverzeichnis

2.1	Login-Fenster des Webinterfaces	21
2.2	Hauptfenster des Webinterfaces	21
2.3	Projektübersicht im Webinterface	22
2.4	Projekterstellung im Webinterface	22
2.5	Versionsauswahl im Webinterface	23
2.6	Einstellungsfenster im Webinterface	23
2.7	Automat <i>2doors</i> mit verschiedenen Declarations	24
2.8	Übersicht: Sichtbarkeitsbereiche (Scopes) der UPPAAL-Declarations	25
3.1	Ausführen eines Hardware-Commands	31

1 Aufbau

- `uppaal2jetracer`: Wurzelverzeichnis aller Daten von UPPAAL2JETRACER .
- `data`: Verzeichnis zum Speichern persistenter Daten, wie Logs und der Projektdatenbank. Dieser Ordner wird mit dem Dockercontainer geteilt. Das *logging*-Paket legt hier die Log-Dateien aller von uns definierten Logger ab.
- `setup`: Verzeichnis der Daten, die zum Aufbau des Dockercontainers notwendig sind.
- `uml-diagrams`: Verzeichnis aller UML-Entwürfe von UPPAAL2JETRACER .
- `docs`: Verzeichnis der Klassen-, Modul- und Paketdokumentation, die mit *Sphinx* erstellt werden.
- `test`: Verzeichnis aller Tests. Die Paketstruktur des Quellcode-Verzeichnisses ist hier gespiegelt, sodass jedes Paket seine Testskripte enthält.
- `uppaal2jetracer`: Quellcode-Verzeichnis.

Die folgenden Unterkapitel bieten eine Übersicht über die Rollen und den Inhalt der verschiedenen Pakete im Quellcode-Verzeichnis.

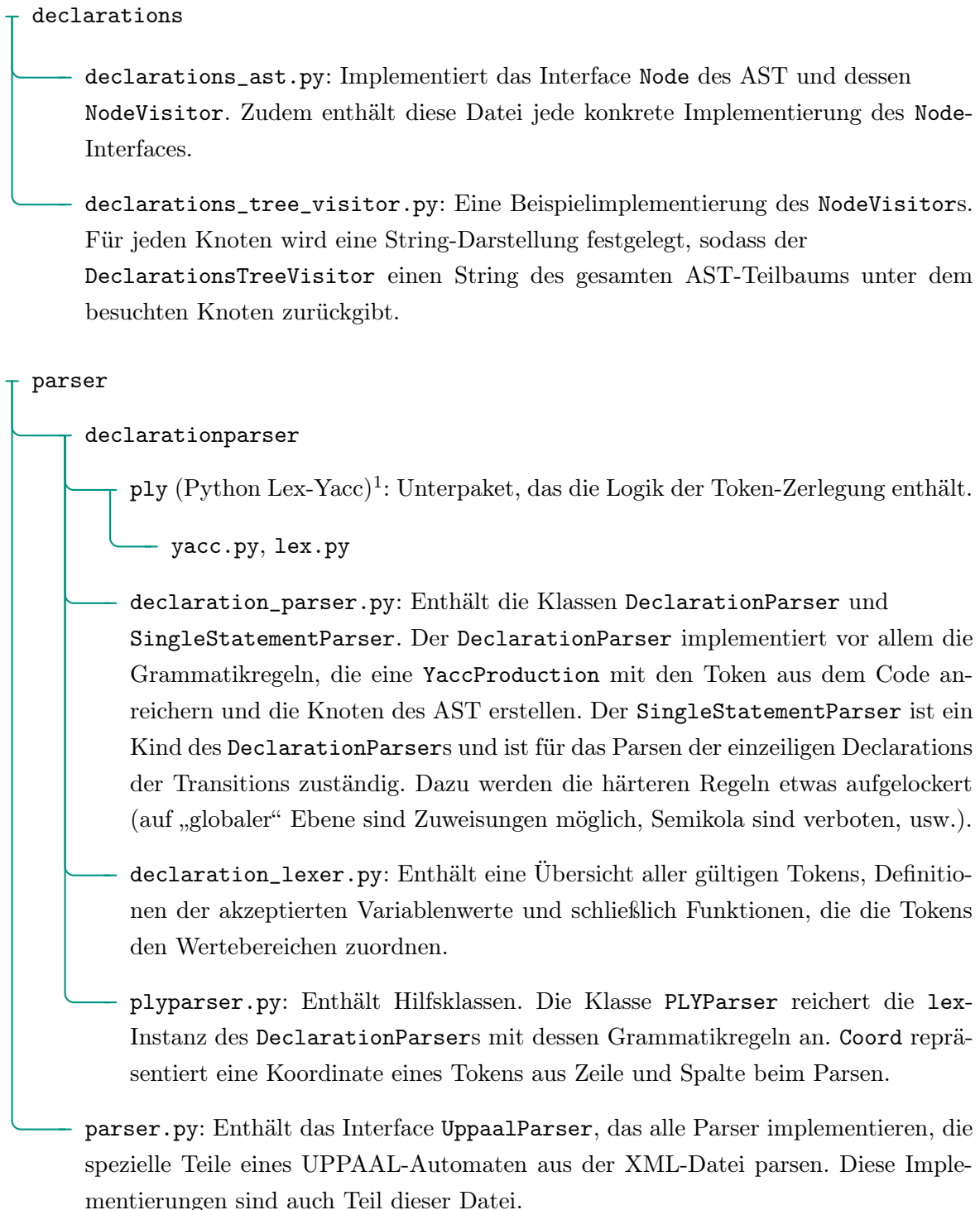
1.1 Globale Dateien & UPPAAL-Modell

- `cli.py`: Verwaltet die Ansteuerung von UPPAAL2JETRACER über die Kommandozeile. Alle Commands werden in der CLI-Implementierung des `CommandHandler`-Interfaces angemeldet, die u2j-Kommandoeingabe wird gestartet und bei jeder Eingabe des Anwenders wird die Gültigkeit der übergebenen Flags und Argumente geprüft.
- `command_system.py`: Enthält die Interfaces `CommandHandler` und `Command`, sowie das `CommandResult`. Außerdem verwaltet die Datei alle konkreten `Command`-Implementierungen.
- `logging.conf`: Konfigurationsdatei für alle Logger aller Komponenten des Projekts. Genaueres in 3.3.

uppaalmodel

- `elements.py`: Enthält das Interface `UppaalElement`, das Komponenten eines UPPAAL-Automaten wie `Locations` und `Transitions` implementiert. Die Datei enthält auch alle Klassen, die Eigenschaften von UPPAAL-Elementen repräsentieren (`Synchronization`, `Guard` und `Update`).
- `system.py`: Enthält die UPPAAL-Komponenten `System` und `Automaton`, die wiederum aus anderen Komponenten aufgebaut sind.
- `frame.py`: Enthält die `Frame`-Klasse. Sie bildet mittels Dictionaries Strings auf Variablen bzw. Funktionen ab.
- `variable.py`: Enthält das Interface `Variable`, sowie alle konkret implementierten Variablentypen. Außerdem gibt hier Fehlerklassen für verschiedene illegale Wertzuweisungen.

1.2 Parser & Declarations



¹<https://www.dabeaz.com/ply/ply.html>

1.3 Controller & JetRacerROS2

`controller`

- `executor.py`: Enthält die `Executor` Klasse, die für die Ausführung von Freitextdeklarationen zuständig ist. Der `Executor` implementiert dabei das Interface `NodeVisitor`.
- `hardware_command_system.py`: Enthält die Klassen `HardwareCommandHandler`, `HardwareCommandResult`, sowie die abstrakte Klasse `HardwareCommand[T]`. Außerdem werden die konkreten Implementierungen der `HardwareCommands` verwaltet.
- `hardware_controller.py`: Enthält die abstrakte Klasse `HardwareController`, sowie ihre konkrete Implementierung `JRHardwareController`.
- `uppaal_controller.py`: Simulation von UPPAAL Systemen

`jetracerros2_interface`

- `action`: Enthält die Definitionen der *Actions*, die vom JetRacer ROS 2 Paket verwendet werden.
- `srv`: Enthält die Definitionen der *Services*, die vom JetRacer ROS 2 Paket verwendet werden.

- `jetracerros2`: Ein ROS 2 Paket zur Steuerung des JetRacers ROS AI Kits.
 - `jetracerros2`
 - `jetracer_controller.py`: Enthält die Klasse `JetracerController` und den Entry-Point für die ROS 2 Launch Infrastruktur für diese Node.
 - `collision_controller.py`: Enthält die Klassen `CollisionController`, `FilteredLaserScanSubscriber` und `LaserFilterParameterSetter`, welche Funktionalität zum Konfigurieren und Auslesen von LIDAR Daten bieten.
 - `servo_motor_controller.py`: Enthält die Klasse `ServoMotorController` und den Entry-Point für die ROS 2 Launch Infrastruktur für diese Node.
 - `speed_controller.py`: Enthält die Klasse `SpeedController` und `SetSpeedClientAsync`. `SpeedController` bietet Funktionalität, um die Geschwindigkeit des JetRacers zu verändern. Hierfür nutzt `SpeedController` `SetSpeedClientAsync`.
 - `turn_controller.py`: Enthält die Klassen `TurnController` und `TurnActionClientAsync`. `TurnController` bietet Funktionalität, um den JetRacer zu drehen. Hierfür nutzt `TurnController` `TurnActionClientAsync`.
 - `laser_filter.py`: Enthält die Klasse `LaserFilter` und den Entry-Point der ROS 2 Launch Infrastruktur für diese Node.
 - `launch`: Enthält ROS 2 Launch-Files, um die Komponenten des JetRacer ROS 2 Pakets zu starten.
 - `jetracerros2_launch.py`: Ein ROS 2 Launch-File, um die gesamte JetRacer ROS 2 Infrastruktur zu starten.
 - `lidar_launch.py`: Ein ROS 2 Launch-File, um `rplidar_node` und `LaserFilter` zu starten.
 - `minimal_launch.py`: Ein ROS 2 Launch-File, um `JetracerController` und `LaserFilter` zu starten.
 - `movement_controller_launch.py`: Ein ROS 2 Launch-File, um `JetracerController` und `ServoMotorController` zu starten.

1.4 Versionsverwaltung

`versioncontrol`

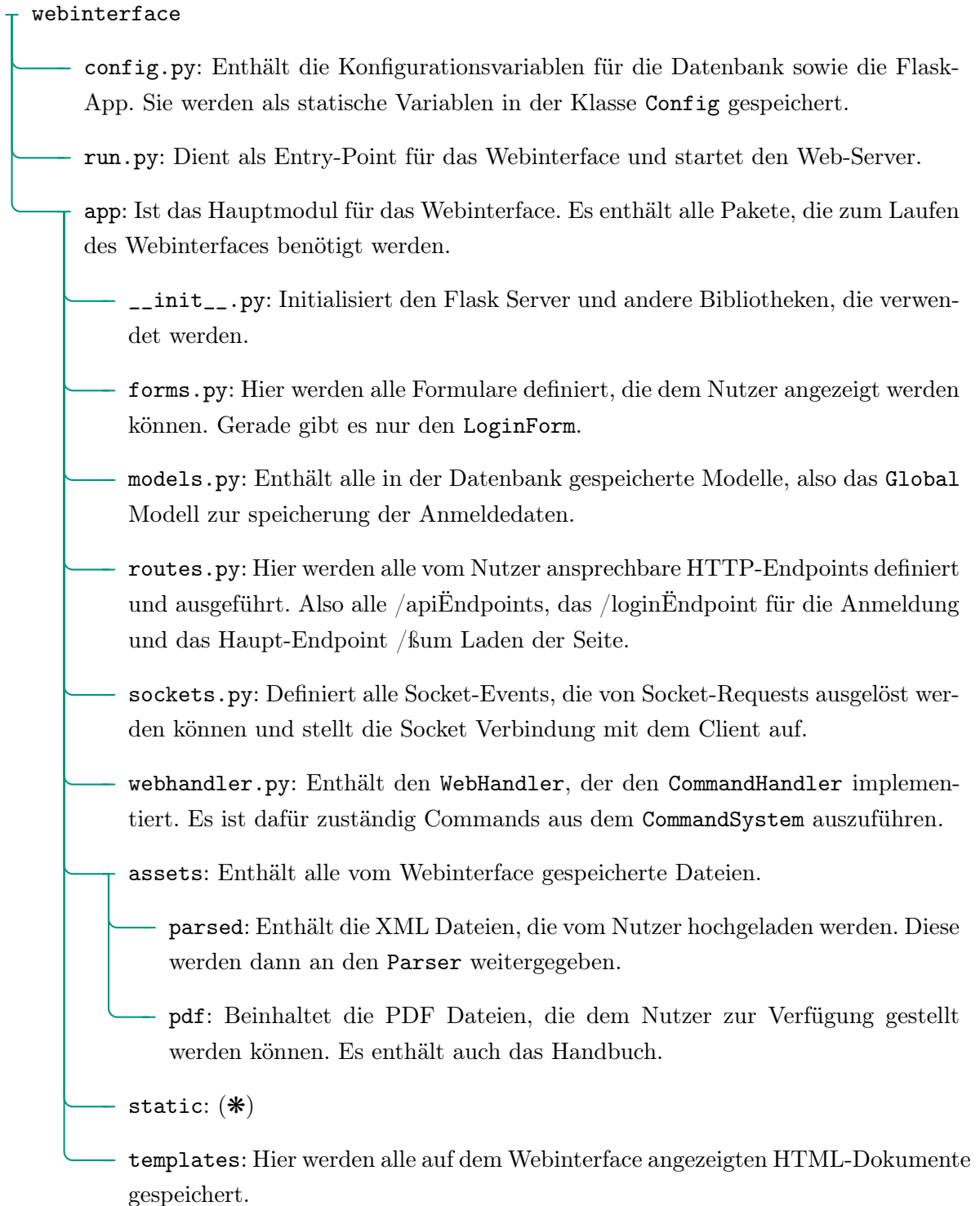
`config.py`: Enthält die Konfigurationsvariablen für die Datenbank. Sie werden als statische Variablen in der Klasse `Config` gespeichert.

`database.py`: In der `DatabaseConnection` Klasse wird die Datenbank zur Speicherung der Projekt- und Versionsdaten erstellt mit der Datenbank Datei. Eine Instanz dieser Klasse wird dann vom `ProjectManager` und `VersionManager` benutzt um mit der Datenbank zu interagieren.

`models.py`: Enthält alle in der Datenbank gespeicherte Modelle, also die `Project`, `Version` und `Global` Modelle, die zur Verwaltung der Projekt- bzw. Versionsdaten sowie globale Daten wie das Versionslimit benötigt werden.

`versioncontrol.py`: Enthält die beiden Manager Klassen, die jeweils die Modelle in der Datenbank abrufen sowie modifizieren. Also hier ist das `ProjectManager` und `VersionManager`.

1.5 Webinterface



- (*) **static**: Enthält alle Dateien, die vom Browser, beim Laden der Seite geholt werden.
 - fonts**: Hier werden alle Schriftarten gespeichert, die auf dem Webinterface zu sehen sind.
 - icons**: Hier werden alle Icons gespeichert, die auf dem Webinterface zu sehen sind.
 - styles**: Für das Design der HTML Seiten werden Stylesheets in CSS geschrieben, die hier gespeichert sind.
 - scripts**: Sind alle JavaScript Dateien, die die Funktionalität der Webseite implementieren.
 - event_handler.js**: Enthält den **EventHandler**, der sich um alle Nutzerinteraktionen kümmert. Hauptsächlich sind das Click-Events.
 - page_handler.js**: Enthält den **PageHandler**, der sich um die Aktualisierung der UI kümmert.
 - request_handler.js**: Enthält den **RequestHandler**, der für alle HTTP-Requests an den Server zuständig ist. Es gibt die Möglichkeit entweder einen **GET** oder **POST** Request zu schicken.
 - socket_handler.js**: Enthält den **SocketHandler**, der für alle eingehenden Socket-Requests zuständig ist.
 - main.js**: Wird beim ersten Laden der Seite ausgeführt und startet die Handler. Dient als Entry-Point der JavaScript Logik.
 - login.js**: Wird nur von der Login Seite verwendet und kümmert sich um kleine Aktualisierungen der UI.
 - events**: Hier werden alle existierenden Events gespeichert, die vom **EventHandler** aufgerufen werden können.

1.6 Verwendete Pakete

Name	Version	Beschreibung
bs4	4.12.3	Mit <i>Beautiful Soup 4</i> können XML-Dateien analysiert und zerlegt werden, um Tags, Attribute oder Labelinhalte auszu-lesen.
eventlet	0.39.0	<i>eventlet</i> ermöglicht nebenläufiges Networking durch Green Threads, was nicht-blockierende I/O-Operationen beschleunigt.
flask	3.1.0	<i>Flask</i> ist ein leichtgewichtiges Webframework, das eine flexible Grundlage für die Entwicklung von Webanwendungen und APIs bietet.
logging	∈ Python	<i>logging</i> ermöglicht es, einen globalen Logger zu importieren, der Nachrichten mit verschiedenen Prioritäten aufnehmen und in einer Datei ausgeben kann. Der Logger ist ein Singleton-Objekt.
pickle	∈ Python	Um Objekte zu serialisieren, kodiert das Paket <i>pickle</i> mit einem Pickler Objekte zu einer Binärdatei. Mit einem Unpickler, wird diese Datei wieder zu Objekten deserialisiert.
pycparser	2.22	<i>pycparser</i> ist ein vollständig in Python implementierter C-Parser nach dem C99-Standard. Intern wird das Paket <i>ply</i> verwendet. <i>pycparser</i> wurde stark modifiziert, um die Declarations aus UPPAAL zu parsen.
rclpy	3.2.1	<i>rclpy</i> ist eine ROS 2 Client-Library für Python.
requests	2.32.3	<i>requests</i> dient dazu, HTTP-Anfragen (z. B. GET, POST) auf einfache Weise zu senden und Webinhalte abzurufen.
rplidar_ros	10.01.25	<i>rplidar_ros</i> stellt ein ROS 2 Node für einen RPLIDAR-Sensor von <i>Slamtec</i> zur Verfügung.
socketio	5.5.1	<i>socketio</i> ist eine Bibliothek für Echtzeitkommunikation, die WebSockets und Fallback-Mechanismen verwendet, um bidirektionale, ereignisbasierte Kommunikation zwischen Client und Server zu ermöglichen.
sqlalchemy	2.0.37	<i>sqlalchemy</i> ist ein ORM-Framework, das SQL-Abfragen abstrahiert und Datenbankoperationen erleichtert.

Neben Python $\geq 3.12.7$ und den in der Tabelle erwähnten externen Paketen werden verschiedene Python-interne Pakete, wie `__future__`, `ABC`, `typing`, o.ä. für die Typisierung des Codes verwendet.

2 Verwendung

Dieses Kapitel behandelt die intendierte Verwendung und das Zusammenspiel der verschiedenen Komponenten von UPPAAL2JETRACER . Es richtet sich an den Anwender.

2.1 Installation und Start

2.1.1 Vorbereitung

UPPAAL2JETRACER ist eine Software, die es ermöglichen soll, zeitgesteuerte Automaten des Model-Checkers UPPAAL zur Steuerung von Robotern zu verwenden, insbesondere des JetRacer ROS AI Kits.

Damit der JetRacer gesteuert werden kann, muss UPPAAL2JETRACER auf dem JetRacer installiert werden. Dazu muss auf dem Roboter ein Betriebssystem-Image laufen, sodass er per SSH angesprochen werden kann. Zur Installation eines Images und der SSH-Verbindung hilft das offizielle Tutorial¹ des Herstellers *Waveshare*.

Um UPPAAL2JETRACER zu starten, muss Docker² auf dem System vorinstalliert sein. Anschließend kann UPPAAL2JETRACER installiert werden.

UPPAAL2JETRACER kann in kleinerem Funktionsumfang auch ohne Roboterhardware verwendet werden, indem man den Docker-Container lokal startet.

¹https://www.waveshare.com/wiki/JetRacer_ROS_AI_Kit_Tutorial_II:_Install_Jetson_Nano_Image

²<https://docs.docker.com/get-started/get-docker/>

2.1.2 Installation

1. Das Repository klonen:

```
git clone git@gitlab.kit.edu:unmsq/uppaal2jetracer.git
cd uppaal2jetracer
```

2. Das Setup-Skript ausführen:

```
cd setup/scripts

# Windows (Admin)
.\setup.bat

# Linux, macOS
sudo chmod +x setup.sh
sudo ./setup.sh
```

3. Ausführungsprofil wählen: Das Ausführungsprofil bestimmt, welche Abhängigkeiten in den Docker-Container aufgenommen werden, der später auf dem JetRacer läuft.

- a) `setup/config.ini` in einem Texteditor öffnen
- b) Setze die Variable `profile` im Abschnitt `[profiles]` auf das gewünschte Profil.
 - **jetracer**: für den Einsatz auf dem JetRacer. *Nur mit diesem Profil funktioniert der Docker-Container auf dem JetRacer.*
 - **minimal**: für ein minimales Setup ohne Hardwareanforderungen. So können alle Komponenten verwendet werden, die keinen Roboter benötigen, wie der Parser oder die Versionsverwaltung.

4. Das Docker-Image bauen:

```
cd setup/scripts

# Windows (Admin)
.\build.bat

# Linux, macOS
sudo chmod +x build.sh
./build.sh
```

Falls `docker image list` ein Image namens `uppaal2jetracer` anzeigt, war der Prozess erfolgreich.

2.1.3 Start

Vor dem Start von UPPAAL2JETRACER muss die Installation abgeschlossen sein. Das `run`-Skript fährt den Docker-Container hoch und der Webserver für das Webinterface wird automatisch gestartet. Um auf den Webserver zuzugreifen, verbindet man sich im Browser eines Computers desselben Netzwerks mit `http://[ip-jetracer]:5000`, wobei die IP des JetRacers auf dessen Display abzulesen ist (siehe auch Tutorial von *Waveshare*).

Möchte man nun klassisch per Kommandozeile auf UPPAAL2JETRACER zugreifen, führt man in einem neuen Terminal das `cli`-Skript aus.

Folgende Commands starten also den Docker-Container, sowie auf Wunsch die UPPAAL2JETRACER-Kommandozeile:

```
cd setup/scripts

# Windows
.\run.bat
.\cli.bat           # in einem neuen Terminal

# Linux, macOS
sudo chmod +x run.sh
./run.sh
sudo chmod +x cli.sh # in einem neuen Terminal
./cli.sh
```

2.2 Bedienung

2.2.1 Kommandozeile

UPPAAL2JETRACER kann vom Anwender über die Kommandozeile textuell bedient werden. Dabei bietet die Kommandozeile folgende Befehle:

parse

Der **parse**-Befehl ist für das Übersetzen einer XML-Datei eines UPPAAL-Automaten in ein ausführbares Automatenmodell zuständig. Das übersetzte Modell wird bei Erfolg als PKL-Datei im gleichen Ordner wie die ursprüngliche XML-Datei abgelegt und in der Versionsverwaltung im aktiven Projekt als Version gespeichert.

Nutzung:

```
parse <path> [-d | --debug] [-h | --help]
```

Argumente:

<path> Ein Pfad zu einer gültigen XML-Datei eines UPPAAL-Automaten.

Optionen:

-d, --debug Bei aktiver Debug-Option werden beim Ausführen des Befehls zusätzliche Debug-Nachrichten in die Log-Dateien ausgegeben.

-h, --help Bei aktiver Hilfe-Option wird eine Hilfsnachricht zum Befehl ausgegeben.

run

Der **run**-Befehl ist für die Ausführung eines übersetzten Automatenmodells zuständig. Das Automatenmodell liegt dabei in Form einer PKL-Datei mit serialisiertem UPPAAL-System vor.

Nutzung:

```
run <path> [-d | --debug] [-h | --help]
```

Argumente:

<path> Ein Pfad zu einer gültigen PKL-Datei eines serialisierten UPPAAL-Modells.

Optionen:

-d, --debug Bei aktiver Debug-Option werden beim Ausführen des Befehls zusätzliche Debug-Nachrichten in die Log-Dateien ausgegeben.

-h, --help Bei aktiver Hilfe-Option wird eine Hilfsnachricht zum Befehl ausgegeben.

pnr

Der **pnr**-Befehl ist für das Übersetzen und Ausführen eines UPPAAL-Automaten in einem Schritt verantwortlich. Dabei ist der Befehl äquivalent zur sequentiellen Nutzung der **parse**- und **run**-Befehle.

Nutzung:

```
pnr <path> [-d | --debug] [-h | --help]
```

Argumente:

<path> Ein Pfad zu einer gültigen XML-Datei eines UPPAAL-Automaten.

Optionen:

-d, --debug Bei aktiver Debug-Option werden beim Ausführen des Befehls zusätzliche Debug-Nachrichten in die Log-Dateien ausgegeben.

-h, --help Bei aktiver Hilfe-Option wird eine Hilfsnachricht zum Befehl ausgegeben.

prj

Der **prj**-Befehl ist für das Verwalten von Projekten zuständig. Ein Projekt ist dabei eine benannte Sammlung an Versionen geparster Automaten.

Nutzung:

```
prj (list | current | config | max <int > 0> | open <name> | new <name> | delete <name>) [-d | --debug] [-h | --help]
```

Unterbefehle:

list Listet alle existenten Projekte auf.

current Zeigt das aktuell ausgewählte Projekt.

config Zeigt die aktuelle Konfiguration des Versionsmaximums.

max <int > 0> Setzt das Versionslimit auf den gegebenen Wert.

open <name> Öffnet das Projekt mit dem gegebenen Namen.

new <name> Erstellt ein neues Projekt mit dem gegebenen Namen.

delete <name> Löscht ein bestehendes Projekt mit dem gegebenen Namen.

Argumente:

<int > 0> Ein Pfad zu einer gültigen XML-Datei eines UPPAAL-Automaten.

<name> Ein Name, bestehend aus Buchstaben, Ziffern und Unterstrichen ohne andere Sonderzeichen.

Optionen:

-d, --debug Bei aktiver Debug-Option werden beim Ausführen des Befehls zusätzliche Debug-Nachrichten in die Log-Dateien ausgegeben.

-h, --help Bei aktiver Hilfe-Option wird eine Hilfsnachricht zum Befehl ausgegeben.

ver

Der **ver**-Befehl ist für das Verwalten von Versionen im aktuellen Projekt zuständig. Eine Version ist dabei ein geparster UPPAAL-Automat.

Nutzung:

```
ver (list | run <id> | fav <id> | delete <id>) [-d | --debug] [-h | --help]
```

Unterbefehle:

list	Listet alle Versionen im aktuellen Projekt auf.
run <id>	Führt eine existente Version analog zum run -Befehl aus.
fav <id>	Favorisiert eine Version und schützt sie somit vor automatischer Löschung.
delete <id>	Löscht eine existente Version.

Argumente:

<id>	Die Identifikationsnummer eines existenten Projekts.
-------------------	--

Optionen:

-d, --debug	Bei aktiver Debug-Option werden beim Ausführen des Befehls zusätzliche Debug-Nachrichten in die Log-Dateien ausgegeben.
-h, --help	Bei aktiver Hilfe-Option wird eine Hilfsnachricht zum Befehl ausgegeben.

help

Der **help**-Befehl gibt eine Hilfmeldung aus, die die verfügbaren Befehle zeigt.

Nutzung:

```
help
```

2.2.2 Webinterface

Hier wird die Verwendung des Webinterfaces beschrieben.

Beim ersten Start des Webinterfaces wird man mit einer Passworteingabe konfrontiert (Abb. 2.1). Das Master-Passwort lautet `u2j`.

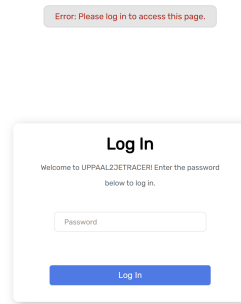


Abbildung 2.1: Login-Fenster des Webinterfaces

Im Hauptfenster (Abb. 2.2) wählt man zuerst ein Projekt unter „Change Project“ aus. Es öffnet sich der Dialog *Projects* (Abb. 2.3) zur Auswahl eines Projekts. Über „New Project“ öffnet sich eine Eingabeaufforderung (Abb. 2.4) für den Namen des neuen Projekts.

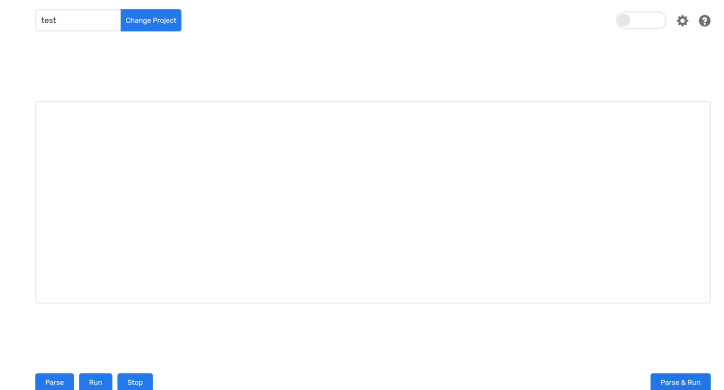


Abbildung 2.2: Hauptfenster des Webinterfaces

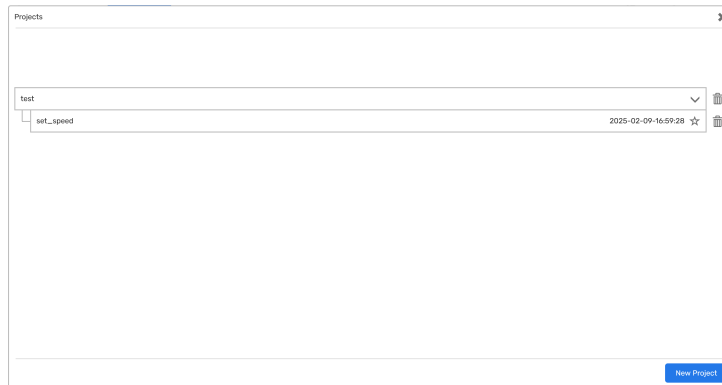


Abbildung 2.3: Projektübersicht im Webinterface

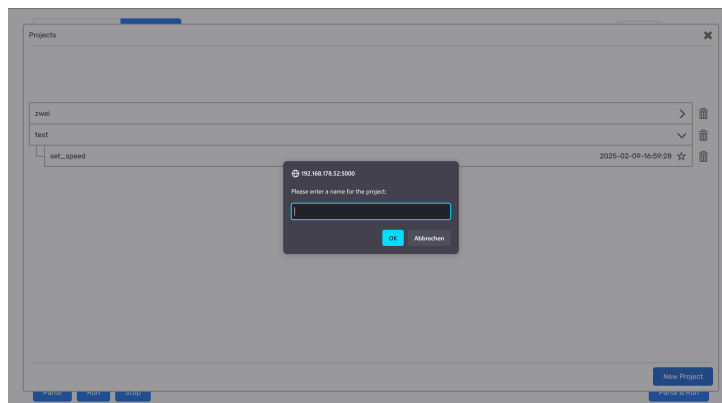


Abbildung 2.4: Projekterstellung im Webinterface

Anschließend kann man mit den unteren Buttons interagieren. „Parse“ öffnet einen Uploaddialog für einen Automaten. Dann kann dieser Automat mit „Run“ ausgeführt werden. Dabei wird für die durch den Parser generierten Steueranweisungen eine Version erstellt und im Projekt abgelegt. Falls anfangs kein Projekt ausgewählt wurde, weist einen „Parse“ durch einen Fehler darauf hin. Falls im gewählten Projekt schon Versionen vorhanden sind, öffnet „Run“ den Dialog *Versions* (Abb. 2.5), um von diesen eine zu wählen. „Parse & Run“ führt beide Aktivitäten hintereinander aus. Unter „?“ öffnet sich die PDF-Datei dieses Handbuchs, das Zahnrad öffnet den Einstellungsdialog *Settings* (Abb. 2.6).



Abbildung 2.5: Versionsauswahl im Webinterface

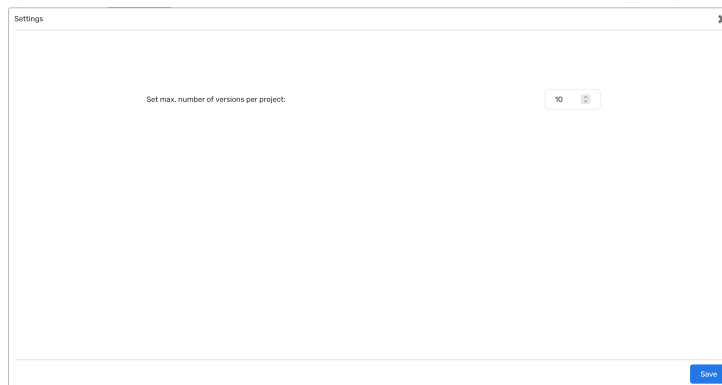


Abbildung 2.6: Einstellungsfenster im Webinterface

Der Dialog *Projects* dient zur Verwaltung der Projekte und kann unter „Change Project“ aufgerufen werden. Mit einem Klick auf den Titel eines Projekts wird dieses im Hauptfenster (2.2) geöffnet. Ganze Projekte oder einzelne Versionen geparster Automaten können manuell über den Mülleimer gelöscht werden. Über den Stern können einzelne Versionen geparster Automaten von der automatischen Löschung ausgeschlossen werden. „New Project“ erlaubt es, ein neues Projekt anzulegen.

Der Dialog *Versions* dient zur Wahl einer Version eines geparsten Automaten innerhalb eines Projekts. Das Projekt wurde vorher bereits festgelegt von (2.3).

In den Einstellungen kann man allgemeine Projekteinstellungen festlegen; insbesondere die maximale Anzahl gespeicherter Versionen geparster Automaten in einem Projekt.

2.3 Eingabefomat

2.3.1 Unterstütze Automaten

Declarations

Declarations kommen in UPPAAL an verschiedenen Stellen vor, hier absteigend in ihrer Sichtbarkeit: globale Declarations, System-Declarations, Declarations pro Automat. Zudem gibt es einzeilige Statements an den Transitions (Guards, Updates, Synchronizations) und in den Locations (Invariants) eines Automaten. In Abbildung 2.7 befinden sich in der linken Leiste die verschiedenen Arten der Declarations. Im gezeigten Automaten *Door* kann man Updates (dunkelblau), Synchronizations (hellblau), Guards (grün) und Invarianten (lila) erkennen.

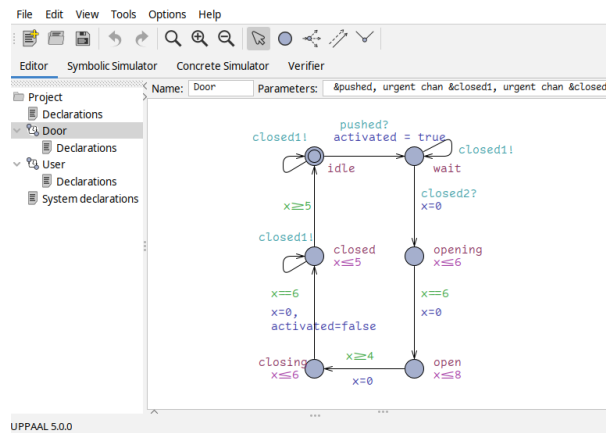


Abbildung 2.7: Automat *2doors* mit verschiedenen Declarations

Variablentypen: `bool`, `int`, `double`, `chan`, `clock`, `range`, `string`, `array`

Typqualifizierer (Qualifier): `const`, `broadcast`

- Sichtbarkeiten: Lokale Automaton-Declarations und System-Declarations haben Zugriff auf eigene Variablen bzw. Funktionen sowie auf alle globalen Declarations. Funktionen haben ihren eigenen Scope. Siehe auch Abb. 2.8.
- Funktionen können definiert werden.
- Variablenzuweisungen sind innerhalb von Funktionen erlaubt.
- Variablendeklarationen können innerhalb des Declaration-Codes erfolgen, im Gegensatz zu den Transitions.
- Systemdeklarationen: Innerhalb der System-Declarations können Templates definiert und ein System spezifiziert werden.

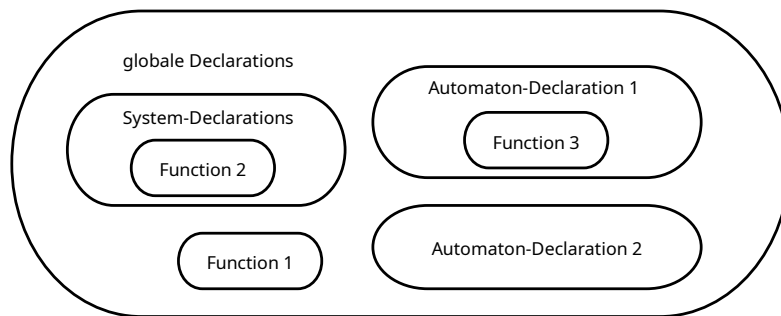


Abbildung 2.8: Übersicht: Sichtbarkeitsbereiche (Scopes) der UPPAAL-Declarations

Automaton

Locations: Invariants

Transitions: Guards, Synchronizations, Updates

- Guards und Invarianten: Erlaubt sind boolesche Ausdrücke.
- Updates: Erlaubt sind Zuweisungen.
- Synchronizations: Erlaubt sind Channelreferenzen in Kombination mit ! oder ?.
- Variablentypen: Alle in den Declarations definierbaren Typen sind zulässig.
- Funktionen mit entsprechendem Rückgabewert können als Invariante, Guard und Update aufgerufen werden.
- Hardware-Commands: Werden analog zu Funktionen definiert. Hardware-Commands haben stets eine höhere Präzedenz als in den Declarations definierte Funktionen. Existiert eine Funktion mit demselben Namen wie ein Hardware-Commands, wird sie ignoriert.
- Parameter können call-by-value und call-by-reference sein.

3 Erweiterung

3.1 Declaration-Parser

In diesem Abschnitt wird basierend auf dem grundlegenden Aufbau der Pakete *DeclarationParser* und *Declarations* (siehe 1.2) beschrieben, wie die Regeln zum Parsen von Declarations erweitert werden können.

Für jede Declaration einer UPPAAL-Datei wird vom Paket *DeclarationParser* mit Hilfe des Pakets *Declarations* ein *Abstract Syntax Tree (AST)* erstellt. *Declarations* enthält alle Knoten für den AST, *DeclarationParser* zerlegt den Eingabestring und generiert den Baum.

Declarations haben eine C-artige Syntax. Daher wurde als Basis für die Implementierung des Declaration-Parsers das Pythonpaket *pycparser*¹ verwendet.

Zur direkten Verwendung der Klassen können die entsprechenden Tests betrachtet oder erweitert werden, wie in 3.4 beschrieben. Innerhalb von UPPAAL2JETRACER und außerhalb der gezeigten Pakete werden sie in `parser.py` verwendet.

In den folgenden Abschnitten werden konkrete Vorgehensweisen zur Änderung des Parsingverhaltens beschrieben. Für gezielte Änderungswünsche lohnt sich immer auch ein Blick ins Netz, da *yacc* und *lex* (gebündelt in *ply*) sehr etablierte Parser und Lexer sind. So können auch die gängigen KI-Modelle die meisten Fragen umfangreich beantworten.

3.1.1 Neuen Datentyp hinzufügen

Am Beispiel des Datentyps `clock` wird hier gezeigt, wie dieser Typ gültig geparkt werden kann. Dies braucht nur zwei Änderungen:

1. *Grammatik erweitern*: Im `DeclarationParser` muss die Regel `p_type_specifier_no_typeid` erweitert werden:

¹<https://github.com/eliben/pycparser>

```
def p_type_specifier_no_typeid(self, p):
    """ type_specifier_no_typeid : VOID
                                   | BOOL
                                   | CHAR
                                   | INT
                                   | CLOCK
                                   | DOUBLE
    """
    p[0] = c_ast.IdentifierType([p[1]], coord=self._token_coord(p, 1))
```

2. *Keyword hinzufügen*: Auch `DeclarationLexer` muss erweitert werden, damit `clock` als gültiges Token erkannt werden kann:

```
keywords = (
    "CLOCK", "CONST", "DOUBLE", "INT", "RETURN",
    "STATIC", "STRING", "STRUCT", "TYPEDEF", "VOID",
)
```

3.1.2 Wertebereich eines Typs verändern

Auch ist es möglich, den Wertebereich eines Typs festzulegen oder zu verändern. Hier wird dies am Beispiel des durch 3.1.1 hinzugefügten Keywords `bool` gezeigt, das den Wertebereich `{true, false}` erhält.

1. Den Lexer erweitern:

- a) Ein neues Token im Tupel `tokens` muss angelegt werden, das den neuen Wertebereich bezeichnet:

```
tokens = keywords + keywords_new + (
    ...
    # constants
    "INT_CONST_DEC", "INT_CONST_OCT", "INT_CONST_HEX",
    "INT_CONST_BIN", "INT_CONST_CHAR",
    "FLOAT_CONST", "HEX_FLOAT_CONST",
    "CHAR_CONST",
    "WCHAR_CONST",
    "U8CHAR_CONST",
    "U16CHAR_CONST",
    "U32CHAR_CONST",
    "BOOL_CONST",
    ...
)
```

- b) Der Wertebereich muss durch einen regulären Ausdruck beschrieben werden. Dieser kann im passenden Abschnitt in der Mitte des `DeclarationLexers` hinzugefügt werden:

```
BOOL_CONST = r"""(true|false)"""
```

- c) Das Token "BOOL_CONST" muss nun noch mit der Variable `BOOL_CONST` verknüpft werden. Dazu wird eine Funktion mit dem `@TOKEN`-Decorator weiter unten hinzugefügt:

```
@TOKEN(BOOL_CONST)
def t_BOOL_CONST(self, t):
    """
    Matches and returns a boolean constant token.
    """
    return t
```

2. Den Parser erweitern: Wie wird der Wertebereich mit dem Variablentyp verbunden? Hierzu muss die Grammatik erweitert werden.

Wir inspirieren uns bei der Implementierung bei den anderen Funktionen. Die kontextfreien Grammatikregeln (Produktionen) werden aus den Docstrings gelesen. Im Allgemeinen gibt es für jedes Nichtterminal `X` einer Produktion auch eine Funktion `p_X(self, p)`. Dabei ist `p` eine `YaccProduction`, denn diese Funktionen werden von `yacc.py` aufgerufen. In `p[0]` soll nach der Funktion das Ergebnis der Produktion des Docstrings stehen, `p[1]`, `p[2]`, usw. entsprechen dabei den Nichtterminalen der rechten Seite der Produktion, von links nach rechts. Die `p[i]` der rechten Seite können entweder Token der lexikalischen Analyse enthalten (`COMMA`, `NUM`, `PLUS`, usw.) oder Zwischenergebnisse von Nichtterminalen, die in anderen Produktionen verarbeitet werden. Betrachte folgendes Beispiel:

```
def p_expression(self, p):
    """ expression : assignment_expression
                  | expression COMMA assignment_expression
    """
    if len(p) == 2:
        p[0] = p[1]
    else:
        if not isinstance(p[1], declarations_ast.ExprList):
            p[1] = declarations_ast.ExprList([p[1]], p[1].coord)
        p[1].exprs.append(p[3])
        p[0] = p[1]
```

Hier sind `assignment_expression` und `expression` Nichtterminale, deren Werte von anderen Produktionen abgeleitet werden. `COMMA` ist ein Token des Lexers. Je nachdem, ob

die erste oder zweite Produktion angewendet wird (dies entscheidet `yacc.py`, basierend auf der Eingabe, hier z.B. ob `a = 5;` oder `x = 1, y = 2;`), entspricht `p[1]` dem Nicht-terminal `assignment_expression` oder `expression`. Falls die zweite Produktion gewählt wird, gibt es auch ein `p[2]` und `p[3]`, andernfalls nicht. Anhand der Länge der Produktion `len(p)` kann also unterschieden werden, welche Produktion angewendet wird und `p[0]` kann verschieden gebildet werden.

Da es sich bei `BOOL_CONST` um einen Wert `true` bzw. `false` handelt, der später im Baum im Knoten `Constant` stehen soll, können wir so schnell den richtigen Abschnitt der Grammatik finden: der, in dem Objekte vom Typ `Constant` erstellt werden. Wir fügen eine vierte Regel mit unserem neu erstellten Token `BOOL_CONST` hinzu:

```
def p_constant_4(self, p):
    """ constant      : BOOL_CONST"""
    p[0] = declarations_ast.Constant(
        "bool", p[1], self._token_coord(p, 1)
    )
```

3.1.3 Neuen Typqualifizierer hinzufügen

Qualifier sind modifizierende Keywords vor einem Variablennamen, z. B. `const`, `static` oder `broadcast`. Der Vorgang wird am Beispiel des `broadcast`-Qualifiers erklärt.

1. Den Lexer erweitern: Der neue Qualifier muss in die `keywords` aufgenommen werden:

```
keywords = (
    "BROADCAST", "CLOCK", "CONST", "DOUBLE", "INT",
    "RETURN", "STATIC", "STRING", "STRUCT", "TYPEDEF", "VOID",
)
```

2. Den Parser erweitern: Um neue Qualifier aufzunehmen, muss die Produktion der Funktion `p_type_qualifier` erweitert werden:

```
def p_type_qualifier(self, p):
    """ type_qualifier : CONST
                        | STATIC
                        | BROADCAST
    """
    p[0] = p[1]
```

3.1.4 Variablendeklarationen einschränken

Oft möchte man auch semantische Aspekte in den Parser einfließen lassen, um Fehler bei unsinnigen Zuweisungen zu werfen, wie z.B. `int x = false;`. Jeder `Decl`-Knoten wird in der Hilfsfunktion `_build_declarations` erstellt. Bevor jedoch ein neuer `Decl`-Knoten erstellt wird, wird die Funktion `_verify_declaration(self, decl, spec)` aufgerufen. `decl` und `spec` enthalten gemeinsam alle Informationen einer Variablendeklaration, u.a. auch Typqualifizierer, Variablenname und Wert, falls vorhanden. Diese Methode prüft bestimmte Kombinationen von Qualifiern und Typen, Typen und Werten auf ihre Sinnhaftigkeit und kann entsprechend erweitert werden.

Fehler können mit der `_parse_error(self, msg, coord)` der Klasse `PLYParser` geworfen werden, die die Nachricht und die Koordinate der Klasse `Coord` des Tokens entgegennimmt, das für den Fehler verantwortlich ist.

3.2 Hardware-Commandsystem

Dieser Abschnitt erklärt, wie neue Hardware-Commands hinzugefügt werden können. Außerdem wird beschrieben, wie auch andere Roboter neben dem JetRacer von UPPAAL2JETRACER gesteuert werden können.

3.2.1 Hardware-Commands

Hardware-Commands kapseln Hardware-Interaktionen als Objekte. Jeder Hardware-Command ist einem eindeutigen Namen zugeordnet. Ein Hardware-Command kann in einem UPPAAL-System zum Beispiel an einer Transition folgendermaßen ausgeführt werden:

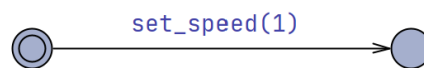


Abbildung 3.1: Ausführen eines Hardware-Commands

Hierbei muss der Name der aufgerufenen Methode genau der Name des auszuführenden Hardware-Commands sein.

Für den JetRacer existieren bereits folgende Hardware-Commands:

- **set_speed**
 - Parameter: **speed**: `float`
 - Rückgabetyt: `void`
 - Beschreibung: Setzt die Geschwindigkeit des JetRacers auf den Wert von **speed** (in $\frac{m}{s}$). $[-1.2, 1.2]$
- **turn**
 - Parameter: **theta**: `float`
 - Rückgabetyt: `void`
 - Beschreibung: Dreht den JetRacer um den gegebenen Winkel **theta** (in Grad).
- **free_ahead**

- Parameter: /
 - Rückgabety: bool
 - Beschreibung: Überprüft, ob die Fläche vor dem JetRacer frei ist und gibt das entsprechende Ergebnis zurück.
- potential_collision
 - Parameter: /
 - Rückgabety: bool
 - Beschreibung: Überprüft, ob der JetRacer dabei ist zu kollidieren und gibt das entsprechende Ergebnis zurück.

UPPAAL2JETRACER kann mit neuen Hardware-Commands erweitert werden. Hierbei sollte sich an bereits implementierten Hardware-Commands orientiert werden.

Zuerst muss im Modul

uppaal2jetracer/uppaal2jetracer/controller/hardware_command_system.py
eine neue Klasse für diesen Hardware-Command angelegt werden.

```
class TurnCommand(JRHardwareCommand[None]):
    """
    A class to represent a hardware command for turning the JetRacer.

    Parameters:
    - theta: The angle to turn the JetRacer in radians.
    """

    def __init__(self, jr_hardware_controller: JRHardwareCommand):
        super().__init__(jr_hardware_controller)

    # execute method
```

Jeder Hardware-Command muss die `execute`-Methode implementieren. Diese definiert die Funktion des Hardware-Commands und gibt ein `HardwareCommandResult` zurück.

Ein `HardwareCommandResult[T]` besitzt ein `result` vom Typ `T` und ein `result_type`. Das `result_type` kann entweder `SUCCESS`, `PARTIAL_SUCCESS` oder `HARDWARE_FAILURE` sein.

```
def execute(self, params: List[object]) -> HardwareCommandResult[None]:

    # Check for valid params

    theta: float = float(params[0])
    hardware_result: HardwareResult = self._hardware_controller
        .turn(math.radians(theta))
```



```
return HardwareCommandResult(HardwareCommandResultType.SUCCESS,
hardware_result)
```

Um Fehler wie invalide Parameter oder Hardware-Fehler auszugeben, können `InvalidParamsAmountError`, `InvalidParamTypeError`, `InvalidParamValueError` und `HardwareError` verwendet werden.

Zuletzt muss der Hardware-Command im `HardwareCommandHandler` registriert werden. Hierfür werden die `_init_commands`- und `_register_command`-Methode verwendet.

```
def _init_commands(self):
    jr_hardware_controller = JRHardwareController()

    self._register_command(self.TURN_COMMAND_NAME,
                           TurnCommand(jr_hardware_controller))
    self._register_command(self.SET_SPEED_COMMAND_NAME,
                           SetSpeedCommand(jr_hardware_controller))
    self._register_command(self.POTENTIAL_COLLISION_COMMAND_NAME,
                           PotentialCollisionCommand(jr_hardware_controller))
    self._register_command(self.FREE_AHEAD_COMMAND_NAME,
                           FreeAheadCommand(jr_hardware_controller))

    self._hardware_controller.append(
        jr_hardware_controller
    )
```

Außerdem benötigt jeder Hardware-Command einen `HardwareController`, um die Hardware zu steuern. `JRHardwareCommand` verwendet den `JRHardwareController`.

3.2.2 Andere Roboter

Das Steuerprogramm des UPPAAL2JETRACER kann auch auf andere Roboter erweitert werden. Hierfür muss zusätzlich zu neuen Hardware-Commands ein neuer `HardwareController` angelegt werden.

```
class DemoController(HardwareController):
    # Provided methods
```

Jeder `HardwareController` muss eine `shutdown`-Methode implementieren, die die Hardware ausschaltet.

```
class DemoController(HardwareController):

    def shutdown(self):
        # Shutdown underlying hardware
```

Außerdem sollte jede bereitgestellte Methode ein `HardwareResult` zurückgeben. Hierdurch kann der derzeitige Zustand der Hardware und das Ergebnis der Aktion zurückgegeben werden.

```
class DemoController(HardwareController):  
  
    # Shutdown  
  
    def defuse_bomb(self) -> HardwareResult:  
        # Defuse bomb  
        return HardwareResult(HardwareState.OK, None)
```

Zuletzt muss der `HardwareController` im `HardwareCommandHandler` registriert werden.

```
def _init_commands(self):  
    jr_hardware_controller = JRHardwareController()  
    demo_controller = DemoController()  
  
    # Register Commands  
  
    self._hardware_controller.append(  
        jr_hardware_controller,  
        demo_controller  
    )
```

3.3 Logger hinzufügen

UPPAAL2JETRACER verwendet das Python-interne *logging*²-Paket, um alle anfallenden allgemeinen Informationen, Debugdaten, Warnungen und Fehler abzufangen und in LOG-Dateien im `data`-Verzeichnis 1 zu speichern. *logging* erlaubt eine Aufgabentrennung, indem man verschiedene Logger für verschiedene Subsysteme definieren kann. Für jeden dieser Logger wird eine eigene LOG-Datei in `data` angelegt. Aktuell definiert UPPAAL2JETRACER folgende Logger:

- **controller**: verantwortlich für den Teil des Steuerprogramms, der UPPAAL-Automaten simuliert. Er loggt im `Executor`, sowie `SystemController` und `AutomataController`.
- **declarations**: verantwortlich für die AST-Struktur des *Declarations*-Pakets und für die Komponenten des *DeclarationParsers*
- **jetracer**: verantwortlich für das *JetRacer-ROS2*-Paket
- **parser**: verantwortlich für den Automatenparser
- **root**: Elter aller anderen Logger, erhält alle Logs der anderen Logger.
- **uppaal_model**: verantwortlich für die Klassen des UPPAAL-Modells
- **user_command**: verantwortlich für User-Command-System
- **version_control**: verantwortlich für die Versionsverwaltung
- **yacc**: verantwortlich für alle Ausgaben des *yacc*-Pakets. Ersetzt *yaccs* interne Logger.

Um einen neuen Logger hinzuzufügen, definiert man diesen in der Datei `logging.conf`, die im Quellcode-Verzeichnis 1 liegt. Dazu muss dessen Name unter `[keys]` sowie `[handlers]` hinzugefügt werden. Bei allen weiteren nötigen Ergänzungen kann man sich an der Struktur der Datei orientieren.

Um einen bestehenden Logger zu verwenden, geht man wie folgt vor:

1. In der Pythondatei, in der geloggt werden soll, muss das Paket `logging` importiert werden.

```
import logging
```

2. Alle Logger, die man verwenden möchte, müssen global initialisiert werden. *logging* identifiziert die Logger eindeutig über ihre Namen, die in der `logging.conf` definiert wurden.

```
yacc_logger = logging.getLogger("yacc")
parser_logger = logging.getLogger("parser")
```

²<https://docs.python.org/3/library/logging.html>

3. Wenn man loggen möchte, ruft man auf seinem gewünschten Logger die Methode mit dem entsprechenden Loglevel auf: `.info(...)`, `.debug(...)`, `.warning(...)`, `.error(...)`. In der `logging.conf` kann man sich auch eigene Loglevel definieren.

3.4 Tests schreiben

UPPAAL2JETRACER wurde testgetrieben mithilfe des Tools *pytest*³ entwickelt. Alle Tests befinden sich im Verzeichnis `test`, wie in der Grafik 1 zu sehen.

Um eigene Testfälle anzulegen und auszuführen, geht man wie folgt vor:

1. Eine bereits erstellte Testdatei auswählen oder eine neue Pythondatei im `test`-Verzeichnis mit dem Präfix `test_` anlegen.
2. Innerhalb dieser Datei im globalen Scope eine neue Funktion mit dem `test_`-Präfix erstellen. Diese muss mit einem `assert`-Statement enden.
3. Um zu testen, kann der `pytest`-Command im Wurzelverzeichnis ausgeführt werden.

Um die Ausgabe des *logging*-Pakets bei allen fehlgeschlagenen Tests anzuzeigen, kann die Option `-v` (verbose) verwendet werden. Dazu müssen vorher alle gewünschten Logger importiert werden, wie in 3.3 beschrieben.

³<https://docs.pytest.org/en/stable/>

3.5 Dokumentation generieren

Nachdem neue Features hinzugefügt oder bestehende Funktionen geändert wurden und alles getestet ist, möchte man ggf. auch eine aktualisierte Version der Dokumentation des Codes generieren lassen. Dazu verwendet UPPAAL2JETRACER das Paket *Sphinx*⁴.

Um die Dokumentation zu generieren, muss Folgendes im Docker-Container des Projekts ausgeführt werden:

1. In das Wurzelverzeichnis des Projekts navigieren:

```
cd uppaal2jetracer
```

2. Die Dokumentation mit folgendem Command generieren lassen:

```
sphinx-apidoc -o docs uppaal2jetracer *ply *declarations \  
*declarationparse *webinterface
```

3. In das Dokumentverzeichnis navigieren:

```
cd docs
```

4. Alte Dokumentation entfernen und neue generieren:

```
make clean  
make html
```

Die Hauptseite der Dokumentation liegt dann in der Datei:
uppaal2jetracer/docs/_build/html/index.html

⁴<https://www.sphinx-doc.org/en/master/>

4 Fehlerbehandlung

4.1 Fehlermeldungen

BelowRangeError:

Wird geworfen, wenn eine Range auf einen Wert unter der oberen Grenze gesetzt wird.

ExceedsRangeError:

Wird geworfen, wenn eine Range auf einen Wert über der oberen Grenze gesetzt wird.

HardwareCommandError:

Wird geworfen, wenn ein Fehler in einem `HardwareCommand` auftritt.

HardwareError:

Erbt von `HardwareCommandError` und wird geworfen, wenn die verwendete Hardware fehlschlägt.

IndexOutOfRangeException:

Wird beim Zugriff auf einen Arrayindex geworfen, der außerhalb der Länge des Arrays liegt.

InvalidGuardError:

Wird geworfen, wenn bei der Evaluierung eines `Guard`s während der Laufzeit ein `InvalidTypeError` aufkommt.

InvalidInvariantError:

Wird geworfen, wenn bei der Evaluierung einer `Invariant` während der Laufzeit ein `InvalidTypeError` aufkommt.

InvalidParamsAmountError:

Erbt von `HardwareCommandError` und wird geworfen, wenn eine falsche Anzahl an Parametern an den `HardwareCommand` übergeben wurde.

InvalidParamTypeError:

Erbt von `HardwareCommandError` und wird geworfen, wenn ein Parameter eines `HardwareCommands` den falschen Typ hat.

InvalidParamValueError:

Erbt von `HardwareCommandError` und wird geworfen, wenn ein Parameter eines `HardwareCommands` einen nicht zulässigen Wert hat.

ImmutableValueError:

Wird geworfen, wenn versucht wird, eine konstante Variable während der Laufzeit zu verändern.

ParseError:

Wird geworfen, falls während des Baus des AST im `DeclarationParser` oder während des Parsings durch `yacc.py` Syntaxverstöße festgestellt werden und daher kein AST gebaut werden kann.

UnexpectedTypeError:

Wird geworfen, wenn bei der Evaluierung von `Guards` oder `Invariants` während der Laufzeit ein unerwarteter Typ errechnet wird. Beispielsweise, falls die Verarbeitung eines `Guards` statt einem Wahrheitswert einen Channel-Wert ergibt.

XMLParseError:

Wird geworfen, wenn beim Parsen einer XML-Datei ein gesuchtes Tag fehlt. Zum Beispiel, wenn ein UPPAAL-Automat keinen Startzustand hat.

4.2 Probleme am JetRacer

Bei Problemen mit dem JetRacer kann auch die [Wiki](#) des Herstellers *Waveshare* helfen.

4.2.1 Fehlende Spurtreue

Der JetRacer weicht von einer geraden Spur ab, obwohl `steering_angle` der Vorderräder auf numerisch 0.0 gesetzt ist.

Zur Behandlung kann entweder die Vorderachse manuell gerade gestellt werden oder `servo_bias` wird entsprechend neu definiert. Hierfür muss in `servo_motor_controller.py` in der `main`-Methode der entsprechende Wert für `servo_bias` folgendermaßen gesetzt werden:

```
def main(args = None):  
    """  
    Entry point for ROS 2 launch script.  
    """  
    # ...  
    servo_motor_controller.set_params(servo_bias = 50)  
    # ...
```

`servo_bias` muss zwischen -500 und 500 liegen. Ein positiver Wert sorgt für eine Abweichung nach rechts, ein negativer Wert für eine Abweichung nach links.