# Fusion von externen Videos und intrinsischen Daten zur Offline-Analyse von Teams mobiler autonomer Roboter

Fusion of external videos and intrinsic data for offline analysis of teams of mobile autonomous robots

Diplomarbeit von Simon Templer
Aufgabenstellung: Prof. Dr. Oskar von Stryk
Betreuer: Dipl.-Inform. Dirk Thomas
Mai 2009

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Fachgebiet Simulation, Systemoptimierung und Robotik

Fusion von externen Videos und intrinsischen Daten zur Offline-Analyse von Teams mobiler autonomer Roboter
Fusion of external videos and intrinsic data for offline analysis of teams of mobile autonomous robots

vorgelegte Diplomarbeit von Simon Templer
Aufgabenstellung: Prof. Dr. Oskar von Stryk
Betreuer: Dipl.-Inform. Dirk Thomas

Tag der Einreichung: 25. Mai 2009

# Erklärung zur Diplomarbeit

Hiermit versichere ich die vorliegende Diplomarbeit ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 25. Mai 2009

_____

(Simon Templer)

## Zusammenfassung

In einem Robotiksystem gibt es viele mögliche Fehlerquellen. Die Ursache für das Fehlverhalten eines Roboters zu identifizieren, ist jedoch meist keine leichte Aufgabe. Ein maßgeblicher Faktor ist dabei die Komplexität des Systems und der zu erfüllenden Aufgaben. Für viele Probleme, die für einen Menschen einfach zu lösen sind, benötigt ein Roboter eine komplexe Steuerungssoftware. Mobile autonome Roboter sind außerdem auf die Ressourcen beschränkt, die sie mit sich führen können, und sind damit neben Softwarefehlern auch besonders anfällig für Fehler in der Hardware. Betrachtet man die Interaktion mehrerer solcher Roboter, so wird die Analyse und Identifikation von Fehlern noch weiter erschwert.

In dieser Arbeit wurde als Ansatz zur Lösung dieses Problems eine Software entwickelt, die eine nachträgliche Analyse auf Basis von aufgezeichneten Daten ermöglicht. Die Identifikation und Beseitigung von Fehlern in komplexen Robotiksystemen soll dadurch vereinfacht werden. Dazu wird das Geschehen aus zwei Blickwinkeln betrachtet - die interne Sicht der Roboter und die externe Sicht auf das Geschehen - und beide zueinander in Bezug gesetzt. Durch Kombination dieser beiden Perspektiven kann die reale Welt mit dem Verständnis, das ein Roboter von ihr hat, verglichen werden. Da das Geschehen nicht in Echtzeit betrachtet wird, ist eine detaillierte und wiederholte Analyse möglich. Die externe Sicht wird durch Videoaufnahmen realisiert, die intrinsischen Roboterdaten werden anschaulich visualisiert. Beide Blickwinkel werden schließlich vereint, indem das Videobild um Informationen aus den Roboterdaten ergänzt wird.

Die Software ist eine Erweiterung des Softwareframeworks *RoboFrame* und wurde im Kontext der *RoboCup German Open* eingesetzt.

## Inhaltsverzeichnis

# 1 Einleitung

Die Steuerungssoftware eines mobilen autonomen Roboters muss viele Aufgaben erfüllen. Da ein Eingreifen von außerhalb je nach Anwendung nicht erlaubt, möglich oder gewünscht ist, liegt die Entscheidungsgewalt über alle Handlungen des Roboters bei dieser Software. Die Daten von in- und externen Sensoren werden verarbeitet, um daraus Modelle von Roboter und Umwelt zu bestimmen. Bei einem System mehrerer kooperierender Roboter können diese Informationen unter den Robotern ausgetauscht und in die jeweiligen Modelle der anderen Roboter integriert werden. Die Verhaltenssteuerung eines Roboters verwendet den aktuell bekannten Zustand von Umwelt und Robotern, um daraus abzuleiten, welche Aktionen er auszuführen hat.

Bei einem derart komplexen System ist es keine einfache Aufgabe, den Grund für das Fehlverhalten eines oder mehrerer Roboter herauszufinden. Ursachen können unter anderem fehlerhafte oder gestörte Sensordaten, eine fehlerhafte Verarbeitung von Sensordaten, Fehler in der Verhaltenssteuerung oder sogar Fehler in der Ausführung von Aktionen sein. Auch fehlerhafte, beschädigte oder überstrapazierte Bauteile können mögliche Fehlerquellen sein.

Um solche Fehler lokalisieren zu können, benötigt man umfassende Information über den Zustand der Roboter und über das Verständnis, das diese Roboter von ihrer Umwelt haben. In vielen Fällen reicht es jedoch nicht aus, die Umwelt aus Sicht der Roboter zu kennen, es werden als weitere Perspektive Informationen über die reale Umwelt benötigt, über die der Roboter selbst nicht verfügt. Ein Blick auf die tatsächliche Umwelt zeigt uns die Aktionen und Interaktionen der Roboter und deren Auswirkungen. Diese Informationen können mit der roboterinternen Repräsentation zum selben Zeitpunkt in Zusammenhang gebracht und verglichen werden.

Ziel dieser Arbeit ist es, eine Software zu entwickeln, welche die Erkennung und Lokalisierung von Fehlern erleichtert, indem sie Informationen über Roboter und Umwelt sammelt und diese synchronisiert zur Analyse bereitstellt. Die Sicht auf die reale Umwelt soll dabei durch Kamerabilder realisiert werden, die möglichst das gesamte Geschehen erfassen. Kamerabilder und intrinsische Daten der Roboter sollen schließlich in einer gemeinsamen Darstellung vereint werden. Die Software soll das Softwareframework *RoboFrame* erweitern, welches zur Steuerung mobiler autonomer Robotersysteme eingesetzt wird.

## 1.1 Szenario RoboCup

*RoboCup*[1] ist eine internationale Forschungsinitiative. Ziel ist die Förderung von Robotik und künstlicher Intelligenz durch die Stellung von Standardaufgaben und der Konkurrenz im internationalen Wettbewerb bei deren Lösungen[8]. Eine dieser Standardaufgaben ist das Problem fußballspielender Roboter. Der *Robot World Cup* findet seit 1997 jährlich statt, zusätzlich zu den offiziellen Weltmeisterschaften gibt es noch verschiedene weitere internationale Wettbewerbe, wie die *RoboCup German Open,* welche zuletzt im April 2009 in Hannover stattfanden.

Beim *RoboCup Soccer* kommen in verschiedenen Ligen Teams mobiler autonomer Roboter zum Einsatz. Die Aufgabe dieser Teams ist jedoch dieselbe - in einem Fußballspiel gegen die gegneri-

[1] Offizielle Homepage des *RoboCup*: `http://www.robocup.org/`

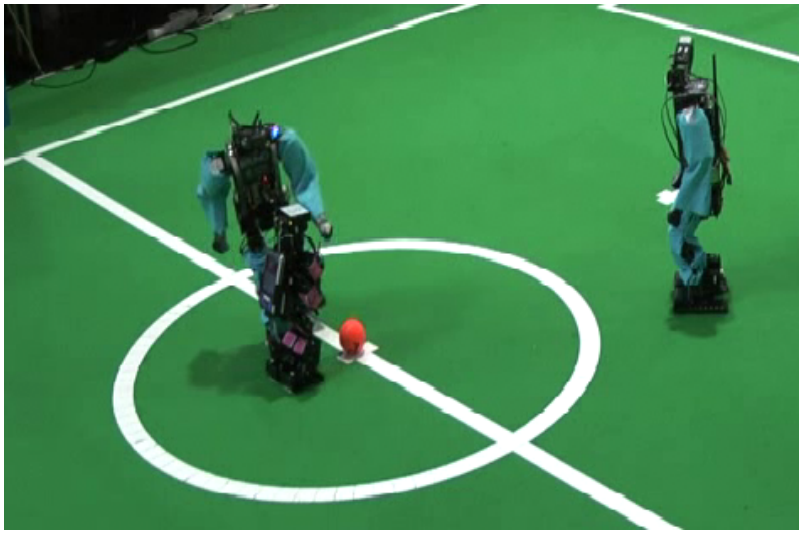

Abbildung 1.1: *Bruno* (links) setzt zum Schuss an.

schen Teams anzutreten, sich auf dem Spielfeld zu orientieren und den Ball in das Tor des Gegners zu befördern.

In der *Humanoid League* besteht ein Team aus drei, in der *Mid Size League* sogar aus fünf Robotern[1, 2]. Die einzigen erlaubten menschlichen Eingriffe sind dabei das Herausnehmen oder Einsetzen der Roboter aus dem bzw. in das Feld, sowie das Senden von Kontroll-Signalen bei Spielstart und bestimmten anderen Spiel-Ereignissen über WLAN. Der Erfolg eines Teams ist hier zu einem großen Teil davon abhängig, wie gut die Roboter miteinander kooperieren oder ob sie sich gegenseitig behindern. Zur Analyse ist es daher wichtig, alle Roboter darin einzubeziehen.

Während eines Spiels ist eine Analyse allerdings sehr schwierig, meist kann nur spekuliert werden, warum ein bestimmtes Fehlverhalten aufgetreten ist. Zeichnet man zur nachträglichen Analyse nur die intrinsischen Daten der Roboter auf, hat man keinen Bezug auf das Spielgeschehen. Verfügt man andererseits nur über ein Video der Begegnung, kann man trotz der Möglichkeit, das Spielgeschehen im Detail zu betrachten, nur Fehler mit offensichtlichen Ursachen identifizieren. Erst in Kombination wird eine sinnvolle Auswertung ermöglicht. Die Abbildungen 1.1 und 1.2 zeigen beispielhaft, dass externes und internes Weltbild nicht immer übereinstimmen. Zwar stimmt die Position des Balls relativ zum Roboter *Bruno*, er ist jedoch nicht zum Tor hin ausgerichtet, wie er annimmt.

Das Team der *Darmstadt Dribblers*[2] tritt mit seinen humanoiden Robotern beim *RoboCup Soccer* in der *Humanoid League* an[5]. Die *Darmstadt Dribblers* verwenden *RoboFrame*[15] für die Steuerungssoftware der Roboter, hier kam auch die zu entwickelnde Software schon während der Entwicklungsphase zum Einsatz.

Viele der Algorithmen und Verfahren, die im *RoboCup* Anwendung finden, können auch in andere Anwendungsbereiche der Robotik übertragen werden. Bei den Wettbewerben findet ein reger Austausch an Erfahrungen und Ergebnissen statt und der Wettbewerb der Teams untereinander sorgt dafür, dass die Entwicklung stetig voranschreitet.

[2] Offizielle Homepage der *Darmstadt Dribblers*: `http://www.dribblers.de`

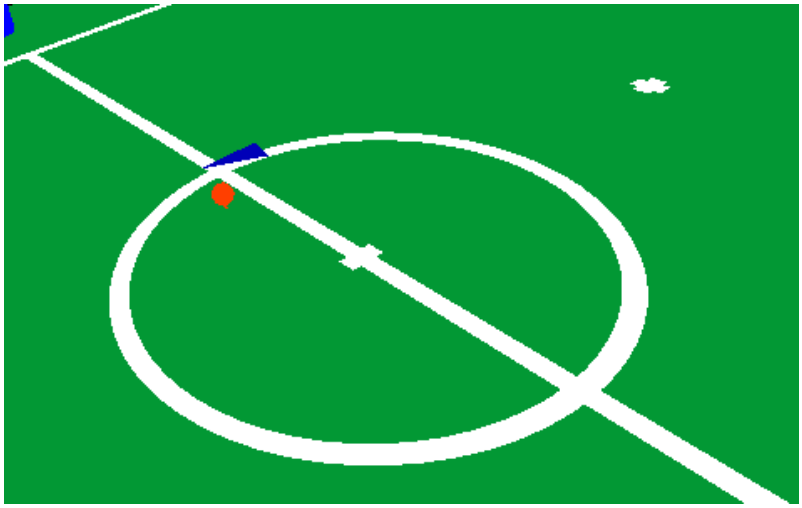

Abbildung 1.2: Der blaue Pfeil repräsentiert *Brunos* gedachte Position und Orientierung.

## 1.2 Aufbau der Arbeit

In Kapitel 2 wird auf themenverwandte Arbeiten eingegangen, dabei werden Unterschiede aufgezeigt und Vor- und Nachteile abgewägt. Kapitel 3 beschreibt die Anforderungen an diese Arbeit.

Die Kapitel 4 und 5 beschreiben das erarbeitete Konzept und dessen Realisierung. In Kapitel 6 wird auf die Ergebnisse eingegangen, bevor schließlich in Kapitel 7 eine kurze Zusammenfassung und ein Ausblick folgen.

Anhang A ist eine Anleitung zur Verwendung und Erweiterung der Software für Benutzer und Entwickler.

# 2 Grundlagen

Bei Robotersystemen, die auf relativ einfacher Modellierung und Verhalten basieren, können die Ursachen eines Fehlverhaltens oft noch aus reiner Beobachtung identifiziert werden. In solchen Fällen ist der Aufwand, ein System zur videogestützten Analyse zu entwickeln, weit größer als der Nutzen der sich daraus ergibt. Bevor eine Verhaltenssteuerung implementiert werden kann sind die grundsätzlicheren Probleme der Sensordatenverarbeitung und Bewegungserzeugung zu lösen.

Der Gewinn an zusätzlichen Erkenntnissen, der durch die Kombination von externen und intrinsischen Daten zur Analyse eines Robotersystems entstehen kann, hängt von den Eigenschaften des Systems ab. Es gibt verschiedene Faktoren, die diese Kombination besonders sinnvoll machen:

**Autonome Roboter** entscheiden selbst über ihre Handlungen. Diese Entscheidungen basieren auf den Sensordaten und der internen Modellierung. Zwar ist das Verhalten meist auf bestimmte Verhaltensweisen und Handlungen begrenzt, die Verhaltens-Entscheidungen sind an sich jedoch nicht vorhersagbar.
Die intrinsischen Daten können eventuell nicht an die Außenwelt kommuniziert werden, für eine Auswertung müssen sie in diesem Fall auf dem Roboter aufgezeichnet werden.

**Mobile Roboter** sind auf die Ressourcen beschränkt, die sie selbst befördern können. Das betrifft unter anderem Rechenkapazität, Speicherplatz und die Energieversorgung. Dadurch sind die Sensordaten begrenzt, die verarbeitet werden können, sowie die Daten, die auf dem Roboter aufgezeichnet werden können. Es stehen also mit großer Wahrscheinlichkeit nicht die vollständigen intrinsischen Daten zur Verfügung, es kann nur ein Teil zur Analyse herangezogen werden.
Da sich der Roboter in seiner Umwelt bewegt, ändert sich seine Wahrnehmung der Umwelt auch in einer statischen Umgebung ständig. Das Verhalten ist dann nicht nur durch die Umgebung selbst bedingt, sondern auch durch die Position und Bewegung des Roboters in dieser Umgebung.

**Komplexes Verhalten** erlaubt es nicht, das Verhalten und die entsprechenden Entscheidungen nur durch visuelle Beobachtung des Roboters nachzuvollziehen. Es gibt eine hohe Anzahl an unterschiedlichen Verhaltensweisen. Handlungen können mehreren dieser Verhaltensweisen zugeordnet sein, durch das Beobachten einer bestimmten Handlung lässt sich die aktuelle Verhaltensweise also nur in manchen Fällen eindeutig identifizieren. Die Komplexität wird weiter erhöht, wenn unterschiedliche Verhaltensweisen gleichzeitig ausgeführt werden.

**Eine unbekannte oder dynamische Umwelt** führt dazu, dass die Plausibilität der internen Modellierung nicht ohne eine externe Sicht bestimmt werden kann. Denn diese ist nötig, um den Zustand der Umwelt zu dem entsprechenden Zeitpunkt zu bestimmen. Zusätzlich ist das Verhalten in einer dynamischen Umgebung noch weniger vorhersagbar, da die äußeren Bedingungen - auch beim Ausführen identischer Tests - jedes Mal unterschiedlich ausfallen können. Zusätzlich spielt noch die Geschwindigkeit von Veränderungen eine Rolle - folgen in kurzer Zeit viele Aktionen lassen sich diese viel leichter nachvollziehen wenn man die Geschehnisse in Zeitlupe betrachten oder pausieren kann.

Beim *RoboCup Soccer* kommen bereits sehr komplexe Verhalten zum Einsatz. Es gilt nicht nur die unterschiedlichen Spielsituationen zu meistern, sondern auch das Regelwerk und die Schiedsrichteranweisungen zu befolgen. Das Verhalten der *Darmstadt Dribblers* von 2008 bestand aus mehr als 60 verschiedenen Modulen, bei manchen Teams sind es sogar mehr als 100[14]. Im Vergleich dazu hatte das Team *Victor-Tango* von *Virginia Tech*, welches 2007 mit seinem autonomen Fahrzeug bei der *DARPA*[1] *Urban Challenge* den dritten Platz belegte, nur neun verschiedene Verhaltensmodule zur Steuerung des Fahrzeugs[7].

Der *LogViewer* des *GermanTeam* ist eine Analyse-Software basierend auf intrinsischen Daten und externen Videodaten, die im Umfeld des *RoboCup Soccer* zum Einsatz kommt. Ein weiterer Bereich, in dem dieser Ansatz zur Anwendung kommt, ist die Analyse der Interaktion zwischen Menschen und Robotern (Human Robot Interaction).

## 2.1 GermanTeam LogViewer

Bis 2008 kamen beim *RobCup Soccer* in der *Standard Platform League,* der früheren *Four-Legged Robot League,* die *Sony Aibo*-Roboter zum Einsatz. Das *GermanTeam* nahm seit 2001 aktiv an der Liga teil und konnte 2004, 2005 und 2008 den Weltmeister-Titel erringen[4]. Es besteht aus Mitarbeitern und Studenten der *Humboldt-Universität zu Berlin*, der *Universität Bremen* und der *Technischen Universität Darmstadt.*

Für die Verhaltenssteuerung der Roboter verwendet das Team die *Extensible Agent Behavior Specification Language* (XABSL)[10]. Die Ausführung des Verhaltens basiert auf einem hierarchischen endlichen Zustandsautomaten. Die Steuerungssoftware des *GermanTeam* erlaubt es, die Verhaltensdaten zur späteren Analyse aufzuzeichnen. Dazu gehört nicht nur der jeweils aktuelle Zustand des Verhaltens, sondern auch verschiedene Verhaltens-Symbole, die die Informationen wiederspiegeln, die der Roboter über sich, sein Team und seine Umwelt gesammelt hat.

Da die Ressourcen auf dem *Aibo*-Roboter stark begrenzt sind, sowohl an Speicherplatz als auch an Rechenzeit, ist eine effiziente Speicherung besonders wichtig. Deshalb werden soweit möglich nur Änderungen des Zustands und der Symbole in der entsprechenden Log-Datei abgelegt. Auch mit dieser Optimierung ist es aufgrund der stark begrenzten Speicherkapazität auf dem *Aibo* meist nicht möglich, die Daten einer kompletten Halbzeit aufzuzeichnen.

Mit dem *LogViewer* ist eine Analyse der aufgezeichneten Daten möglich[14]. Zusätzlich zu der Log-Datei eines Roboters kann auch eine Video-Datei geladen werden. Das Video muss manuell über bestimmte Ereignisse, die man sowohl dem Video als auch den Verhaltensdaten entnehmen kann, synchronisiert werden. Um diese Synchronisierung schnell und hinreichend genau vorzunehmen, ist eine gute Kenntnis des Roboterverhaltens nötig. Nach erfolgreicher Synchronisierung können Video und Verhaltensdaten gemeinsam abgespielt werden. Die Navigation durch die Daten kann über einen Zeitstrahl geschehen, an dem die Aktivierung und die Dauer eines bestimmten Verhaltenszustands angezeigt werden kann. Zusätzlich zur Anzeige von Video und Verhaltensdaten wird noch eine graphische Visualisierung bestimmter Verhaltens-Symbole aufbereitet (siehe Abbildung 2.1). Durch Einbeziehen der Team-Kommunikation in die Verhaltens-Symbole kann - trotz der Auswertung der Daten von nur einem Roboter - ein guter Überblick über das gesamte Spielgeschehen gewonnen werden.

[1] *Defense Advanced Research Projects Agency*, die Forschungsorganisation des Verteidigungsministeriums der USA

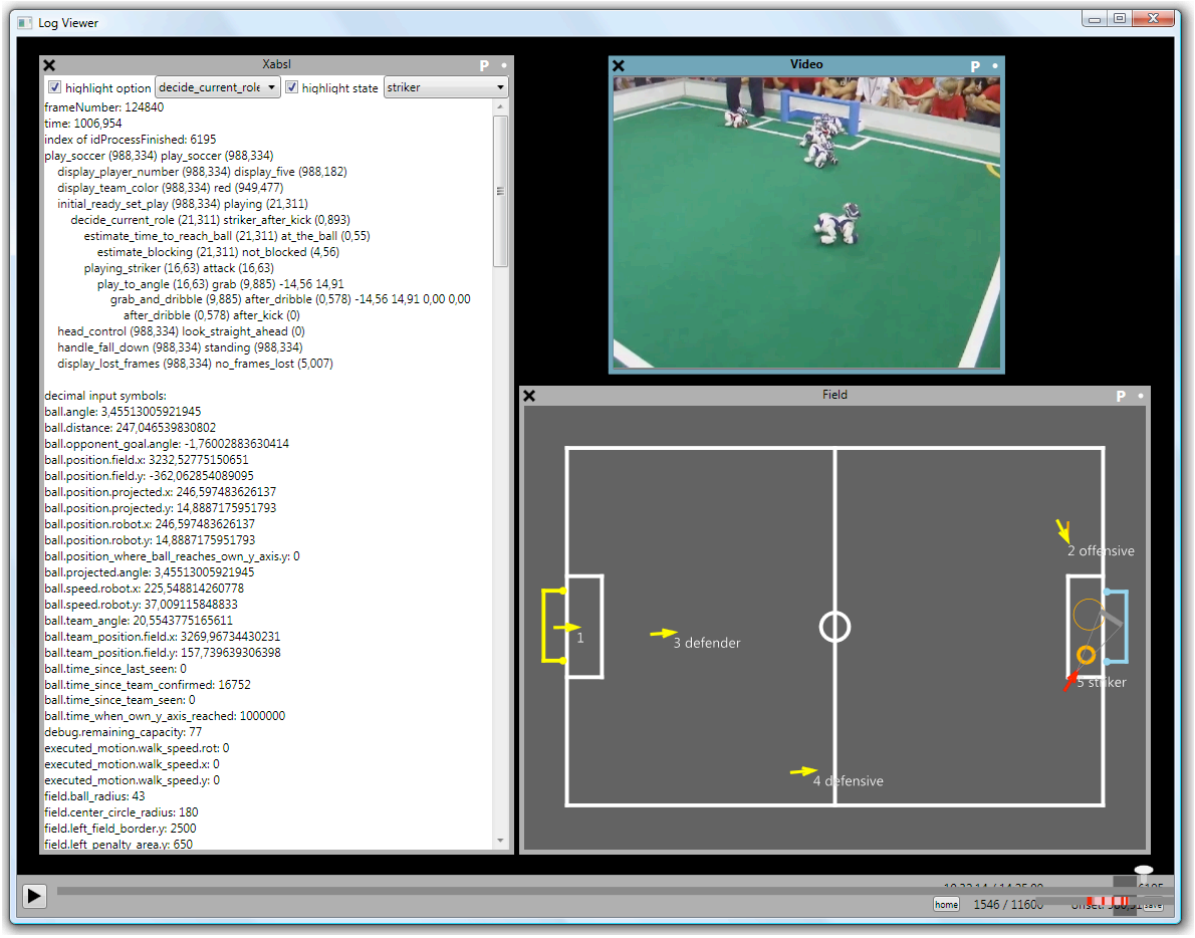


Abbildung 2.1: Der *LogViewer* zeigt Verhaltensdaten (links), Video (rechts oben) und graphische Visualisierung bestimmter Verhaltens-Symbole (rechts unten)[14]

## 2.2 Human Robot Interaction

Um mit einem Menschen zu interagieren muss ein Roboter die Fähigkeit haben, Menschen und deren Aktionen zu erkennen und zu deuten. Da das Verhalten von Menschen sehr komplex ist, ergibt sich für einen Roboter, der entsprechend auf die Aktionen eines Menschen reagieren soll, ein komplexes Verhalten auf Basis einer komplexen Modellierung. Außerdem ist eine Umgebung, in der sich Menschen aufhalten und bewegen, auch eine dynamische Umgebung.

Zur Verbesserung der Interaktion wird das Verhalten - sowohl das der Menschen als auch das der Roboter - analysiert. Der herkömmliche Ansatz ist dabei die Aufzeichnung und spätere Analyse von Video- oder Audiodaten. Aufgrund der beschriebenen Faktoren bietet es sich jedoch an, diese Informationen zur Analyse mit der internen Darstellung des Roboters zu kombinieren.

### 2.2.1 Interaction Debugger

Der *Interaction Debugger* ist eine Software zur Analyse der Interaktion zwischen Menschen und Robotern[9]. Daten von unterschiedlichen Quellen werden aufgezeichnet und zur späteren Analyse zusammengefasst. Der *Interaction Debugger* ermöglicht ein gemeinsames Abspielen und Visualisieren dieser Daten. Er wird eingesetzt um das Verhalten des Roboters *Robovie*[6] im Umgang mit Menschen zu verbessern.

#### Aufzeichnen der Daten

Zu den Daten, die aufgezeichnet werden, gehören intrinsische Daten des Roboters, insbesondere Sensordaten und der Verhaltenszustand. Weiterhin werden durch verschiedene speziell ausgestatte Computer, die in der Umgebung platziert sind, externe Sensordaten aufgenommen. Diese Computer verfügen über Kameras, Mikrophone oder weiteren Sensoren wie Entfernungs- und Lautstärkesensoren.

Die aufgezeichneten Daten werden mit einem Zeitstempel versehen. Danach werden Sie an einen zentralen Server geschickt, der die Daten aus allen Quellen gemeinsam verwaltet (siehe Abbildung 2.2). Alternativ können diese Daten auch direkt im *Interaction Debugger* visualisiert werden (mit Ausnahme der Audio- und Videodaten).

Um die zeitlich korrekte Abfolge der Daten aus den verschiedenen Quellen zu gewährleisten, wird das *Network Time Protocol* (NTP) verwendet. Roboter und Computer befinden sich in einem gemeinsamen lokalen Netzwerk und die Systemzeit wird untereinander synchronisiert.

Audio- und Videodaten werden in Abschnitten von einer Minute Länge als Dateien auf dem Server abgespeichert. Die restlichen Daten werden in einer Datenbank abgelegt.

#### Visualisierung und Analyse

Zur Analyse der Daten muss im *Interaction Debugger* das Zeitintervall gewählt werden, aus dem Daten bereitgestellt werden sollen. Für die verschiedenen Datentypen gibt es verschiedene Visualisierungsfenster. Der Benutzer wählt die für seine Analyse relevanten Anzeigen aus (siehe Abbildung 2.3).

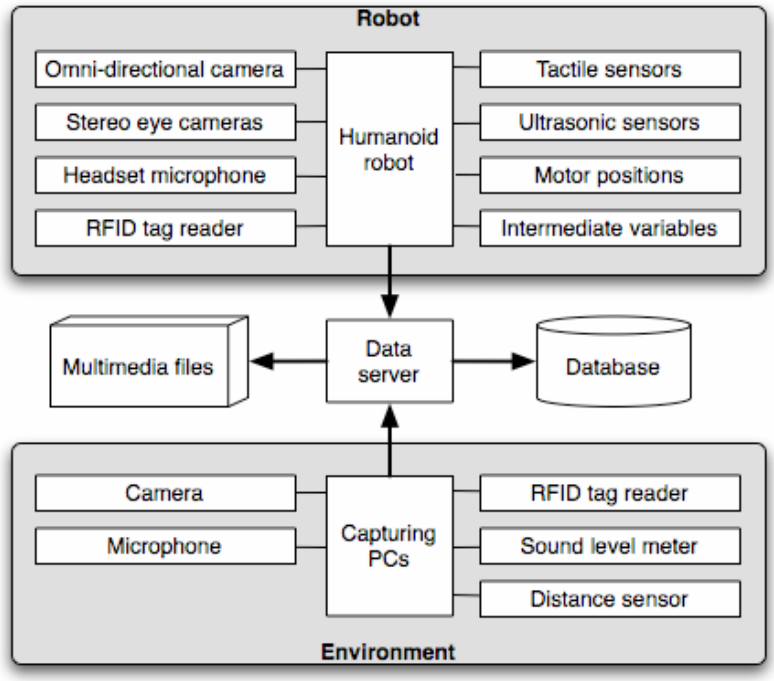


Abbildung 2.2: Infrastruktur zum Aufzeichnen der Daten für den *Interaction Debugger*[9]

Das Abspielen und die Navigation durch die Daten erfolgt anhand eines Zeitstrahls innerhalb des gewählten Intervalls. Jeder Zeitpunkt kann mit einer Anmerkung versehen werden. Diese Anmerkungen werden gespeichert und können dazu verwendet werden, später wieder an denselben Zeitpunkt zu springen. Eine weitere Möglichkeit zur Navigation innerhalb der Daten ist die Auswahl eines bestimmten Verhaltenszustands. Zu diesem Verhaltenszustand werden alle Intervalle angezeigt, in denen dieser aktiv war.

## 2.3 Zusammenfassung

Mit dem *LogViewer* und dem *Interaction Debugger* sind zwei Beispiele für Systeme gegeben, die intrinsische und externe Daten kombinieren, um das Verhalten von Robotern zu analysieren.

Der *LogViewer* ermöglicht eine Analyse der kompletten XABSL-Verhaltensdaten in Kombination mit einem Video. Durch die Definition von Verhaltens-Symbolen kann der interne Zustand des Roboters aufgezeichnet und visualisiert werden. Leider ist jeweils nur die Analyse von Daten von einem Roboter möglich. Daten von anderen Robotern stehen nur begrenzt durch die Kommunikation der Roboter untereinander zur Verfügung.

Für die Verarbeitung von anderen Daten als den Verhaltensinformationen ist der *LogViewer* jedoch nicht konzipiert. Die Visualisierung ist auf bestimmte Symbole, wie sie im Verhalten des *GermanTeam* auftreten, spezialisiert. Dies macht eine Übertragung auf andere Anwendungen schwierig, besonders wenn dort kein XABSL zur Verhaltenssteuerung eingesetzt wird.

Im Gegensatz dazu erlaubt der *Interaction Debugger* die Erweiterung um neue Datentypen und Visualisierungen. Der modulare Aufbau soll die Anwendung auf andere Roboter und Architekturen vereinfachen.

Ein Nachteil des *Interaction Debuggers* ist, dass zum Aufzeichnen der Daten eine umfangreiche Infrastruktur benötigt wird. Die Daten müssen auf dem zentralen Server gesammelt werden, der

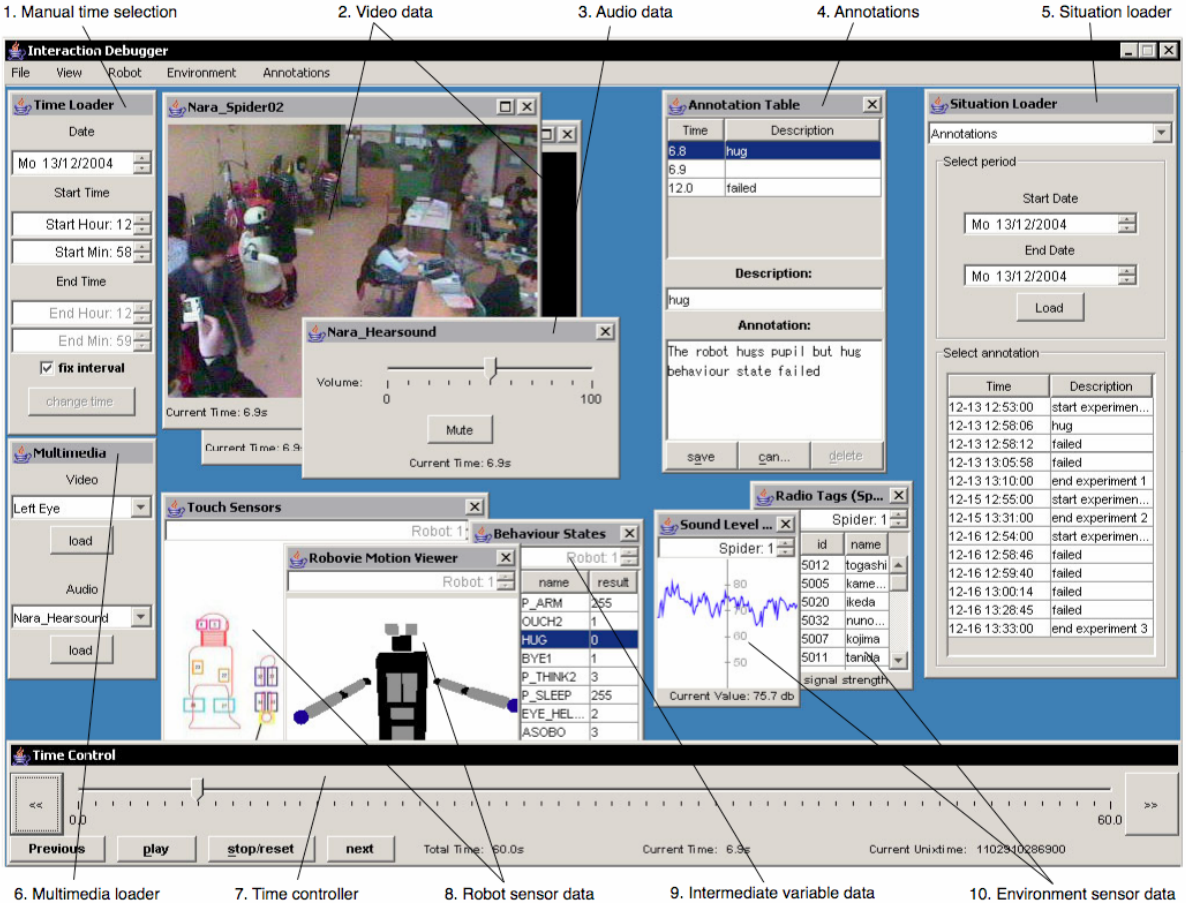


Abbildung 2.3: Analyse eines Datenabschnitts im *Interaction Debugger*[9]

Roboter und speziell ausgestattete Computer liefern diese dort ab. Video- und Audiodaten werden in kurzen Abschnitten gespeichert und können so ohne weitere Verarbeitung nicht anderweitig verwendet werden. Für den *LogViewer* des *GermanTeam* reicht es aus, für die Aufnahme der Videos auf ein handelsübliches Gerät zurückzugreifen, wie zum Beispiel eine digitale Videokamera oder sogar ein Mobiltelefon mit Videofunktion. Den Nachteil, den dies wiederum mit sich bringt, ist die Notwendigkeit einer manuellen Synchronisation.

Zur Navigation innerhalb der Daten erlauben beide Ansätze die Orientierung an bestimmten Verhaltenszuständen. Im *Interaction Debugger* ist es zusätzlich möglich, jeden Zeitpunkt manuell mit einer eigenen Anmerkung zu versehen.

Für die im Rahmen dieser Arbeit zu entwickelnde Software sollen Vorteile der beiden Ansätze kombiniert und mögliche weitere Verbesserungen implementiert werden. Es soll möglich sein, die Software auf verschiedenen Roboter-Plattformen und für unterschiedliche Anwendungsszenarien einzusetzen, sowie sie entsprechend der jeweiligen Anwendung zu erweitern. Um dies zu erreichen wird auf dem Softwareframework *RoboFrame* aufgebaut, welches es ermöglicht, plattformunabhängige Roboter-Anwendungen zu entwickeln. *RoboFrame* kommt unter anderem bei Teams in *RoboCup Soccer* und *RoboCup Rescue*[2] , sowie für die Steuerung eines bionischen Roboterarms[3] zum Einsatz[5, 3]. Ein wichtiger Aspekt ist außerdem die Unterstützung für die Analyse mehrerer Roboter. Für das Aufzeichnen intrinsischer und externer Daten soll möglichst wenig Aufwand betrieben werden müssen, optimalerweise sollen dafür bereits die Roboter selbst und eine Videokamera ausreichen. Die Navigation innerhalb der Daten soll anhand von beliebigen Datentypen geschehen können, dazu kann die Annotation der Daten automatisiert werden. Schließlich soll eine Fusion von intrinsischen und externen Daten in einem Video geschehen, welches eine weitere Analyse oder Präsentation ohne die Software möglich macht.

[2] `http://www.robocuprescue.org/`
[3] `http://www.biorob.de/`

# 3 Anforderungen

Das Ziel dieser Arbeit ist es, eine umfangreiche visuelle Analyse von komplexen Roboter-Systemen zu ermöglichen. Dazu soll ein Software-Tool entwickelt werden, das den Benutzer in dieser Aufgabe unterstützt. Ein wichtiger Aspekt dabei ist, dass die Software für den Einsatz auf verschiedenen Roboter-Architekturen und für eine Vielzahl von möglichen Anwendungen geeignet ist. Zu diesem Zweck soll das bereits erwähnte Softwareframework *RoboFrame* erweitert werden, welches auf die Entwicklung von Steuerungssoftware für unterschiedlichste Roboter-Architekturen ausgelegt ist.

## 3.1 Integration in RoboFrame

Durch die Erweiterung des Softwareframeworks *RoboFrame* wird die zu entwickelnde Software allen Anwendungen auf Basis dieses Frameworks nutzbar gemacht. Das bedeutet aber auch, dass die Software an die Architektur des Frameworks gebunden ist und bestimmte Richtlinien erfüllen muss. Wie sich die Software genau in das Framework einfügt, wird in den jeweiligen Abschnitten in Kapitel 5 erläutert.

Anwendungen auf Basis von *RoboFrame* bestehen grundsätzlich aus zwei Teilen - einer Applikation (*RoboApp*), die auf den Robotern installiert ist und diese steuert, sowie eine Applikation mit grafischer Benutzeroberfläche (*RoboGui*), die mit den Applikationen der Roboter kommunizieren kann. Die grafische Benutzeroberfläche (GUI) dient der Anzeige der intrinsischen Daten der Roboter, aber auch dem Senden von Befehlen oder Einstellungen. Wegen der hohen Diversität an Roboter-Architekturen ist die Unterstützung möglichst vieler verschiedener Plattformen von hoher Bedeutung. Abbildung 3.1 zeigt die Plattformabstraktion in *RoboFrame*. Die GUI verwendet die *Qt*-Bibliothek, welche unter anderem die gängigen Betriebssysteme Windows, Linux und Mac OS X unterstützt. Die zu entwickelnde Software sollte möglichst viele dieser Plattformen unterstützen, um die Nutzung so wenig wie möglich weiter einzuschränken.

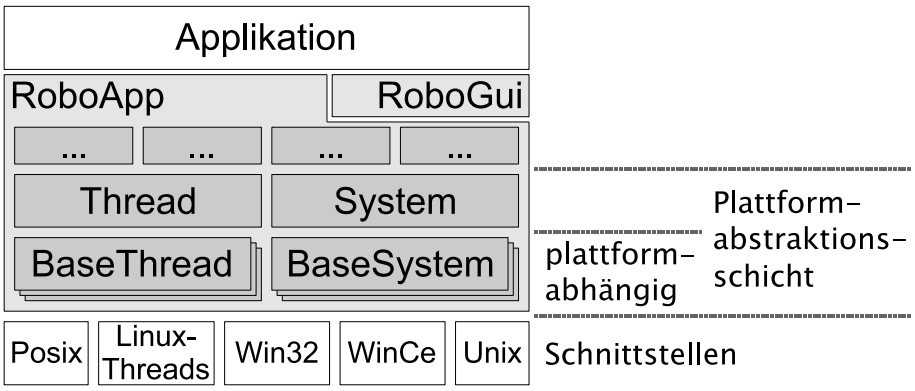


Abbildung 3.1: Plattformabstraktionsschicht in *RoboFrame*[15]

## 3.2 Roboter-Architekturen

Prinzipiell sollte der Einsatz der Software auf beliebigen Architekturen möglich sein. Allerdings ergeben sich auch Anforderungen aus den Architekturen, auf denen das System primär eingesetzt werden soll. Die Entscheidung, auf dem *RoboFrame*-Framework aufzubauen lässt zwar viele Möglichkeiten offen, ist aber trotzdem eine Einschränkung. Bei der Definition der Anforderungen ist es wichtig, auf spezielle Anforderungen einzugehen ohne das Anwendungsgebiet zu weit einzuschränken.

### 3.2.1 Darmstadt Dribblers

Haupteinsatzgebiet soll die Roboter-Architektur der *Darmstadt Dribblers* sein. Dort werden zwei verschiedene Typen von humanoiden Robotern eingesetzt, die auf den Modellen *HR18* und *HR30* von *Hajime Research Institute Ltd.*[1] basieren[5].

Beide Modelle verfügen über 21 Freiheitsgrade und sind 55 bzw. 57,5 cm hoch. Der *HR30* ist im Vergleich mit stärkeren Servo-Motoren und einem leistungsfähigeren Mikrocontroller zur Steuerung der Bewegungen ausgestattet. Die Sensorik der Roboter bilden eine Kamera im Kopf, sowie ein Gyroskop und ein Beschleunigungssensor in der Hüfte.

Die Steuerungssoftware läuft unter einem Linux-Betriebssystem auf einem PC/104-Board von *DigitalLogic* mit einem *AMD Geode LX800* Prozessor mit 500 MHz. Eine *CompactFlash*-Karte dient als Datenspeicher.

Die Kamera liefert etwa 30 Bilder pro Sekunde in einer Auflösung von 640x480 Pixeln. Die Bildverarbeitung ist der rechenaufwendigste Teil der Roboterapplikation. Deshalb können zur Zeit nur nur etwa die Hälfte der Bilder überhaupt verarbeitet werden. Die Bilder werden farblich segmentiert und durchlaufen verschiedene Algorithmen zur Objekterkennung, unter anderem für Tore, Feldlinien, Ball und Hindernisse. Die erkannten Objekte gehen in die Weltmodellierung ein, dabei ist besonders die Bestimmung der eigenen Position und der des Balls wichtig.

Ein weiterer wichtiger Aspekt ist die Teamkommunikation. Jeder Roboter bindet die Informationen aus den Nachrichten seiner Teamkollegen in seine Modellierung mit ein. Die Teamkommunikation wird außerdem eingesetzt, um die Rollen der verschiedenen Spieler untereinander abzustimmen.

Für die Steuerung des Verhaltens der Roboter wird die *Extensible Agent Behavior Specification Language* (XABSL)[10] eingesetzt. Das komplexe Verhalten umfasst die verschiedenen möglichen Rollen der Roboter in einem Fußballspiel, wie zum Beispiel die Rolle des Torwarts oder Stürmers. Außerdem wird dort dafür gesorgt, dass der Roboter nach einem Sturz wieder aufsteht oder erkannten Hindernissen ausweicht.

Abbildung 3.2 zeigt einen Überblick über die Struktur und den Informationsfluss in der Steuerungssoftware.

Dadurch dass bei einem Spiel in der *Humanoid League* pro Team drei Roboter zum Einsatz kommen hat es einen sehr dynamischen Charakter. Das komplexe Verhalten und die Vielzahl an möglichen Fehlerquellen tragen ebenfalls dazu bei, dass hier der Einsatz einer videogestützten Analyse sehr erfolgversprechend ist. In der grafischen Benutzeroberfläche der Software der Darmstadt Dribblers sind bereits viele Werkzeuge zur Analyse vorhanden. Hauptsächlich handelt es sich

[1] *http://www.hajimerobot.co.jp*

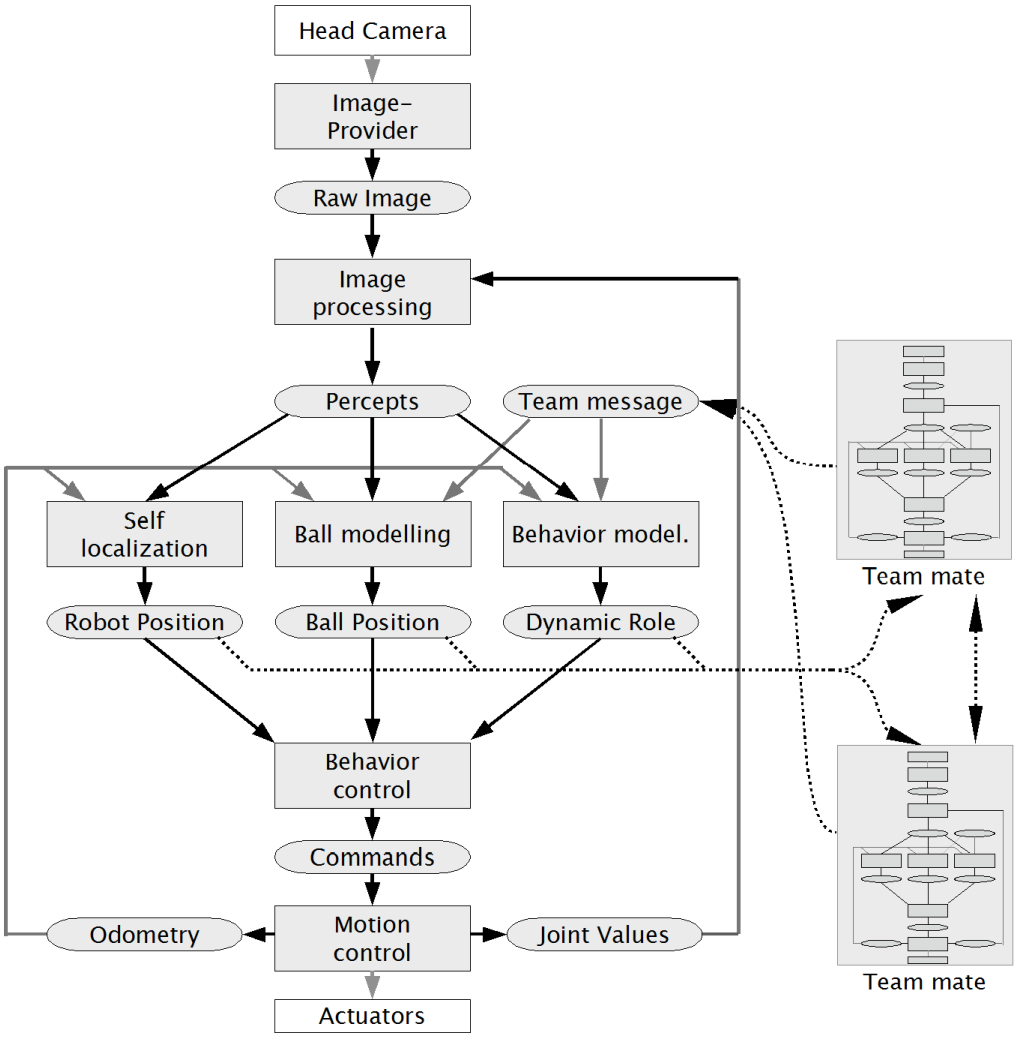


Abbildung 3.2: Struktur der Steuerungssoftware der *Darmstadt Dribblers*[12]

Abbildung 3.3: Roboter *Monstertruck* des Darmstadt Rescue Robot Team[3]

dabei um Visualisierungen intrinsischer Daten. Da in diesen Visualisierungen bereits sehr viel Zeit und Arbeit steckt, ist eine wichtige Anforderung die Möglichkeit zur Wiederverwendung.

### 3.2.2 Darmstadt Rescue Robot Team

Eine weitere potentielle Einsatzmöglichkeit ist die Roboter-Architektur des *Darmstadt Rescue Robot Team*. Der Gedanke hinter der *Rescue Robot League* des *RoboCup* ist die Förderung der Forschung und Entwicklung auf dem Gebiet des Einsatzes von Robotern im Katastrophenmanagment. Die Aufgaben umfassen unter anderem das Erkennen von simulierten Opfern, die Navigation in einer unbekannten Umgebung und die Fortbewegung auf unterschiedlichen Arten von Gelände.

Der vom *Darmstadt Rescue Robot Team* eingesetzte Roboter mit dem Namen *Monstertruck* basiert auf dem ferngesteuerten Modellauto *Kyosho Twin Force*. Dieses wurde um einen Onboard-Computer auf Basis eines PC/104-Boards erweitert, sowie um einen Laserscanner zur autonomen Navigation und Kartografierung. Zur Erkennung von Opfern dient die *Vision Box*. Sie besteht aus einer visuellen und einer Infrarot-Kamera, sowie einem weiteren integrierten PC zur Verarbeitung der Bilddaten und Erkennung von Objekten[3]. Abbildung 3.3 zeigt den *Monstertruck* mit montierter *Vision Box*.

Die Steuerungssoftware basiert auch bei dieser Architektur auf *RoboFrame*. Es werden unter anderem auch Komponenten, die bei den *Darmstadt Dribblers* entwickelt wurden, wiederverwendet.

## 3.3 Funktionale Anforderungen

Die intrinsischen Daten mehrerer Roboter müssen aufgezeichnet und synchronisiert in der grafischen Benutzeroberfläche wieder abgespielt werden können. Dabei muss es möglich sein, die einzelnen Roboter im Nachhinein zu identifizieren und die entsprechenden Daten zuzuordnen. Beim Abspielen der Daten sollen diese entsprechend visualisiert werden können. Da zu den verschiedenen Daten oft bereits Visualisierungen vorhanden sind, bietet es sich an, hierfür nach einem Weg zu suchen, diese wiederzuverwenden.

Der Abspielmechanismus soll es erlauben, die Daten schneller oder langsamer abzuspielen, sowie das Abspielen zu pausieren oder an eine bestimmte Position im Aufnahmezeitraum zu springen. So kann das Auftreten eines Fehlers oder einer Anomalie leicht genauer untersucht werden. Um auch bei großen Datenmengen eine einfache Navigation zu ermöglichen, sollen gezielt bestimmte Ereignisse innerhalb der Daten betrachtet werden können.

Die Software soll die Möglichkeit bieten, beim Abspielen der intrinsischen Daten der Roboter zusätzlich synchronisiert die Videodaten anzuzeigen, die jeweils zeitgleich aufgenommen wurden. Dadurch erhält man die beiden gewünschten Perspektiven auf das Geschehen - die Sicht auf die reale Umwelt durch die Kamerabilder und die Umwelt aus Sicht des Roboters durch die intrinsischen Daten.

Beide Perspektiven sollen schließlich fusioniert werden. Durch gezielte Einblendung intrinsischer Daten in das Videobild werden die wesentlichen Informationen dort zusammengefasst. Der anschließende Export in eine Videodatei macht eine Analyse oder Präsentation auch mit einer Software nur zum Abspielen von Videos möglich.

# 4 Konzept

Das Konzept beschreibt, wie die in Kapitel 3 beschriebenen Anforderungen umgesetzt werden. In Kapitel 5 wird später konkret auf die Implementierungsdetails der einzelnen Aspekte der Software eingegangen.

## 4.1 Aufbereitung der intrinsischen Daten

Die intrinsischen Daten der Roboter müssen für die Analyse aufbereitet werden. Dazu gehören als erste Schritte das Aufzeichnen der Daten und deren Synchronisierung. Zur Analyse sollen die aufgezeichneten Daten dann visualisiert werden.

### 4.1.1 Aufzeichnen der Daten

Das Aufzeichnen der intrinsischen Daten der Roboter ist bereits über den *LogRecorder* in der GUI möglich. Dafür wird eine Netzwerkverbindung mit den jeweiligen Roboter-Applikationen benötigt. Spielt man die aufgenommenen Daten über den *LogRecorder* ab, gehen die Informationen über die Zugehörigkeit der Daten zu den einzelnen Robotern aber verloren - der *LogRecorder* ist aus Sicht der Anwendung die Quelle der Daten.

Ziel ist es, beim Aufzeichnen der Daten möglichst unabhängig von der Netzwerkinfrastruktur zu sein. Dazu soll ein dezentraler Ansatz verwendet werden, nämlich die Daten direkt auf den Robotern aufzuzeichnen. Dabei muss konfigurierbar sein, welche Daten aufgezeichnet werden, wie oft und wie lange. Es soll möglich sein, die Daten automatisch mit Start der Roboter-Applikation aufzeichnen zu lassen, aber auch während dem Betrieb die Aufzeichnung steuern zu können.

Um die aufgezeichneten Daten einem bestimmten Roboter zuordnen zu können, soll automatisch ein Identifikator mit aufgezeichnet werden. Der Identifikator soll den Namen der Applikation, sowie eine eindeutige, der Applikation zugeordnete Identifikationsnummer (ID) beinhalten.

Analog zum Aufzeichnen mit dem *LogRecorder* sollen Metainformationen wie der lokale Zeitstempel oder die Herkunft der Daten innerhalb der Applikation mitgespeichert werden. Das verwendete Dateiformat soll kompatibel zu dem im *LogRecorder* verwendeten Format sein um auch eine Kombination von Aufnahmen aus GUI und Roboter zu ermöglichen.

### 4.1.2 Synchronisierung

Da die aufgezeichneten intrinsischen Daten einem lokalen Zeitstempel zugeordnet sind, muss dieser bei der Betrachtung von Daten mehrerer Applikationen in eine gemeinsame Zeit konvertiert werden. Dazu wird die Zeitdifferenz der Systemzeiten der verschiedenen Applikationen benötigt.

Eine Zeitsynchronisierung, und damit auch die Bestimmung der Zeitdifferenz, ist zwischen Roboter-Applikation und GUI in *RoboFrame* bereits implementiert. Zusätzlich muss nun auch eine Bestimmung der Zeitdifferenz zwischen verschiedenen Applikationen möglich gemacht werden. Die

bestimmten Zeitdifferenzen müssen den jeweiligen Applikationen zugeordnet werden können und zum Abspielen der Daten verfügbar gemacht werden.

### 4.1.3 Visualisierung

Das übliche Szenario zur Verwendung der GUI in *RoboFrame* ist das Verbinden zu einer einzigen Roboter-Applikation und das Empfangen von Daten von bzw. Senden von Daten an diese Applikation. So gibt es im Fall der *Darmstadt Dribblers* mehr als 15 verschiedene Dialoge, die zur Visualisierung von intrinsischen Roboterdaten dienen.

Das Problem bei der Verwendung der vorhandenen Visualisierungen ist, dass sie fast ausschließlich für die Visualisierung von Daten von nur einem Roboter konzipiert sind. Ein Dialog in *RoboFrame* fordert Daten, die er verarbeiten kann, über den jeweiligen *Schlüssel* an. Dazu werden, zum Beispiel beim Öffnen des Dialogs, Anfragen verschickt, die die gewünschten Daten über den *Schlüssel* identifizieren.

Grundsätzlich besteht die Möglichkeit, nur Daten von bestimmten Quellen anzufordern. Allerdings verfügt ein Dialog nicht über die nötigen Informationen, um selbstständig entscheiden zu können, von welcher Quelle er Daten empfangen soll. Letztendlich weiß nur der Benutzer, welche Daten von welchem Roboter ihn interessieren. Außerdem kann ein Dialog nur Daten von einer Quelle anfordern, die es zu diesem Zeitpunkt auch gibt. Über neu hinzugekommene oder ungültig gewordenen Datenquellen wird ein Dialog nicht informiert.

Deshalb fordern die meisten Dialoge Daten von allen Datenquellen an, gehen dabei aber davon aus, dass sie nur Daten von einer Quelle erhalten. Ist mehr als ein Roboter mit der GUI verbunden, werden so natürlich auch Daten von mehreren Robotern angefordert und an den jeweiligen Dialog übermittelt. Dies führt zu einer Vermischung der Daten und man erhält keine sinnvolle Visualisierung.

Um nicht jeden einzelnen Dialog an die veränderten Anforderungen anpassen zu müssen, soll eine Zwischenschicht eingeführt werden, die den Dialogen die Verwaltung der verschiedenen Datenquellen abnimmt und dem Benutzer eine einfache Konfiguration erlaubt. So können Dialoge wie gewohnt Daten anfordern und müssen nicht angepasst werden. Die Zwischenschicht verwaltet die tatsächliche Anforderung von Daten und reagiert auf Ereignisse wie das Hinzufügen oder Entfernen von Datenquellen. Abbildung 4.1 zeigt die Visualisierung von intrinsischen Daten bei mehreren verbundenen Robotern mit und ohne filternde Zwischenschicht.

Daten, für die es keine dedizierten Dialoge zur Visualisierung gibt, sollen direkt als sog. Overlays (siehe Abschnitt 4.5) im Video oder als Ereignisse (siehe Abschnitt 4.4) visualisiert werden können.

## 4.2 Video

Für die Verarbeitung von Kamerabildern bieten sich drei Methoden an:

- die direkte Aufnahme von Kamerabildern über eine angeschlossene Kamera,
- das Verarbeiten von Videodateien, die mit einer anderen Software oder auf einem anderen Gerät aufgenommen wurden oder
- eine Kombination von beidem, indem aus der Aufnahme eine Videodatei erzeugt wird.

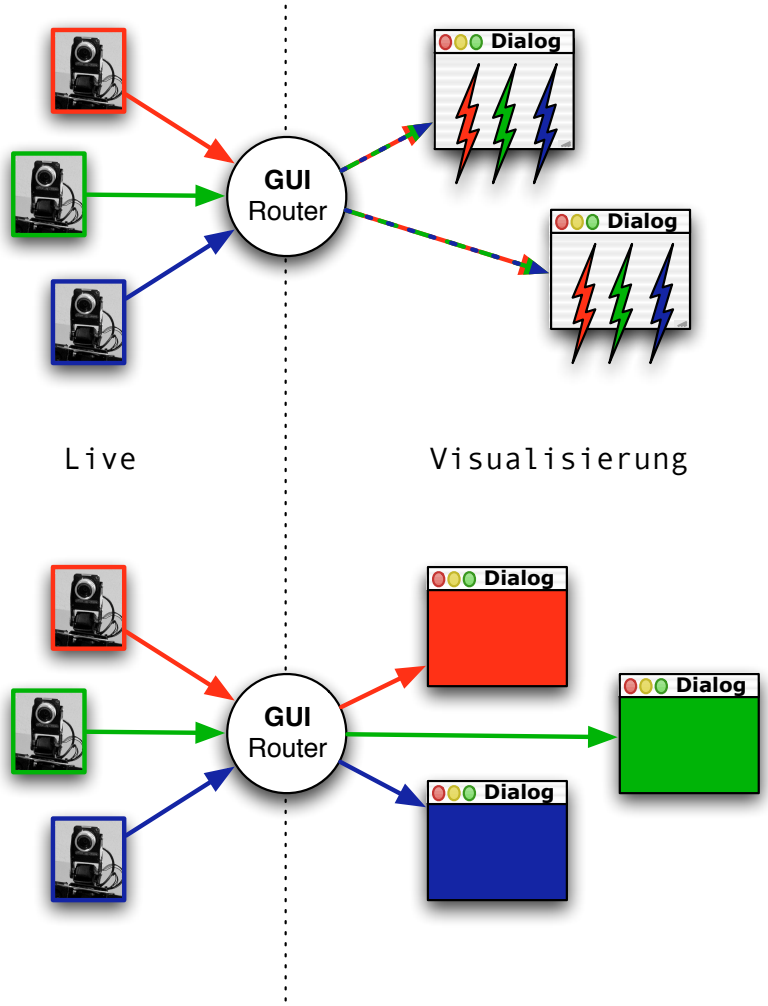


Abbildung 4.1: Visualisierung von intrinsischen Daten bei mehreren verbundenen Robotern ohne (oben) und mit filternder Zwischenschicht (unten).

Die Aufnahme von Kamerabildern direkt über die GUI brächte den Vorteil mit sich, dass eine Synchronisation mit den intrinsischen Daten der Roboter automatisch bei der Aufnahme geschehen könnte. Dazu würde lediglich eine kurze Verbindung zu den Robotern - beispielsweise über WLAN - benötigt, in der die Zeitdifferenzen zwischen GUI und Applikationen ausgetauscht werden. Ein Problem bei diesem Ansatz ist jedoch, dass die Realisierung für verschiedene Plattformen meist auch die Verwendung von unterschiedlichen APIs erfordert. Beispielsweise wird unter Windows üblicherweise *DirectShow* verwendet, um Kamerabilder aufzuzeichnen, unter Linux hingegen *Video4Linux* oder unter Mac OS X *Quicktime*.

Die Verwendung von Videodateien sorgt für ein hohes Maß an Flexibilität. Die Dateien können von einer beliebigen Quelle stammen, beispielsweise von einer digitalen Kamera, von einer an einen PC angeschlossenen WebCam oder sogar von einem Mobiltelefon. Allerdings gibt es in diesem Fall keine einfache Methode, die Synchronisation mit den intrinsischen Daten automatisch durchzuführen. Um die Verarbeitung von Videodateien möglich zu machen, müssen die entsprechenden Videocodecs dekodiert werden können.

Da das wahrscheinlichste Szenario der Einsatz einer digitalen Videokamera oder einem anderen Gerät mit Videoaufnahme-Funktion ist, wird die Software zunächst auf die Verarbeitung von Videodateien beschränkt. Es soll beobachtet und untersucht werden, wie schwierig sich eine manuelle Synchronisation von Videodatei und intrinsischen Daten gestaltet. Bei Bedarf kann die Software noch um die direkte Aufnahme der Kamerabilder und die automatische Synchronisierung erweitert werden.

Die manuelle Synchronisation eines Videos mit den intrinsischen Daten soll für den Benutzer möglichst einfach sein. Deshalb soll für nicht synchronisierte Videos eine Abspielkomponente bereitgestellt werden, deren Position mit der Position der Abspielkomponente der intrinsischen Daten abgeglichen werden kann.

## 4.3 Abspielen

Zum Abspielen von Daten und Videos sollen in der GUI eine beliebige Anzahl von Log- und Video-Dateien geladen werden können. Die Log-Dateien können sowohl von den Roboter-Applikationen als auch mit der GUI aufgenommen worden sein. Sie enthalten wie in Abschnitt 4.1.2 beschrieben, die Informationen über die Zeitdifferenzen zwischen den einzelnen Applikationen und gegebenenfalls der GUI.

Synchronisiertes Video und intrinsische Daten sollen über eine gemeinsame Abspielkomponente verfügen, über die der Abspielvorgang gesteuert werden kann. Wie in Abschnitt 3.3 beschrieben soll man dabei an eine beliebige Stelle im Aufnahmezeitraum springen, sowie die Aufnahme langsamer oder schneller abspielen können. Das Abspielen zu einem beliebigen Zeitpunkt zu pausieren soll ebenfalls möglich sein.

Das Abspielen der intrinsischen Daten der Roboter soll sich für die GUI so verhalten, als wäre sie mit den verschiedenen Applikationen der Roboter direkt verbunden. Für jeden Roboter, von dem Daten aufgezeichnet wurden, wird also ein simulierte Verbindung zu dessen Applikation erstellt. Dadurch können die Daten in den Dialogen der GUI visualisiert werden. Durch die in Abschnitt 4.1.3 beschriebenen Anpassungen können die Daten für Dialoge, die nur Daten von einem Roboter verarbeiten können, entsprechend gefiltert werden. Für die Visualisierung der Daten mehrerer Roboter können in diesem Fall einfach mehrere Instanzen des entsprechenden Dialogs geöffnet und den einzelnen simulierten Roboter-Applikationen zugeordnet werden.

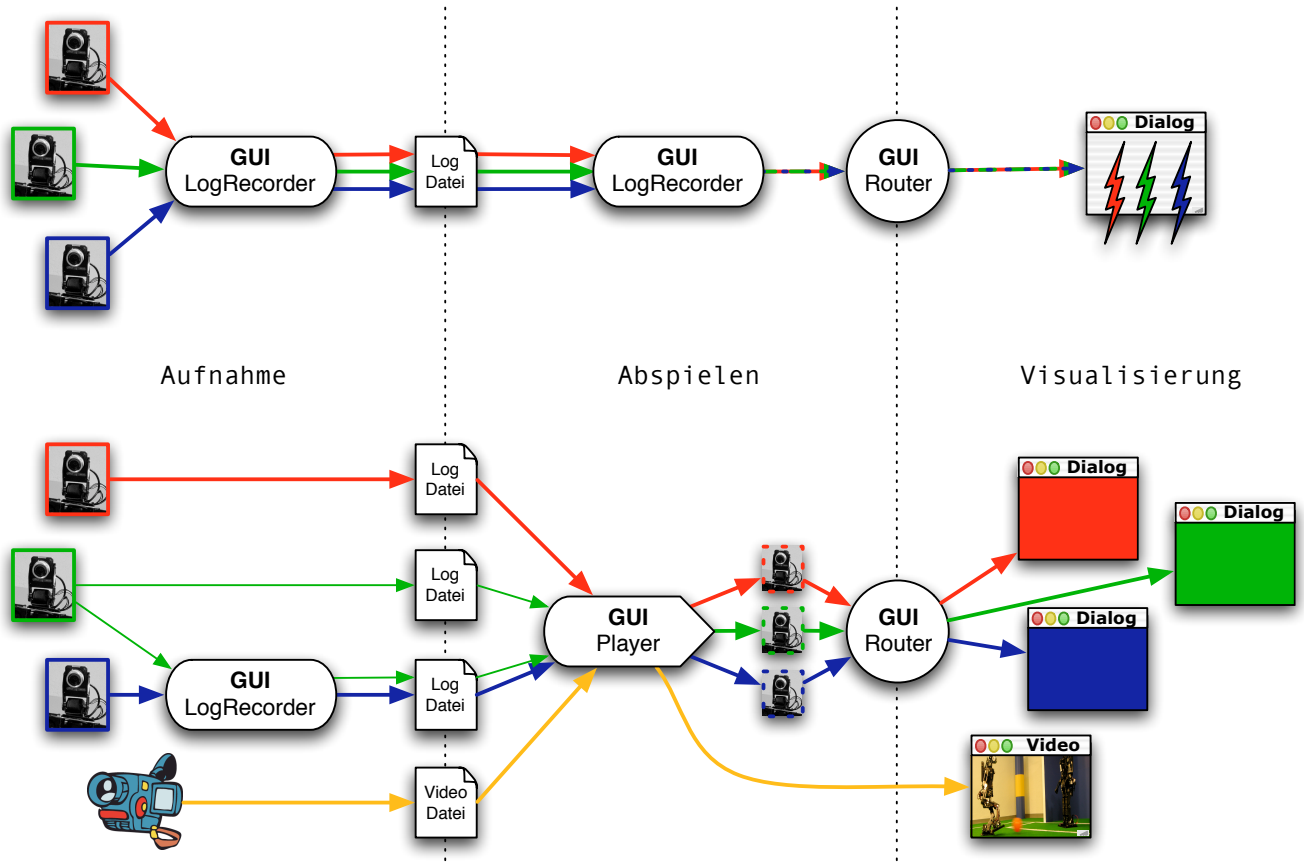


**Abbildung 4.2: Die Aufnahme, das Abspielen und die Visualisierung der Daten über den *LogRecorder* wie bisher (oben) und verteilt auf verschiedene Applikationen und mit Simulation verbundener Roboter (unten).**

Abbildung 4.2 zeigt eine Zusammenfassung von Aufnahme, Abspielen und Visualisierung der Daten.

## 4.4 Ereignisse

Ereignisse sollen die Navigation innerhalb des Aufnahmezeitraums und die Synchronisierung von Videos mit intrinsischen Daten erleichtern. Ein Ereignis bezeichnet zum Beispiel eine Änderung im Zustand oder Verhalten des Roboters. Sie sollen entweder aus den aufgenommenen Daten generiert oder direkt auf einer Applikation ausgelöst und aufgezeichnet werden können.

Für die Synchronisierung der Daten mit Videos sind solche Ereignisse interessant, die in einem Video leicht erkannt und identifiziert werden können. Beispiel für ein solches Ereignis ist das Start-Signal für die Roboter bei einem Spiel im *RoboCup Soccer*. Es kann in einem Video sehr leicht dadurch identifiziert werden, dass die Roboter still standen und plötzlich los laufen.

Andere interessante Ereignisse, sind solche Ereignisse, die bestimmte Stellen in der Aufnahme markieren, die für Analyse und Debugging besonders relevant sind. Außerdem können Ereignisse nützlich sein, die auf einen kritischen Zustand oder das Auftreten eines fatalen Fehlers aufmerksam machen.

Es soll in der GUI möglich sein, während dem Abspielen den Ablauf der Ereignisse zu verfolgen. Zusätzlich soll man direkt zu einem bestimmten Ereignis springen können.

## 4.5 Fusion von Daten und Video

Die intrinsischen Daten sollen in das Videobild integriert werden. Dazu sollen sie als sog. Overlays in das Videobild gezeichnet werden. Einem Overlay wird ein bestimmter Bereich im Bild zugeordnet, in den es zeichnen darf. Diese Bereiche sollen über die GUI einfach festgelegt, verschoben und in der Größe geändert werden können. Overlays sollen gegebenenfalls noch weiter konfiguriert werden können, beispielsweise um Farben oder Schriften einzustellen.

Beim Abspielen in der GUI werden die Overlays angezeigt und können konfiguriert und platziert werden. Für den Export in eine Videodatei werden diese Konfigurationen übernommen, so erhält man eine Vorschau auf das resultierende Video. Für den Export wählt man den gewünschten Zeitraum und die gewünschte Auflösung und startet den Vorgang. Die Videodatei kann dann außerhalb der Anwendung weiterverwendet werden.

# 5 Realisierung

Im folgenden Kapitel wird auf die Realisierung des in Kapitel 4 vorgestellten Konzepts eingegangen. Die einzelnen Teile der Realisierung werden detailliert erläutert. Die Software wird eng in das Softwareframework *RoboFrame* eingebunden, die Grundlagen zu *RoboFrame* werden zu Beginn kurz erläutert.

## 5.1 Einführung in RoboFrame

*RoboApp* und *RoboGui* sind modular aufgebaut, die einzelnen Komponenten legen dabei nur Ein- und Ausgabedaten fest. Komponenten in *RoboApp* sind *Module*, die in *Prozessen* zusammengefasst sind. Komponenten in *RoboGui* sind *Dialoge*. *Prozesse* und *Dialoge* sind *Konnektoren*. In *RoboApp* und *RoboGui* gibt es jeweils einen *Router*, der die Kommunikation zwischen den *Konnektoren* regelt. Abgesehen von lokalen *Konnektoren* kann es auch entfernte *Konnektoren* geben, diese repräsentieren eine über Netzwerk verbundene GUI oder Applikation. Ein *Konnektor* hat innerhalb des *Routers* eine eindeutige Identifikationsnummer. *Konnektoren* können Daten über einen *Schlüssel* anfordern, der diese identifiziert.

Ein *Prozess* in der Applikation besteht aus einem oder mehreren *Modulen*. Diese *Module* können über Datenpuffer untereinander oder mit anderen *Konnektoren* kommunizieren. Abbildung 5.1 zeigt beispielhaft die Kommunikation zwischen und innerhalb von Applikationen.

## 5.2 Identifikation von Applikationen und GUI

Um Daten einer Applikation auch über mehrere Log-Dateien aus verschiedenen Quellen zuordnen zu können, wird jeder Applikation beim Start eine Identifikationsnummer (ID) zugeordnet. Die ID ist ein kombinierter Hash-Wert aus der Systemzeit und der Adresse der Applikation im Speicher.

Zusammen mit dem Namen der Applikation ergibt sich aus der ID ein Identifikator (`AppIdentifier`) für die Applikation. Ein *Konnektor* hält diesen Identifikator auf Anfrage für andere lokale oder entfernte *Konnektoren* bereit.

Der Identifikator wird für die Zuordnung und Synchronisierung von intrinsischen Daten, sowie zur Bestimmung des Applikationsnamens bei Anzeige in der GUI verwendet.

Die GUI verfügt ebenfalls über eine eigene ID und einen Identifikator.

## 5.3 Intrinsische Daten

Die folgenden Abschnitte beschreiben die Realisierung des Aufzeichnens und der Synchronisierung der intrinsischen Daten der Roboter. Anschließend wird auf die Visualisierung dieser Daten eingegangen.

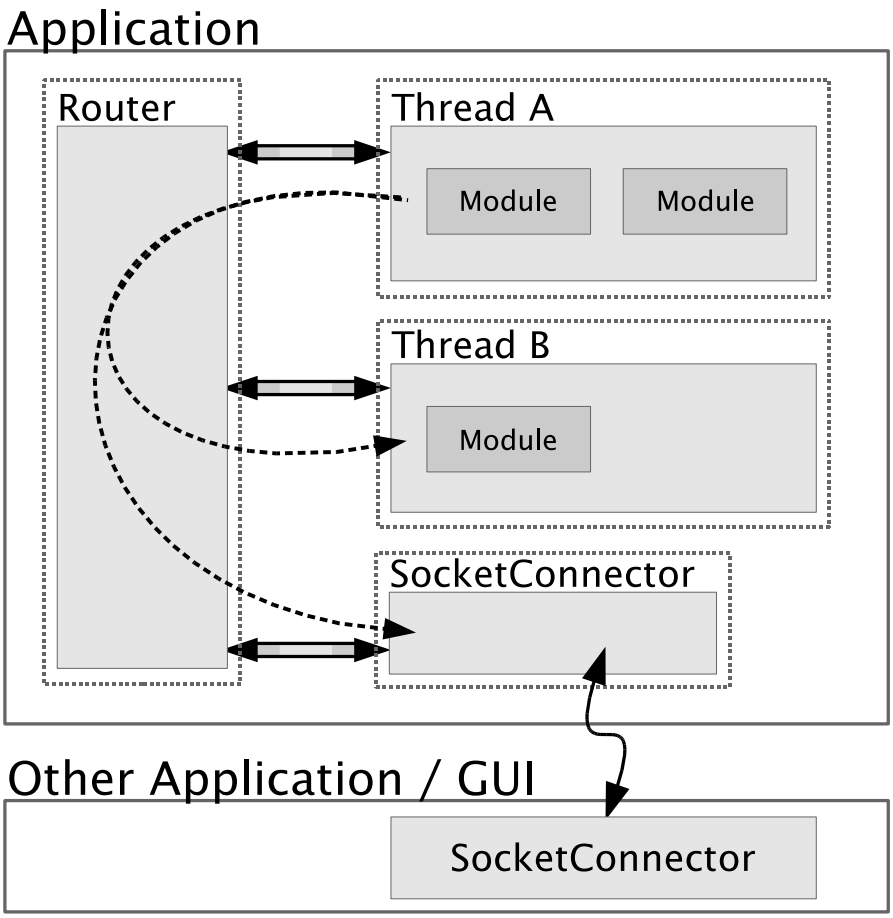


Abbildung 5.1: Kommunikation zwischen Modulen, Prozessen und Applikationen in RoboFrame[13]

### 5.3.1 Aufzeichnen

Für das Aufzeichnen der Daten in der Roboter-Applikation wurde ein neuer *Konnektor* implementiert. Der `LogFileConnector` fordert Daten an und schreibt eingehende Daten in eine Log-Datei.

Der `LogFileConnector` wird über eine `LogConfiguration` konfiguriert. Die `LogConfiguration` gibt folgendes an:

- ob das Aufnehmen der Log-Datei aktiviert werden soll
- den Namen der aufzunehmenden Log-Datei
- ob die letzte Log-Datei mit gleichem Namen überschrieben werden soll
- die maximale Größe der Log-Datei
- die maximale Aufnahmedauer
- die *Schlüssel*, deren Daten angefordert werden sollen

Außerdem kann zu jedem *Schlüssel* angegeben werden, ob jedes Paket dieser Daten angefordert werden soll oder nur bestimmte Pakete. Es können beispielsweise auch nur $n$ Pakete oder jedes $n$-te Paket angefordert werden. Es besteht auch die Möglichkeit, Daten zu einem *Schlüssel* nur für alle $n$ Millisekunden anzufordern.

Außer den Daten, die in der Konfiguration spezifiziert sind, werden automatisch auch folgende Daten mit angefordert:

- der Identifikator der Applikation (siehe Abschnitt 5.2) um die Datenquelle zu identifizieren
- die Informationen über die Zeitdifferenzen zu anderen Applikationen oder einer GUI (siehe Abschnitt 5.3.2)
- in der Applikation ausgelöste Ereignisse (siehe Abschnitt 5.6.1)
- Marker, die die Ausführung eines Prozesses in der Applikation markieren. Über diese Marker kann bestimmt werden, wie oft und mit welcher Frequenz ein Prozess ausgeführt wurde. Um das Senden der Marker für einen Prozess zu aktivieren muss die Methode `setSendExecutionMarker` entsprechend aufgerufen werden.

Der `LogFileConnector` liest seine Konfiguration - wenn vorhanden - aus einer bestimmten Datei, andernfalls ist er standardmäßig deaktiviert. Die Konfiguration kann in der GUI über den Dialog *RemoteLog* (siehe Abbildung 5.2) bearbeitet werden. Der Dialog bietet die Möglichkeit die Konfiguration in einer Datei abzuspeichern, um diese zusammen mit der Applikation auf dem Roboter abzulegen. Die Konfiguration kann auch direkt an eine verbundene Roboter-Applikation gesendet werden, wo diese dann vom `LogFileConnector` übernommen wird.

Zusätzlich kann der *RemoteLog* Dialog den Status des *Konnektors* überwachen. Dabei wird angezeigt, ob das Schreiben der Log-Datei gerade aktiviert ist, wie der Name der Log-Datei ist und wie viel Speicher noch auf dem Roboter vorhanden ist.

Für das Schreiben der Log-Datei verwendet der `LogFileConnector` den `LogFileWriter`. Der `LogFileWriter` ist auf das Schreiben von Log-Dateien optimiert und hält im Gegensatz zu der Klasse `LogFile`, die im *LogRecorder* Dialog verwendet wird, nicht alle Daten im Speicher. Neue Daten werden einfach an die Datei angehängt, wobei ein Puffer verwendet wird, um nicht für alle eingehenden Daten direkt eine Schreiboperation aufzurufen.

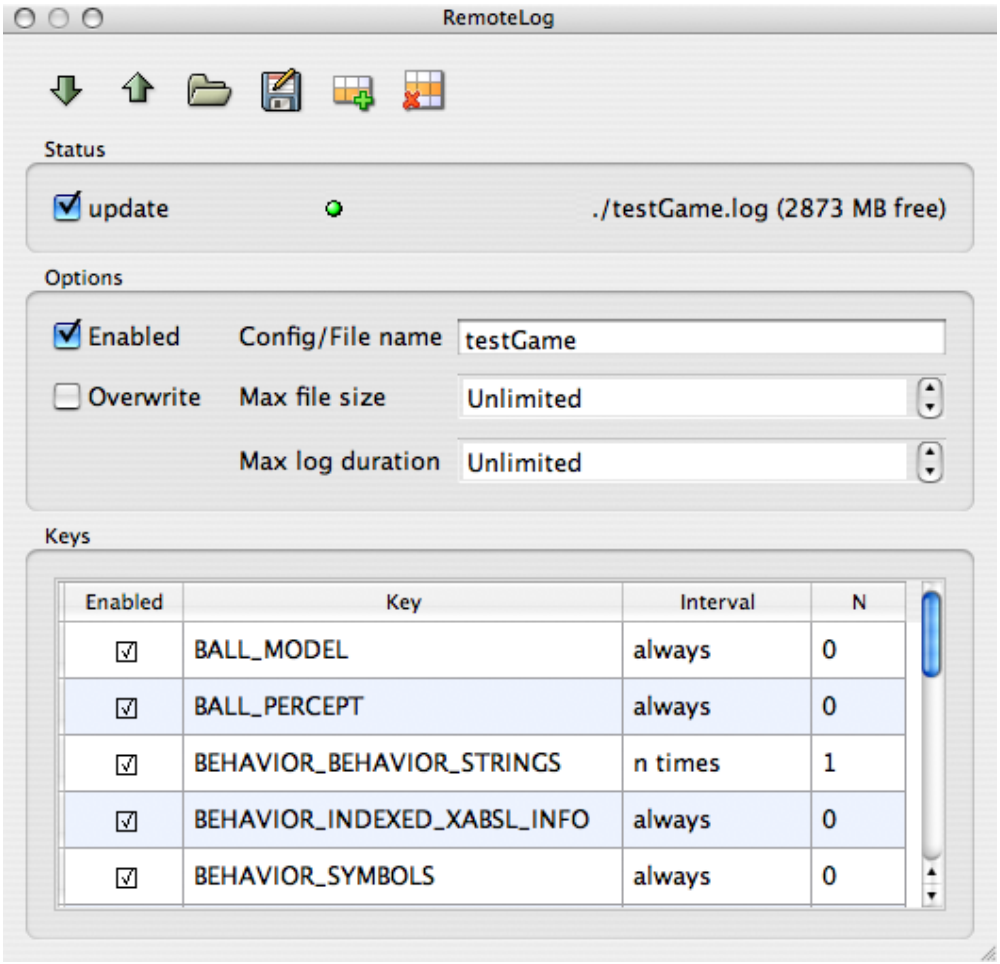


Abbildung 5.2: Der *RemoteLog* Dialog mit dem aktuellen Log-Status (oben), der allgemeinen Konfiguration (mitte) und der Liste der *Schlüssel* der angeforderten Daten (unten).

### Aufzeichnen in der GUI

Um die Daten, die in der GUI über den *LogRecorder* aufgezeichnet werden, in der Analyse den einzelnen Robotern zuordnen zu können, werden die entsprechenden Identifikatoren (siehe Abschnitt 3.1) angefordert und mit aufgezeichnet. Außerdem wird ein Identifikator in die Log-Datei geschrieben, der die GUI als Autor identifiziert.

## 5.3.2 Synchronisieren

Die Informationen zur Zeitsynchronisierung, die zwischen Applikation und GUI ausgetauscht werden, wurde um die IDs (siehe Abschnitt 5.2) der beiden Kommunikationspartner erweitert. Nach dem Bestimmen der Zeitdifferenz wird diese zusammen mit den IDs gespeichert. Ein *Konnektor* hält sie auf Anfrage für den `LogFileConnector` bereit.

Zur Bestimmung der Zeitdifferenz zwischen verschiedenen Roboter-Applikationen wird ein weiterer *Konnektor* verwendet. Der `UDPSyncConnector` sendet periodisch Anfragen zur Bestimmung der Zeitdifferenz an andere Roboter-Applikationen. Die Anfragen werden per *UDP Broadcast* oder falls möglich - wenn die Netzwerk-Adressen der anderen Roboter bekannt sind - per *UDP Unicast* verschickt. Erhält der `UDPSyncConnector` einer Applikation eine Anfrage, antwortet er dem Sender per *UDP Unicast*. Die Zeitdifferenz wird dabei über ein SNTP[11]-basiertes Protokoll bestimmt. Die bestimmte Zeitdifferenz wird unter den Kommunikationspartnern ausgetauscht und ebenfalls inklusive der IDs der Applikationen über einen *Konnektor* zugänglich gemacht. Der `LogFile-Connector` zeichnet diese Informationen automatisch auf, falls er aktiviert ist.

## 5.3.3 Visualisieren

Die Dialoge der GUI sind spezialisierte *Konnektoren*, wobei die *Konnektoren* die Infrastruktur für das Senden und Empfangen von Daten bereitstellen. Um die in Abschnitt 4.1.3 beschriebene Zwischenschicht zu implementieren werden die *Konnektoren* der GUI von einem *Dekorierer* (*Decorator*) gekapselt.

Dazu wurde aus der abstrakten *Konnektor*-Basisklasse `Connector` das Interface `IConnector` extrahiert. Der *Dekorierer* implementiert das Interface und hält die Instanz des gekapselten *Konnektors*. Aufrufe auf den Methoden des *Dekorierers* werden entweder direkt an den gekapselten *Konnektor* weitergereicht oder im *Dekorierer* neu implementiert. Die Klasse des *Dekorierers* heißt `FilteredConnector`. Von `FilteredConnector` werden nur *Konnektoren* gekapselt, die das Interface `IFilterOptions` implementieren. Nach dem Kapseln des *Konnektors* wird dann der *Dekorierer* statt dem *Konnektor* zum *Router* hinzugefügt. Abbildung 5.3 zeigt die wichtigsten Klassen der Filter-Zwischenschicht und deren Beziehungen untereinander.

Den Kern des Filter-Mechanismus bildet der `ConnectorFilter`. Er speichert die Zuordnung der gefilterten *Konnektoren* zu den anderen *Konnektoren*, von denen der gefilterte *Konnektor* Daten empfangen oder an diese senden darf. Ein solcher zugeordneter *Konnektor* kann auch eine verbundene Roboter-Applikation repräsentieren. In diesem Fall fordert der `ConnectorFilter` den Identifikator der Applikation (siehe Abschnitt 5.2) an, um den *Konnektor* mit dem Namen der Applikation zu versehen. Der `ConnectorFilter` kann über den *FilterDialog* konfiguriert werden (siehe Abbildung 5.4). Über das Interface `IFilterOptions` kann ein *Konnektor* angeben, ob er

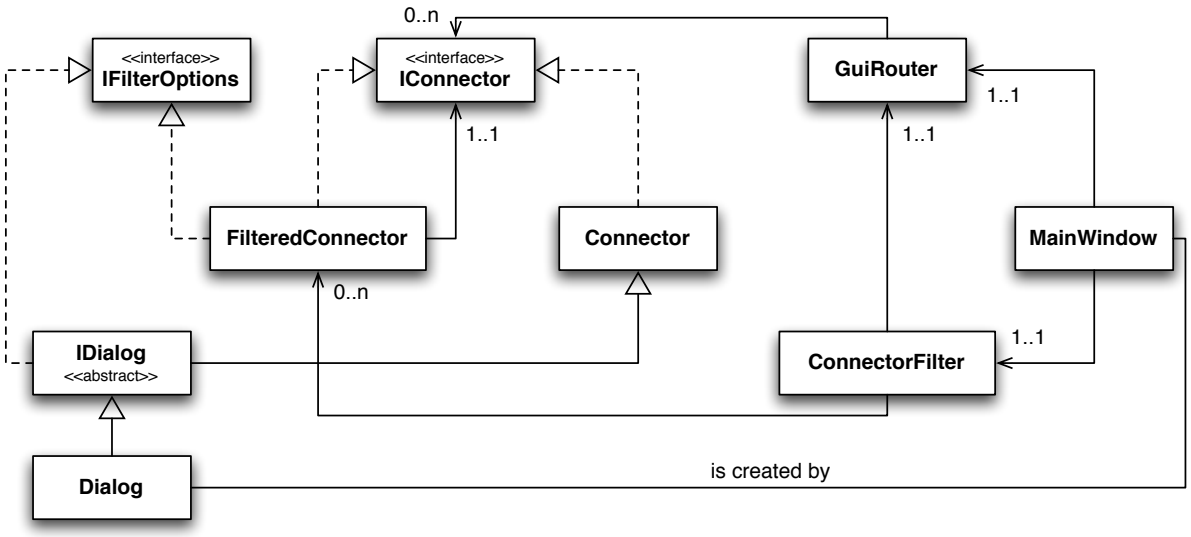


Abbildung 5.3: Klassendiagramm der Filter-Zwischenschicht

Verbindungen zu nur einem oder mehreren *Konnektoren* unterstützt. Entsprechend kann man im *FilterDialog* nur einen einzigen oder mehrere *Konnektoren* zuordnen. Ist der Filter für Senden und Empfangen gleich konfiguriert, wirken sich Änderungen für das Senden auch auf das Empfangen aus. Standardmäßig wird Senden und Empfangen an bzw. von allen *Konnektoren* erlaubt, da dies das gewohnte Verhalten ohne den Filter-Mechanismus ist.

Der `ConnectorFilter` wird vom *Router* über neu hinzugefügte oder gelöschte *Konnektoren* informiert und aktualisiert gegebenenfalls die vorhandenen Zuordnungen. Bei Änderung einer Filter-Konfiguration informiert er den entsprechenden `FilteredConnector`, welcher das Ereignis wiederum an den gekapselten *Konnektor* weitergibt. So zeigt ein Dialog in der Titelleiste an, mit welchen *Konnektoren* er verbunden ist und kann durch Überschreiben der `reset()`-Methode auf die Änderung der Zuordnung reagieren und ungültig gewordene Daten verwerfen.

Der `FilteredConnector` sammelt alle Datenanforderungen, die der gekapselte *Konnektor* sendet und leitet diese nur an die für den Empfang von Daten zugeordneten *Konnektoren* weiter (siehe auch Abbildung 5.5). Bei Änderung der Zuordnung werden die weitergeleiteten Anforderungen entsprechend widerrufen oder an neu zugeordnete *Konnektoren* gesendet. So werden nur Daten angefordert, die auch wirklich verwendet werden und es müssen keine eingehenden Daten verworfen werden.

Umgekehrt werden auch nur Datenanforderungen, die von *Konnektoren* kommen, für die das Senden in der Zuordnung aktiviert ist, an den gekapselten *Konnektor* weitergeleitet. Diese Anforderungen werden bis zu ihrer Aufhebung gespeichert so dass sie bei einer Änderung der Zuordnung entsprechend aktualisiert und weitergeleitet werden können.

#### 5.3.3.1 Verbesserung der Bedienbarkeit

Sollen Daten von verschiedenen Robotern in der GUI visualisiert werden, so werden tendenziell viele Dialoge dafür benötigt. Viele geöffnete Dialoge machen die GUI jedoch schnell sehr unübersichtlich. Außerdem lässt sich dadurch das Hauptfenster und damit das Hauptmenü nur schlecht erreichen.

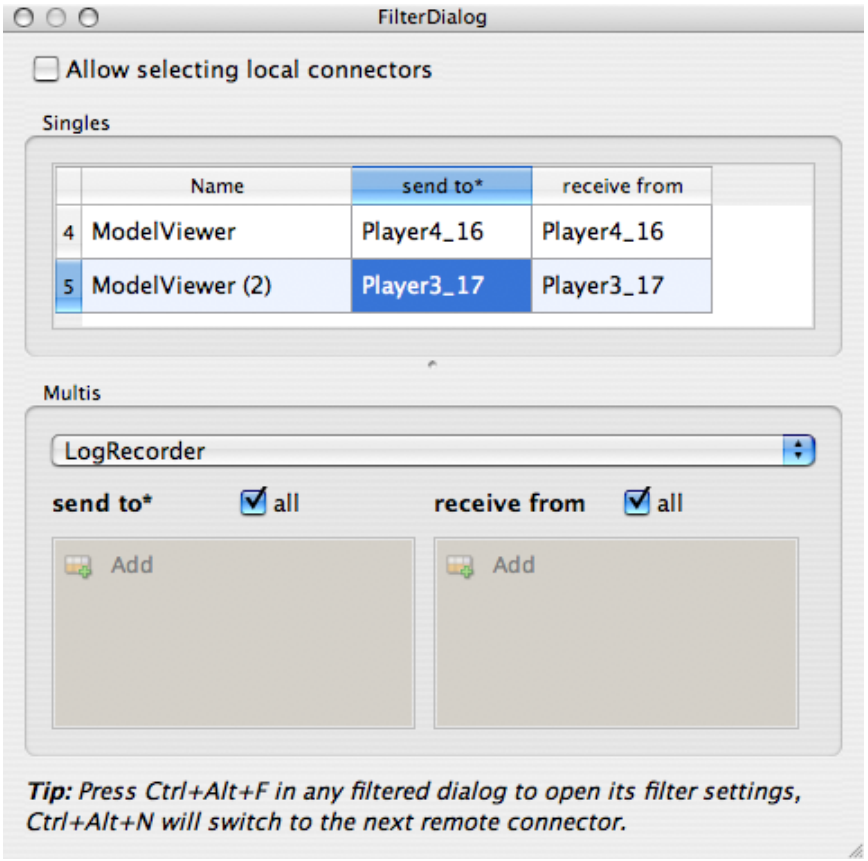


Abbildung 5.4: Der *FilterDialog* zur Konfiguration der gefilterten *Konnektoren*

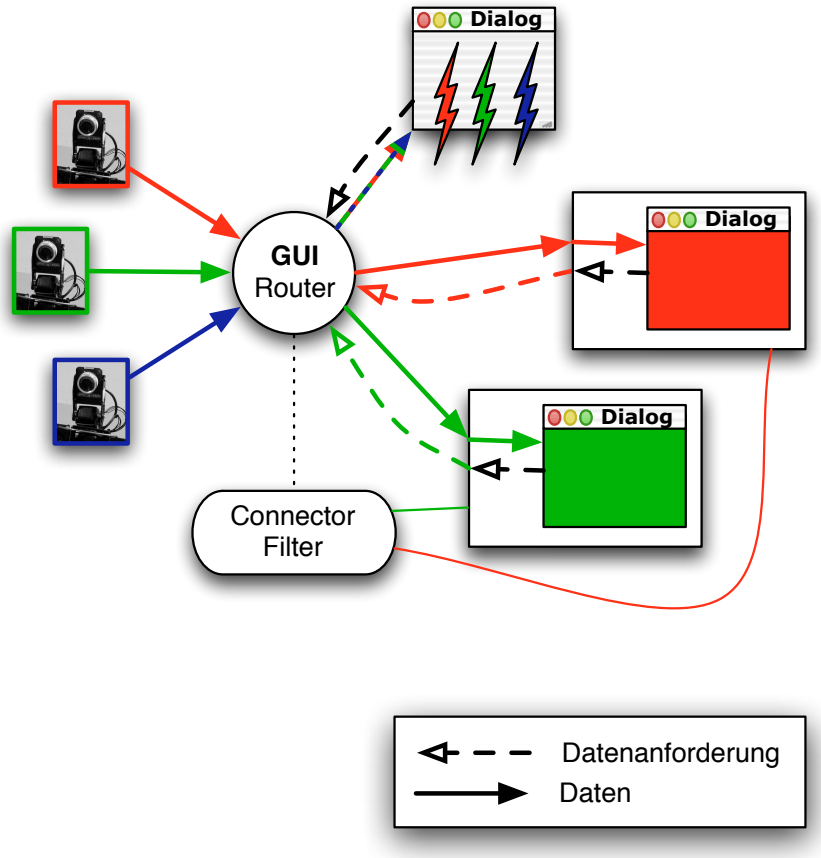


Abbildung 5.5: Datenanforderungen und Datenfluss bei ungefilterten und gefilterten Dialogen.

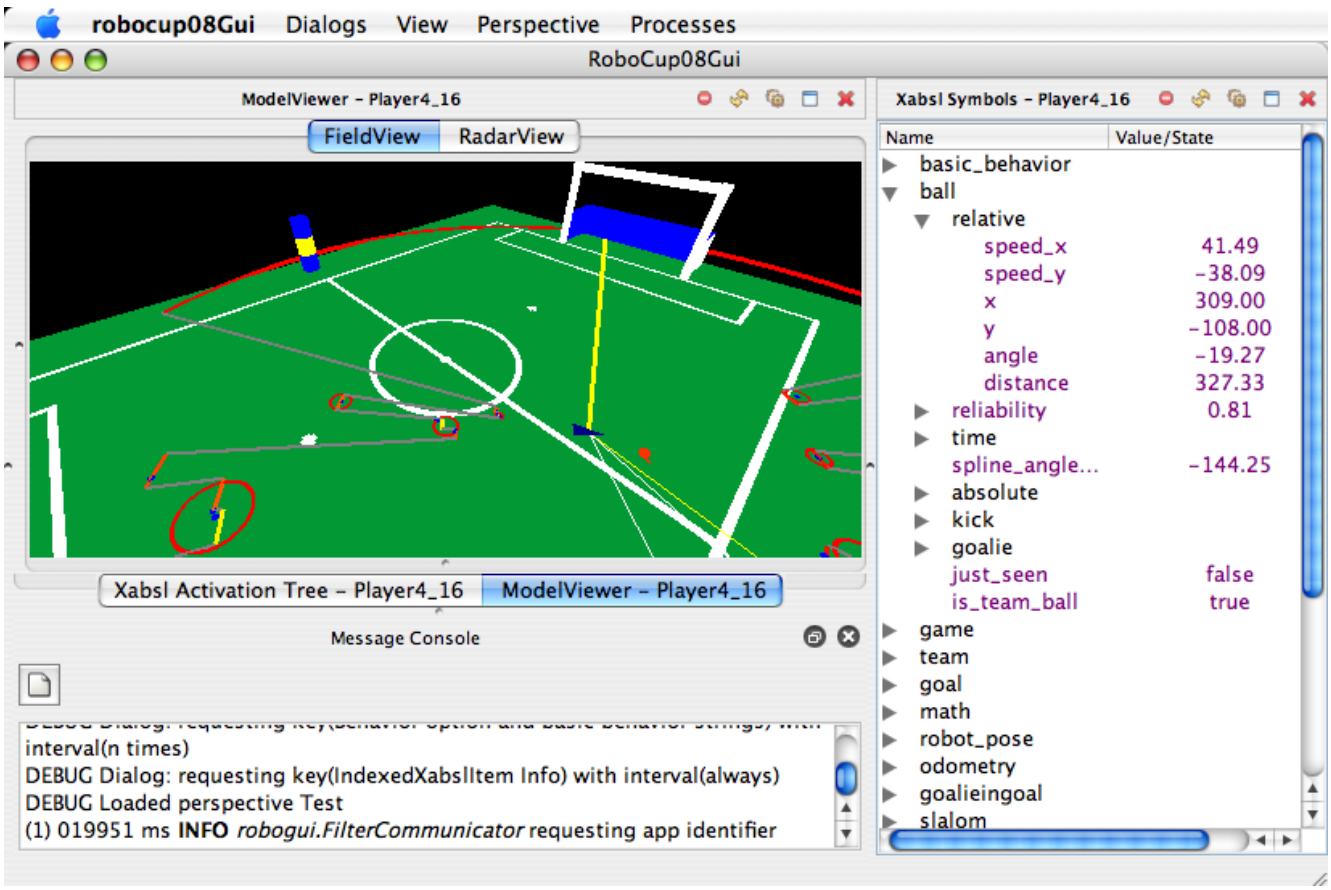


Abbildung 5.6: GUI mit *Qt Dock Widgets*, das Hauptmenü (oben) ist immer erreichbar, die Titelleiste der gefilterten Dialoge beinhaltet die verbundenen Applikationen und zusätzliche Optionen zur schnellen Filter-Konfiguration.

Um diesem Problem Abhilfe zu schaffen wurden die Dialoge in *Qt Dock Widgets* integriert. Dadurch lassen sich die Dialoge im Hauptfenster beliebig anordnen, unter anderem können mehrere Dialoge auch in einem Register angezeigt werden. Das Hauptmenü ist so immer erreichbar, bei Bedarf können einzelne Dialoge auch weiterhin als eigene Fenster angezeigt werden.

Die Komponenten der GUI, die zuvor fest im Hauptfenster untergebracht waren, sind jetzt ebenfalls als *Qt Dock Widgets* verfügbar und können beliebig angeordnet werden. Sie können über das Menü *View* ein- und ausgeblendet werden, so dass man die GUI auf die Anzeigen beschränken kann, die man auch wirklich verwendet.

Dialoge, die im Hauptfenster angezeigt werden, haben eine angepasste Titelleiste, die zusätzliche Optionen bereitstellt, die die Filtereinstellungen betreffen. So kann ein Dialog mit einem Klick einer anderen Roboter-Applikation zugeordnet werden. Es können auch alle Zuordnungen des Dialogs aufgehoben werden, um den Empfang bzw. das Anfordern von Daten zu verhindern. Eine weitere Option öffnet ein Fenster zur Konfiguration der Filter-Optionen für den zugehörigen Dialog. Abbildung 5.6 zeigt ein Beispiel einer GUI mit im Hauptfenster angeordneten Dialogen.

Weiterhin ist es jetzt möglich, verschiedene Sets von Einstellungen zu verwalten, sogenannte Perspektiven. Zu den Einstellungen gehören die geöffneten Dialoge, deren Anordnung und deren eigene Einstellungen. Über das Hauptmenü können verschiedene Perspektiven angelegt und verwaltet werden. Beim Wechseln zwischen Perspektiven werden alle Einstellungen der einen Perspektive gespeichert und alle Einstellungen der anderen Perspektive geladen.

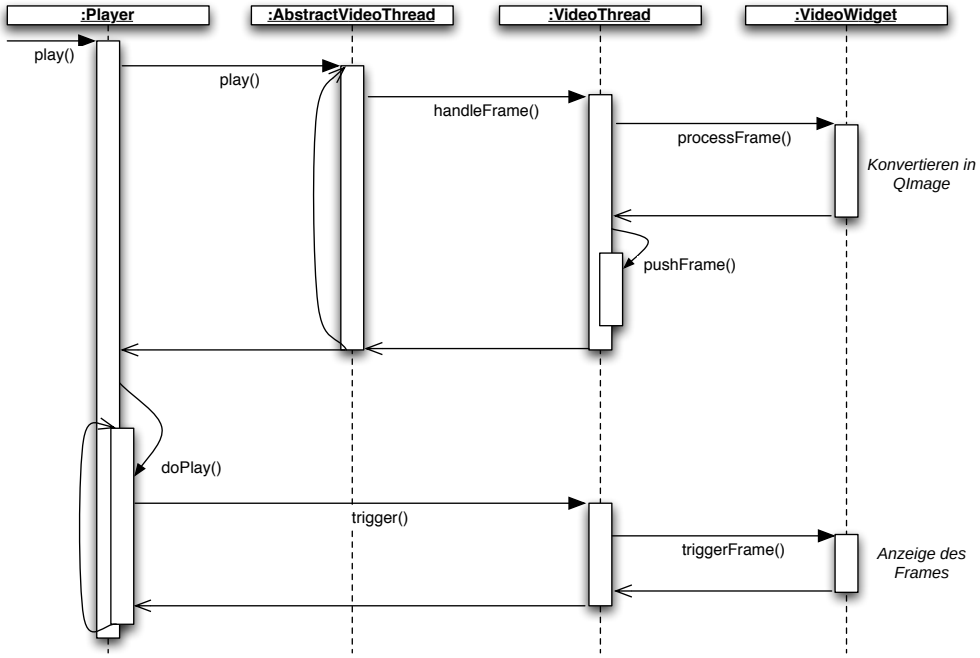


Abbildung 5.7: Dekodierung und Anzeige von Video-Frames

## 5.4 Video

Für das dekodieren und kodieren von Videos wird *FFmpeg*[1] verwendet. *FFmpeg* ist freie Software lizenziert unter der *GNU Lesser General Public License* (LGPL) oder der *GNU General Public License* (GPL), wobei hier nur Teile verwendet werden, die unter der LGPL verfügbar sind. *FFmpeg* bietet eine Unterstützung einer breiten Menge an Codecs und unterstützt unter anderem die Plattformen Windows, Linux und Mac OS X, welche auch für die GUI hauptsächlich eingesetzt werden.

### 5.4.1 Dekodierung und Anzeige

Die Dekodierung und Anzeige der Videos erfolgt in drei Stufen (siehe auch Abbildung 5.7).

Die Klasse `AbstractVideoThread` implementiert die grundlegenden Funktionen wie Abspielen und Pausieren. Sie dekodiert in einem eigenen Thread die Frames des Videos. Zu jedem Frame wird der Anzeige-Zeitstempel (Presentation TimeStamp, PTS) bestimmt.

`VideoThread` ist eine Template-Klasse, die von `AbstractVideoThread` erbt und den Frame, der in einem *FFmpeg*-internen Format vorliegt, in ein anderes Format konvertieren lässt. Der konvertierte Frame wird zusammen mit dem PTS zeitlich in eine Liste einsortiert.

Das `VideoWidget` erbt wiederum von `VideoThread` und übernimmt die Konvertierung der Frames in ein `QImage`, die Bildklasse von *Qt*. Die Anzeige der Frames wird durch den Abspielvorgang gesteuert (siehe Abschnitt 5.5.1). Der Abspielvorgang ruft das `VideoWidget` mit der jeweiligen Abspielzeit auf, welches das nächste `QImage` aus der Liste holt und dabei gegebenenfalls einige davon verwirft. Ein Neuzeichnen des Widgets wird ausgelöst und der aktuelle Frame angezeigt.

[1] Offizielle Homepage von FFmpeg: `http://www.ffmpeg.org/`

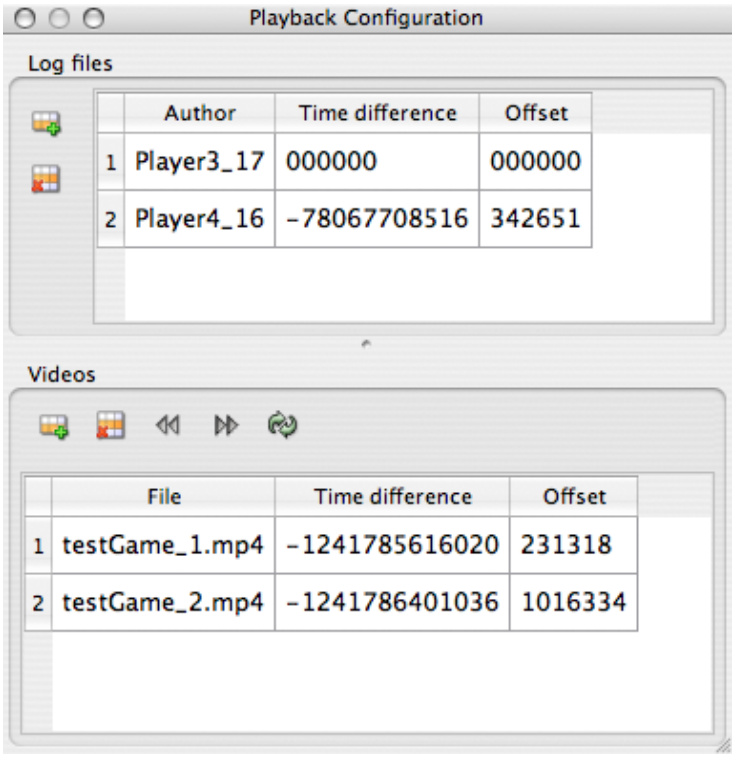


Abbildung 5.8: Ansicht *Playback Configuration* zur Verwaltung von Log-Dateien und Videos

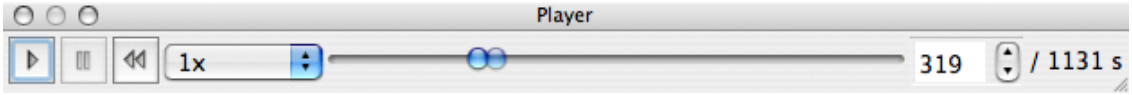


Abbildung 5.9: Zentrale Abspielkomponente

Der `VideoThread` merkt sich intern die Abspielzeit und sorgt dafür, dass der `AbstractVideoThread` maximal zehn Sekunden im voraus dekodiert oder Frames vor dem Konvertieren überspringt wenn die Dekodierung hinter der Abspielzeit zurückliegt.

## 5.5 Abspielen

In der Ansicht *Playback Configuration* (siehe Abbildung 5.8) können Log-Dateien und Videos geladen werden. Durch das Laden einer Log-Datei wird die Abspielkomponente (siehe Abbildung 5.9) aktiviert. Beim Hinzufügen eines Videos wird ein Dialog erstellt, der das entsprechende `VideoWidget` enthält.

### 5.5.1 Steuerung des Abspielvorgangs

Die Klasse `Player` steuert den Abspielvorgang. Ein `Player` verwaltet verschiedene `IPlayables`. `IPlayable` ist ein Interface, das die Methoden zur Steuerung des Abspielens der jeweiligen Komponente definiert. `IPlayable` wird zum einen von der Komponente zum Abspielen der Log-Dateien (siehe Abschnitt 5.5.2) und zum anderen von der Komponente zum Abspielen der Videos - `Ab-`

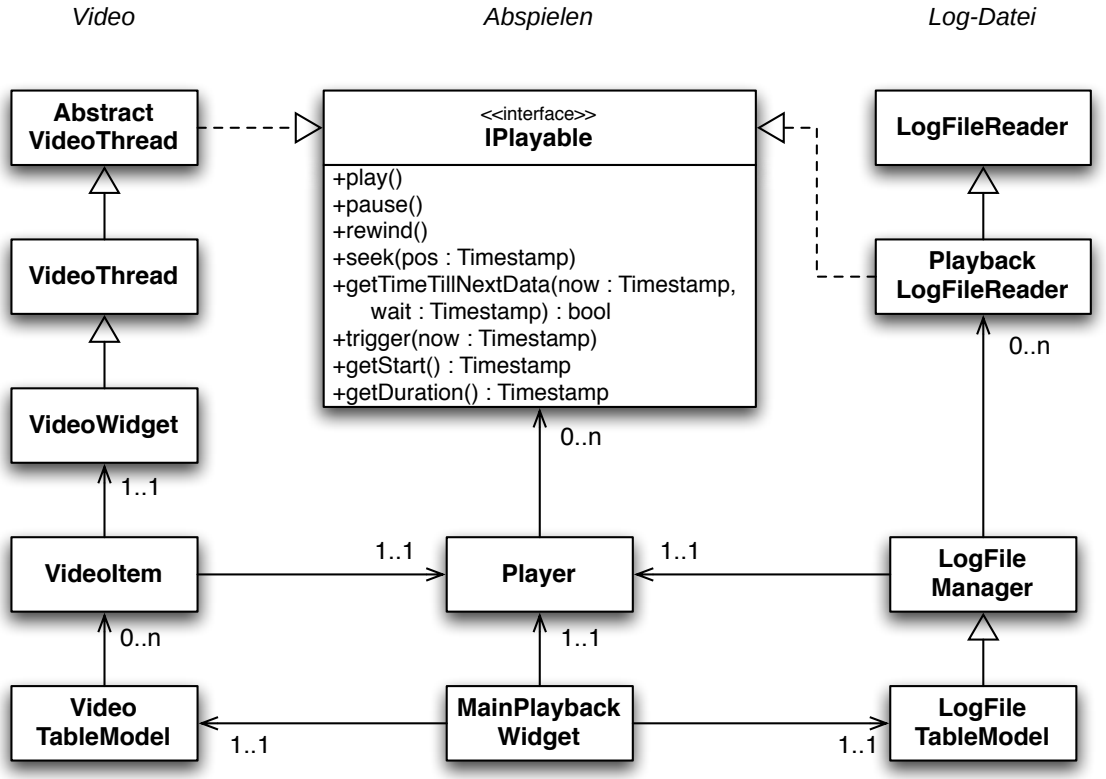


Abbildung 5.10: Implementierungen von `IPlayable`

`stractVideoThread` bzw. `VideoThread` - implementiert. Abbildung 5.10 gibt eine Übersicht über die Implementierungen von `IPlayable`.

Jedes `IPlayable` hat eine bestimmte Startzeit und Dauer. Im `Player` ist jedem `IPlayable` die Zeitdifferenz zu einer globalen Abspielzeit zugeordnet. Für Aufrufe eines `IPlayable` wird die globale Abspielzeit in die lokale Abspielzeit des jeweiligen `IPlayable` umgerechnet. Für die globale Abspielzeit wird üblicherweise ein `IPlayable` als Referenz genommen, dessen lokale Abspielzeit ist dann die globale Abspielzeit. Die Startzeit des `Player` ergibt sich aus dem `IPlayable` mit der - in globaler Abspielzeit - frühesten Startzeit. Die Dauer ergibt sich aus dem `IPlayable`, welches zuletzt endet.

Listing 5.1 zeigt einen Auszug aus dem Interface `IPlayable`. Die Methoden `play`, `pause`, `rewind` und `seek` bereiten auf das Abspielen vor und informieren über Zustandsänderungen des `Player`. Das eigentliche Abspielen erfolgt unter Verwendung der Methoden `getTimeTillNextData` und `trigger`.

Der `Player` bestimmt über den Aufruf von `getTimeTillNextData` die Wartezeit ab der momentanen Abspielzeit. Die Abspielzeit wird dementsprechend erhöht, nach Ablauf der Wartezeit wird `trigger` auf den `IPlayables` aufgerufen. Der Kreis schließt sich wieder mit Bestimmung der Wartezeit, wobei diese um die Dauer reduziert wird, die für die `trigger`-Aufrufe benötigt wurde.

Dieser Kreislauf wird erst unterbrochen, wenn das Abspielen pausiert wird oder keine Daten mehr vorhanden sind. Der erneute Aufruf des `Player` nach der Wartezeit erfolgt über einen *Qt Timer*. Dadurch wird der Abspielvorgang in der *Qt Event Queue* abgearbeitet. Das hat den Vorteil, dass der nächste Abspielzyklus immer erst nach der Visualisierung der Daten aufgerufen wird, welche auch in der *Event Queue* erfolgt. So werden die Daten stets synchron visualisiert. Außer-

Listing 5.1: Auszug aus dem `IPlayable` Interface

```
37  /**
38   * Signals that playing has started, the IPlayable can start to prepare data
39   */
40  virtual void play() {};
41
42  /**
43   * Signal that playing has paused, the IPlayable can stop preparing data
44   */
45  virtual void pause() {};
46
47  /**
48   * Rewind to the start / first elemnt
49   */
50  virtual void rewind() = 0;
51
52  /**
53   * Jump to the given position in time
54   *
55   * @param pos the playback position
56   */
57  virtual void seek(roboapp::Timestamp pos) = 0;
58
59  /**
60   * Get the time till the next data from the given time
61   *
62   * @param now the current playback time
63   * @param wait the variable to set with the time till the next data
64   *
65   * @return if there is any data left
66   */
67  virtual bool getTimeTillNextData(roboapp::Timestamp now, roboapp::Timestamp&
        wait) = 0;
68
69  /**
70   * Play all data for the given playback time
71   *
72   * @param now the current playback time
73   */
74  virtual void trigger(roboapp::Timestamp now) = 0;
```

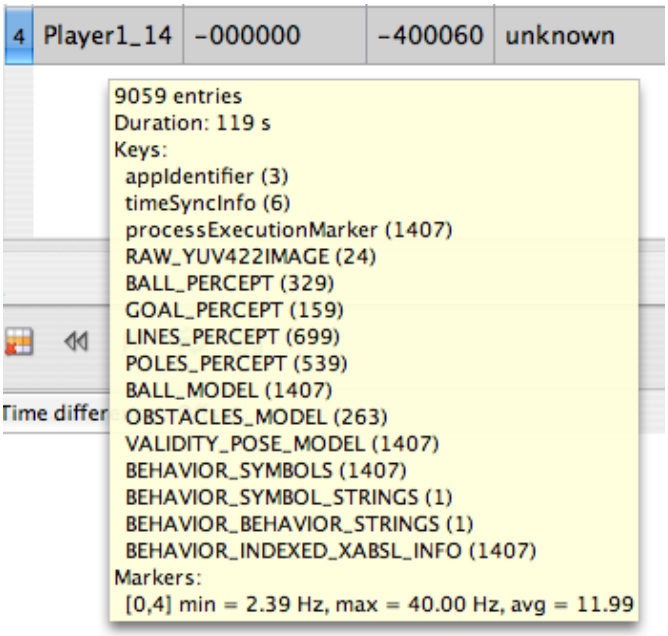


Abbildung 5.11: Der Tooltip gibt zu jeder Log-Datei eine kurze Zusammenfassung

dem wird die Interaktion des Benutzers mit der GUI so nicht eingeschränkt, der Abspielvorgang wird lediglich verlangsamt, wenn der Rechenaufwand bei der Verarbeitung der Daten zu groß ist.

Das `PlayerControlWidget` (siehe Abbildung 5.9) ist die Benutzerschnittstelle für die Steuerung des `Player`. Hier kann die Geschwindigkeit des Abspielvorgangs angepasst oder an eine bestimmte Position im Aufnahmezeitraum gesprungen werden.

### 5.5.2 Abspielen der Log-Dateien

Zum Einlesen der Log-Dateien dient der `LogFileReader`. Im Gegensatz zur vorhandenen `LogFile`-Klasse lädt er nicht die gesamten Daten in den Speicher, sondern nur die Metainformationen zu den einzelnen Dateneinträgen. Dazu gehören der Daten-*Schlüssel*, der Zeitstempel, die Länge der Daten in Bytes und die Datenquelle. Die Datenquelle enthält die Identifikationsnummer des entsprechenden *Konnektors*.

Beim Laden einer Log-Datei erstellt der `LogFileReader` einen Index über die Einträge der Datei. Dazu werden alle Dateneinträge der Datei durchgegangen. Die Metainformationen werden zusammen mit der Position der zugehörigen Daten in der Datei im Index hinterlegt. Die Tabelle der Log-Dateien stellt zu jeder Datei einen Tooltip zur Verfügung, der eine kurze Zusammenfassung liefert (siehe Abbildung 5.11). Unter anderem wird dort angegeben, wie viel Dateneinträge die jeweilige Datei hat und welche Informationen wie oft dort enthalten sind.

Ein `LogFileReader` kann über verschiedene Module zur Analyse der Einträge verfügen. Diese werden beim Einlesen der Datei für jeden Eintrag aufgerufen. Die Daten des jeweiligen Eintrags werden jedoch nur gelesen, wenn eines der Analyse-Module explizit darauf zugreifen möchte - andernfalls werden die Daten übersprungen und mit dem nächsten Eintrag fortgefahren. Die Analyse-Module implementieren das Interface `ILogEntryAnalyzer`. Ein einfaches Analyse-Modul ist der `HeaderStatisticsAnalyzer`. Er bestimmt Startzeit und Dauer der Log-Datei, sowie die Anzahl der Einträge. Außerdem sammelt er die Informationen zu den vorhandenen Ausführungs-Markern

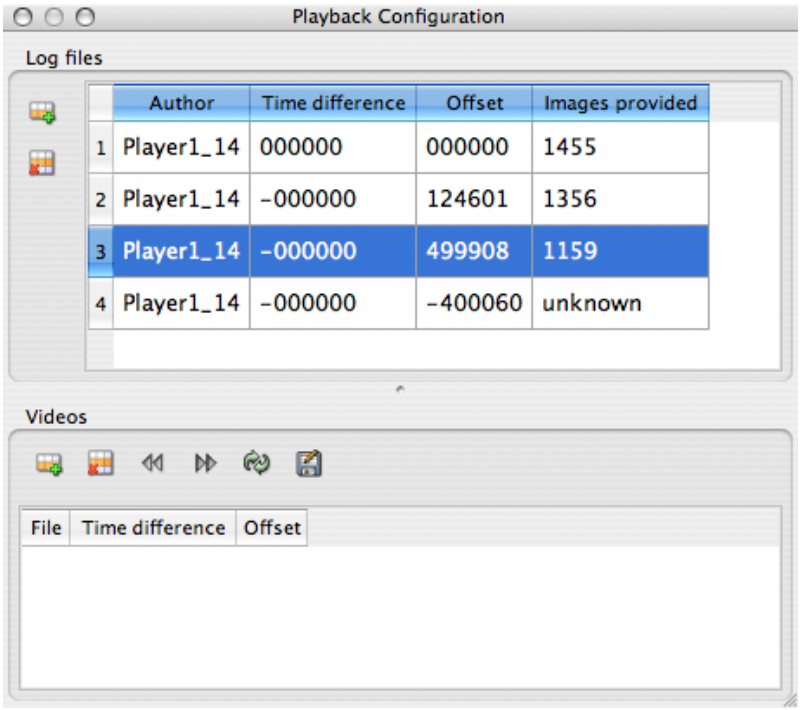


Abbildung 5.12: Tabelle der Log-Dateien (oben) mit einer zusätzlichen Spalte “Images provided”, die durch ein benutzerdefiniertes Analyse-Modul bereitgestellt wird. Der Wert gibt an, wie viele Bilder auf dem Roboter während des Aufzeichnens der Log-Datei verarbeitet wurden. Der Tooltip in der Spalte kann zusätzliche Informationen liefern.

der Prozesse und gibt die Anzahl der Ausführungen, sowie die durchschnittliche, minimale und maximale Frequenz der Ausführungen an.

Zusätzlich zu den bereits integrierten Analyse-Modulen können benutzerdefinierte Module hinzugefügt werden. Ein benutzerdefiniertes Modul wird ebenfalls beim Laden der Datei ausgeführt und kann die ermittelten Ergebnisse in der Tabelle der Log-Dateien über eine zusätzliche Spalte bereitstellen (siehe Abbildung 5.12).

Der `PlaybackLogFileReader` erbt von `LogFileReader` und implementiert das `IPlayable` Interface (siehe Abbildung 5.10). Er dient zum Abspielen einer Log-Datei im Kontext eines `Player`. Für den Abspielvorgang hält er intern die aktuelle Indexposition. Diese entspricht dem ersten Eintrag ab der aktuellen Abspielzeit. Beim Aufruf von `rewind` wird diese auf den ersten Eintrag gesetzt, beim Aufruf von `seek` wird die neue Indexposition durch eine binäre Suche bestimmt.

#### 5.5.2.1 Virtuelle Roboter

Ein weiteres Analyse-Modul des `PlaybackLogFileReader` ist der `AppIdentifierAnalyzer`. Er sammelt alle in der Log-Datei enthaltenen Identifikatoren (siehe Abschnitt 5.2) und ordnet diese der jeweiligen *Konnektor*-Identifikationsnummer zu. Für jeden Dateneintrag kann mit dieser Information bestimmt werden, welche ID und welchen Namen die Applikation hatte, die diese Daten

versendet hat. Außerdem wird für jede Log-Datei bestimmt, welche Applikation der Autor dieser Log-Datei war[2].

Alle `PlaybackLogFileReader` verwenden einen gemeinsamen `PlaybackConnectorPool`. Dieser stellt für die verschiedenen Identifikatoren entsprechende *Konnektoren* bereit, die die ursprünglichen Applikationen repräsentieren. Identifikatoren mit demselben Namen wird dabei derselbe *Konnektor* zugeordnet, so können Daten aus verschiedenen Log-Dateien aber mit gleicher Quelle über denselben *Konnektor* versandt werden.

Die *Konnektoren* aus dem `PlaybackConnectorPool` werden dem *Router* als *entfernter Konnektor* hinzugefügt, für die GUI verhält es sich so, als wäre sie mit einer Roboter-Applikation verbunden.

#### 5.5.2.2 Synchronisierung

Der `LogManager` verwaltet alle `PlaybackLogFileReader` und fügt sie dem `Player` hinzu. Dazu benötigt er die jeweiligen Zeitdifferenzen. Ein `PlaybackLogFileReader` wird dabei grundsätzlich als Referenz verwendet.

Zur Bestimmung der Zeitdifferenzen für weitere `PlaybackLogFileReader` sammelt der `TimeSyncAnalyzer` alle Log-Einträge mit Informationen zu Zeitdifferenzen zwischen verschiedenen Applikationen. Der `TimeSyncAnalyzer` ist als Analyse-Modul in den `PlaybackLogFileReader` integriert. Der `LogFileManager` kombiniert die Informationen aus allen Log-Dateien und erstellt eine Datenstruktur, die es ermöglicht, die Zeitdifferenz zwischen zwei Applikationen direkt oder - falls nötig - indirekt - zu bestimmen. Eine Log-Datei bekommt die Zeitdifferenz im `Player` zugeordnet, die zwischen ihrem Autor und dem Autor der Referenz-Log-Datei besteht. Ist auch eine indirekte Bestimmung nicht möglich, wird die jeweilige Log-Datei nicht zum `Player` hinzugefügt.

#### 5.5.2.3 Visualisierung

Wird ein `PlaybackLogFileReader` vom `Player` durch den Aufruf von `trigger` aktiviert, so verarbeitet dieser alle Einträge ab der aktuellen Indexposition, deren Zeitstempel der Abspielzeit entspricht oder vor der Abspielzeit liegt.

Als erster Schritt wird die Datenquelle und der Daten-*Schlüssel* für einen solchen Eintrag bestimmt. Damit wird der entsprechende *Konnektor* aus dem `PlaybackConnectorPool` identifiziert. Für diesen *Konnektor* wird überprüft, ob Daten mit dem *Schlüssel* angefordert sind. Erst dann werden die Daten aus der Datei gelesen und über den *Konnektor* verschickt. Der *Router* verteilt die Daten dann weiter an die *Konnektoren* und Dialoge, die diese weiter verarbeiten oder visualisieren.

#### 5.5.2.4 Statische Daten

Manche Daten ändern sich während der Laufzeit der Roboter-Applikation nicht, werden aber für die Visualisierung ergänzender Daten benötigt oder sind in anderer Weise interessant. Solche Daten sollten nur einmalig aufgezeichnet werden, schon allein wegen der Ersparnis an Speicherplatz.

[2] Die Identifikationsnummer des *Konnektors* ist in diesem Fall gleich Null

Bei dem herkömmlichen Abspielvorgang würden solche Daten jedoch nur an die jeweiligen Dialoge verschickt werden, wenn genau der Zeitpunkt abgespielt wird, an dem die statischen Daten aufgenommen wurden. Um dieses Problem zu umgehen, kann man für das Abspielen statische *Schlüssel* registrieren. Der `StaticDataAnalyzer` ist ein weiteres Analyse-Modul im `PlaybackLogFileReader`, welches die Daten zu den statischen *Schlüsseln* speichert und diese immer verschickt, wenn sie angefordert sind.

### 5.5.3 Synchronisierung mit Videos

Für Videos, die nicht mit den intrinsischen Daten synchronisiert sind, wird eine eigene Abspielkomponente zur Verfügung gestellt. Für jede geladene Video-Datei wird diese in dem jeweiligen Fenster bereitgestellt (siehe Abbildung 5.13). Video und intrinsische Daten können so getrennt voneinander abgespielt werden. Zur Synchronisierung muss beim Abspielen von Video und intrinsischen Daten nach einer markanten Stelle gesucht werden, über die man beides einander zuordnen kann. Hat man eine entsprechende Stelle gefunden, pausiert man das Abspielen beider Abspielkomponenten an der jeweiligen Stelle. Eine Option in der Abspielkomponente des Videos ordnet die aktuellen Abspielzeiten der intrinsischen Daten der Abspielzeit des Videos zu. Das Video ist jetzt mit den Daten synchronisiert und kann nur noch über die gemeinsame Abspielkomponente gesteuert werden.

Um diesen Vorgang nur einmal durchführen zu müssen, wird die resultierende Zeitdifferenz zwischen Video und Log-Dateien automatisch in eine Datei gesichert. Dabei wird die Zeitdifferenz zu jeder geöffneten Log-Datei bestimmt und gespeichert, wenn die jeweilige Log-Datei eine gültige Zeitdifferenz zur Abspielzeit hat. So kann zu einem späteren Zeitpunkt das synchronisierte Video verwendet werden, auch wenn nur eine der beteiligten Log-Dateien geladen ist. Der Name der Datei, in der die Zeitdifferenzen gespeichert werden, entspricht dem Namen der Video-Datei mit der zusätzlichen Erweiterung `.offsets`. Da diese beim Laden der Video-Datei automatisch auch geladen wird, lassen sich die Informationen zur Synchronisierung der Video-Datei so einfach an dritte weitergeben.

In der Ansicht *Playback Configuration* (siehe Abbildung 5.8) hat man die Möglichkeit, die Zeitdifferenz der ausgewählten Video-Datei in Schritten von 100 Millisekunden nach vorne oder hinten zu korrigieren oder wieder zurückzusetzen. Dadurch lässt sich nach dem Synchronisieren eine Feineinstellung vornehmen.

## 5.6 Ereignisse

Ein Ereignis besteht aus einem Namen bzw. einer Beschreibung des Ereignisses und dem Ort und der Zeit an denen es aufgetreten ist. Dadurch ist es einfach, aber auch aussagekräftig. Zusätzlich kann ein Ereignis als globales Ereignis gekennzeichnet werden, was es ermöglicht, gleiche Ereignisse von unterschiedlichen Orten zusammenzufassen.

### 5.6.1 Ereignisse auslösen

Ein Ereignis kann in einer Roboter-Applikation einfach ausgelöst werden, indem ein Objekt vom Typ `Event` verschickt wird, beispielsweise über ein *Modul* oder einen *Konnektor*. Ein `Event` enthält


Abbildung 5.13: Video-Fenster mit eigener Abspielkomponente zur Synchronisierung (unten)

nur den Namen des Ereignisses und kann ein lokales oder globales Ereignis kennzeichnen. Ort und Zeit werden in diesem Fall automatisch bestimmt, der Ort ist die auslösende Applikation. Ist das Aufzeichnen von Daten in der Applikation aktiviert, so wird das `Event` automatisch mit aufgezeichnet.

Die zweite Methode Ereignisse auszulösen, ist diese aus anderen Daten zu generieren. Generatoren für Ereignisse implementieren das `IEvents` Interface und müssen in der `EventsRegistry` registriert werden.

### 5.6.2 Navigation mit Ereignissen

Die Ereignisse der geladenen Log-Dateien werden in einer Liste angezeigt (siehe Abbildung 5.14). Zusätzlich zu den aus den Daten generierten Ereignissen werden automatisch für den Anfang und das Ende jeder Log-Datei und jeder Video-Datei Ereignisse hinzugefügt.

In der Ereignis-Liste kann nach den verschiedenen Orten und Generatoren der Ereignisse gefiltert werden. Dadurch kann die Liste leicht auf die Ereignisse eingeschränkt werden, die gerade von Interesse sind. Die Ereignis-Liste kann mit dem `Player` synchronisiert werden, so dass immer das zuletzt aufgetretene Ereignis ausgewählt wird. Durch Doppelklick auf ein Ereignis in der Liste springt der `Player` an die entsprechende Position.

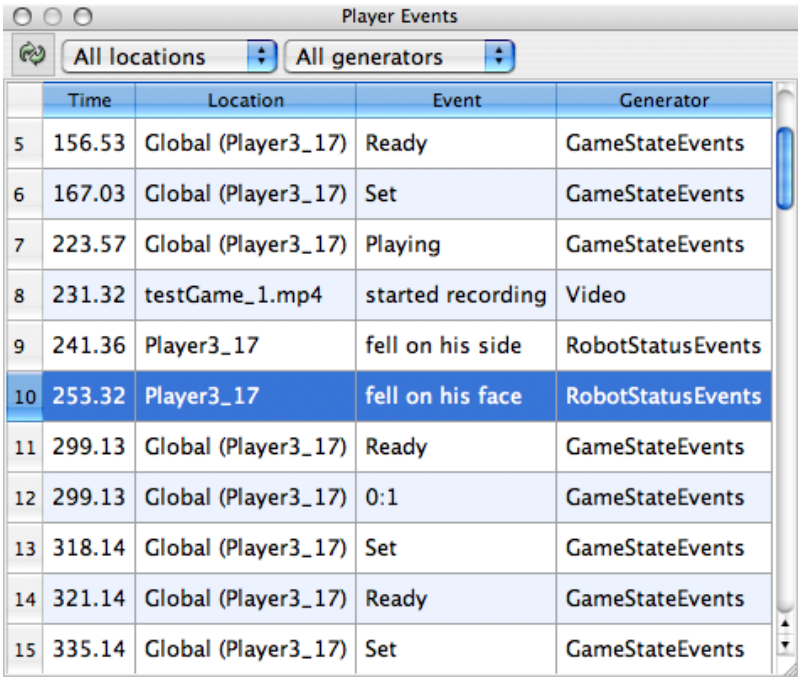


Abbildung 5.14: Ereignis-Liste mit Filter-Optionen (oben)

## 5.7 Fusion von Daten und Video

Die Video-Overlays werden über das `VideoWidget` verwaltet. Dieses stellt ein Menü zur Verfügung, in dem die einzelnen Overlays aktiviert oder deaktiviert werden können. Außerdem kann über das Menü, falls vorhanden, ein Konfigurationsdialog für ein Overlay geöffnet werden.

Im *Bearbeiten-Modus* erscheint ein Rahmen um die Overlay-Bereiche (siehe Abbildung 5.15). Mit der Maus können Overlays einfach während der Laufzeit verschoben und ihre Bereiche vergrößert oder verkleinert werden.

Overlays können statische und normale *Schlüssel* bestimmen, um die entsprechenden Daten zur Verfügung gestellt zu bekommen. Ein benutzerdefiniertes Overlay muss sich zum einen um eingehende Daten kümmern und zum anderen bei einem entsprechenden Aufruf seinen Zustand in den angewiesenen Bereich im Videobild zeichnen. Zum Zeichnen wird die *Qt Painter* Klasse verwendet, die einen umfangreiche Sammlung von Zeichenmethoden bietet, sowohl für Texte, als auch für Formen und Bilder.

### 5.7.1 Ereignis-Overlay

Das Ereignis-Overlay informiert innerhalb des Videobilds über aufgetretene Ereignisse (siehe Abbildung 5.16). Ein aufgetretenes Ereignis wird dabei jeweils für einige Sekunden eingeblendet. Die Ereignisse werden für das Ereignis-Overlay aus den abspielenden Daten generiert, deshalb können auch Zustands-Änderungen, die beim Springen an eine bestimmte Position in der Aufnahme geschehen, abgebildet werden.

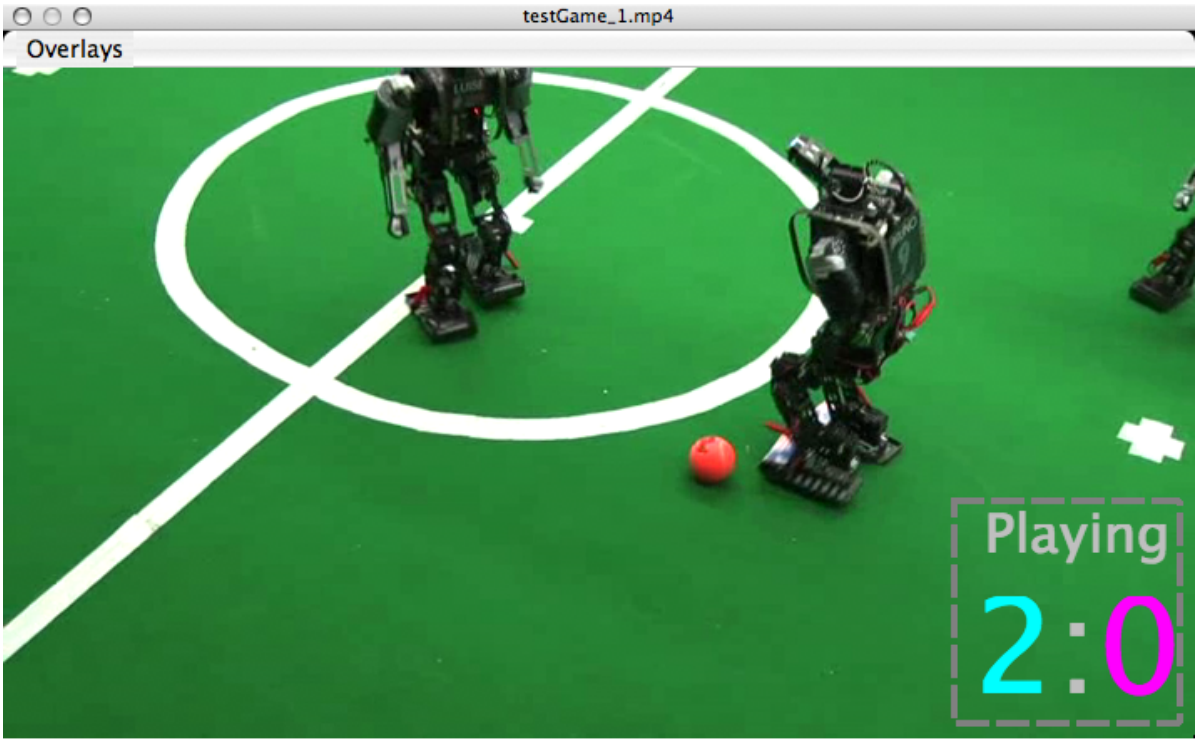


Abbildung 5.15: Video mit einem aktivierten Overlay im *Bearbeiten-Modus*

### 5.7.2 Export

Video und Overlays können nun zusammen in eine Video-Datei exportiert werden. Dazu müssen lediglich die gewünschten Overlays aktiviert und eine Zieldatei angegeben werden. Aus der Dateiendung wird der zu verwendende Video-Codec bestimmt. Standardmäßig wird als Auflösung des Videos die im entsprechenden Dialog sichtbare Größe des Videos verwendet. Die Auflösung kann jedoch, ebenso wie die Bitrate, auf einen benutzerdefinierten Wert gesetzt werden.

Der Export des Videos startet an der aktuellen Position des *Players*. Während des Exportvorgangs wird alle halbe Sekunde das zuletzt vorbereitete Video-Bild angezeigt (siehe Abbildung 5.17). Zusätzlich kann über die Zeitleiste der Fortschritt des Vorgangs mitverfolgt werden. Die Overlays können auch während des Exports aktiviert, deaktiviert oder umkonfiguriert werden. Dies wirkt sich ab dem Änderungszeitpunkt auch auf das exportierte Video aus.

Der Export kann zu einem beliebigen Zeitpunkt abgeschlossen werden oder endet automatisch wenn das Ende des Quell-Videos erreicht ist. Das fertige Video kann dann in herkömmlicher Software zum Abspielen von Videos betrachtet, präsentiert und analysiert werden (siehe Abbildung 5.18).

Während des Exports werden die anderen Visualisierungen nicht mit Daten versorgt. Nur die verwendeten Overlays erhalten die aktuell abgespielten Daten aus den Log-Dateien, diese werden direkt an die Overlays weitergegeben, um Videokodierung und das Zeichnen der Daten synchron zu halten.


Abbildung 5.16: Videobild mit Ereignis-Overlay (oben) und die Ereignis-Liste (unten)

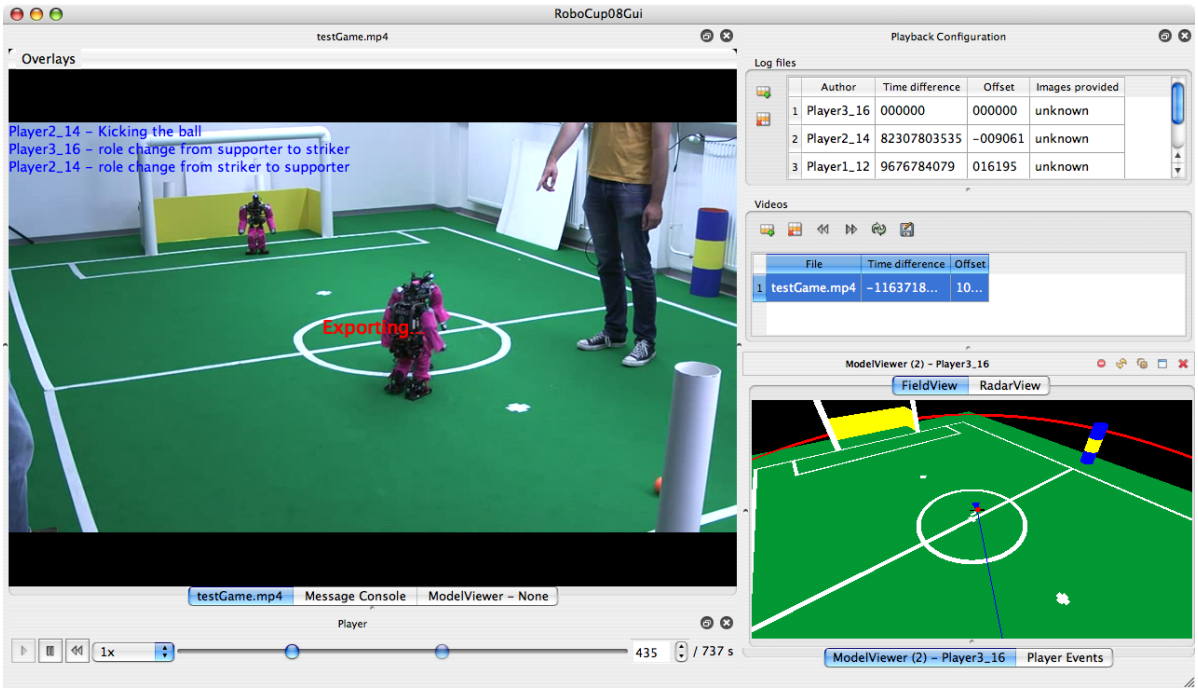


Abbildung 5.17: Export in eine Video-Datei.


Abbildung 5.18: Abspielen des exportierten Videos

| Zustand | Bedeutung |
|---|---|
| *Initial* | Spielvorbereitung, die Roboter haben Zeit, die Informationen zu Team-Farbe und Anstoß zu verarbeiten. |
| *Ready* | Die Roboter sollen sich auf ihre jeweiligen Position auf dem Spielfeld bewegen. |
| *Set* | Die Roboter sollten nun ihre Positionen erreicht haben und dürfen sich nicht mehr bewegen. |
| *Playing* | Das Spiel startet oder wird fortgesetzt. |
| *Finished* | Das Spiel ist beendet. |

Tabelle 5.1: Spielzustände im *RoboCup* (*Humanoid League*)

## 5.8 RoboCup-spezifische Erweiterungen

Für den Einsatz im *RoboCup* beim Team *Darmstadt Dribblers* wurden bereits einige Erweiterungen implementiert. Die implementierten Ereignisse dienen vor allem der einfacheren Synchronisierung von Daten und Videos.

### 5.8.1 Ereignisse

Die *GameStateEvents* geben Änderungen im Spielzustand (siehe Tabelle 5.1) und im Spielstand wieder. Außerdem können die Strafzeiten der einzelnen Roboter darüber verfolgt werden. Die Ereignisse zum Spielzustand eignen sich gut zur Synchronisierung der Videos, da bei bestimmten Zuständen alle Roboter gleichzeitig stehen bleiben oder anfangen sich zu bewegen. Die entsprechenden Stellen im Video lassen sich relativ einfach identifizieren.

Der `RobotStatus` enthält Angaben zur Lage eines Roboters, dem aktuellen Bewegungsmodul und den Batterien. Entsprechend informieren die *RobotStatusEvents* über den Zustand der Batterien und geben an, ob ein Roboter umgefallen ist.

### 5.8.2 ScoreOverlay

Das *ScoreOverlay* ist ein Beispiel für ein benutzerdefiniertes Overlay. Es erweitert das Videobild um die Anzeige des aktuellen Spielstands. Die Punktestände sind entsprechend der Team-Farbe eingefärbt (siehe auch Abbildung 5.15).

# 6 Ergebnisse

Die zu entwickelnde Software wurde schon während der Entwicklung auf der Architektur der *Darmstadt Dribblers* (siehe Abschnitt 3.2.1) eingesetzt und getestet. Durch das Konzept der virtuellen Roboter und der Filter-Schicht zur Unterstützung mehrerer Roboter können die vorhandenen Visualisierungen ohne Änderungen weiter verwendet werden. Dadurch bleibt die Handhabung der GUI für den Benutzer vertraut, zusätzlich stehen aber auch die neuen integrierten Funktionalitäten zur Verfügung.

## 6.1 Aufzeichnen der intrinsischen Daten

Die Aufzeichnung der intrinsischen Daten direkt auf dem Roboter ermöglicht eine Speicherung der Daten zur späteren Analyse. Dabei wird keine Netzwerkinfrastruktur benötigt und der Roboter kann völlig autonom agieren. Aufgrund der beschränkten Ressourcen ist es jedoch wichtig, dass der Vorgang des Aufzeichnens die Ausführung der eigentlichen Roboteranwendung nicht zu sehr beeinträchtigt.

Kritisch wird es vor allem bei großen Daten wie Kamerabildern. Die Daten werden innerhalb der Applikation kopiert um sie dem `LogFileConnector` zum Aufzeichnen verfügbar zu machen, danach werden sie in die Log-Datei auf dem Datenspeicher geschrieben. Ein weiteres Problem in diesem Zusammenhang kann die Geschwindigkeit des Datenspeichers selbst sein. Die *Compact Flash*-Speicherkarten, die bei den *Darmstadt Dribblers* eingesetzt werden unterstützen beispielsweise zum Teil nur eine Schreibgeschwindigkeit von bis zu 9 MB/s. Da die Kamerabilder unkomprimiert vorliegen, haben sie in diesem Fall (640x480 Pixel, YUV422[1]) etwa ein Größe von 600 KB. Dadurch ist bereits die theoretische maximale Bildrate beim Aufzeichnen auf etwa 15 Bilder pro Sekunde beschränkt.

Auf dem Roboter *Bruno* wurden verschiedene Aufnahmekonfigurationen unter jeweils möglichst gleichen Bedingungen getestet. Der Roboter stand dabei still, hat jedoch den Kopf mit der Kamera in einem sich wiederholenden Muster bewegt. Im Sichtfeld der Kamera waren je nach Kopfposition der Ball, ein Tor, Eckpfosten und Feldlinien. Die Aufnahmedauer für jeden der Testfälle betrug zwei Minuten. Tabelle 6.1 zeigt die Ergebnisse dieser Tests. Die Angaben beziehen sich auf den Prozess *Cognition* in der Roboterapplikation, der die komplette Kette von Datenerfassung über Bildverarbeitung und Modellierung bis hin zur Verhaltenssteuerung realisiert. Die Testspiel-Konfiguration fordert eine umfangreiche Mischung an Daten zur Analyse an, unter anderem die kompletten Verhaltensinformationen, alle fünf Sekunden ein Kamerabild, die Lokalisierungsinformationen, das Hindernismodell und verschiedene Perzepte von erkannten Objekten wie Ball, Tor oder Feldlinien. Die Log-Datei der Minimal-Konfiguration hingegen enthält nur die Daten, die automatisch mit aufgezeichnet werden.

Mit der Minimal-Konfiguration wird im Durchschnitt eine Verarbeitung von mehr als 14 Bildern pro Sekunde erreicht. Für die anderen Konfigurationen kann mit steigender Menge an angeforderten Daten auch eine steigende Leistungseinbuße festgestellt werden. Werden beispielsweise

[1] YUV-Farbmodell mit je vier Byte pro zwei Pixel

| Aufnahmekonfiguration | *Cognition*-Frequenz (Durchschnitt) | Maximale Ausführungsdauer | Verarbeitete Bilder | Aufgezeichnete Bilder | Datenverlust beim Aufzeichnen | Effektive Datenrate |
|---|---|---|---|---|---|---|
| Minimal-Konfiguration | 14,53 Hz | 134 ms | 1730 | - | - | 0,7 KB/s |
| Komplettes Verhalten | 13,64 Hz | 133 ms | 1626 | - | - | 14,5 KB/s |
| Jedes 15. Bild | 13,08 Hz | 364 ms | 1455 | 84 | 13% | 421,5 KB/s |
| Jedes fünfte Bild | 12,68 Hz | 435 ms | 1356 | 83 | 69% | 416,4 KB/s |
| Ein Bild alle fünf Sekunden | 13,75 Hz | 154 ms | 1639 | 24 | - | 121,1 KB/s |
| Ein Bild pro Sekunde | 13,10 Hz | 299 ms | 1440 | 84 | 39% | 421,5 KB/s |
| Alle Bilder | 10,59 Hz | 490 ms | 1160 | 73 | 94% | 366,1 KB/s |
| Testspiel-Konfiguration | 11,99 Hz | 418 ms | 1407 | 24 | - | 140,8 KB/s |

Tabelle 6.1: Performanz beim Aufzeichnen intrinsischer Daten

zusätzlich die kompletten Verhaltensinformationen aufgezeichnet, sinkt die Bildrate bereits um durchschnittlich etwa ein Bild pro Sekunde. Bei Verwendung der Testspiel-Konfiguration sinkt sie weiter auf etwa 12 Bilder pro Sekunde. Es ist hier also sehr wichtig abzuwägen, welche Daten sinnvoll für die Analyse sind und wie viel Leistungseinbuße dafür in Kauf genommen werden kann. Die konkrete Beeinträchtigung der Leistung hängt natürlich maßgeblich von dem jeweiligen System ab - ist die Rechenkapazität wie in diesem Fall ohnehin schon voll ausgelastet, hat das zwangsweise Einbußen zur Folge. Durch die verschiedenen Werkzeuge, die die entwickelte Software bereitstellt, ist es jedoch einfach, verschiedene Konfigurationen zu testen und die Beeinträchtigung der Leistung zu messen und zu bewerten.

Beim Aufzeichnen von Kamerabildern fällt auf, dass bei vielen angeforderten Bildern ein großer Teil nicht aufgezeichnet wird. In diesem Fall läuft die Warteschlange des `LogFileConnector` über, die Daten können nicht schnell genug geschrieben werden. Bei dem Versuch alle Kamerabilder aufzuzeichnen sind nur sechs Prozent der gesamten verarbeiteten Bilder in der Log-Datei, weniger als beim Aufzeichnen von nur jedem 15. Bild. Die Speicherung der Verhaltensdaten wurde im Zuge dieser Arbeit bereits optimiert, um möglichst wenig Speicher zu verbrauchen. Vor dieser Optimierung kam es auch beim Aufzeichnen des Verhaltens zu einem Überlauf in der Warteschlange.

## 6.2 Synchronisierung

Die Synchronisierung von intrinsischen Daten und externer Videodatei geschieht manuell. Das erlaubt das Verwenden von Videomaterial aus einer beliebigen Quelle, bedeutet aber einen zusätz-

lichen Aufwand bei der Vorbereitung der Daten für die Analyse. Um eine gute Benutzbarkeit der Software zu gewährleisten muss es möglich sein, mit relativ geringem Aufwand eine hinreichend gute Synchronisierung zu erreichen. Zur manuellen Synchronisierung muss ein bestimmtes Ereignis in den Daten und im Video identifiziert werden können. Dabei spielen die Benutzeroberfläche und die Daten selbst eine wichtige Rolle.

Es hat sich gezeigt, dass - bei Aufnahme geeigneter Daten - die manuelle Synchronisierung durch jemanden mit Kenntnis über die Bedeutung der Daten leicht vorgenommen werden kann. Da mehrere Roboter, die gleichzeitig im Einsatz sind, ohnehin untereinander Synchronisierungsinformationen austauschen, muss der Vorgang nur für einen der Roboter vorgenommen werden. Alternativ können natürlich auch die Daten von allen Robotern gleichzeitig dafür zu Rate gezogen werden. Außerdem reicht es aus, wenn der Vorgang nur ein einziges Mal für ein Video vorgenommen wird, die Synchronisierungsinformationen können zusammen mit dem Video an andere weitergegeben werden.

Im Kontext des *RoboCup Soccer* hat sich der Spielzustand (siehe Tabelle 5.1) als gute Basis für die Synchronisierung erwiesen. Die Roboter erhalten über WLAN in regelmäßigen Abständen Kontrollsignale mit dem aktuellen Spielzustand. Wurde der Spielzustand mit den intrinsischen Daten aufgezeichnet, so werden beim Laden der Log-Datei Ereignisse generiert, die die Übergänge zwischen verschiedenen Spielzuständen markieren. Über die Ereignis-Liste (siehe auch Abschnitt 5.6.2) kann direkt zu der Position in den Daten gesprungen werden, an der der Zustandswechsel stattfindet. Im Video lässt sich das Ereignis auch leicht identifizieren. Wenn beispielsweise vor Beginn des Spiels das erste Mal in den Zustand *Ready* gewechselt wird, fangen die zuvor stillstehenden Roboter alle an, sich auf ihre Positionen zu bewegen. Hat man das selbe Ereignis in Daten und Video erkannt, kann man die Synchronisierung vornehmen. Diese lässt sich durch die Feinabstimmung gegebenfalls korrigieren, indem man sich auf weitere Merkmale in Daten und Video stützt.

Die Visualisierung der Lokalisierungsinformationen, der erkannten Objekte oder der relativen Fußpositionen eines Roboters können sich auch gut für eine Synchronisierung eignen. Beispielsweise kann der erkannte Ball und die Änderung der Fußpositionen Aufschluss über das Ausführen eines Schusses geben, was im Video ebenfalls gut zu erkennen ist.

## 6.3 Einsatz beim RoboCup

Die entwickelten Erweiterungen kamen auf den *RoboCup German Open 2009* in Hannover und bei den Vorbereitungen auf den *RoboCup 2009* erstmals vollständig bei den *Darmstadt Dribblers* zum Einsatz. Bei nahezu allen Begegnungen der *German Open* wurden auf den Robotern Daten aufgezeichnet, die Spiele selbst wurden mit einer digitalen Videokamera festgehalten.

Durch die Kombination der Videos mit der Visualisierung der Lokalisierungsinformationen konnte die falsche Lokalisierung der Roboter nach einer Drehbewegung als Ursache für viele falsche Spielentscheidungen identifiziert werden (Vergleich Abbildung 1.1 und 1.2). Infolgedessen wurde die Lokalisierung nach den *German Open* angepasst, um die Sensordaten des Gyroskops mit einzubeziehen. Die folgenden Tests zeigten bei Gegenüberstellung von Video und Lokalisierung eine deutliche Verbesserung. Weiterhin wurde die Software um einige Ereignisse basierend auf den Verhaltensdaten erweitert. Dies erlaubt eine gezielte Analyse des Roboterverhaltens. Beispielsweise wird durch die Ereignisse zum Rollenwechsel der einzelnen Roboter die Analyse von kooperativem Verhalten vereinfacht. Es bleibt zu hoffen, dass die Verbesserungen im Vorfeld des *RoboCup* den

*Darmstadt Dribblers* die Möglichkeit geben, ihren zweiten Platz von den *German Open* noch zu verbessern.

# 7 Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde eine Erweiterung des Softwareframeworks *RoboFrame*[15] entwickelt, welche die Analyse und Identifikation von Fehlern in einem System mobiler autonomer Roboter entschieden erleichtert. Dazu werden die intrinsischen Daten der Roboter und die externe Sicht auf das Geschehen kombiniert. Als externe Sicht dienen Videodateien, die mit den in Log-Dateien enthaltenen aufgezeichneten Roboterdaten synchronisiert werden. In der GUI können Daten und Videos synchron abgespielt und visualisiert werden. Die Navigation durch die Daten kann anhand der Zeit oder bestimmten, in den Daten aufgetretenen, Ereignissen erfolgen. Zusätzlich lässt sich das Geschehen beschleunigt oder in Zeitlupe abspielen.

Die Aufnahme von Log-Dateien kann jetzt direkt durch die Roboterapplikation geschehen. Eine Konfiguration legt die aufzuzeichnenden Daten und verschiedene Optionen fest. Das Aufzeichnen der Daten kann auf Wunsch automatisch mit dem Starten der Applikation beginnen. Zum Lesen und Abspielen der Log-Dateien wurde eine Lösung entwickelt, die nur die Metainformationen im Speicher hält und nur die benötigten Daten aus der Datei liest. Zusätzlich können beim initialen Einlesen einer Log-Datei Analyse-Module angewandt werden, zum Beispiel zum Erstellen von Statistiken.

Durch das Einführen einer filternden Zwischenschicht in die Datenkommunikation wurde die Möglichkeit zum Verarbeiten und Visualisieren von Daten mehrerer Roboter in *RoboFrame* verbessert. Beim Abspielen von Roboterdaten wird für jede der unterschiedlichen Roboterapplikation ein virtueller Roboter simuliert. Für die Visualisierungen der GUI besteht kein Unterschied zwischen einem verbundenen echten Roboter und einem virtuellen. Deshalb können sie ohne Anpassungen auch für die abgespielten Daten verwendet werden.

Video-Overlays (siehe Abschnitt 4.5 und 5.7) dienen zur Fusion von Daten und Videos. Sie integrieren Informationen zu den intrinsischen Roboterdaten im Videobild. Das Video kann inklusive der Overlays wieder in eine Videodatei exportiert werden und ermöglicht so eine Analyse auch außerhalb von *RoboFrame*.

## 7.1 Mögliche Erweiterungen

Bei der entwickelten Software wurde, wie schon bei dem zugrunde liegenden Framework, großer Wert auf Erweiterbarkeit gelegt. So lassen sich eigene Video-Overlays, Ereignis-Generatoren oder Analyse-Module für Log-Dateien integrieren. In der Roboterapplikation lassen sich eigene Ereignisse einfach durch das Verschicken eines entsprechenden Datenpakets auslösen.

Die bereits vorhandenen Erweiterungen sind relativ einfach, es gibt viele Möglichkeiten, das Potential der Software besser zu nutzen. Natürlich gibt es auch einige Ansatzpunkte, an denen die Software selbst noch verbessert werden kann.

### 7.1.1 Verbesserter Video-Export

Bei der Verarbeitung von Videos fehlt bisher die Unterstützung für Audio. Dies wäre zum einen für den Export wünschenswert, zum Beispiel zum Präsentieren eines Videos, zum anderen kann es auch für die Synchronisierung mit den Roboterdaten nützlich sein.

Momentan kann nur ein zusammenhängender Teil eines Videos exportiert werden. Für viele Anwendungsfälle wäre es sicher sinnvoll, das exportierte Video aus verschiedenen Abschnitten zusammensetzen zu können. Diese Abschnitte könnten mit unterschiedlich Overlays konfiguriert sein oder jeweils aus verschiedenen Video-Dateien stammen.

### 7.1.2 Konfiguration von Ereignissen

Ereignisse können entweder in einer Applikation ausgelöst oder aus anderen Daten generiert werden. Alle Ereignisse werden in der Ereignis-Liste oder im Ereignis-Overlay angezeigt. Oft sind jedoch nicht alle Ereignisse auch wirklich interessant. Zwar bietet die Ereignis-Liste die Option zur Filterung nach Generator oder Ort des Ereignisses, diese wirkt sich aber nicht auf das Ereignis-Overlay aus. Außerdem kann auch innerhalb der Ereignisse eines Generators nur eine bestimmte Art von Ereignissen interessant sein, besonders wenn viele andersartige Ereignisse auftreten. Besonders im Ereignis-Overlay wäre es wünschenswert, wirklich nur die relevanten Ereignisse angezeigt zu bekommen. Um dieses Problem zu lösen könnte man zum Beispiel die Möglichkeit geben, bestimmte Generatoren zu aktivieren oder zu deaktivieren, sowie Optionen für einen Generator festzulegen.

### 7.1.3 Komplexe Overlays

Die bisher implementierten Overlays blenden hauptsächlich Textinformationen in das Videobild ein. Im Vergleich zu den zum Teil sehr ausgefeilten Visualisierungen in der GUI gehen bei einer Textrepräsentation viele Informationen verloren oder sind viel weniger anschaulich. Stattdessen könnten zum Beispiel erkannte Objekte, wie der Ball oder ein Hindernis, an die korrekte Position relativ zum Roboter in das Videobild gezeichnet werden. Eine andere mögliche Anwendung wäre die Anzeige der gedachten Positionen der verschiedenen Teammitglieder oder ein Hinweis auf die Position eines Roboters der momentan nicht im Videobild zu sehen ist.

Um ein derartiges Overlay zu entwickeln muss einer Position im Videobild eine Position im Raum zugeordnet werden können und - in einem bestimmten Bereich - auch umgekehrt. Dies lässt sich lösen, wenn man die Position und Orientierung der Kamera relativ zum Spielfeld kennt. Dies würde jedoch nur mit einer statisch montierten Kamera funktionieren. Ein anderer Ansatz wäre eine Objekterkennung im Videobild durchzuführen, um so die entsprechende Transformation zu bestimmen.

### 7.1.4 Kamerabild-Overlay

Eine weitere mögliche Anwendung für Overlays wäre die Einblendung des Kamerabilds eines oder mehrerer Roboter. Dafür müssten jedoch zuerst genügend Daten zur Verfügung stehen. Aktuell ist nur die Aufzeichnung von einem Bild alle paar Sekunden möglich. Für den Fall, dass zusätzliche

Rechenleistung in der Applikation zur Verfügung steht, könnten die Bilder dort in ein platzsparenderes Format konvertiert werden um diese in der Log-Datei abzulegen.

# A Verwendung und Erweiterung der Software

## A.1 Aufnahme

Bei der Aufnahme der intrinsischen Daten ist die Auswahl der Daten entscheidend. Wichtig ist es einerseits, solche Daten aufzuzeichnen, die sich gut zur Synchronisierung mit den Videos eignen. Diese Daten können Ereignisse markieren, die sich im aufgezeichneten Video leicht erkennen und identifizieren lassen. Zusätzlich sollten Daten aufgezeichnet werden, die für die Analyse interessant sind.

Der Dialog *RemoteLog* (siehe Abbildung 5.2 auf Seite 34) ermöglicht das Erstellen einer Konfiguration zur Aufnahme von intrinsischen Daten. Die Konfiguration kann als Datei abgespeichert werden oder direkt an eine verbundene Roboter-Applikation verschickt werden. Um eine Konfiguration automatisch beim Start einer Applikation zu laden, muss die Konfigurationsdatei unter dem Namen `autolog.conf` im Verzeichnis der Applikation abgelegt werden. Ist die Konfiguration aktiviert, wird in diesem Fall nach dem Applikations-Start direkt mit der Aufnahme der Log-Datei begonnen.

Zusätzlich oder alternativ zur Aufnahme auf den Robotern kann auch der Dialog *LogRecorder* verwendet werden, um Daten über die GUI aufzuzeichnen.

In den aufgenommenen Video-Dateien sollte möglichst das gesamte Verhalten aller Roboter dokumentiert sein. Mit zunehmender Anzahl von Video-Dateien steigt der Aufwand zur Synchronisierung mit den intrinsischen Daten. Es macht also durchaus Sinn, das Geschehen am Stück aufzunehmen, anstatt viele kurze Aufnahmen zu machen. Allerdings kann es sich lohnen, das Geschehen aus unterschiedlichen Perspektiven aufzuzeichnen. Dies bedeutet zwar auch einen Mehraufwand bei der Synchronisierung, bei der Analyse können dann jedoch beide Perspektiven gleichzeitig zu Rate gezogen werden.

## A.2 Abspielen und Analyse

Der erste Schritt zur Analyse der aufgenommenen Daten ist das Laden der entsprechenden Log- und Video-Dateien. Diese können in der Ansicht *Playback Configuration* (siehe Abbildung 5.8 auf Seite 41) hinzugefügt werden.

Die erste geladene Log-Datei wird als Referenz für die Synchronisierung der anderen Dateien verwendet. Die Spalten *Time difference* und *Offset* geben die Zeitdifferenz und die Abweichung des Startzeitpunkts zu dieser Referenz-Datei an. Erscheint bei einer Datei der Wert *N/A*, so ist keine entsprechende Synchronisierungsinformation vorhanden und die Datei wird nicht in das Abspielen integriert.

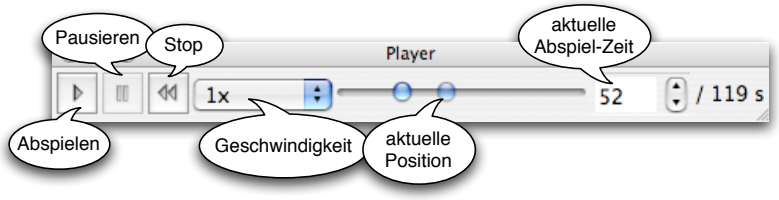


Abbildung A.1: Steuerung des Abspielvorgangs mit dem *Player*

### A.2.1 Steuerung des Abspielvorgangs

Die Steuerung des Abspielvorgangs erfolgt über den *Player* (siehe Abbildung A.1). Nur für Videos, die noch nicht mit den Log-Dateien synchronisiert sind, wird eine eigene Abspielkomponente zur Verfügung gestellt (siehe Abschnitt A.2.2).

Der Player erlaubt es unter anderem, den Abspielvorgang zu pausieren oder die Abspielgeschwindigkeit zu regeln. Außerdem ist es möglich, an eine bestimmte Abspielposition zu springen. Dazu kann entweder die entsprechende Position in der Zeitleiste angeklickt oder der gewünschte Abspiel-Zeitpunkt eingetragen und mit der Eingabetaste bestätigt werden.

Für eine einfachere Navigation innerhalb der Daten kann die Ereignis-Liste verwendet werden.

Zur Verbesserung der Leistung beim Abspielen kann eine Video-Datei bereits im Vorfeld in eine geringere Auflösung oder ein anderes Format konvertiert werden. Auch durch die Verkleinerung der Video-Dialoge kann ein Leistungsgewinn erfolgen.

### A.2.2 Synchronisierung von Videos

Zum Synchronisieren eines Videos mit den Daten muss sowohl im Video als auch in den Daten eine Position gefunden werden, die mit der jeweils anderen übereinstimmt. Dazu genügt es, zuerst nur eine relativ grobe Übereinstimmung zu finden. Beide Abspielkomponenten müssen an den jeweiligen Positionen pausiert werden, dann können beide synchronisiert werden (siehe Abbildung A.2).

Jetzt können Video und Daten gemeinsam über den *Player* gesteuert werden. Zur Feinabstimmung der Synchronisierung kann diese in Schritten von 100 Millisekunden angepasst werden (siehe auch Abbildung A.3). So kann behoben werden, dass das Video beim Abspielen merklich hinter den Daten voraus ist oder zurück liegt.

## A.3 Konfiguration zur Visualisierung

Bei der Verarbeitung von Daten mehrerer Roboter müssen die einzelnen Dialoge zur Visualisierung der Daten konfiguriert werden, von welchen Robotern bzw. Applikationen sie Daten empfangen sollen.

Der *FilterDialog* (siehe auch Abbildung 5.4) bietet die Möglichkeit, diese Konfiguration für alle geöffneten Dialoge vorzunehmen. Es gibt zwei verschiedene Arten von Dialogen - Dialoge die nur Daten einer Applikation sinnvoll verarbeiten können und Dialoge die auch mit Daten von mehreren Applikationen umgehen können. Entsprechend ist der *FilterDialog* in *"Singles"* und *"Multis"* unterteilt. Die Einstellungen, von wo ein Dialog Daten empfängt und wohin er Daten


Abbildung A.2: Synchronisation des Videos: Grobe Übereinstimmung von Video und Daten

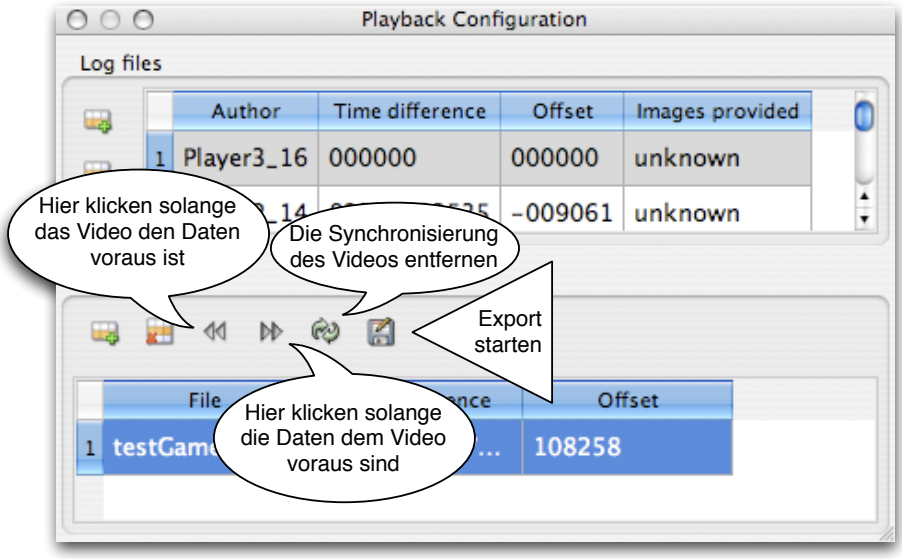


Abbildung A.3: Video-Optionen: Feinabstimmung der Synchronisation, Entfernen der Synchronisation und Export in eine Video-Datei

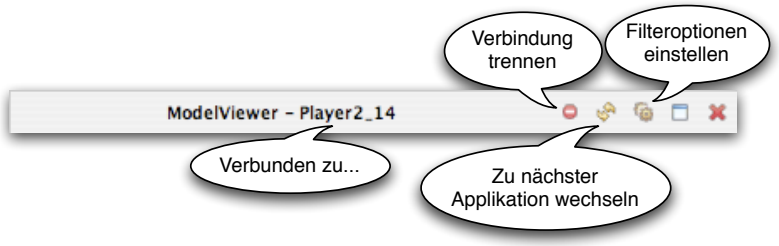


Abbildung A.4: Titelleiste mit Aktionen zu den Filtereinstellungen des Dialogs

sendet, sind unabhängig voneinander. Da diese jedoch im Normalfall gleich sind, wird in diesem Fall bei einer Änderung des Datenempfängers auch der potenzielle Datensender entsprechend gesetzt. In der Titelleiste eines Dialogs steht, für welche Applikationen er für den Empfang von Daten konfiguriert ist.

Zusätzlich zum *FilterDialog* werden weitere Methoden bereitgestellt, die eine einfachere Konfiguration eines Dialogs ermöglichen. Wenn ein Dialog als *DockWidget* im Hauptfenster integriert ist, werden diese als zusätzliche Buttons in der Titelleiste des Dialogs bereitgestellt (siehe Abbildung A.4).

Ist ein Dialog vom Hauptfenster abgetrennt stehen diese Methoden nur über Tastaturkürzel zur Verfügung, welche wiederum für die *DockWidgets* nicht verfügbar sind. Das Kürzel *Ctrl+Alt+F* öffnet das Fenster mit den Filteroptionen des Dialogs, das Kürzel *Ctrl+Alt+N* wechselt den Filter zur nächsten vorhandenen Applikation.

## A.4 Eigene Ereignisse

Ereignisse können entweder in einer Applikation ausgelöst oder aus anderen Daten generiert werden.

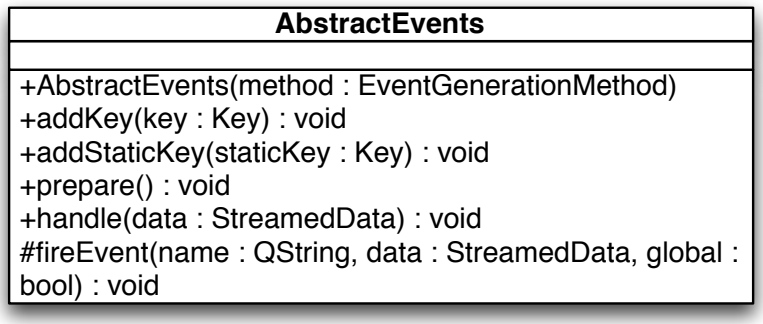


Abbildung A.5: Für Unterklassen relevante Methoden in `AbstractEvents`

Um ein Ereignis auszulösen muss lediglich ein Objekt vom Typ `Event` erstellt und mit dem Schlüssel `FrameworkKeys::event` verschickt werden. Zu einem Ereignis muss dessen Name bzw. Beschreibung angegeben werden, sowie ob es ein globales Ereignis ist.

### A.4.1 Ereignisse generieren

Für die Umsetzung eigener Ereignisse muss eine benutzerdefinierte Generator-Klasse implementiert werden. Dazu sollte auf der Klasse `AbstractEvents` aufgebaut werden. Diese stellt bereits die Verwaltung der Daten-*Schlüssel* bereit und setzt die verschiedenen Methoden zur Generierung von Ereignissen um.

Die Generator-Klasse muss im Konstruktor über den Aufruf der Methoden `addKey` bzw. `addStaticKey` über Angabe der jeweiligen *Schlüssel* die Daten definieren, die sie zur Generierung der Ereignisse benötigt. In der `handle`-Methode können diese Daten verarbeitet und mit Hilfe der Methode `fireEvent` ein Ereignis ausgelöst werden. Durch Überschreiben der Methode `prepare` können vor dem ersten Aufruf von `handle` vorbereitende Maßnahmen getroffen werden. Bei der Implementierung von `handle` ist zu beachten, dass die eingehenden Daten auch von unterschiedlichen Robotern stammen können, dieser Fall ist entsprechend zu behandeln. Über den `IConnectorContext` kann zu einer Datenquelle der entsprechende Applikationsname bestimmt werden. Abbildung A.5 zeigt die Definitionen der relevanten Methoden aus `AbstractEvents`.

Beispiele für Ereignis-Generatoren sind die Klassen `RobotStatusEvents` und `GameStateEvents` (siehe auch Abschnitt 5.8.1).

Um einen Ereignis-Generator zu aktivieren muss dieser in der `EventsRegistry` registriert werden. Dazu wird ein `IEventsWrapper` benötigt, der Instanzen des Generators erstellen kann und bestimmt, ob dieser standardmäßig aktiviert ist und unter welchem Namen er registriert wird. Listing A.1 zeigt beispielhaft die Registrierung zweier Ereignis-Generatoren. Der `KeyEventsWrapper` übergibt zusätzlich noch einen Schlüssel an den erstellten Generator.

## A.5 Eigene Overlays

Um ein eigenes Video-Overlay zu implementieren muss man von der Klasse `VideoOverlay` erben. Abbildung A.6 zeigt die Definitionen der relevanten Methoden und Attribute der Klasse `VideoOverlay`. Die einzige Methode, die implementiert werden muss, ist die Methode `paint`. Hier erfolgt

Listing A.1: Registrieren von Ereignissen in der Klasse `RoboCup08Events`

```
21 void RoboCup08Events::registerEvents() {
22   // GameStateEvents | default enabled
23   EventsRegistry::registerEvents(new EventsWrapper<gamecontroller::GameStateEvents
       <GameStateKey> >("GameStateEvents", true));
24
25   // RobotStatusEvents | default enabled
26   EventsRegistry::registerEvents(new KeyEventsWrapper<robocup::motion::
       RobotStatusEvents>(ROBOT_STATUS, "RobotStatusEvents", true));
27 }
```

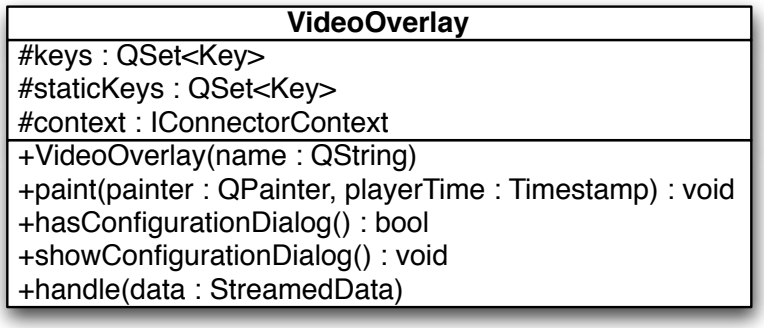


Abbildung A.6: Für Unterklassen relevante Methoden und Attribute in `VideoOverlay`

das Zeichnen über das Video-Bild. Der Bereich des *Qt Painter* auf den gezeichnet werden kann, ist bereits auf den Bereich beschnitten, der für das Overlay vorgesehen ist. Um die Berücksichtigung eines zeitlichen Verlaufs zu ermöglichen, wird die aktuelle Abspielzeit beim Aufruf von `paint` mit übergeben.

Overlays, die auf der Verarbeitung intrinsischer Daten basieren, müssen zusätzlich die `handle`-Methode überschreiben und die entsprechenden *Schlüssel* zu den Mengen `keys` bzw. `staticKeys` hinzufügen. Die Methode `handle` wird immer dann aufgerufen, wenn Daten eines der registrierten Schlüssel vorliegen. Schlüssel in `staticKeys` werden nur einmalig angefordert. Der `IConnectorContext` kann dazu verwendet werden, die Quelle der Daten zu identifizieren.

Um ein Overlay verwenden zu können, muss es in der GUI registriert werden. Dazu wird eine Factory benötigt, die Instanzen des Overlays erstellen kann und es mit benötigten Einstellungen konfiguriert. Die Factory-Klasse muss `IVideoOverlayFactory` implementieren. Es gibt bereits zwei Implementierungen, `VideoOverlayFactory` und `KeyVideoOverlayFactory`. Letztere erlaubt es, der `VideoOverlay`-Instanz einen *Schlüssel* zu übergeben. Listing A.2 zeigt eine Beispielkonfiguration, in der das `ScoreOverlay` registriert wird.

## A.6 Eigene Analysemodule

Analysemodule für Log-Dateien ermöglichen es, einfache Statistiken über die Daten in einer Log-Datei zu erstellen. In der Tabelle der Log-Dateien wird für jedes benutzerdefinierte Analysemodul eine zusätzliche Spalte bereitgestellt (siehe auch Abbildung 5.12 auf Seite 45). Das Analysemodul bestimmt den angezeigten Wert und den Tooltip. Für die Implementierung eines eigenen Analyse-

Listing A.2: Beispielkonfiguration für die Wiedergabe von Daten und Videos

```
void RoboCup08Gui::configurePlayback(ILogPlaybackConfiguration& logPlayback,
    IVideoPlaybackConfiguration& videoPlayback) {
  GuiApplication::configurePlayback(logPlayback, videoPlayback);

  // Register events
  RoboCup08Events::registerEvents();

  // add static keys
  logPlayback.registerStaticKey(BEHAVIOR_SYMBOL_STRINGS);
  logPlayback.registerStaticKey(BEHAVIOR_BEHAVIOR_STRINGS);
  logPlayback.registerStaticKey(YUV_COLOR_TABLE_TO_GUI);

  // add custom analyzers
  logPlayback.addAnalyzerFactory(new KeyLogEntryAnalyzerFactory<
      ImageInformationAnalyzer>("Images provided", IMAGE_INFORMATION));

  // add overlays
  videoPlayback.addVideoOverlay(new KeyVideoOverlayFactory<gamecontroller::
      ScoreOverlay, GameStateKey>(0.01f, 0.8f, 0.35f, 0.2f), true);
}
```

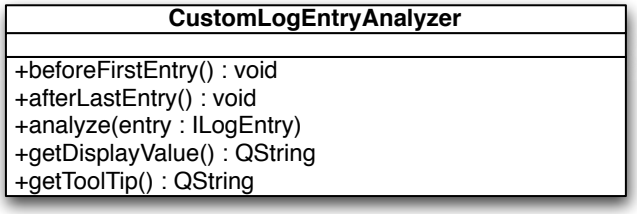


Abbildung A.7: Relevante Methoden für die Implementierung eines Analysemoduls für Log-Dateien.

moduls muss eine Klasse erstellt werden, die von `CustomLogEntryAnalyzer` erbt. Abbildung A.7 zeigt die Definitionen der für die Implementierung relevanten Methoden. Die Methode `analyze` wird für jeden Eintrag einer Log-Datei aufgerufen, das Analysemodul hat hier über den `ILogEntry` die Möglichkeit, auf Metainformationen und Inhalt eines Eintrags zuzugreifen. Vor dem ersten bzw. nach dem letzten Aufruf von `analyze` wird `beforeFirstEntry` bzw. `afterLastEntry` aufgerufen.

Ähnlich zu den Overlays muss zur Verwendung des Analysemoduls eine Factory registriert werden (siehe Listing A.2). Die Factory-Klasse muss die abstrakte Klasse `CustomLogEntryAnalyzerFactory` erweitern. Die `KeyLogEntryAnalyzerFactory` übergibt jeder Instanz eines Analysemoduls im Konstruktor einen *Schlüssel*.

## Abbildungsverzeichnis

## Listings

# Tabellenverzeichnis

## Literaturverzeichnis


[1] Robocup soccer humanoid league rules and setup for the 2009 competition in graz, austria. http://www.tzi.de/humanoid/pub/Website/Downloads/HumanoidLeagueRules2009.pdf, March 2009.

[2] MSL Technical Committee 1997-2009. Middle size robot league rules and regulations for 2009. http://www.er.ams.eng.osaka-u.ac.jp/robocup-mid/index.cgi?action=ATTACH&page=Rules+and+Regulations&file=msl%2Drules%2D2008%2D12%2D12%2Epdf, December 2008. Version - 13.1 20081212.

[3] Micha Andriluka, Martin Friedmann, Stefan Kohlbrecher, Johannes Meyer, Karen Petersen, Christian Reinl, Peter Schauß, Paul Schnitzspan, Armin Strobel, Dirk Thomas, and Oskar von Stryk. Robocuprescue 2009 - robot league team: Darmstadt rescue robot team (germany). Technical report, Technische Universität Darmstadt, 2009.

[4] David Becker, Jörg Brose, Daniel Göhring, Matthias Jüngel, Max Risler, and Thomas Röfer. Germanteam 2008 - the german national robocup team. Technical report, DFKI Bremen, TU Darmstadt, HU Berlin, 2008.

[5] M. Friedmann, K. Petersen, S. Petters, K. Radkhah, D. Thomas, and O. von Stryk. Darmstadt dribblers: Team description for humanoid kidsize league of robocup 2008. Technical report, Technische Universität Darmstadt, 2008.

[6] Michita Imai Takeshi Maeda Takayuki Kanda Ryohei Nakatsu Hiroshi Ishiguro, Tetsuo Ono. Robovie: an interactive humanoid robot. *Industrial Robot: An International Journal*, 28:498–504, 2001.

[7] Jesse Gutierrez Hurdus. A portable approach to High-Level behavioral programming for complex autonomous robot applications. http://scholar.lib.vt.edu/theses/available/etd-05082008-133149/, June 2008.

[8] H. Kitano, M. Asada, I. Noda, and H. Matsubara. RoboCup: robot world cup. *Robotics & Automation Magazine, IEEE*, 5(3):30–36, 1998.

[9] Tijn Kooijmans, Takayuki Kanda, Christoph Bartneck, Hiroshi Ishiguro, and Norihiro Hagita. Interaction debugging: an integral approach to analyze human-robot interaction. In *HRI '06: Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*, pages 64–71, New York, NY, USA, 2006. ACM.

[10] M. Lötzsch, M. Risler, and M. Jüngel. Xabsl - a pragmatic approach to behavior engineering. In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, pages 5124–5129, Beijing, China, October 9-15 2006.

[11] D. Mills. Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI. http://portal.acm.org/citation.cfm?id=RFC2030, 1996.

[12] S. Petters, D. Thomas, M. Friedmann, and O. von Stryk. Multilevel testing of control software for teams of autonomous mobile robots. In S. Carpin et al., editor, *Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2008)*, number 5325 in Lecture Notes in Artificial Intelligence, pages 183–194. Springer, November 2008.

[13] S. Petters, D. Thomas, and O. von Stryk. Roboframe - a modular software framework for lightweight autonomous robots. In *Proc. Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware of the 2007 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, San Diego, CA, USA, Oct. 29 2007.

[14] Max Risler. *Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines*. PhD thesis, Technische Universität Darmstadt, May 15 2009.

[15] Sebastian Petters und Dirk Thomas. Roboframe - softwareframework für mobile autonome robotersysteme. Master's thesis, TU Darmstadt, FB Informatik, 2005.