Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research Report
RR-10-01

Schlussbericht des Projektes SAMS

Christoph Lüth

March 2010

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-67608 Kaiserslautern
Tel.: + 49 (631) 205 75-0
Fax: + 49 (631) 205 75-503

Stuhlsatzenhausweg 3
D-66123 Saarbrücken
Tel.: + 49 (681) 302-5151
Fax: + 49 (681) 302-5341

Robert-Hooke-Str. 5
D-28359 Bremen, Germany
Tel.: +49 (421) 218-64 100
Fax: +49 (421) 218-64 150

E-Mail: info@dfki.de WWW: http://www.dfki.de

Deutsches Forschungszentrum für Künstliche Intelligenz

DFKI GmbH

German Research Center for Artificial Intelligence

Founded in 1988, DFKI today is one of the largest nonprofit contract research institutes in the field of innovative software technology based on Artificial Intelligence (AI) methods. DFKI is focussing on the complete cycle of innovation — from world-class basic research and technology development through leading-edge demonstrators and prototypes to product functions and commercialization.

Based in Kaiserslautern, Saarbrücken and Bremen, the German Research Center for Artificial Intelligence ranks among the important “Centers of Excellence” worldwide.

An important element of DFKI’s mission is to move innovations as quickly as possible from the lab into the marketplace. Only by maintaining research projects at the forefront of science can DFKI have the strength to meet its technology transfer goals.

The key directors of DFKI are Prof. Wolfgang Wahlster (CEO) and Dr. Walter Olthoff (CFO).

DFKI’s research departments are directed by internationally recognized research scientists:

- Image Understanding and Pattern Recognition (Director: Prof. T. Breuel)
- Knowledge Management (Director: Prof. A. Dengel)
- Deduction and Multiagent Systems (Director: Prof. J. Siekmann)
- Language Technology (Director: Prof. H. Uszkoreit)
- Intelligent User Interfaces (Prof. Dr. Dr. h.c. mult. W. Wahlster)
- Institute for Information Systems at DFKI (Prof. Dr. P. Loos)
- Robotics (Prof. F. Kirchner)
- Safe and Secure Cognitive Systems (Prof. B. Krieg-Brückner)

and the associated Center for Human Machine Interaction (Prof. Dr.-Ing. Detlef Zühlke)

In this series, DFKI publishes research reports, technical memos, documents (eg. workshop proceedings), and final project reports. The aim is to make new results, ideas, and software available as quickly as possible.

Prof. Wolfgang Wahlster
Director

Schlussbericht des Projektes SAMS

Christoph Lüth

DFKI-RR-10-01

Das diesem Bericht zugrunde liegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung und Forschung unter dem Förderkennzeichen 01 IMF 02 A gefördert.
Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

ISSN 0946-008X

Schlussbericht des Projektes SAMS

Christoph Lüth

16. März 2010

Zusammenfassung

Im Projekt *SAMS (Sicherungskomponente für Autonome Mobile Serviceroboter)* ist eine Komponente zur Berechnung von Schutzfeldern in Abhängigkeit von der Geschwindigkeit und dem Lenkwinkel eines sich bewegenden autonomen Fahrzeugs oder Serviceroboters entwickelt, implementiert, und für den Einsatz bis zu einem Sicherheitslevel (SIL) 3 geeignet zertifiziert worden. Das Schutzfeld überdeckt die beim Bremsen mit der momentanen Geschwindigkeit bis zum Stillstand überstrichene Fläche zuzüglich normativer Sicherheitszuschläge. Durch die Überwachung dieser Fläche mit einem Laserscanner kann die Sicherheit gegen Kollisionen mit statischen Hindernissen garantiert werden. Die Ergebnisse des Projektes sind zusammengefasst

(i) die Entwicklung eines einfach konfigurierbaren, konservativen Bremsmodells für autonome Fahrzeuge und mobile Serviceroboter,

(ii) ein als Sicherheitsfunktion nach der Norm IEC 61508:3 bis zu SIL 3 zertifizierter Algorithmus zur Berechnung von Schutzfeldern, und

(iii) eine Verifikationsumgebung, welche zur Spezifikation und Verifikation von MISRA-C-Software gemäß IEC 61508:3 bis zu SIL 3 verwendet werden kann.

Die Normenkonformität wurde durch den TÜV Süd schriftlich in *letters of conformance* bestätigt. Kern der Zertifizierung ist die formale mathematische Modellierung und der Korrektheitsbeweis der Implementierung mit dem computergestützten Theorembweiser Isabelle. Die entwickelten Techniken zur Verifikation algorithmisch orientierter Programme sind nicht auf die konkrete Anwendungsdomäne beschränkt, und für weitere Anwendungen einsetzbar.

Inhaltsverzeichnis

I Kurzdarstellung

I.1 Aufgabenstellung

Ziel des Projektes SAMS war die Entwicklunge einer verifizierten und zertifzierten Komponente für die sichere Servicerobotik, welche durch Auswerten der Entfernungsmessdaten eines Sicherheits-Laserscanners einen mobilen Roboter durch gezielte und gesteuerte Geschwindigkeitsreduktion rechtzeitig vor einem Hindernis stoppt. Diese prinzipielle Funktionalität, kurz *Fahrwegsicherung* genannt, ist für nicht-sichere Forschungsprototypen von Servicerobotern Stand der Technik; vor dem Einsatz beim Endanwender liegt jedoch die Hürde einer sicherheitstechnischen Zulassung, beispielsweise durch den TÜV. Ähnlich ist die Situation bei industriell eingesetzten fahrerlosen Transportsystemen (FTS), die auch die Basis für Serviceroboter in der Produktion bilden. Dort überwacht ein Laserscanner als zugelassenes Sicherheits-Bauteil eine feste Sicherheitszone um das Fahrzeug und leitet Maßnahmen zum sicheren Halt des Fahrzeugs ein, wenn sich ein Hindernis in der Sicherheitszone (dem *Schutzfeld*) befindet. Die mangelnde Flexibilität und eng begrenzte Anzahl solch fester Zonen schränkt Bahnführung und Geschwindigkeit unnötig ein, und verhindert in vielen Fällen eine effiziente und kostengünstige Bahnführung.

Das Projekt hat sich daher zum Ziel gesetzt, eine zulassungsfähige Fahrwegsicherung für Serviceroboter und FTS zu entwickeln, die mit einem zertifizierten Sicherheits-Laserscanner die überwachte Sicherheitszone dynamisch der Geschwindigkeit und der gefahrenen Kurve anpasst. Kernvorhaben war dabei die formal mathematische Modellierung und der Korrektheitsbeweis der Implementierung mit einem computergestützten Beweissystem. Auf Grundlage des Beweises und mit zusätzlichen, normativen Tests sollte die Zulassungsfähigkeit durch ein TÜV-Gutachten nachgewiesen werden.

I.2 Voraussetzungen

Zum Zeitpunkt des Projektstartes wurde der Forschungsbereich *Sichere Kognitive Systeme* (SKS) als Forschungsgruppe am (ebenfalls neuen) DFKI Labor Bremen aus dem Bremer Institut für Sichere Systeme (BISS) an der Universität Bremen heraus neu gegründet. Gründungsmitglieder waren viele Mitglieder der Arbeitsgruppe Krieg-Brückner an der Universität Bremen. Dadurch konnten auf die langjährigen Erfahrungen des BISS bei formalen Methoden und Werkzeugen im Bereich der sicherheitskritischen eingebetteten Systeme zurückgegriffen werden; als Referenzprojekte seien hier die International Space Station, die Innenkommunikation im Airbus A380 oder Projekte aus der Bahnsteuerung genannt.

Die Universität Bremen, und damit auch der Forschungsbereich Sichere Kognitive Systeme seit seinen Anfängen, sind und waren international eine der wenigen Institutionen, die die Verbesserung der Sicherheit in der Robotik durch den Einsatz formaler Methoden untersuchen, wie im DFG-Projekt SafeRobotics (2001– 2004) am Beispiel

des Bremer autonomen Rollstuhls Rolland. Hierbei wurde u.a. das zentrale Verfahren zur Sicherung des Rollstuhls (automatisches Anhalten mit Hilfe eines dynamischen Schutzfeldes) betrachtet.

Die fachlichen Voraussetzungen waren damit bei Projektstart exzellent, aber auf Grund der Aufbausituation am Forschungsbereich SKS musste das einzustellende Personal erst extern angeworben werden.

I.3 Planung und Ablauf des Vorhabens

I.3.1 Arbeitsplan und Meilensteine

Der Arbeitsplan des Projektes sah insgesamt zwanzig Arbeitspakete mit drei Meilensteinen vor. Diese Planung wurde im Rahmen der Möglichkeiten eingehalten, aber es ergaben sich zwei wesentliche Abweichungen, die im folgenden erläutert werden.

Verzögerter Anfang. Auf Grund der Aufbausituation am DFKI Bremen verzögerte sich der Start des Projektes effektiv um vier Monate; dieser Zeitverzug blieb im Projektverlauf konstant, und führt am Ende zu einer viermonatigen kostenneutralene Verlängerung.

Fokussierung auf Softwarezertifikation. Der erste Meilenstein (M1) beinhaltete die Vorstellung des Konzeptes bei der Zertifizierungsbehörde (TÜV Süd Rail). Bei dieser Präsentation wurde mit dem TÜV vereinbart, die Zertifizierung auf die Software zu fokussieren, und die Hardware nicht zu zertifizieren, aus folgenden Gründen:

(1) Während bei der Zertifikation der Hardware der Stand der Technik zur Anwendung gekommen wäre, musste für die Zertifikation der Software neue Techniken der Softwareverifikation entwickelt werden, was ein wesentlich größeres, auch für andere nutzbares Innovationspotential bietet.

(2) Ferner ist eine Hardwarezertifikation nicht ohne weiteres wiederverwendbar; potentielle Nutzer müßten bei jeder Änderung der Hardware — die für den Übergang in Serienproduktion unumgänglich ist— eine erneute Qualifikation durchführen. Somit ist der praktische Nutzen der Hardwarequalifikation für einen Forschungsprototypen fraglich, da es sich in diesem Falle ohnehin um *off-the-shelf*-Komponenten handelt.

Damit wurde das Projektziel auf eine wiederverwendbare, zertifizierte Softwarekomponente fokussiert, sowie die Entwicklung und Zertifikation einer dafür nötigen Softwareverifikationsumgebung.

I.3.2 Organisatorischer Ablauf.

Die Meilensteine konnten mit dem sich aus der verzögerten Start ergebenden konstanten Verzug eingehalten werden, insbesondere die Zertifikation des Gesamtsystems zum Ende.

Es fanden insgesamt acht Projekttreffen statt, sechs Präsentationen beim TÜV, und eine öffentliche Abschlussveranstaltung (siehe Tabelle 1).

09./10.11.07	Kick-Off Meeting	Fürstenfeldbruck
02./03.04.07	2. Projekttreffen	Bremen
29./30.05.07	3. Projekttreffen	Fürstenfeldbruck
30./31.08.07	4. Projekttreffen	Bremen
10./11.10.07	5. Projekttreffen	Fürstenfeldbruck
05./06.12.07	6. Projekttreffen	Bremen
21.07.08	7. Projekttreffen	München
17.02.09	8. Projekttreffen	Bremen
13.10.09	Öffentliche Abschlussveranstaltung	Bremen

11.10.07	1. Konzeptpräsentation	TÜV, München
21.07.08	2. Konzeptpräsentation	TÜV, München
11.11.08	3. Konzeptpräsentation	Bremen
20.01.09	4. Konzeptpräsentation	TÜV, München
15./16.06.09	Abnahmepräsentation	Bremen
13.08.09	Nachbereitung der Abnahmepräsentation	TÜV, München

Tabelle 1: Übersicht über Projekttreffen und TÜV-Präsentationen

I.4 Stand der Wissenschaft und Technik zu Projektstart

Industrielle Sicherheitssysteme Fahrerlose Transportsysteme (FTS) wurden früher meist durch Kontaktsensoren (Bumper) abgesichert, die jedoch die zulässige Fahrzeuggeschwindigkeit auf ein unproduktives Maß reduzieren. So hat sich der Sicherheits-Laserscanner nach seiner Einführung sehr schnell als berührungslos wirkende Schutzeinrichtung mit mehreren Metern großen Schutzfeldern in diesem Markt etabliert. Die Fa. Leuze electronic gehört zu den führenden Herstellern von berührungslos wirkenden Sicherheits-Bauteilen und zu den weltweit drei Herstellern, die Sicherheits-Laserscanner entwickeln. Ein Laserscanner tastet mit einem rotierenden Laserstrahl die Umgebung nach Objekten ab und vergleicht die gemessene Entfernung des Objektes mit einer fest eingestellten Schutzzonen-Distanz. Ist ein Hindernis in der Zone, schaltet der Scanner das Gefahr bringende System ab. Bei einem Fahrzeug hängt die zum sicheren Anhalten benötigte Zone von der Geschwindigkeit, dem Beladungszustand, dem Bauteileverschleiß und einer eventuell möglichen Kurvenfahrt ab. Deshalb unterstützen

Laserscanner mehrere Schutzfelder, die je nach Zustand, Geschwindigkeit und Kurvenfahrt umgeschaltet werden. Solche feste Schutzfelder müssen aber entsprechend konservativ und mit großen Zuschlägen gewählt werden, wodurch Bahnführung und Geschwindigkeit unnötig eingeschränkt werden. Bei dynamischen Schutzfeldern wird hingegen immer so spät wie möglich, aber so früh wie nötig gebremst; dynamische Schutzfelder bieten also maximale Verfügbarkeit bei garantierter Sicherheit.

Fahrwegsicherung in der Robotik Mobile Serviceroboter nutzen ebenfalls oft Laserscanner, lesen aber deren Entfernungsdaten aus, um sie im Hauptrechner für die Bewegungssteuerung auszuwerten, weil die Verwendung fester Schutzzonen zu unflexibel ist. Das populärste Verfahren ist der *dynamic window approach* (DWA), neben *nearness diagrams* und *potential fields*. Der DWA wurde z.B. bei Museumsrobotern im Deutschen Museum in Bonn, dem Carnegie Museum in Pittsburgh, und der SwissExpo 2003 in Lausanne eingesetzt. Er setzt einen runden Roboter voraus, lässt sich aber verallgemeinern. Alle Algorithmen steuern die Fahrt des Roboters und sichern ihn nicht nur gegen Kollisionen ab. Aus der Perspektive der Zulassung ist dies ungünstig, weil der sicherheitsrelevante Teil unnötig komplex und mit dem Rest des Systems integriert wird, was die gezielte Fehleranalyse, Verifikation, Tests und damit die Zulassung erschwert.

I.5 Zusammenarbeit mit Anderen

Das Verbundprojekt SAMS wurde vom Forschungsbereich Sichere Kognitive Systeme des Deutschen Forschungszentrums für Künstliche Intelligenz als Konsortialführer zusammen mit den Projektpartnern Universität Bremen und Leuze electronic (vormals Leuze lumiflex) durchgeführt; es fanden insgesamt acht Projekttreffen statt (siehe Tab. 1).

Eine enge Zusammenarbeit war natürlich mit der Zertifierungsstelle, dem TÜV Süd Rail, gegeben. Mit diesem fanden vier Konzeptpräsentation statt (siehe Tab. 1), und eine Abnahmepräsentation am 15/16.06.09 in Bremen.

Während des Projektes fand mit anderen Projekten der Leitinnovation Servicerobotik ein reger Erfahrungsaustausch statt. Projektmitarbeiter von SAMS nahmen mit eigenen Vorträgen teil am Architekturworkshop am Fraunhofer-Institut für Produktionstechnik und Automation (IPA) in Stuttgart (11.05.2007), am Workshop *Sicherheit* am Fraunhofer-Institut für Fabrikbetrieb und -automatisierung (IFF) in Magdeburg (12.10.2007), und am Workshop *Sicherheit in der Mensch-Roboter-Interaktion* am IFF in Magdeburg (18.06.2009).

II Eingehende Darstellung

II.1 Erzielte Ergebnisse

Die im Projekt SAMS durch das Deutsche Forschungszentrum für Künstliche Intelligenz erzielten Ergebnisse lassen sich wie folgt kurz zusammenfassen:

(1) Entwicklung eines einfach konfigurierbaren, konservativen Bremsmodells für autonome Fahrzeuge und mobile Serviceroboter;

(2) ein als Sicherheitsfunktion nach der Norm IEC 61508:3 bis zu SIL-3 zertifizierter Algorithmus zur Berechnung von Schutzfeldern;

(3) eine Verifikationsumgebung, welche zur Spezifikation und Verifikation von MISRA-C-Software gemäß IEC 61508:3 bis zu SIL 3 verwendet werden kann;

(4) und über die ursprünglichen Projektziele hinausgehend eine Verallgemeinerung des Schutzfeldverfahrens im dreidimensionalen Raum.

Die Punkte (2) und (3) wurden uns vom TÜV Süd Rail durch *Letters of conformance* bestätigt (s. Abb. 1). Weitere Einzelheiten zu der vom TÜV bestätigten Normenkonformität finden sich im technischen Bericht des TÜV, der diesem Bericht in Anlage B beliegt.

II.1.1 Konservatives Bremsmodell

Das *Bremsmodell* beschreibt das Verhalten des autonomen Fahrzeugs (*equipment under control*, EUC) während des Bremsvorgangs, vom Einsetzen der Bremsung bis zum Stillstand. Die Anforderungen an das Bremsmodell waren folgende:

(i) Es muss *sicher* sein, d.h. es muss mindestens die tatsächlich zum Bremsen benötigte Fläche beschreiben; mit anderen Worten, es muss eine *konservative Abschätzung* bieten.

(ii) Es muss eine möglichst große der in der *Praxis* verwendeten Menge von autonomen Fahrzeugen (insbesondere industriell verwendete FTS) *abdecken*.

(iii) Es muss *einfach parametrierbar* sein, im Idealfall mit einer Messung. Diese Anforderung entstammt der industriellen Praxis und Erfahrung der Firma Leuze electronic als Anbieter von parametrierbaren Schutzeinrichtungen.

Schreiben des TÜV Süd Rail vom 26.09.10:

Betreff: SAMS

Hiermit bestätigen wir Ihnen, dass die Sicherungskomponente für Autonome Mobile Systeme (SAMS) *zur Berechung von geschwindigkeitsabhängigen Schutzfeldern einschließlich des zu Grunde liegenden Bremsmodells der Norm IEC 61508-3:2008 (SIL 3) entspricht.*

Die durchgeführten Analysen und Reviews der von Ihnen übersandten Dokumente und Tests haben gezeigt, dass keine sicherheitsrelevanten Bedenken gegen den Einsatz der SAMS-Software zur Schutzfeldberechnung bestehen.

Schreiben des TÜV Süd Rail vom 06.10.10:

Betreff: SAMS Verifikationsumgebung

Hiermit bestätigen wir Ihnen, dass die von Ihnen entwickelte SAMS Verifikationsumgebung *und der Theorembeweiser* Isabelle *zur Spezifikation und Verfikation von MISRA-C Softwaremodulen gemäß der Norm IEC 61508-3:1998 bis zum SIL 3 verwendet werden kann. Insbesondere werden durch den ordnungsgemäßen Einsatz der Verifikationsumgebung die folgenden Verfahren und Maßnahmen als abgedeckt betrachtet:*

Tabelle A.4: 1c, 2, 5, 6
Tabelle A.9: 1, 3
Tabelle B.1: vollständig
Tabelle B.8: 1, 3, 4, 5, 8

Die durchgeführten Analysen und Reviews der von Ihnen übersandten Dokumente und Beweisskripte haben gezeigt, dass keine sicherheitsrelevanten Bedenken gegen den Einsatz der SAMS Verifikationsumgebung *zur Software-Verifikation der Phasen Software-Systementwurf, Modulentwurf und Implementierung bestehen.*

Abbildung 1: Schreiben des TÜV (*letters of conformance*), welche die Normenkonformität der Schutzfeldberechnung und der Entwicklungsumgebung bescheinigen. Kopien der Schreiben finden sich in der Anlange (Anhang B).

Das entwickelte Bremsmodell erfüllt diese Anforderungen. Es kann mit allen Fahrzeugen verwendet werden, die sich in einer Ebene fortbewegen, und die mindestens zwei ungelenkte Räder haben, so dass sich das Fahrzeug nicht seitwärts bewegen kann. Beispiele für geeignete Fahrzeuge sind: Fahrzeuge mit Differentialantrieb, wenn diese die Lenkwinkel entsprechend steuern, und Fahrzeuge mit einer gelenkten und einer ungelenkten Achse, wenn der Lenkwinkel während des Bremsvorganges fix ist. Die genauen Anforderungen an das Fahrzeug finden sich im Anwenderhandbuch des Schutzfeldberechnungsmoduls (S. 10f), welche diesem Bericht in Anlage B beiliegt. Weitere Annahmen an das Fahrzeug sind

(1) eine konvexe Fahrzeugkontur,

(2) dass die Bremsbeschleunigung höchstens proportional zur Geschwindigkeit steigt,

(3) und dass der zeitliche Verlauf des Abbaus der kinetischen Energie während des Bremsvorgangs für Kurvenfahrten und Geradeausfahrten gleich ist.

Dieses sind alles realistische Annahmen (beispielsweise kann eine konvexe Fahrzeugkontur immer durch die Bildung der konvexe Hülle erreicht werden, was auch nie eine die Verfügbarkeit beinträchtigende Verallgemeinerung darstellen dürfte).

Das Bremsmodell geht von einer Messung des Bremsverhaltens bei Geradeausfahrt aus. Durch Annahme (2) oben läßt sich das Bremsverhalten als konvexe Kurve mit wenigen Messungen gut bestimmen. Abb. 2 zeigt links mit der roten Kurve ein exaktes Bremsverhalten, welches schon durch eine einzige Messung (grüne Gerade) gut, und mit zwei Messungen (blauer Streckenzug) fast exakt angenähert wird.

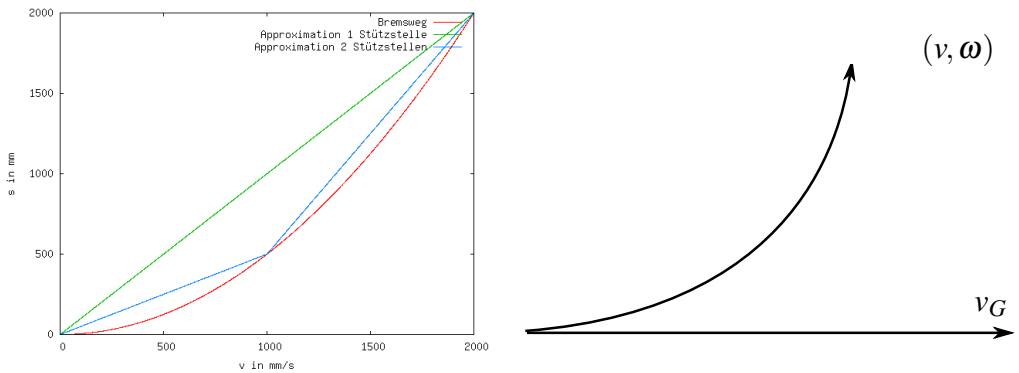


Abbildung 2: Parametrierung des Bremsmodells (links), Geradeaus- und Kurvenfahrt (rechts)

Aus dem Bremsweg bei einer Geradeausgeschwindigkeit v_G wird der Bremsweg unter der Annahme (3) oben bei einer Geschwindigkeit v und Lenkwinkel ω berechnet (Abb. 2 rechts). Das Modell geht dabei davon aus, dass die kinetische Energie beim

Bremsen in der Kurve auch durch die Zentripetalkraft abgebaut wird; Einzelheiten hierzu finden sich im Konzeptpapier Bremsmodell, welches diesem Bericht beliegt (Anlage B).

Das Bremsmodell ist in dieser Form vom TÜV Süd als sicherheitstechnisch geeignet begutachtet worden, und hat sich in bei Versuchen als praktisch nutzbar herausgestellt. Insbesondere schränkt es nicht durch zu große Überapproximation die Verfügbarkeit übermäßig ein.

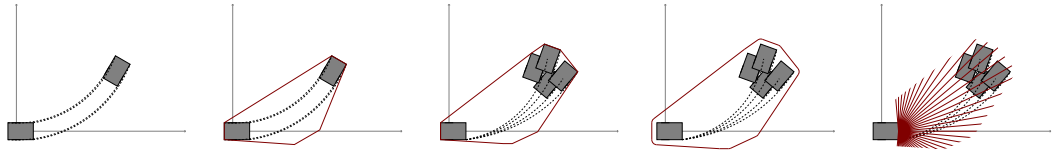
II.1.2 Algorithmus zur Berechnung der Schutzfelder

Mathematisch beschreibt das Bremsmodell eine Funktion, welche das im Ursprung befindliche Fahrzeug bei einer Bremsung mit einer Geschwindigkeit $\vec{v}$ auf den Punkt abbildet, bei dem das Fahrzeug zum Stillstand kommt. Durch Integration über die Zeit erhält man die beim Bremsvorgang überstrichene Fläche. Diese muss noch um verschiedene, unter anderem durch die relevanten Normen geforderten Zuschläge vergrößert werden:

(i) Aufgrund der Ungenauigkeiten der Odometrie (Messung des Geschwindigkeitsvektors) wird das Schutzfeld nicht für eine Geschwindigkeit, sondern für ein Intervall von Geschwindigkeiten berechnet.

(ii) Ferner wird das Schutzfeld am Ende um eine feste Größe nach allen Seiten ausgepuffert; diese Konstante modelliert vorgeschriebene Sicherheitsabstände und Latenzzeiten (wie Ansprechzeiten der Bremse oder Bremsabnutzung).

Nach dieser Berechnung wird das Schutzfeld von einem Polygon durch Abtasten in ein für einen Sicherheitslaserscanner geeignetes Schutzfeld überführt. Abb. 3 illustriert den Vorgang.



(a) (b) (c) (d) (e)

Abbildung 3: Berechung des Schutzfeldes. Die Bewegung auf einer Kurvenbahn bis zum Stillstand (a) wird durch ein Polygon eingefasst (b), es werden geschwindigkeitsabhängige (c) und geschwindigkeitsunabhängige (d) Zuschläge aufgeschlagen, und zum Schluß durch Abtasten ein Schutzfeld erstellt (e).

Wenn das so berechnete Schutzfeld durch einen Sicherheitslaserscanner überwacht wird, und bei einem Hindernis im Schutzfeld innerhalb der bei der Berechnung des

Schutzfeldes berücksichtigten Ansprechzeit eine Bremsung eingeleitet wird, kann die Sicherheit vor der Kollision mit stationären Hindernissen gewährleistet werden. Diese Eigenschaft wurde mit der im Rahmen des Projektes entwickelten Verifikationsumgebung (siehe Abschnitt II.1.3) formal nachgewiesen. Der TÜV Süd hat den Algorithmus und die von uns entwickelte Implementation in MISRA-C- begutachtet, und für den Einsatz bis zu einem Sicherheitsintegritätslevel SIL-3 nach IEC 61508:3 geeignet zertifiziert.

Die Begutachtung umfaßte sämtliche von der Norm IEC 61508:3 geforderten Projektdokumente, einschließlich der Konzeptpapiere, welche das Bremsmodell und die Berechnung der Schutzfelder beschreiben, eine Software-FMEA und Kritikalitätsanalyse, Anforderungs- und Sicherheitsanforderungsspezfikation, Detailspezifikation, Verifikations- und Validationsplan und das Anwenderhandbuch. Die nicht-vertraulichen Projektdokumente sind online unter `http://www.dfki.de/sks/sams/papers/documents/` verfügbar.

Gegenüber der prototypischen Implementierung, die am Anfang des Projektes verfügbar war und für den autonomen Bremer Rollstuhl Rolland genutzt wurde, hat die im Rahmen von SAMS entwickelte Implementierung neben der Sicherheitszertfikation, die das eigentliche Ziel des Projektes war, den zusätzlichen Vorteil, dass sie um Größenordnungen schneller ist. Damit entfällt die Vortabellierung der Schutzfelder; Schutzfelder können im laufenden Betrieb berechnet werden. Die dafür benötigte Rechenleistung ist auch bei den im Bereich eingebetteter System verbreiteten Architekturen (z.B. ARM-basiert) gegeben, soweit hardwareseitige Fließkommaunterstützung verfügbar ist.

II.1.3 Verifikationsumgebung

Die SAMS-Verifikationsumgebung dient dazu, die Korrektheitsbedingungen für die in MISRA-C implementierte Schutzfeldberechnung formulieren, und ihre Einhaltung beweisen zu können. Für die Entwicklung der Verifikationsumgebung galten folgende Anforderungen:

(1) Die Spezifikation soll in einer *verständlichen, formalen Sprache* nahe am Code erfolgen.

(2) Die Umgebung sollte den *formalen Korrektheitsbeweis* unterstützen, d.h. den rechnergestützt geführten, maschinell nachvollziehbaren Beweis, und nicht auf stichproben-basiertem Testen basieren.

(3) Die Umgebung sollte auf SW-Systeme nach IEC 61508 SIL-3 anwendbar sein.

Die Spezifikation erfolgt nach dem *design by contract*-Prinzip: die C-Funktionen werden mit Vor- und Nachbedingungen annotiert. Abb. 4 zeigt eine Beispielspezifikation. Zusätzlich zu Vor- und Nachbedingungen muss auch angegeben werden, welche

```
/*@
  @requires \separated(v, len, v_res, len)
    && $!istSKT(m)
  @ensures 0 <= \result <= len &&
    ${ ^PSet{v_res, \result} =
       ^SKT{m} ‘ ^PSet{v, \result} }
  @modifies v_res[:len]
  @*/

int transformiere(Punkt * v, Punkt * v_res,
                  int len, Matrix * m);
```

Abbildung 4: Spezifikation einer Funktion, die eine Starrkörpertransformation implementiert. **@requires** ist die Vorbedingung, **@ensures** ist die Nachbedingung, und **@modifies** die veränderten Speicherstellen.

Veränderungen im Speicher die Funktion vornimmt. Wenn eine Funktion als korrekt bewiesen wird, dann ist folgendes garantiert:

(i) Die *funktionale Korrektheit*: wenn beim Aufruf der Funktion die Vorbedingung gilt, gilt nach Beendigung der Funktion die Nachbedingung.

(ii) Die *Termination*: wenn beim Aufruf der Funktion die Vorbedingung gilt, dann terminiert die Funktion.

(iii) Die *Programmsicherheit*: wenn beim Aufruf der Funktion die Vorbedingung gilt, dann sind alle Zeigerdereferenzierungen und Feldzugriffe wohldefiniert, und es findet keine Division durch Null statt.

(iv) Die *Speichersicherheit*: wenn beim Aufruf der Funktion die Vorbedingung gilt, dann werden nur die angegeben Stellen im Speicher modifiziert.

Die Spezifikationssprache ist an gängige Sprachen wie JML (für Java) oder ACSL (für C) angelehnt, bietet aber darüber hinaus noch die Möglichkeit, Isabelle-Ausdrücke direkt in die Spezifikation einzubetten, und dadurch bei der Formulierung auf Isabelles reiche Datenstrukturen (Mengen, Listen, ganze und reelle Zahlen u.v.m.) zurückzugreifen. Diese zusätzliche Ausdrucksstärke hat sich in der Praxis als unschätzbarer Vorteil herausgestellt (und wurde aus den Anforderungen der Spezifikationspraxis heraus entwickelt). Die genaue Syntax und Semantik der Spezifikationssprache findet sich im *Referenzhandbuch der SAMS-Verifikationsumgebung*, welche diesem Bericht beliegt (Anlage B).

Die Verifikationsumgebung benutzt eine in dem Theorembeweiser Isabelle eingebettete totale Hoare-Logik. Ein syntaktisches Front-End übersetzt Programm und Spezi-

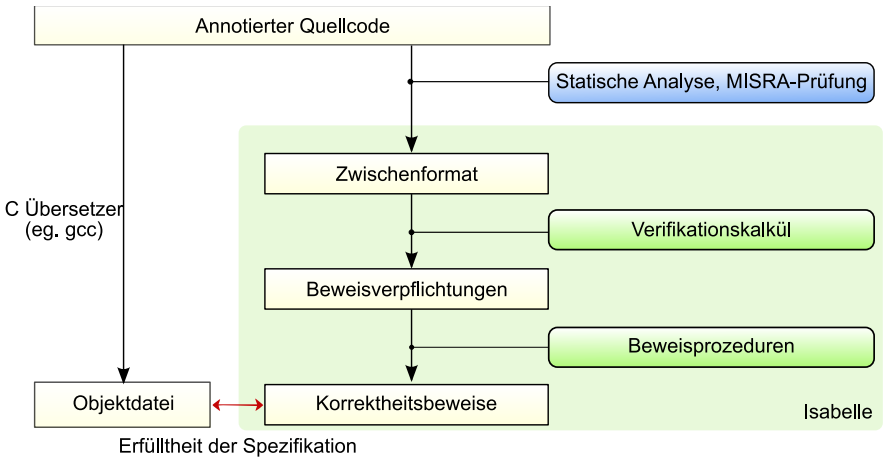


Abbildung 5: Architektur der Verifikationsumgebung.

fikation in ein von Isabelle lesbares Format, und durch syntaxgesteuerte Regelanwendung wird die Spezifikation zu Beweisverpflichtungen reduziert, die dann mit Isabelle interaktiv oder automatisch beweisen werden können (Abb. 5). Die technischen Einzelheiten wurden in dem Papier [1] auf der Konferenz *Formal Methods 09* publiziert.

Entscheidend für die Realisierung der dritten Anforderung oben ist, dass der Korrektheitsbeweis sich zu großem Teil auf Isabelle stützt. Setzt man die Korrektheit von Isabelle voraus (i.d.S dass nur korrekte Theorem in Isabelle bewiesen werden können), dann reduziert sich die Korrektheit der Beweisumgebung auf die Korrektheit des Beweiskalküls, welche sich wieder auf die korrekte Umsetzung der Semantik der Sprache C nach dem Standard ISO 9899:1990 in Isabelle reduziert, aus welcher der Beweiskalkül konservativ entwickelt wird. Diese Argumentation ist im Projektdokument *Validation der SAMS Verifikationsumgebung* ausgeführt.

Die von uns verwendete Verifikationstechnik, und insbesondere die von uns entwickelte Verfikationsumgebung, wurde vom TÜV begutachtet, und kann zur Spezifikation und Verifikation von MISRA-C-Software gemäß der Norm 61508:3 bis zu SIL 3 verwendet werden; dieses wurde in einem *letter of conformance* bestätigt, der dem Bericht beigefügt ist.

II.1.4 Schutzfelder im dreidimensionalen Raum

Die Verallgemeinerung des Verfahrens erlaubt die Berechnung Schutzfeldern im Raum, und dient der zentralen Überwachung von im dreidimensionalen Raum bewegten Anlagenteile, wie beispielsweise Roboterarme oder Manipulatoren, zum Schutz vor Kollisionen miteinander oder der mit anderen Hindernissen.

Wesentlicher Bestandteil der dreidimensionalen Schutzfeldberechnung ist die Verallgemeinerung der Berechung des überstrichenen Volumens von zwei auf drei Dimensionen, indem das überstrichene Volumene als konvexe Hülle einer endlichen Punkt-

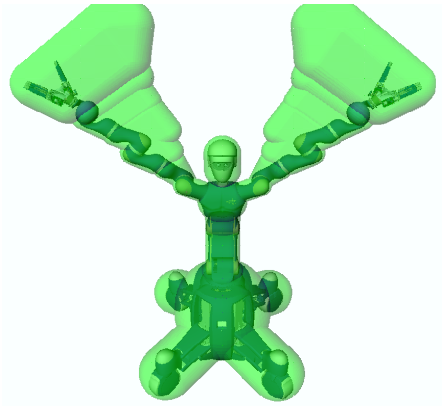


Abbildung 6: Beispiel für dreidimensional Schutzfelder. Roboter Justin des DLR beim „Zusammenklatschen“ der Hände. Die berechneten Schutzfelder sind grün überlagert.

menge zuzüglich eines Pufferradius im Raum dargstellt wird. Kern des Verfahrens ist ein neuer und echtzeitfähiger Algorithmus, der mit dieser Darstellung effizient Bewegungen berechnen kann. Abb. 6 zeigt ein Anwendungsbeispiel.

II.2 Positionen des Nachweises

Es gab nur geringfügige Abweichungen von den Positionen der Vorkalkulationen.

II.3 Notwendigkeit und Angemessenheit

Die aufgeführten Arbeiten waren angemessen und zur Erreichung der Projektziele notwendig.

II.4 Voraussichtlicher Nutzen

Durch die Fokussierung auf ein zertifiziertes Softwaremodul ist die Möglichkeit, die in SAMS entwickelte Bücherei in eine Fremdanwendung zu integrieren, wesentlich erleichtert worden, was die wirtschaftliche Verwendbarkeit erhöht. Zur Zeit wird mit dem Konsortialpartner Leuze die Portierung der Lösung auf eine reine Festkommaarithmetik diskutiert, da nicht in allen Anwendungsplattformen Fließkommaarithmetik zur Verfügung steht.

Weiterhin besteht großes Interesse von Seiten der Industrie an den im Projekt SAMS angewandten Methoden zur Zertifizierung von Software. Hier ist es noch notwendig, die im Projekt entwickelte Software mit einer benutzerfreundlicheren Bedieneroberfläche auszustatten, und in moderne integrierte Softwareeentwicklungsumgebungen wie beispielsweise Eclipse zu integrieren.

Ein Ausdruck des Interesses sowohl an der entwickelten Sicherheitskomponente als auch an den Methoden war die Einladung für einen Vortrag bei der *safetronic 2008* Konferenz in München.

Weiterhin ist in Gesprächen mit industriellen Partnern der Wunsch nach Korrektheitsbeweisen für Programme in der Sprache C++ geäußert worden. Nachdem zwischenzeitlich eine MISRA Programmierrichtlinie für C++ veröffentlicht worden ist, kann jetzt untersucht werden, inwieweit die in SAMS entwickelte Werkzeugunterstützung auch für MISRA-C++ Programme erweitert werden kann.

Die in Abschnitt II.1.4 geschilderte Verallgemeinerung des Verfahrens wurde mit der Patentanmeldung 102009006256.4-32 zum Patent angemeldet, und als technischer Bericht RR-09-01 des DFKI veröffentlicht. Anwendungsfelder dieses Patentes wären insbesondere die berührungslose Absicherung von Manipulatoren.

II.5 Fortschritt bei anderen Stellen

Die Firma Sick bietet die Produktfamilien der Laserscanner S300 und S3000 jeweils ab der Variante Professional sowie die Laserscanner-Interfaces LSI 10X mit dynamischen Steuereingängen für Inkrementalgeber an. Diese Eingänge dienen dazu dynamisch (z.B. geschwindigkeitsabhängig) einen aus mehreren möglichen Feldsätzen auszuwählen. Ein Feldsatz besteht dabei aus einem Schutz- und einem Warnfeld.

Jeder Feldsatz für sich ist im laufenden Betrieb fest und muss vor Inbetriebnahme konfiguriert werden. Die dynamische Schutzfeldanpassung erfolgt lediglich durch die Umschaltung zwischen den vorkonfigurierten festen Feldsätzen. Dazu wird jedem Feldsatz ein Geschwindigkeitsintervall zugewiesen, so dass im laufenden Betrieb derjenige Feldsatz verwendet wird, der dem Geschwindigkeitsintervall zugeordnet ist, in dem sich die aktuelle Geschwindigkeit befindet. Es existiert allerdings nur eine beschränkte Menge an Feldsätzen. (Beim Sick S300 Professional vier, beim Sick S3000 Professional sowie beim LSI 10X acht unterschiedliche Feldsätze.)

Der in SAMS realisierte Ansatz ist aus zwei Gründen dem von Sick am Markt angebotenen überlegen:

(1) Da im SAMS-Modul die Schutzfelder online berechnet werden, ist die Anzahl der Schutzfelder prinzipiell beliebig; dadurch können die Schutzfelder wesentlich enger gewählt werden.

(2) Die Schutzfelder bei Kurvenfahrt müssen im Sick-Scanner von Hand parametriert werden; im SAMS-Modul werden sie aus dem Bremsmodell berechnet, welches durch einfach Konfiguration in Geradeausfahrt berechnet werden kann.

Weitere Arbeiten im Bereich der dynamischen Schutzfeldberechnung, oder andere Ansätze in diesem Gebiet sind uns nicht bekannt.

II.6 Veröffentlichung

II.6.1 Schutzfeldberechnung

Es ist beabsichtigt, die zertifzierte Implementation der Schutzfeldberechnung quellöffentlich verfügbar zu machen, sofern dem nicht Verwertungsabsichten der Firma Leuze electronic entgegenstehen. Hierzu soll die Software noch geeignet aufbereitet, und insbesondere in ein existierendes Rahmenwerk wie Player/Stage integriert werden, um Interessenten die Nutzung zu erleichtern.

II.6.2 Verifikationsumgebung

Auch die Verifikationsumgebung soll quelloffen zur Verfügung gestellt werden. Hierzu müssen insbesondere noch eine robuste, plattformübergreifende Installationsprozedur und geeignete Dokumentation für Installation und Benutzung zur Verfügung gestellt werden. Diese Arbeiten sollen zur Cebit 2010 abgeschlossen sein.

II.6.3 Publikationen und Präsentationen

Die Ergebnisse des Projektes sind auf verschiedenen Kongressen präsentiert worde; weitere Präsentation sind in Vorbereitung. Insbesondere werden die Ergebnisse des Projektes wie die SAMS-Verifkationsumgebung im Rahmen des Messauftritts des DFKI auf der Cebit 2010 präsentiert.

Die zentrale Präsentationsplattform für die Ergebnisse im Internet ist die SAMS-Webseite, `http://www.sams-projekt.de` oder `http://www.sams-project.org`, auf der die Ergebnisse zweisprachig präsentiert werden. Dort finden sich die Projektdokumente, sofern öffentlich verfügbar, die *letters of conformance* des TÜV, viele Präsentationen, und die Veröffentlichungen. Die Softwareprodukte des Projektes werden dort auch veröffentlicht werden.

Am 13. Oktober 2009 fand in Bremen die Abschlusspräsentation des Projektes vor über fünfzig Teilnehmern statt. Die dort gehaltenen Vorträge finden sich auf der Webseite.

A Liste der Veröffentlichungen

[1] Christoph Lüth and Dennis Walter. Certifiable specification and verification of C programs. In Ana Cavalcanti and Dennis Dams, editors, *Formal Methods (FM 2009)*, volume 5850 of *Lecture Notes in Computer Science*, pages 419–434. Springer, 2009.

[2] Udo Frese and Holger Täubig. Verfahren zur Vermeidung von Kollisionen gesteuert beweglicher Teile einer Anlage. Research Report RR-09-01, Deutsches Forschungszentrum für Künstliche Intelligenz, 2009.

[3] Maksym Bortin, Christoph Lüth, and Dennis Walter. A certifiable formal semantics of C. In Tarmo Uustalu, Jüri Vain, and Juhan Ernits, editors, *20th Nordic Workshop on Programming Theory NWPT 2008*, Technical Report, pages 19– 21, Tallinn, Estonia, November 2008. Institute of Cybernetics, Tallinn University of Technology.

[4] Christoph Lüth, Udo Frese, Holger Täubig, Dennis Walter, and Daniel Hausmann. SAMS: Sicherungskomponente für Autonome Mobile Serviceroboter. In *VDI-Bericht*, volume 2012. VDI-Verlag, 2008.

[5] Udo Frese, Daniel Hausmann, Christoph Lüth, Holger Täubig, and Dennis Walter. The importance of being formal. In Hardi Hungar, editor, *International Workshop on the Certification of Safety-Critical Software Controlled Systems SafeCert'08*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2008.

[6] Udo Frese, Daniel Hausmann, Christoph Lüth, Holger Täubig, and Dennis Walter. Zertifizierung einer Sicherungskomponente mittels durchgängig formaler Modellierung. In *Software Engineering 2008*, volume P-122 of *Lecture Notes in Informatics*, pages 335– 338. Gesellschaft für Informatik, 2008.

[7] Christoph Lüth and Bernd Krieg-Brückner. Sicherheit in der Künstlichen Intelligenz. *Künstliche Intelligenz*, 1:51– 52, 2007.

B Weitere Anlagen

B.1 Schreiben des TÜV

- *Letter of conformance* für die SAMS Schutzfeldberechnung
- *Letter of conformance* für die SAMS Verifikationsumgebung
- Prüfbericht

B.2 Konzeptpapier Bremsmodell

- *Autor:* Holger Täubig.

B.3 Konzeptpapier Schutzfeldberechnung

- *Autor:* Holger Täubig.

B.4 Anwenderhandbuch

- *Autoren:* Maksym Bortin, Udo Frese, Christoph Lüth, Stefan Mohr, Holger Täubig, Dennis Walter.

B.5 Referenzhandbuch Verifikationsumgebung

- *Autoren:* Christoph Lüth, Dennis Walter.

Mehr Sicherheit.
Mehr Wert.

TÜV SÜD Rail GmbH· Ridlerstr. 57· D-80339 München· Germany

Deutsches Forschungszentrum für Künstliche Intelligenz

Universität Bremen und

Leuze Lumiflex GmbH + Co. KG

Ihre Zeichen/Nachricht vom	Unsere Zeichen/Name	Tel.-Durchwahl/E-Mail	Fax-Durchwahl	Datum	Seite
	ps Peerasan Supavatanakul	+49 (89) 5190-3524 peerasan.supavatanakul@tuev-sued.de	-2933	28.Sep 2009	1 von 1

Betreff: SAMS

Hiermit bestätigen wir Ihnen, dass die *Sicherungskomponente für Autonome Mobile Systeme (SAMS)* zur Berechnung von geschwindigkeitsabhängigen Schutzfeldern einschließlich des zu Grunde liegenden Bremsmodells der Norm IEC 61508-3:2008 (SIL 3) entspricht.

Die durchgeführten Analysen und Reviews der von Ihnen übersandten Dokumente und Tests haben gezeigt, dass keine sicherheitsrelevanten Bedenken gegen den Einsatz der SAMS-Software zur Schutzfeldberechnung bestehen.

TÜV SÜD Rail GmbH

Rail Automation

i.V. Günter Greil
Fachzertifizierer

i.A. Dr. Peerasan Supavatanakul
Functional Safety Expert

TÜV SÜD Rail GmbH· Ridlerstr. 57· D-80339 München· Germany

Mehr Sicherheit.
Mehr Wert.

Deutsches Forschungszentrum für Künstliche Intelligenz

Universität Bremen und

Leuze Lumiflex GmbH + Co. KG

Ihre Zeichen/Nachricht vom	Unsere Zeichen/Name	Tel.-Durchwahl/E-Mail	Fax-Durchwahl	Datum	Seite
	ps Peerasan Supavatanakul	+49 (89) 5190-3524 peerasan.supavatanakul@tuev-sued.de	-2933	06.10. 2009	1 von 1

Betreff: SAMS Verifikationsumgebung

Hiermit bestätigen wir Ihnen, dass die von Ihnen entwickelt *SAMS Verifikationsumgebung* und der Theorembeweiser *Isabelle* zur Spezifikation und formalen Verifikation von MISRA-C Softwaremodulen gemäß der Norm IEC 61508-3:1998 bis zum SIL 3 verwendet werden kann. Insbesondere werden durch den ordnungsgemäßen Einsatz der Verifikationsumgebung die folgenden Verfahren und Maßnahmen als abgedeckt betrachtet:

Tabelle A.4: 1c, 2, 5, 6
Tabelle A.9: 1, 3
Tabelle B.1: vollständig
Tabelle B.8: 1, 3, 4, 5, 8

Die durchgeführten Analysen und Reviews der von Ihnen übersandten Dokumente und Beweisskripte haben gezeigt, dass keine sicherheitsrelevanten Bedenken gegen den Einsatz der *SAMS Verifikationsumgebung* zur Software-Verifikation der Phasen Software-Systementwurf, Modulentwurf und Implementierung bestehen.

TÜV SÜD Rail GmbH

Rail Automation

i.V. Günter Greil
Fachzertifizierer

i.A. Dr. Peerasan Supavatanakul
Functional Safety Expert

Technical Report

on the evaluation of

SAMS – Safety Component for Autonomous Mobile System

Applicants

Deutsches Forschungszentrum für Künstliche Intelligenz,

Universität Bremen and

Leuze Lumiflex GmbH + Co. KG

Report no. LF82746T
rev. 1.0 of September 25th 2009

Test and Certification Body

TÜV SÜD Rail GMBH
Rail Automation
D-80339 Munich

TABLE OF CONTENTS

Revision

Version	Status	Date	Author	SW ver.	HW ver.	Changed chapters	Reason of change
1.0	initial	25.09.2009	Dr. Supavatanakul				

Table 1: Revision

1 Target of Evaluation (ToE)

In the letter from October 17th,2007, Leuze Lumiflex GmbH + Co. KG requested TÜV SÜD Rail GmbH to evaluate the control software system for automatic navigation of the service robots and the automobile transportation system according to the software standard IEC 61508-3:1998. The evaluation of the software according to IEC 61508-3:1998 is within the scope of the project "SAMS" (Safety Component for Autonomous Mobile Systems). The project partners include DFKI (Deutsches Forschungszentrum für Künstliche Intelligenz), Universität Bremen and Leuze Lumiflex GmbH + Co. KG.

The aims of SAMS project is to develop a safety component for a service robot or an autonomous transport vehicles using an already certified safety laser scanner to calculate the protection zone for the service robot and stop the robot if an obstacle is detected within the protection zone. SAMS is based on the mathematical modeling which represents the specifications related to the service robot and a formal proof which is used to verify the correctness of the implementation. The scope of the evaluation by TÜV SÜD Rail GmbH is on the control software. Therefore, the software process and software safety are assessed. The hardware component and the hardware/software integration are beyond the scope of this evaluation.

2 Product overview

The objective of SAMS is to design and implement the safety component for the service robot or autonomous transport vehicle. The safety component consists of the control software and the safety sensor. Within the scope of SAMS, the ROTOSCAN RS4 from Leuze Lumiflex GmbH + Co. KG is used. The ROTOSCAN RS4 is an already TÜV certified device according to IEC 61496-1 and IEC 61496-3. The sensing function of the ROTOSCAN RS4 is performed by opto-electronic emitting and receiving elements that detect the diffused reflection of optical radiations generated within the device by an object presented in the two-dimensional protection zone. The safety component obtains the sensing information provided by the ROTOSCAN RS4. The control software has the objective to avoid the collision of the service robot or transport vehicle by issuing the stop signal based on the current state of the vehicle and the measurement of the safety laser scanner. Therefore, the safety functions of the control software are:

1. Determine the protection zone.
2. Output the "stop" signal upon detection of an object within the defined protection zone.

The detection of the presence of object in the protection zone can be interpreted mathematically as the detection of points or a set of points within the geometry of the protection zone which is changing with time as the service robot moves. Therefore, SAMS provides the solution to compute the real time protection zone based on the service robot's translational velocity and angular velocity. If any obstacles are detected within this protection zone, the service robot is stopped.

The control software for the safety component is examined according to IEC 61508-3. The assessment by TÜV SÜD Rail GmbH is on the software specification, development, verification, and validation according to the V model as shown in Figure 1. It is a top-down design approach starting from the software safety requirement specification (mathematical description for the protection zone) and ending at the bottom with the actual codes of the control software. The testing of the code is executed for each level of design. The verification of the control software is in the theorem prover Isabelle in typed higher- order logic (see [D21]).

The control software for SAMS was examined by TÜV SÜD Rail GmbH during the period from October 2007 to August 2009 with regards to the following evaluation segments:

Functional safety

- Analysis of the specified safety measures
- Analysis of software architecture
- Software testing
- Software validation

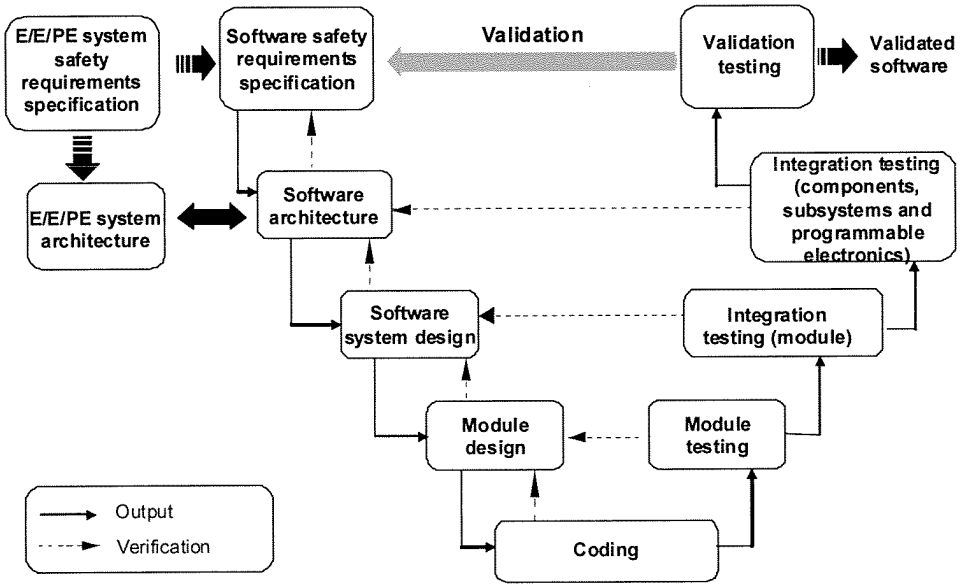


Figure 1 Software development lifecycle (V-model) according to IEC 61508-3

3 Identification of the system and context

The following versions were the base for the evaluation:

Hardware: Not applicable
Software: SAMS software version 0.95

The following table identifies the software version by means of MD5 checksum.

File names	Checksum
abtasten.c	a59cc8475052ebd94bd16dc77dc371f1
abtasten.h	bd785b31aa62ced08540ce0347e9c3a3
bremsmodell.c	2cf71c63d3770142652654cb3ca34098
bremsmodell.h	3b25d4a28e996862a90bf6ba55f1c870
CMakeLists.txt	43aa711e14e4283e8ff85126160d8570
ereignisprotokoll.c	7ec3c19be290388d3d5d9c1abdd578b6
ereignisprotokoll.h	6eebaa49d19697ef91c6868c417df4cc
hilfsfunktionen.c	4aeba9a93823f58b39d0e9cddaf684cf
hilfsfunktionen.h	71c539b0d9d8a8077c777c75dfa14149
hol_dekl.h	c6782f469fba5fe4ef11600e5528f357
kreisbogen.c	a576e24451ddcd86f675239b62ce137e
kreisbogen.h	f7b24274781de2615bb3968d8c4217e5
nutzerkonf_vorverarbeiten.c	2790d5575387c272c38bc67974f095fe
nutzerkonf_vorverarbeiten.h	10ea61d01ca8e0a22e01414d923042ae
pak.c	6e6ff72d68486aedd96474766fc50543
pak.h	f0a875bd73f5a7d4618ad1c110ff2017
sams_konfig.h	a2a38d5f1887bd9cdc679640ba40c8e9
sams_konstanten.c	33405ffb7f8474cbdd61c2630f248f0c
sams_konstanten.h	e3bf18511d04c826b9806d8394d775c0
sams_system.h	28ebc77b2ff294fc12319076facc7b74
sams_typen.c	5389741828eafb9b31dadc548a6bb064
sams_typen.h	9ddf600ee10f7ec5a3ee49ce74f3a360
schutzfeld_berechnen.c	5da651c85de8539796c0db464c10e842
schutzfeld_berechnen.h	02bffbd6084c0b669e84352ef4e6c8b1
speicher.c	f2562b25eedc06839a371c7fc29cc786
speicher.h	9eea9ba8160caf0219e050e5fc28c2ba
wertebereich.h	86ed24fca09c42473daa75e9a77b6117

4 Basis of Testing

The standards which form the basis of the type testing are listed below.

4.1 Software standards

[N1]	IEC 61508-3:1998	Functional safety of electrical/electronic/programmable electronic safety related systems Part 3: Software requirements
[N2]	MISRA C:2004	Motor industry software reliability association C-standard

4.2 Relevant standards

[N3]	IEC 61496-1:2004	Safety of machinery – Electro-sensitive protective equipment – Part 1: General requirements and tests
[N4]	IEC 61496-3:2008	Safety of machinery – Electro-sensitive protective equipment- Part 3: Particular requirements for Active Opto-electronic Protective Devices responsive to Diffuse Reflection (AOPDDR)

5 Document List for the Testing of SAMS

5.1 Documents provided for the evaluation of SAMS

[D1]	SAMS Ueberblick (Powerpoint Folien)	-	17.10.2007
[D2]	Schutzfeldberechnung und Bremsmodell (Powerpoint Folien)	-	17.10.2007
[D3]	Konfigurationsprogramm (Powerpoint Folien)	-	17.10.2007
[D4]	Verifikation (Powerpoint Folien)	-	17.10.2007
[D5]	Dokumentenplan	-	15.07.2009
[D6]	Anforderung (Lastenheft)	Ver 2.1	28.01.2008
[D7]	Konzeptpapier Bremsmodell	Ver 1.7	15.07.2009
[D8]	Konzeptpapier Schutzfeldberechnung	Ver 2.6	22.09.2009
[D9]	Sicherheitsanforderungsspezifikation Schutzfelder	Ver1.0	06.06.2009
[D10]	Software-Kritikalitätsanalyse und FMEA	Ver 1.2	18.06.2009
[D11]	Detailsspezifikation Software Schutzfeldberechnung	Ver 1.2	08.06.2009
[D12]	Anwenderhandbuch	Ver 1.5	24.08.2009
[D13]	Software Qualitätsmanagement Plan (QM Plan)	Ver 1.0	20.05.2009
[D14]	Verifikations- und Validationsplan	Ver 1.3	08.06.2009
[D15]	Maßnahmen zur Prüfung der Einhaltung der MISRA-Richtlinien	Ver 1.1	05.06.2009
[D16]	Testplan	Ver 1.2	05.06.2009
[D17]	Protokolle Code-Review	Ver 1.0	02.06.2009
[D18]	SAMS Verifikationsumgebung – Referenzhandbuch	Ver 2.3	01.07.2009
[D19]	01Stueckliste	-	22.09.2009
[D20]	02Pruefsummen	-	22.09.2009
[D21]	Isabelle course material (reference from TU-München) http://isabelle.in.tum.de/coursematerial/IJCAR04/index.html	-	Accessed in July 2009

5.2 Evaluation documents and reports

[R1]	Minutes of Meeting	-	11.10.2007
[R2]	Minutes of Meeting	-	21.07.2008
[R3]	Minutes of Meeting	-	11.11.2008
[R4]	Minutes of Meeting	-	20.01.2009
[R5]	Minutes of Meeting	-	15.06.2009
[R6]	Review Report Bremsmodelle	-	23.02.2009
[R7]	Review Report Schutzfeld	-	25.02.2009
[R8]	Review Report Spezifikation Daten	-	22.06.2009
[R9]	Review Report Testplan	-	27.05.2009
[R10]	Review Report Handbook	-	27.05.2009
[R11]	Review Report MISRA Check	-	22.06.2009
[R12]	Checklist for Functional Safety of E/E/PE Safety Related Systems according to IEC 61508-3 Software	-	20.08.2009

6 Performance and result of evaluation

The evaluation of the control software for SAMS was made according to the requirements according to [N1]. This evaluation demonstrates that the software architecture specification for the control software is complete and correct with respect to the mathematical description and the safety-related measures for fault control in the software are applied.

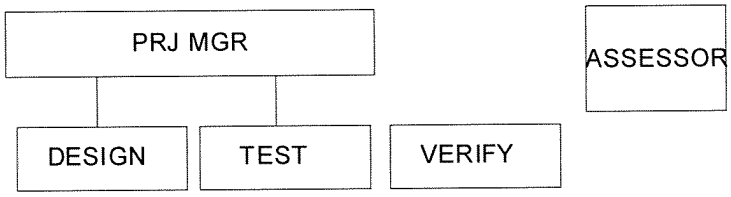
6.1 Software development and quality management system

The software development for the control software was assessed to evaluate the management and technical activities during the overall project. The software development and quality management system shall:

- apply administrative and technical control throughout the software lifecycle
- maintain accurately and with unique identification. The software development shall include: safety analysis and requirement, software specification and design documents, software source codes modules, test plans and results and verification documents.
- apply change control procedure
- ensure that the appropriate methods are implemented
- include the information to allow a subsequent audit
- formally document the release of the safety related software.

Results:

The software development and quality management system for the control software is documented in [D13]. It includes and gives the references to the requirements according to the software development and quality management system. The following order of the person-independencies was chosen for the development of the control software.



where:
PRJ MGR: Project managers (see [D13])
DESIGN: Designers/Developers from the project team (see [D13])
TEST: Designer/Developers from the project team (see [D13])
VERIFY: Designer/Developers from the project team (see [D13]).
ASSESSOR: TÜV SÜD Rail GmbH

The required separation of roles and responsibility of persons is strictly adhered to.

The software lifecycle model according to the V-model is used. For each phase of the lifecycle model, appropriate techniques and measures to avoid failures are used. This is document in [D14].

The requirements according to the software development and quality management system are met.

6.2 Assessment of the Mathematical Specification of the System

The objective of SAMS's control software is to have safe collision avoidance of the service robot by the real time computation of safety zone based on the service robot's translational and angular velocities. This control objective has been translated in the SAMS project as the specification of the breaking model and specification of the protection zone (see [D7] and [D8]). These are mathematical models used for specifying the behavior of the control software. Therefore, these specifications have to be

- complete with respect to the safety needs
- correct with respect to the safety needs
- free from ambiguity
- understandability of the safety requirements
- free from interference of non-safety functions.

Based on the concepts of the breaking model and the protection zones from [D7] [D8], the safety requirement specification is given in [D9] and the software specification in [D11]. Both the breaking models and the protection zone model can be described in the following:

- **Breaking model.** The breaking model describes the behavior of service robot if the stop command is issued. It considers both the translational and rotational behavior of the service robot when the break or stop command is issued.
- **Protection zone model.** The protection zone for the service robot is defined as the area $[v_{\min}, v_{\max}] \times [\omega_{\min}, \omega_{\max}]$, where v represents the translational velocity and ω the angular velocity. The protection zone model is used to determine the protective area for the service robot based on its current translational and angular velocities as well as the parameters such as the reaction time and the uncertainties. Figure 2 illustrates the protection zone and warning zone which depend on the translational and angular velocities of the robot as well as the position of the robot. In case any obstacles lie within the protection zone, the stop command is issued.

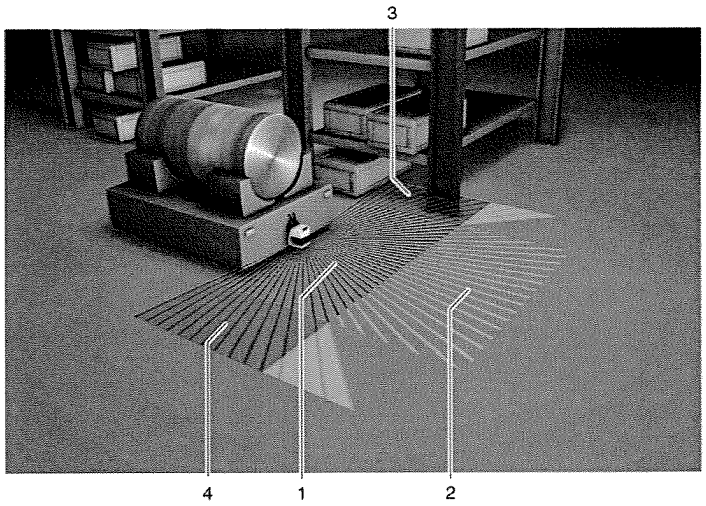


Figure 2 Protection zone for the service robot (1), (2) protection zone and warning zone for straight movement and (3) and (4) protection zones for curve.

Results:

The specification of breaking model and the protection zone model were reviewed by the SAMS project team as well as by TÜV SÜD Rail GmbH (see [R4], [R6], and [R7]). These models were assessed according to the specification for completeness and correctness. The review shows that the specifications for breaking and protection zone suit the intended application and no unintended functions from the models may occur. The software specification in [D11] specifies the software modules, data transfer and interface of the software. It follows directly from [D8] and laid down using natural language and diagrams.

The software requirement specification meets the requirements of SIL 3 according to IEC 61508-3.

6.3 Software design and development

The assessment of software design and development of the control software has the objective to ensure that the software architecture fulfils the specified requirements (cf. [D11]) with respect to SIL 3 of [N1]. The safety critical modules of the control software were identified by means of software criticality analysis (see [D10]). This allows the classification of safety relevance for each software elements. In [D10], it is shown that the most of the software elements are safety critical because they directly impact the calculation of the protection zone for the service robot.

The control software of SAMS is written in C according to [N2] (see [D15]). Therefore the unsafe or unstructured use of the language is minimized. The implementation results in manageable software modules. It is shown at the onsite audit of TÜV SÜD Rail GmbH on 15 - 16.06.2009 that the software development for the control software is:

- modular which control the complexity of the software
- derived from the defined specification
- unambiguously defined and the coding standard is adhered to
- understandable for the project members and developer and others who need to understand the design
- testable and allow modification if necessary

In [D17], it is demonstrated that the codes were reviewed by the project members. Each module of the control software was tested according to the test plan (cf. [D16]). Based on [D16], the following properties were checked:

- no overflow or underflow
- the maximal rounding error is limited
- no access to uninitialized variables
- division by zero
- array size

Results:

Based on the software design review, review of coding guidelines and review of the test plan (see [R11], [R9]) as well as the onsite audit by TÜV SÜD Rail GmbH on 15-16.06.2009, the software architecture, design and implementation meet the SIL 3 requirements according to IEC 61508-3.

6.4 Software verification and validation

The objective of the software verification and validation is to ensure that the developed software complies with the software safety requirement specification. [D14] describes the verification and validation plan used for SAMS. It includes

- details of persons who carry out the verification and validation
- reference for verification (see also in detail [D18])
- identification of the software which needs to be verified and validated
- technical methods used
- tools or relevant software used
- procedure for evaluating the results of verification and validation

The technical methods used for verification of the control software for SAMS consist of:

- code review
- formal specification of the program functions
- review of specifications
- formal proof for SAMS verification environment
- dynamical analysis and test

The central issue for the verification of the software is the formal proof for each program function. All software functions in SAMS together with the formal specifications were transformed to the theorems of the theorem prover Isabelle/HOL (see [D21]). Isabelle is a tool which allows mathematical formulas to be expressed in a formal language and provides tools for proving these formulas in a logical calculus. It mains application is in particular the formal verification which proves the correctness of the software. It has been used by scientists and mathematicians in the academia. Hence, it is considered as proven tool for the formal verification.
The output of the verification is the Isabelle theories which are formally proven. Once all functions of the safety routines satisfy their specification, the overall safety assertion is ensured, i.e. the software is validated.

Results:

Based on the review of the verification and validation plan and demonstration of the verification and validation during the onsite audit by TÜV SÜD Rail GmbH on 15-16.06.2009 (see [R5]), the software verification and validation fulfil the SIL 3 requirements according to IEC 61508-3.

6.5 Inspection of user's documentation

The user's manual [D12] was examined to verify the completeness of the technical documentation.

Results:

The results are documented in the report [R10]. The review results are positive.

7 Additional considerations

This section describes additional considerations which need to be taken into account when implementing the control software for SAMS.

- The user of control software for SAMS has to ensure that the version of the control software is compatible with the laser scanner used.
- Adequate competency: The user who carries out the control software configuration shall be competent (see [D12]).
- Bug reports: The user shall be informed about the bug report information and shall be advised with solutions once known.

8 Statement of compliance

The review found that the SAMS control software for automatic navigation of service robot or autonomous transport vehicle, with software version 0.95, complies with the requirements of SIL 3 according to IEC 61508-3:1998.

TÜV SÜD Rail GmbH
Rail Automation

i.V.
Günter Greil

i.A.
Dr. Peerasan Supavatanakul

Sicherungskomponente für
Autonome Mobile Systeme

Eine Kooperation zwischen
DFKI-Labor Bremen • Leuze lumiflex • Universität Bremen

Konzeptpapier Bremsmodell

Zusammenfassung

Dieses Dokument beschreibt das Konzept des im SAMS Projekt verwendeten und entwickelten Bremsmodells, d.h. eine formale Beschreibung des Verhaltes des EUC bei einer Notbremsung. Dieses Modell bildet die Basis der Schutzfeldberechnung.

Projektbezeichnung	SAMS
Verantwortlich	Holger Täubig
Erstellt am	28.01.2008
Version	1.8
Bearbeitungszustand	vg. (TÜV)
Revision	4326
Letzte Änderung	08.10.2009
Dokumentablage	Projektdokumente/Gesamtsystemspezifikation/Konzept-Bremsmodell.tex

Änderungsliste

07.02.08 – *Version:* 1.0 – *Bearbeiter:* Holger Täubig

Initiale Version 1.0 erstellt.

07.02.08 – *Version:* 1.1 – *Bearbeiter:* Holger Täubig

Abschnitt Wortbedeutungen, Konventionen, Variablen nur noch hier.

12.02.08 – *Version:* 1.2 – *Bearbeiter:* Christoph Lüth

Einige Formulierungen vereinfacht und/oder präzisiert.

12.02.08 – *Version:* 1.3 – *Bearbeiter:* Holger Täubig

Handskizzen durch ordentliche Zeichnungen ersetzt.

02.05.08 – *Version:* 1.4 – *Bearbeiter:* Christoph Lüth

Abschnitt über das Absschätzung des Bremsweges oberhalb v_{max} hinzugefügt.

05.12.08 – *Version:* 1.5 – *Bearbeiter:* Holger Täubig

Neue Berechnung der Bremskonfiguration (s, ϕ). Annahme bzgl. der kinetischen Energie bei Geradeaus- und Kurvenfahrt etwas verschärft.

10.02.09 – *Version:* 1.6 – *Bearbeiter:* Holger Täubig

Unnötige und falsche Integration von $\lambda(t)$ in Gleichung 24 korrigiert. Der Fehler hatte keine Auswirkungen auf andere Gleichungen. Er war lediglich Teil einer Begründung, die nun korrekt ist. Nicht benötigte integrierte Form von Gleichung 25 ebenfalls entfernt.

23.02.09 – *Version:* 1.7 – *Bearbeiter:* Holger Täubig

Abschnitt Bremsweg mit verzögertem Bremsbeginn eingefügt.

06.08.09 – *Version:* 1.8 – *Bearbeiter:* Christoph Hertzberg

Bremswegüberapproximation durch kubische Funktion ersetzt. Faktor zur Überschreitung der Maximalgeschwindigkeit auf 1.2 erhöht.

Prüfverzeichnis

12.02.08 – *Version:* 1.1 – *Prüfer:* Christoph Lüth
Neuer Produktzustand: **i. B.**

Einige Formulierungen noch unklar, sonst korrekt.

12.02.08 – *Version:* 1.2 – *Prüfer:* Dennis Walter
Neuer Produktzustand: **vg.(TÜV)**

13.05.08 – *Version:* 1.4 – *Prüfer:* Holger Täubig
Neuer Produktzustand: **vg.(TÜV)**

15.01.09 – *Version:* 1.5 – *Prüfer:* Dennis Walter
Neuer Produktzustand: **vg.(TÜV)**

23.02.09 – *Version:* 1.5 – *Prüfer:* Dr. P. Supavatanakul
Neuer Produktzustand: **fg.**

Siehe Review Report.

14.06.09 – *Version:* 1.7 – *Prüfer:* Dennis Walter
Neuer Produktzustand: **vg.(TÜV)**

25.09.09 – *Version:* 1.7 – *Prüfer:* TÜV (P. Supavatanakul)
Neuer Produktzustand: **fg.**

Siehe Prüfbericht (Techical Report no. LF82764T)

07.08.09 – *Version:* 1.8 – *Prüfer:* Holger Täubig
Neuer Produktzustand: **vg.(TÜV)**

Inhaltsverzeichnis

1 Kurzbeschreibung

Das Bremsmodell beschreibt das Verhalten des Fahrzeuges im Falle einer Notbremsung, wie sie von der Sicherungskomponente ausgelöst wird. Es dient dazu, abschätzen zu können, welche Fläche bei einer solchen Notbremsung überstrichen wird, bevor das Fahrzeug zum Stillstand kommt; diese ist die Basis der Berechnung der Schutzfelder (siehe dazu das Konzeptpapier (Täubig, 2009)). Zu diesem Zweck werden die Bewegung eines Referenzpunktes sowie die Orientierung des Fahrzeuges, welches sich in einer Ebene bewegt, modelliert. Seine Lage und damit auch die Lage aller seiner Konturpunkte ist durch den Vektor $(x(t), y(t), \Theta(t))^T$, bestehend aus Position des Referenzpunktes (x, y) und Orientierung des Fahrzeuges Θ, zu jedem Zeitpunkt t vollständig beschrieben.

Das in SAMS verwendete Bremsmodell betrachtet sowohl die Distanz als auch die Kurve, die während des Bremsvorganges gefahren wird. Dem Bremsmodell liegt die Annahme zugrunde, dass das Fahrzeug während des gesamten Bremsvorganges den Lenkwinkel konstant hält. Es kann sich dann als Ortskurve des Referenzpunktes eine Gerade oder ein Kreis ergeben. Im Fall einer Geradeausfahrt bei Auslösung der Notbremsung ergibt sich als Ortskurve die Gerade. Befand sich bei Auslösung der Notbremsung das Fahrzeug in einer Kurve, so entsteht eine kreisförmige Bremstrajektorie. Der Radius dieses Kreises ist vom Lenkwinkel bei Bremsbeginn abhängig.

Zur Berechnung des Weges der auf der Ortskurve bis zum Stillstand des Fahrzeuges zurückgelegt wird, wird eine Überabschätzung $\hat{s}_G(v)$ angegeben, die die Konvexität der Funktion $s_G(v)$ verwendet. $s_G(v)$ ist dabei der in Abhängigkeit von der Geschwindigkeit benötigete Bremsweg für Geradeausfahrten. Weil diese obere Grenze nur für den Bremsweg bei Geradeausfahrt gültig ist, wird daraus durch eine energetische Betrachtung eine Überabschätzung des Bremsweges auf der Kreisbahn berechnet.

2 Anforderungen an das EUC

1. Das Fahrzeug bewegt sich in einer Ebene. Seine Position kann durch 2D-Koordinaten beschrieben werden.

2. Das Bremsverhalten ist unabhägig von Ort und Zeitpunkt der Bremsung.

3. Das Fahrzeug bewegt sich während einer Notbremsung auf einer Kreisbahn, deren Radius durch den Lenkwinkel bei Bremsbeginn bestimmt ist.

4. Zwischen dem Bremsweg bei Kurvenfahrt und bei Geradeausfahrt besteht folgender Zusammenhang: Beim Bremsen wird die kinetische Energie einer Kurvenfahrt genauso abgebaut wie die einer Geradeausfahrt.

5. Das Bremsverhalten des Fahrzeuges erfüllt die Monotonieeigenschaft in Gleichung (18).

3 SAMS Bremsmodell

3.1 Wortbedeutungen, Konventionen, Variablen

Im weiteren Text werden Vokabular und Notation wie im Folgenden definiert verwendet:

Position des Fahrzeuges	Position des Referenzpunktes des Fahrzeuges
Orientierung des Fahrzeuges	Winkel zwischen X-Achse und aktueller Fahrtrichtung des Fahrzeuges
Pose $\underline{p}(t) = \begin{pmatrix} x(t) \\ y(t) \\ \Theta(t) \end{pmatrix}$	Position und Orientierung des Fahrzeuges zum Zeitpunkt t
Geschwindigkeitsvektor $\underline{v} = \begin{pmatrix} v(t) \\ \omega(t) \end{pmatrix}$	Geschwindigkeitsvektor zum Zeitpunkt t bestehend aus Vorwärtsgeschwindigkeit $v(t)$ und Winkelgeschwindigkeit $\omega(t)$
Bremsweg s(t)	Weg der auf der Bremstrajektorie (Kreisbogen) bis zum Zeitpunkt t zurückgelegt wurde
Bremsweg $s_G(v)$	Bremsweg für Geradeausfahrt (Bremsung aus Geschwindigkeit v bis zum Stillstand)
$\operatorname{sinc} x = \begin{cases} \frac{\sin x}{x} & x \neq 0 \\ 1 & x = 0 \end{cases}$	sinc-Funktion (*Sinus cardinalis* oder *Kardinalsinus*)

Es werden außerdem folgende Variablen mit fest zugeordneter Bedeutung verwendet:

Startzeitpunkt $t_0 = 0$	Zeitpunkt zu Beginn des Bremsvorganges
Stoppzeitpunkt T	Zeitpunkt zu dem das Fahrzeug vollständig zum Stillstand kommt
Startgeschwindigkeit $\begin{pmatrix} v \\ \omega \end{pmatrix} = \begin{pmatrix} v(0) \\ \omega(0) \end{pmatrix}$	Geschwindigkeit zum Startzeitpunkt
Stoppose $\begin{pmatrix} x \\ y \\ \Theta \end{pmatrix} = \begin{pmatrix} x(T) \\ y(T \\ \Theta(T) \end{pmatrix}$	Pose zum Stoppzeitpunkt
Bremswegschätzung $\hat{s} \geq s(T)$	*obere Schranke* für den Bremsweg s des gesamten Bremsvorganges

Die folgende Tabelle zeigt die verwendeten Einheiten (entsprechend der Projektfestlegung (Täubig, 2008)):

Längen	mm	Winkel	rad
Geschwindigkeiten	$\frac{mm}{s}$	Winkelgeschwindigkeiten	$\frac{rad}{s}$

3.2 Start der Notbremsung und Koordinatensystem

Das Bremsverhalten des Fahrzeuges ist unabhängig von Ort und Zeitpunkt der Bremsung. Damit sind Ortsfunktion und Bremsweg auf dieser Ortsfunktion durch den Geschwindigkeitsvektor $\underline{v}$ zu Beginn der Notbremsung, also durch gefahrene Vorwärtsgeschwindigkeit v und die Winkelgeschwindigkeit ω zum Startzeitpunkt, eindeutig bestimmt.

Im Bremsmodell wird das Koordinatensystem so gewählt, dass sich der Referenzpunkt des Fahrzeuges zum Startzeitpunkt im Koordinatenursprung befindet und die Orientierung entlang der X-Achse ausgerichtet ist. Ferner beginnt der Bremsvorgang zum Zeitpunkt $t_0 = 0$.

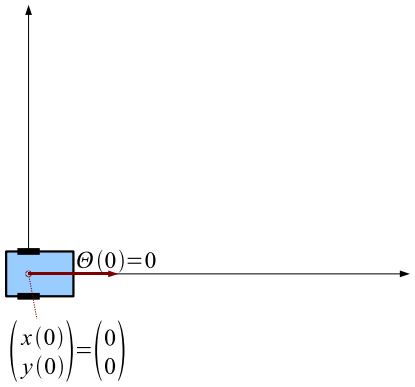


Abbildung 1: Roboterpose zum Startzeitpunkt

Zum Startzeitpunkt $t_0 = 0$ gilt demnach:

$$\underline{p}(t_0) = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{1}$$

$$\underline{v}(t_0) = \begin{pmatrix} v \\ \omega \end{pmatrix} \tag{2}$$

3.3 Ortskurve

Die Ortskurve wird abhängig von der Bremskonfiguration (s, ϕ) dargestellt, die den während der Notbremsung auf der Ortskurve zurückgelegten Weg s und den zugehörigen Winkel ϕ angibt (vgl. Abbildung 2). Für Kreisfahrten ist ϕ der dem Kreisbogen zugehörige Winkel, für Geradeausfahrten gilt $\phi = 0$. Während die Berechnung von (s, ϕ) aus $(v, \omega)^T$ in den Abschnitten 3.4 und 3.5 dargestellt wird, soll dieser Abschnitt zeigen, wie die Stoppose $(x, y, \Theta)^T$ aus der Bremskonfiguration (s, ϕ) berechnet wird.

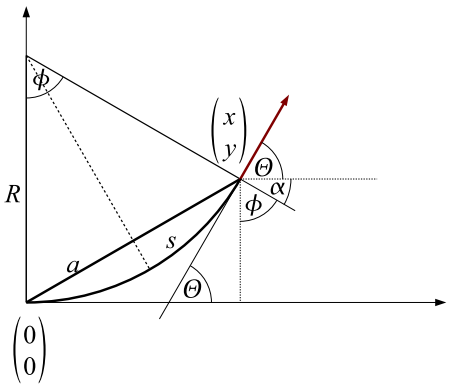


Abbildung 2: Bewegung auf der Kreisbahn

Es wird zunächst der Fall einer Kurvenfahrt betrachtet. In diesem Fall nehmen wir an, dass sich das Fahrzeug auf einer Kreisbahn mit Radius R[1] bewegt.

Betrachtet man nun den Weg s der auf dem Kreis zurückgelegt wird und den dazugehörigen Kreissektor (Abb. 2), so kann (mit $\phi + \alpha = \frac{\pi}{2} = \Theta + \alpha$) leicht bewiesen werden, dass der dem Kreissektor entsprechende Winkel ϕ gleich der Änderung der Orientierung Θ ist:

$$\Theta = \phi \tag{3}$$

Weiterhin gilt für einen Kreissektor der Bogenlänge s und die Länge a der Kreissehne zwischen Anfangs- und Endpunkt des Kreissektors

$$a = 2R \sin \frac{\phi}{2} \tag{4}$$

$$s = \phi R \tag{5}$$

woraus sich die Länge der Sehne ergibt:

$$a = s \frac{\sin \frac{\phi}{2}}{\frac{\phi}{2}} = s \operatorname{sinc} \frac{\phi}{2} \tag{6}$$

Aus Gleichung (6) und der Eigenschaft der Sehne a, dass diese mit der x-Achse den Winkel $\frac{\phi}{2}$ einschließt (Sehnensatz), ergibt sich nun die Stoppposition des Fahrzeugs zu:

$$\begin{pmatrix} x \\ y \end{pmatrix} = s \operatorname{sinc} \tfrac{\phi}{2} \begin{pmatrix} \cos \frac{\phi}{2} \\ \sin \frac{\phi}{2} \end{pmatrix} \tag{7}$$

Gleichung (3) und (7) geben die gewünschte Berechnungsvorschrift für die Stoppose $(x, y, \Theta)^T$ an.

[1] Es gilt $R = \frac{s}{\phi} = \frac{v}{\omega}$. Wichtig ist, dass R niemals explizit ausgerechnet werden muss, da R für den Sonderfall einer Geradeausfahrt nicht definiert ist.

Da die Gleichungen den Radius R der Kurve nie direkt ausrechnen und beim Übergang einer Kurvenfahrt in den Extremfall der Geradeausfahrt stetig fortsetzbar sind, können sie ebenfalls zur Berechnung der Geradeausfahrt verwendet werden. Diese ergibt sich genau dann wenn $\phi = 0$. Auch in diesem Fall berechnet Gleichung (7) die korrekte Stoppposition des Fahrzeuges.

$$\begin{pmatrix} x \\ y \end{pmatrix} = s \operatorname{sinc} 0 \begin{pmatrix} \cos 0 \\ \sin 0 \end{pmatrix} = \begin{pmatrix} s \\ 0 \end{pmatrix} \tag{8}$$

Offensichtlich gilt auch hier für die Orientierungsänderung $\Theta = \phi = 0$.

3.4 Bremsweg $s_G(v)$ bei Geradeausfahrt

3.4.1 Obere Schranke $\hat{s}_G(v)$

$s_G(v)$ ist der Bremsweg des Fahrzeuges aus der Startgeschwindigkeit v bei Geradeausfahrt ($\omega = 0$). Es ist bereits bekannt, dass es sich bei der zugehörigen Bremstrajektorie um eine Gerade handelt. Damit handelt es sich bei der Betrachtung der Funktion $s_G(v)$ und der zugehörigen Ortsfunktion um ein eindimensionales Problem.

Für $s_G(v)$ wird eine obere Schranke $\hat{s}_G(v)$ berechnet, die vom Nutzer durch Eingabe von mindestens einer Stützstelle (bestehend aus einer Geschwindigkeit v_i und dem zugehörigen Bremsweg s_i) parametriert werden muss. Seien n Stützstellen $\{(v_i, s_i)\}_{i=1\ldots n}$ mit $n \geq 1$ gegeben, und sei ferner $(v_0, s_0) = (0,0)$, dann definieren wir $\hat{s}_g(v)$ wie folgt:

$$\hat{s}_G(v) = s_{i-1} + \frac{s_i - s_{i-1}}{v_i - v_{i-1}}(v - v_{i-1}) \qquad \text{für } i \text{ so dass } v_{i-1} \leq v \leq v_i \tag{9}$$

Die Funktion $\hat{s}_G(v)$ ist im Intervall $[0, v_n]$ definiert. Sinnvoll ist die Verwendung der beiden Stützstellen $v_1 = \frac{v_{max}}{2}$ und $v_2 = v_{max}$. Es können aber auch eine einzelne Stützstelle ($v_1 = v_{max}$) oder beliebig viele Stützstellen genutzt werden. Bei Verwendung einer einzelnen Stützstelle ($n = 1$, vgl. Abb. 3) vereinfacht sich $\hat{s}_G(v)$ mit $i = 1$ zu:

$$\hat{s}_G = \frac{s_{max}}{v_{max}} v \tag{10}$$

Die verwendete Abschätzung $\hat{s}_G(v)$ ist eine stückweise lineare Approximation der Funktion $s_G(v)$ aus den Stützstellen $(0,0), (v_1, s_1), \ldots, (v_n, s_n)$ (siehe Abb. 3). Dass es sich um eine obere Grenze der Funktion $s_G(v)$ handelt, wird im Folgenden daraus hergeleitet, dass $s_G(v)$ unter Beachtung der physikalischen Eigenschaften des Bremsvorganges eine konvexe Funktion ist. Dies führt dann zu der gesuchten Aussage

$$\hat{s}_G(v) \geq s_G(v) \qquad \text{für } v \in [0, v_n] \tag{11}$$

3.4.2 Konvexität von $s_G(v)$

Definition: Man nennt eine Funktion f *konvex* auf einem Intervall I, wenn für alle $x < y$ aus I und $t \in [0,1]$ gilt:

$$f(tx + (1-t)y) \leq tf(x) + (1-t)f(y) \tag{12}$$

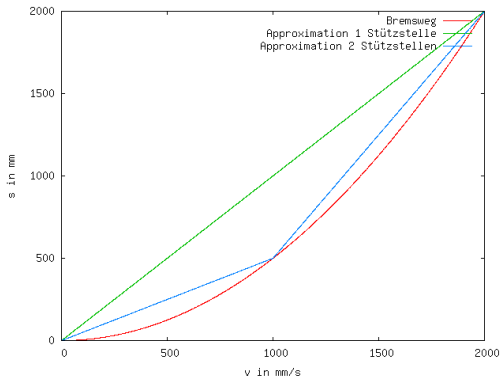


Abbildung 3: Bremsweg bei Geradeausfahrt

Anschaulich bedeutet die Definition: Die Funktionswerte zwischen zwei Werten x,y liegen unterhalb der Verbindungsgeraden der beiden Funktionswerte an den Stellen x und y. Siehe hierzu ggf. (Walter, 2001, § 11.17).

Satz: Ist $f : \Re \to \Re$ differenzierbar, dann gilt:

f ist genau dann konvex, wenn ihre Ableitung f' monoton wachsend ist. (13)

Man betrachtet nun die Ableitung der wiefolgt definierten Funktion $q(t)$ nach der Zeit t:

$q(t)$ ist der Weg, der von Zeitpunkt t bis zum Stoppzeitpunkt T zurückgelegt wird (also der zum Zeitpunkt t noch verbleibende Weg)

Es gilt dann

$$q(t) = s_G(v(t)) \tag{14}$$

$$\frac{dq}{dt} = \frac{ds}{dv}\frac{dv}{dt} \tag{15}$$

und wegen $\frac{dx}{dt} = v$ ist

$$\frac{dq}{dt} = -v \tag{16}$$

Woraus gemeinsam mit $\frac{dv}{dt} = a$ folgt:

$$\frac{ds}{dv}(t) = \frac{v(t)}{-a(t)} \tag{17}$$

Aus der Unabhängigkeit der Funktion $s_G(v)$ von Zeit und Ort der Ausführung des Bremsvorganges, also weil das Bremsverhalten bei Geradeausfahrt für eine bestimmte Geschwindigkeit v immer das gleiche ist, gilt statt Gleichung (17):

$$\frac{ds}{dv} = \frac{v}{-a(v)} \tag{18}$$

Für Gleichung (18) kann nun anhand der physikalischen Eigenschaften des Bremsvorganges gezeigt werden, dass $\frac{ds}{dv}$ eine monoton steigende Funktion ist, woraus sich nach dem aufgeführten Satz zur Konvexität von Funktionen ergibt, dass $s_G(v)$ eine konvexe Funktion ist.

Dies wiederum begründet unter Verwendung der Definition von konvexen Funktionen, dass für alle Punkte der stückweisen linearen Approximation $\hat{s}_G(v)$ die Ungleichung (11), also $\hat{s}_G \geq s_G$ gilt. Damit ist der mittels $\hat{s}_G(v)$ berechnete Weg eine obere Abschätzung des tatsächlichen Bremsweges.

3.5 Bremskonfiguration bei Kurvenfahrt

Wir gehen von der Annahme aus, dass die kinetische Energie bei Kurvenfahrt genauso abgebaut wird wie die kinetische Energie einer Geradeausfahrt mit gleicher kinetischer Energie. Außerdem bewegt sich das Fahrzeug auf einem Kreisbogen dessen Radius sich aus dem Lenkwinkel zum Zeitpunkt des Bremsbeginns ergibt.

Die kinetische Energie eines sich in der Kurve befindlichen Fahrzeuges mit Geschwindigkeitsvektor $(v,\omega)^T$ berechnet sich durch

$$E = \frac{1}{2}mv^2 + \frac{1}{2}J\omega^2 \tag{19}$$

wobei m die Masse des Fahrzeuges und J dessen Trägheitsmoment ist.

Berechnung der Kurvenfahrt aus der Geradeausfahrt Zunächst wird die Abhängigkeit der Bremskonfiguration (s,ϕ) vom Geschwindigkeitsvektor $(v_0,\omega_0)^T$ zu Bremsbeginn dargestellt. Da sich das Fahrzeug auf einer Kreisbahn bewegt, verhalten sich Geschwindigkeit und Winkelgeschwindigkeit proportional und es gilt für den Geschwindigkeitsvektor $(v(t),\omega(t))^T$ zu jedem Zeitpunkt t

$$\frac{v(t)}{\omega(t)} = \frac{v}{\omega} \tag{20}$$

Damit ergibt sich für den Geschwindigkeitsvektor eine von t abhängige charakteristische Bremsfunktion

$$\lambda(t) = \frac{v(t)}{v} = \frac{\omega(t)}{\omega} \tag{21}$$

In Abhängigkeit dieser charakteristischen Bremsfunktion λ lassen sich die Komponenten der Bremskonfiguration darstellen als

$$s = \int_0^T v(t)dt = v\int_0^T \lambda(t)dt \tag{22}$$

$$\phi = \int_0^T \omega(t)dt = \omega\int_0^T \lambda(t)dt \tag{23}$$

Der Abbau der kinetischen Energie während des Bremsvorganges ist ebenfalls abhängig von λ.

$$E(t) = \lambda(t)^2 E \tag{24}$$

Basierend auf der Annahme, dass die kinetische Energie E während des Bremsvorganges aus $(v, \omega)^T$ genauso abgebaut wird wie die kinetische Energie eines Bremsvorganges aus Geradeausfahrt mit Geschwindigkeit v_G, die die gleiche kinetische Energie $E_G = E$ besitzt, sind nun die charakteristischen Funktionen λ und λ_G beider Bremsvorgänge identisch

$$\forall t : \qquad \lambda(t) = \lambda_G(t) \tag{25}$$

und es gilt für den Bremsweg s_G der Geradeausfahrt

$$s_G = v_G \int_0^T \lambda_G(t)dt = v_G \int_0^T \lambda(t)dt \tag{26}$$

Damit gilt für die beiden Bremsvorgänge die Eigenschaft

$$\frac{s_G}{v_G} = \frac{s}{v} = \frac{\phi}{\omega} \tag{27}$$

Gelingt es nun eine obere Schranke $\hat{v}_G \geq v_G$ für die Geschwindigkeit einer Geradeausfahrt v_G mit gleicher kinetischer Energie wie die Kurvenfahrt (v, ω) zu finden, so gilt aufgrund der in Abschnitt 3.4.2 bewiesenen Konvexität von $s_G(v)$

$$\frac{\hat{s}_G}{\hat{v}_G} \geq \frac{s_G}{v_G} \tag{28}$$

Daraus ergibt sich dann eine sichere Abschätzung der Bremskonfiguration (s, ϕ)

$$\hat{s} = v\frac{\hat{s}_G}{\hat{v}_G} \geq s \tag{29}$$

$$\hat{\phi} = \omega\frac{\hat{s}_G}{\hat{v}_G} \geq \phi \tag{30}$$

die den korrekten Kreisradius definiert

$$\frac{\hat{s}}{\hat{\phi}} = \frac{v}{\omega} \tag{31}$$

und außerdem auch für Geradeausfahrten korrekt und damit vor allem auch in Zuständen nahe der Geradeausfahrt ohne numerische Probleme anwendbar ist. Für die Geschwindigkeit v_G einer Geradeausfahrt, die die gleiche kinetischer Energie wie die Kurvenfahrt $(v, \omega)^T$ besitzt, muss im Folgenden also noch eine sichere obere Schranke gefunden werden. Diese obere Schranke $\hat{v}_G$ wird dann in den Gleichungen (29) und (30) eingesetzt.

Obere Schranke der kinetischen Energie der Geradeausfahrt Sowohl m als auch J können als Integrale über der Fahrzeugfläche und in Abhängigkeit der Dichte dargestellt werden:

$$m = \int \rho(x)\, dx \tag{32}$$

$$J = \int \rho(x)\, x^2\, dx \tag{33}$$

Ist nun die Ausdehnung des Fahrzeuges durch eine Konstante D beschränkt, so ergibt sich aus der Beschränkung der Ausdehnung $|x| \leq D$ ein Beschränkung für das Trägheitsmoment J

$$J \leq D^2 m \tag{34}$$

welche dann wiederum eine obere Schranke für die kinetische Energie liefert:

$$E \leq \frac{1}{2} m \left(v^2 + D^2 \omega^2\right) \tag{35}$$

Diese obere Schranke (35) ist vom Trägheitsmoment unabhängig. Sie liefert außerdem eine obere Schranke für die Geschwindigkeit v_G einer Geradeausfahrt mit kinetischer Energie E.

$$\hat{v}_G = \sqrt{v^2 + D^2 \omega^2} \tag{36}$$

Es gilt $\hat{v}_G \geq v_G$.

3.6 Bremsweg oberhalb der Maximalgeschwindigkeit

Zur Ermittlung des Bremsmodells wird der Bremsweg für Geradeausfahrt mit mindestens einer Stützstelle $v_1 = v_{max}$ gemessen. Um nicht für jede minimal Überschreitung der Maximalgeschwindigket v_{max} einen Nothalt auszulösen (und andererseits zu verhindern, dass die Maximalgeschwindigkeit überschätzt werden muss, was ein zu pessimistisches Bremsmodell zur Folge hätte), wird die Überschreitung der Maximalgeschwindigkeit v_{max} bis zu einem Faktor μ_{max} erlaubt, der höchstens 1.2 betragen darf. Die effektive Maximalgeschwindigkeit beträgt damit $v_{max}\mu_{max}$.
Da der Bremsweg in dem Bereich $v > v_{max}$ nicht gemessen wurde, wird er kubisch approximiert, und ergibt sich damit als

$$\hat{s}_G(v) = \frac{s_{max}}{v_{max}^3} v^3. \tag{37}$$

Die kubische Approximation beruht auf der Annahme, dass die Bremsleistung (welche proportional zum Produkt aus Geschwindigkeit und Beschleunigung ist) mit zunehmender Geschwindigkeit nicht abnimmt. Anschaulich bedeutet dies, dass die Bremsen bei höherer Geschwindigkeit mehr Energie absorbieren als bei niedriger. In einem Domänenlemma wird bewiesen, dass aus dieser Annahme folgt, dass $\hat{s}_G(v)$ eine obere Schranke für den tatsächlichen Bremsweg liefert.

3.7 Bremsweg mit verzögertem Bremsbeginn

Vom Zeitpunkt der Berechnung eines Bremsweges bzw. der Anwendung eines Schutzfeldes bis zum tatsächlichen Startzeitpunkt des betrachteten Bremsvorganges vergeht eine Zeitverzögerung t_r (siehe Schutzfeldaufschläge in (Täubig, 2009)), in der sich das Fahrzeug ungebremst mit seiner aktuellen Geschwindigkeit, also der Startgeschwindikgeit $(v, \omega)^T$, auf dem gleichen Kreisbogen wie während des Bremsvorganges bewegt. Um die Auswirkung dieser Zeitverzögerung auf den Bremsvorgang zu beachten, müssen die Gleichungen (29) und (30) zur Berechnung der Bremskonfiguration bei Kurvenfahrt wiefolgt erweitert werden:

$$\hat{s} = t_r\, v + \frac{\hat{s}_G}{\hat{v}_G} v \tag{38}$$

$$\hat{\phi} = t_r\, \omega + \frac{\hat{s}_G}{\hat{v}_G} \omega \tag{39}$$

Literatur

[Täubig 2008] TÄUBIG, Holger: *Datenspezifikation*. SAMS Projektdokumentation, 2008. – DFKI, Forschungsbereich Sichere Kognitive Systeme

[Täubig 2009] TÄUBIG, Holger: *Konzeptpapier Schutzfeldberechnung*. SAMS Projektdokumentation, 2009. – DFKI, Forschungsbereich Sichere Kognitive Systeme

[Walter 2001] WALTER, Wolfgang: *Analysis 1*. Springer-Verlag, 2001

Sicherungskomponente für
Autonome Mobile Systeme

Eine Kooperation zwischen
DFKI-Labor Bremen • Leuze lumiflex • Universität Bremen

Konzeptpapier Schutzfeldberechnung

Zusammenfassung

Dieses Dokument beschreibt das Konzept der Schutzfeldberechnung des SAMS Projektes. Das Dokument verwendet das SAMS Bremsmodell, wie es in *Konzeptpapier Bremsmodell* (Täubig, 2009) beschrieben wird.

Projektbezeichnung	SAMS
Verantwortlich	Holger Täubig
Erstellt am	29.01.2008
Version	2.6
Bearbeitungszustand	fg.
Revision	4326
Letzte Änderung	08.10.2009
Dokumentablage	Projektdokumente/Gesamtsystemspezifikation/Konzept-Schutzfeldberechnung.tex

Änderungsliste

06.02.08 – *Version:* 1.0 – *Bearbeiter:* Holger Täubig

Version 1.0 erstellt.

07.02.08 – *Version:* 1.1 – *Bearbeiter:* Holger Täubig, Christoph Lüth

Abschnitt Wortbedeutungen, Konventionen, Variablen nur noch im Konzeptpapier Bremsmodell (Täubig, 2009). Änderungen aus Prüfung vom 07.02.08 eingearbeitet

12.02.08 – *Version:* 1.2 – *Bearbeiter:* Holger Täubig

Handskizzen durch ordentliche Zeichnungen ersetzt.

23.02.08 – *Version:* 1.3 – *Bearbeiter:* Holger Täubig

Darstellung des Algorithmus `SCHUTZFELD` überarbeitet und an Abb. 10 angepasst.

12.03.08 – *Version:* 1.4 – *Bearbeiter:* Dennis Walter

2-Punkt-Methode zur Bestimmung der konvexen Hülle des Bremsbogens beschrieben.

10.12.08 – *Version:* 1.5 – *Bearbeiter:* Holger Täubig

L-Punkt-Methode, konvexe Hüllen der Trajektorien am Rand des Geschwindigkeitsbereiches, Abtasten mit Pufferradien begonnen

13.01.09 – *Version:* 2.0 – *Bearbeiter:* Holger Täubig

Abtasten mit Pufferradien und Berechnung des Pufferradius hinzugefügt.

15.01.09 – *Version:* 2.1 – *Bearbeiter:* Holger Täubig

Berechnung des Pufferradius korrigiert.

10.02.09 – *Version:* 2.1 – *Bearbeiter:* Holger Täubig

Abbildung 11 von Farb- in s/w-Darstellung geändert. Keine Versionsänderung und Prüfung notwendig.

23.02.09 – *Version:* 2.2 – *Bearbeiter:* Holger Täubig

Ergänzung für Schutzfeldaufschlag t_r aus *Konzept-Bremsmodell* (Täubig, 2009) übernommen.

03.07.09 – *Version:* 2.3 – *Bearbeiter:* Holger Täubig

Repräsentation des Laserscan geändert; Abtastalgorithmus geändert

13.07.09 – *Version:* 2.4 – *Bearbeiter:* Holger Täubig

Berechnung des Pufferradius geändert

24.08.09 – *Version:* 2.5 – *Bearbeiter:* Holger Täubig

Algorithmus 13 korrigiert; Algorithmus 16 ergänzt

08.09.09 – *Version:* 2.6 – *Bearbeiter:* Holger Täubig

Berechnung von s_{min}, s_{max}, Θ_{min}, Θ_{max} korrigiert, da sie in den Quadranten 2 und 4 fehlerhaft war

Prüfverzeichnis

07.02.08 – *Version:* 1.0 – *Prüfer:* Christoph Lüth
Neuer Produktzustand: **i. B.**

Bedeutung von *P* und *S* zu Beginn von 3.2.4 einfügen; Extrempunkte des Gültigkeitsbereiches erklären (3.2.6); Darstellung Algorithmus `SCHUTZFELD_VW` überarbeiten; Formulierungen überarbeitet

12.02.08 – *Version:* 1.1 – *Prüfer:* Dennis Walter
Neuer Produktzustand: **vg.(TÜV)**

Korrekturlesung

11.01.09 – *Version:* 1.5 – *Prüfer:* Christoph Lüth
Neuer Produktzustand: **i.B.**

Korrekturlesung

15.01.09 – *Version:* 2.1 – *Prüfer:* Dennis Walter
Neuer Produktzustand: **vg.(TÜV)**

Korrekturlesung

25.02.09 – *Version:* 2.1 – *Prüfer:* Dr. P. Supavatanakul
Neuer Produktzustand: **fg.**

Siehe Review Report.

03.07.09 – *Version:* 2.3 – *Prüfer:* D. Walter
Neuer Produktzustand: **vg.(TÜV)**

Prüfung Abschnitt 3.4.5. Tippfehler bereinigt; sonst keine Anmerkungen.

24.08.09 – *Version:* 2.5 – *Prüfer:* Christoph Lüth
Neuer Produktzustand: **vg.(TÜV)**

Korrekturlesung, keine Anmerkungen.

21.09.09 – *Version:* 2.6 – *Prüfer:* Christoph Lüth
Neuer Produktzustand: **vg.(TÜV)**

Korrekturlesung Abschnitt 3.2.6, kleine Anmerkung eingearbeitet.

25.09.09 – *Version:* 2.6 – *Prüfer:* TÜV (P. Supavatanakul)
Neuer Produktzustand: **fg.**

Siehe Prüfbericht (Techical Report no. LF82764T)

Inhaltsverzeichnis

1 Kurzbeschreibung

Ein Schutzfeld wird für einen Gültigkeitsbereich $[v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$ definiert und darf dann nur für Geschwindigkeitsvektoren verwendet werden, die innerhalb des Gültigkeitsbereiches liegen. Das berechnete Schutzfeld muss für jeden Vektor $(v, \omega)^T \in [v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$ eine Obermenge der Fläche sein, die bei einer Notbremsung aus der Geschwindigkeit $(v, \omega)^T$ von der Roboterkontur überstrichen werden kann.

In die Berechnung eines Schutzfeldes fließen die durch das Bremsmodell gegebene Bewegung (abhängig von $(v, \omega)^T$), die vom Nutzer als konvexes Polygon eingegebene Fahrzeugkontur und die verschiedenen zu beachtenden Schutzfeldaufschläge (Reaktionszeiten, Fehlergrößen, Meßgenauigkeiten, usw.) ein. Das Ergebnis der Berechnung ist zunächst das Schutzfeld als geometrische Figur, die durch ein um einen Pufferradius erweitertes, konvexes Polygon beschrieben wird. Diese Darstellung wird abschließend in eine der Darstellung eines Laserscanns analoge Repräsentation umgewandelt. Dazu wird das vorher ermittelte konvexe Polygon zzgl. Pufferradius von einem festen Punkt aus mit einer festen Winkelauflösung radial abgetastet. Als Strahlenursprung (Abtastzentrum) wird die dem Laserscannerzentrum entsprechende Position und als Winkelauflösung die Winkelauflösung des Laserscanners verwendet.

Die Bedeutung der erzeugten Schutzfeldrepräsentation kann als „minimal nötiger Laserscan“ beschrieben werden.

2 Anforderungen an das EUC

1. EUC muss die Anforderungen erfüllen, die sich aus dem verwendeten Bremsmodell ergeben (siehe *“Konzept-Bremsmodell”(Täubig, 2009)*).

2. Die 2-dimensionale Ausdehnung des Roboters (Projektion in die Bewegungsebene) ist eine Teilmenge der konvexen Hülle der übergebenen Konturpunkte des Roboters.

3 Schutzfeldberechnung

3.1 Wortbedeutungen, Konventionen, Variablen

Wortbedeutungen, Konventionen und Einheiten sind identisch mit denen aus dem Konzeptpapier Bremsmodell (Täubig, 2009). Weiterhin wird verwendet:

$\mathfrak{K}(M)$	Konvexe Hülle der Punktmenge M (z.B. Trajektorie)
$A \oplus B$	Minkowski-Summe (auch punktweise Summe von A und B)

$$A \oplus B = \{x+y | x \in A, y \in B\} \tag{1}$$

3.2 Schutzfeldberechnung ohne Schutzfeldaufschläge

Zunächst wird die Schutzfeldberechnung ohne Aufschläge vorgestellt, d.h. der Bremsvorgang beginnt im Moment der Aktivierung des Nothaltsignals. Es werden dabei keine Reaktionszeiten betrachtet. In Abschnitt 3.3 wird diese Berechnung dann um die notwendigen Aufschläge zu dem tatsächlich verwendendeten Algorithmus erweitert.

3.2.1 Bewegung des Referenzpunktes

Entsprechend des in (Täubig, 2009) spezifizierten Bremsmodells ist die Trajektorie des Referenzpunktes des Fahrzeuges als Kreisbogen zwischen der Startpose $(0,0,0)^T$ und der Stopppose $(x,y,\Theta)^T$ definiert (Abb. 1). Es gilt:

$$\hat{v}_G = \sqrt{v^2 + D^2\omega^2} \tag{2}$$

$$\mathrm{s} = t_r\, v + \frac{\hat{\mathrm{s}}_G(\hat{v}_G)}{\hat{v}_G} v \tag{3}$$

$$\Theta = t_r\, \omega + \frac{\hat{\mathrm{s}}_G(\hat{v}_G)}{\hat{v}_G} \omega \tag{4}$$

$$\begin{pmatrix} x \\ y \end{pmatrix} = \mathrm{s}\,\mathrm{sinc}\frac{\Theta}{2} \begin{pmatrix} \cos\frac{\Theta}{2} \\ \sin\frac{\Theta}{2} \end{pmatrix} \tag{5}$$

t_r ist die Zeitverzögerung bis zum Start des Bremsvorganges. Sie realisiert einen in Abschnitt 3.3 erläuterten Schutzfeldaufschlag.

3.2.2 Transformationsmatrix für Roboterpose der Konfiguration $T(\mathrm{s},\Theta)$

Während des Bremsvorganges definiert zu jedem Zeitpunkt die Pose des Roboters ein Koordinatensystem, das egozentrische Roboterkoordinatensystem. Sein Koordinatenursprung ist der Referenzpunkt des Roboters, die Richtung seiner X-Achse ist durch die Orientierung Θ des Fahrzeuges definiert. Die Transformation zwischen diesen Roboterkoordinatensystemen zum Start- und zum Stoppzeitpunkt wird nun verwendet um die Bewegung der Roboterkontur während des Bremsvorganges zu beschreiben. Dabei bewegt sich der Referenzpunkt des Fahrzeuges auf einem Kreisbogen (siehe (Täubig, 2009)). Dies gilt ebenfalls für alle anderen Konturpunkte des Roboters (siehe 3.2.3).

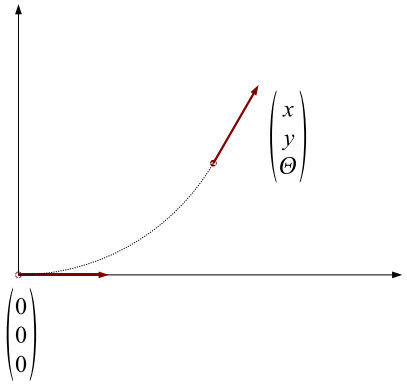


Abbildung 1: Trajektorie des Referenzpunktes

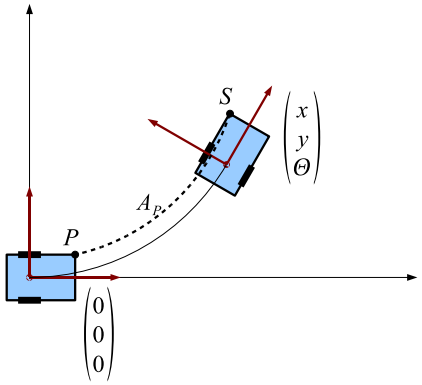


Abbildung 2: Transformation aus Roboterkoordinaten zum Stoppzeitpunkt in Roboterkoordinaten zum Startzeitpunkt

Das Roboterkoordinatensystem zum Startzeitpunkt ist gleichzeitig das Basiskoordinatensystem, in dem der Vorgang beschrieben werden soll (in Übereinstimmung mit (Täubig, 2009)). Deshalb transformiert $T(\mathrm{s},\Theta)$ Punkte aus dem Roboterkoordinatensystem zum Stoppzeitpunkt in das Roboterkoordinatensystem zum Startzeitpunkt. Da die Konturpunkte in Roboterkoordinaten gespeichert sind, lassen sich somit Konturpunkte P durch Anwendung der Transformation $T(\mathrm{s},\Theta)$ in ihre Stoppposition S umrechnen. P ist gleichzeitig die Position des Konturpunktes beim Start des Bremsvorganges (Abb. 2). Sowohl P als auch S sind in Basiskoordinaten angegeben, also im Roboterkoordinatensystem zum Startzeitpunkt.

$$\begin{pmatrix} x_S \\ y_S \\ 1 \end{pmatrix} = T(\mathrm{s},\Theta) \begin{pmatrix} x_P \\ y_P \\ 1 \end{pmatrix} \tag{6}$$

Allgemeiner betrachtet ist die Transformation $T(u,\phi)$ für eine Bremskonfiguration (u,ϕ) zu einem beliebigen Zeitpunkt des Bremsvorganges definiert, für den u,ϕ den bis dahin auf dem Kreisbogen zurückgelegten Weg u und den zum Bogen gehörigen Winkel ϕ angeben. Die Transformationsmatrix von $T(u,\phi)$ ist aus Gleichung (5) ergänzt um die Rotation des Fahrzeuges um

Winkel ϕ gegeben:

$$T(u,\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & u\,\mathrm{sinc}\frac{\phi}{2}\cos\frac{\phi}{2} \\ \sin\phi & \cos\phi & u\,\mathrm{sinc}\frac{\phi}{2}\sin\frac{\phi}{2} \\ 0 & 0 & 1 \end{pmatrix} \tag{7}$$

3.2.3 Grundidee der Schutzfeldberechnung (konvexe Hüllen)

Genau wie der Referenzpunkt des Fahrzeuges bewegt sich auch jeder Konturpunkt während des Bremsvorganges auf einer Kreisbahn (bzw. Geraden bei Geradeausfahrt). Dies ist unmittelbar daraus ersichtlich, dass die Bewegung des Referenzpunktes eine lineare Abbildung ist. Alle anderen Punkte der Roboterkontur unterliegen der gleichen Abbildung, die sich dann aufgrund der Eigenschaften linearer Abbildungen ebenfalls als Kreis(Drehung) bzw. Gerade(Verschiebung) darstellt. Beschrieben ist die lineare Abbildung, wie in Abschnitt 3.2.2 dargestellt, durch die Transformation $T(\mathrm{s},\Theta)$.

Die Grundidee der Berechnung eines Schutzfeldes ist nun, für jeden Punkt P in der Roboterkontur die Trajektorie A_P (Kreisbogen oder Gerade), auf der sich der Punkt bewegt, durch ihre konvexe Hülle $\mathfrak{K}_P$ zu umschließen. Danach werden die konvexen Hüllen für alle Konturpunkte vereinigt und man erhält eine Obermenge V aller Punkte, die vom Fahrzeug während des Bremsvorganges berührt werden könnten.

$$A_P = \left\{ \begin{pmatrix} x_P(t) \\ y_P(t) \end{pmatrix} : 0 \le t \le T \right\} \tag{8}$$

$$\mathfrak{K}_P = \mathfrak{K}(A_P) \tag{9}$$

$$V = \bigcup_{P \in Fahrzeug} \mathfrak{K}_P \tag{10}$$

Die Berechnung der gesuchten Vereinigung V wird vereinfacht indem die Roboterkontur als ein konvexes Polygon mit den Eckpunkten $P_1,\ldots,P_n$ definiert wird. Es ist ausreichend, für alle Eckpunkte P_i dieses konvexen Polygons die konvexen Hüllen K_i ihrer Bremstrajektorien A_{P_i} (für $i = 1\ldots n$) zu bestimmen, und danach wiederum die konvexe Hülle der Vereinigung aller konvexen Hüllen K_i zu berechnen. Das Ergebnis ist die gesuchte Vereinigung V.

$$\mathfrak{K}_i = \mathfrak{K}(A_{P_i}) \tag{11}$$

$$V = \mathfrak{K}\left(\bigcup_{i \in \{1,\ldots,n\}} \mathfrak{K}_i \right) \tag{12}$$

3.2.4 Konvexe Hülle eines Kreisbogens

Im ersten Schritt des Algorithmus werden die konvexen Hüllen $\mathfrak{K}_i$ der Bremstrajektorien der Eckpunkte P_i des Konturpolygons benötigt. Hierzu müssen konvexe Hüllen von Kreisbögen bzw. Geraden berechnet werden.

Im weiteren werden alle Trajektorien als Kreisbögen betrachtet. Dies ist möglich, weil die Geraden Kreisbögen mit Radius $R \to \infty$ entsprechen. Jeder Kreisbogen beginnt an einem Eckpunkt P_i der Fahrzeugkontur zum Startzeitpunkt und endet an der Position S_i des gleichen Konturpunktes nach der Bremsung (Abb. 3). Es gilt $S_i = T(\mathrm{s}, \Theta)\, P_i$.

Die konvexen Hüllen der Kreisbögen werden dann durch eine der folgend beschriebenen Methoden überabgeschätzt. Charakteristisch ist für beide Methoden, dass eine Obermenge der konvexen Hülle berechnet wird, die durch ein konvexes Polygon dargestellt ist.

1-Punkt-Methode. Die 1-Punkt-Methode berechnet ein Dreieck als Obermenge der konvexen Hülle eines Kreisbogens. Das Dreieck besteht aus den beiden Endpunkten des Kreisbogens und einem weiteren, geeignet gewählten Punkt K. Wir verwenden als dritten Punkt K den Schnittpunkt der beiden Tangenten g_S und g_P an den Kreis, die diesen in P und S berühren (siehe Abbildung 3).

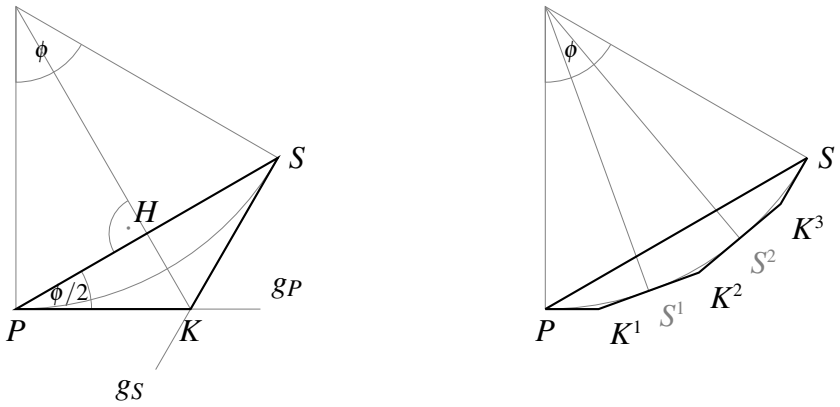


Abbildung 3: *links:* Berechnung des dritten Dreieckspunktes K für die 1-Punkt-Methode. *rechts:* Berechnung der fehlenden Eckpunkte $K_1, \cdots, K_L$ des konvexen Polygones mit der L-Punkt-Methode für L=3

Diese Methode ist nur anwendbar für Kreisbögen mit $\phi < \pi$. Zur Berechnung des Punktes K ist neben den Koordinaten der Punkte P und $S = T(\mathrm{s}, \Theta)\, P$ der Winkel $\angle KPS = \frac{\phi}{2}$ bekannt. Dieser ergibt sich aus dem Sehnensatz für den dem Kreisbogen zugehörigen Winkel ϕ. Außerdem gilt, dass ϕ für alle Fahrzeugpunkte P gleich groß ist, da diese alle die gleiche lineare Abbildung durchführen (vgl. Abschnitt 3.2.3) und sich damit alle um das gleiche Drehzentrum um den gleichen Winkel drehen. Da dies auch für den Referenzpunkt gilt, ist dieser Winkel $\phi = \Theta$:

$$\angle KPS = \frac{\Theta}{2} \tag{13}$$

Außerdem gilt $\overline{HK} = \tan\frac{\phi}{2}\,\overline{PH}$ und die Orientierung von $\overrightarrow{HK}$ geht aus der Orientierung von $\overrightarrow{PH}$

durch Drehung um -90° hervor. Damit läßt sich der Vektor $\overrightarrow{HK}$ wie folgt berechnen:

$$\overrightarrow{PH} = \tfrac{1}{2}(S-P) \tag{14}$$

$$\overrightarrow{HK} = \tan\tfrac{\Theta}{2}\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\overrightarrow{PH} \tag{15}$$

Unter Verwendung der Gleichungen (14) und (15) können dann die Koordinaten des Punktes K durch

$$\begin{aligned} K &= P + \overrightarrow{PH} + \overrightarrow{HK} \\ &= P + \begin{pmatrix} 1 & \tan\frac{\Theta}{2} \\ -\tan\frac{\Theta}{2} & 1 \end{pmatrix}\frac{1}{2}(S-P) \end{aligned} \tag{16}$$

berechnet werden. Zur Berechnung von Gleichung (15) kann in der Implementation außerdem die Halbwinkelformel

$$\tan\frac{\Theta}{2} = \frac{1-\cos\Theta}{\sin\Theta} = \frac{\sin\Theta}{1+\cos\Theta} \tag{17}$$

verwendet werden. Ihr Nutzen ensteht vor allem daraus, dass $\sin\Theta$ und $\cos\Theta$ bereits zur Berechnung von $T(\mathrm{s},\Theta)$ berechnet worden sind.

L-Punkt-Methode Die L-Punkt-Methode teilt den Kreisbogen in L gleiche Teile, approximiert jeden Teil getrennt und fasst dann alle Punkte zusammen. Die Berechnung bestimmt dabei zunächst die drei Approximationspunkte P^1, S^1 und K^1 für den ersten Teilbogen und errechnet anschließend daraus entsprechende Punkte für die Teilbögen 2 bis L. Für den k-ten Teilbogen müssen die drei Punkte dazu mit $T(\frac{k-1}{L}\mathrm{s}, \frac{k-1}{L}\Theta)$ transformiert werden.

$$P^k = T(\frac{k-1}{L}\mathrm{s}, \frac{k-1}{L}\Theta)P^1 \tag{18}$$

$$K^k = T(\frac{k-1}{L}\mathrm{s}, \frac{k-1}{L}\Theta)K^1 \tag{19}$$

$$S^k = T(\frac{k-1}{L}\mathrm{s}, \frac{k-1}{L}\Theta)S^1 = P^{k+1} \tag{20}$$

Die Trennpunkte P^k und S^k werden, mit Ausnahmen von Start- und Endpunkt des gesamten Kreisbogens, nicht für die Approximation benötigt. Es entstehen damit $L+2$ anstatt 3 Punkte zu jedem Punkt der Fahrzeugkontur.

Da diese Methode auf der L-fachen Anwendung der 1-Punkt-Methode beruht, ist es durch geeignete Wahl von L möglich, Bögen mit beliebig großem Winkel Θ zu approximieren. Es empfiehlt sich die Verwendung eines $L \geq 2\frac{\Theta_{max}}{\pi}$. Dies garantiert für die Anwendung der 1-Punkt-Methode $\phi \leq \frac{\pi}{2}$ und vermeidet dadurch eine unnötig große Überapproximation wie sie diese Methode im Bereich $[\frac{\pi}{2}, \pi[$ aufweist (Beispiel in Abbildung 4).

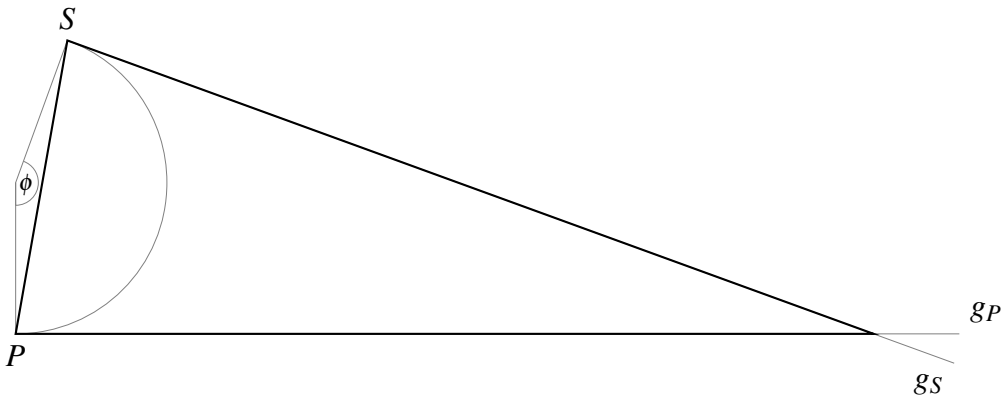


Abbildung 4: Korrekte, aber unerwünscht großflächige Approximation der 1-Punkt-Methode für $\phi \in [\frac{\pi}{2}, \pi[$.

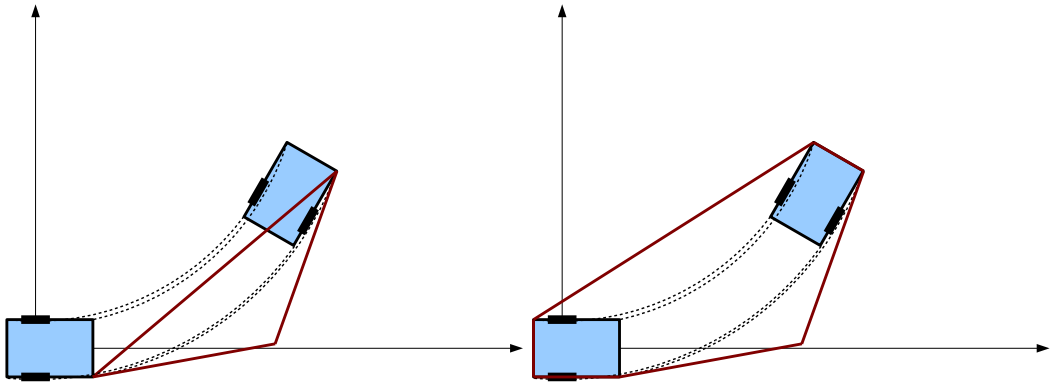


Abbildung 5: Berechnung der konvexen Hülle für eine Bremskonfiguration $(s, \alpha)^T$.

3.2.5 Konvexe Hülle der Roboterbewegung für einen Geschwindigkeitsvektor $(s, \Theta)^T$

Aus den bisher vorgestellten Teilbetrachtungen und Einzelalgorithmen ergibt sich nun entsprechend der in Abschnitt 3.2.3 vorgestellten Grundidee (Betrachtung der Trajektorien der Eckpunkte des konvexen Konturpolygones) der in Abb. 6 dargestellte Algorithmus zur Berechnung eines Schutzfeldes für eine einzelne Bremskonfiguration $(s, \Theta)^T$. Im folgenden Abschnitt 3.2.6 wird gezeigt, wie diese Berechnung verwendet wird, um das Schutzfeld für seinen gesamten Gültigkeitsbereich zu berechnen. Dieser Gültigkeitsbereich ist anwendungsbedingt in $(v, \omega)^T$ und nicht in (s, Θ) definiert.

3.2.6 Konvexe Hülle der Roboterbewegung für den gesamten Gültigkeitsbereich eines Schutzfeldes

Die Erweiterung des Schutzfeldes von einer einzelnen Bremskonfiguration (s, Θ) auf den gesamten Gültigkeitsbereich $[v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$ erfolgt, indem zunächst ein Gültigkeitsbereich

Name: `SCHUTZFELD_STheta`

Input: s,Θ, Eckpunkte des konvexen Konturpolygons $P_1,\ldots,P_n$
Output: Punktmenge V (Das Schutzfeld ist die konvexe Hülle der Punkte in V)

Berechnung:

(i) Berechne $T(\mathrm{s},\Theta)$ enstprechend Gl. (7) und daraus die Stoppposi-tionen aller Konturpunkte enstprechend Gl. (6).

Für $i = 1\ldots n$:

$$S_i = T(\mathrm{s},\Theta)\,P_i$$

(ii) Berechne mittels L-Punkt-Methode die Approximationspunkte $K_i^1,\cdots,K^L$ der konvexen Hülle des Kreisbogens für jeden Kontur-punkt P_i.

Für $i = 1\ldots n$:

(a) Approximationspunkt K_i^1 des ersten Teilbogens aus $P_i^1 = P_i$ und $S_i^1 = T(\frac{s}{L},\frac{\Theta}{L})P_i$ mittels 1-Punkt-Methode.

$$K_i^1 = P_i^1 + \begin{pmatrix} 1 & \tan\frac{\Theta}{2L} \\ -\tan\frac{\Theta}{2L} & 1 \end{pmatrix} \frac{1}{2}\left(S_i^1 - P_i^1\right)$$

entsprechend Gl. (16) für $\frac{\Theta}{L}$

(b) Berechne die Approximationspunkte $K_i^2,\cdots,K_i^L$ für jeden Konturpunkt P_i

Für $k = 2\ldots L$:

$$K_i^k = T(\frac{k-1}{L}\mathrm{s},\frac{k-1}{L}\Theta)K_i^1$$

(iii) Bilde

$$V = \bigcup_{i=1\ldots n}\left\{P_i, S_i, \bigcup_{k=1\ldots L} K_i^k\right\}$$

Abbildung 6: Algorithmus `SCHUTZFELD_STheta`

v_{min}	v_{max}	s_{min}	s_{max}
+	+	$(v_{min}, \mathrm{B}_{\min}(\omega_{min}, \omega_{max}))$	$(v_{max}, \mathrm{B}_{\max}(\omega_{min}, \omega_{max}))$
-	+	$(v_{min}, \mathrm{B}_{\max}(\omega_{min}, \omega_{max}))$	$(v_{max}, \mathrm{B}_{\max}(\omega_{min}, \omega_{max}))$
-	-	$(v_{min}, \mathrm{B}_{\max}(\omega_{min}, \omega_{max}))$	$(v_{max}, \mathrm{B}_{\min}(\omega_{min}, \omega_{max}))$

ω_{min}	ω_{max}	Θ_{min}	Θ_{max}
+	+	$(\mathrm{B}_{\min}(v_{min}, v_{max}), \omega_{min})$	$(\mathrm{B}_{\max}(v_{min}, v_{max}), \omega_{max})$
-	+	$(\mathrm{B}_{\max}(v_{min}, v_{max}), \omega_{min})$	$(\mathrm{B}_{\max}(v_{min}, v_{max}), \omega_{max})$
-	-	$(\mathrm{B}_{\max}(v_{min}, v_{max}), \omega_{min})$	$(\mathrm{B}_{\min}(v_{min}, v_{max}), \omega_{max})$

Tabelle 1: Diese Tabelle gibt abhängig von den Vorzeichen die Vektoren (v, ω) an, aus denen mittels der Gleichungen (2), (3) und (4) die Grenzen s_{min}, s_{max}, Θ_{min} und Θ_{max} berechnet werden.

im (s, Θ)-Raum bestimmt wird, der $[v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$ vollständig enthält.
Das Schutzfeld für den gesamten Gültigkeitsbereich $[s_{min}, s_{max}] \times [\Theta_{min}, \Theta_{max}]$ muss dann die Vereinigung der Schutzfelder für alle Konfigurationen $(s, \Theta) \in [s_{min}, s_{max}] \times [\Theta_{min}, \Theta_{max}]$ sein. Diese Vereinigung lässt sich wieder als konvexe Hülle darstellen. Sie ist die konvexe Hülle der Schutzfelder für den Rand des Gültigkeitsbereiches und für diese wird eine Überabschätzung ermittelt. Dazu wird zunächst die konvexe Hülle der Schutzfelder der vier Extrempunkte $E_1, \ldots, E_4$ des Gültigkeitsbereiches $[s_{min}, s_{max}] \times [\Theta_{min}, \Theta_{max}]$ berechnet und diese anschließend so erweitert, dass auch die Ränder des Gültigkeitsbereiches im Schutzfeld enthalten sind.

Gültigkeitsbereich $[\mathbf{s}_{min}, \mathbf{s}_{max}] \times [\Theta_{min}, \Theta_{max}]$ Die Berechnung der Grenzen s_{min}, s_{max}, Θ_{min} und Θ_{max} des Gültigkeitsbereiches im (s, Θ)-Raum wird entsprechend der Festlegung in Tabelle 1 durchgeführt.
Die Tabelle beschreibt die Abhängigkeit jeder Grenze des (s, Θ)-Raumes von den Vorzeichen im (v, ω)-Raum und den Stellen an denen im (v, ω)-Raum die Betragsmaxima und Betragsminima angenommen werden. Sie gibt für jede Grenze des (s, Θ)-Raumes in jeder möglichen Vorzeichenkombination den Vektor im (v, ω)-Raum an, aus dem sich mittels der Gleichungen (2), (3) und (4) die Grenze des (s, Θ)-Raumes ergibt. Die Funktionen $\mathrm{B}_{\min}$ und $\mathrm{B}_{\max}$ beschreiben dabei jeweils das Element eines Wertebereiches, das den größten bzw. kleinsten Betrag aufweist (wobei argmin und argmax den den Funktionswert minimierenden bzw. maximierenden Argumentwert auswählen):

$$\mathrm{B}_{\min}(x_{min}, x_{max}) = \operatorname*{argmin}_{x \in [x_{min}, x_{max}]} |x| \tag{21}$$

$$\mathrm{B}_{\max}(x_{min}, x_{max}) = \operatorname*{argmax}_{x \in [x_{min}, x_{max}]} |x| \tag{22}$$

Es ist zu beachten, dass $\mathrm{B}_{\max}(x_{min}, x_{max})$ lediglich die Werte x_{min} oder x_{max} annehmen kann, während $\mathrm{B}_{\min}(x_{min}, x_{max})$ die Werte x_{min}, x_{max} oder 0 annehmen kann.

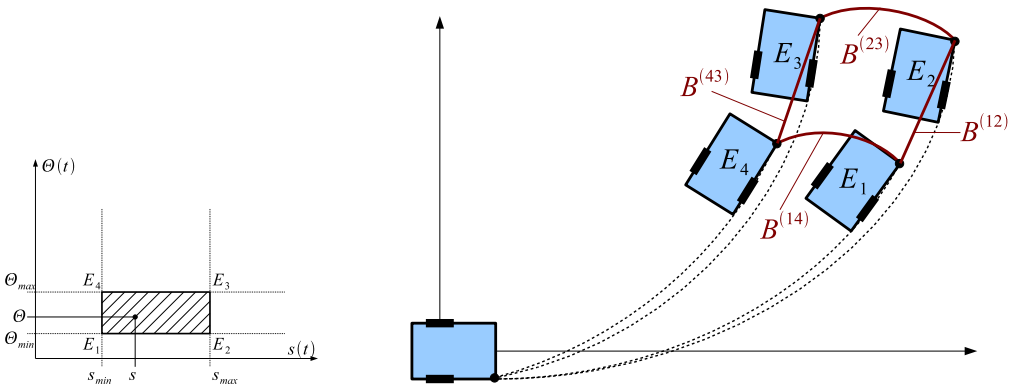


Abbildung 7: Berechnung der konvexen Hülle für den gesamten Gültigkeitsbereich eines Schutzfeldes. Die Abbildung rechts ist ungenau, eigentlich haben die Konfigurationen E_1 und E_2 sowie die Konfigurationen E_3 und E_4 die gleiche Orientierung!

Die Korrektheit der Berechnung in Tabelle 1 wird an dieser Stelle nicht detailliert begründet. Sie wird durch die formale Verifikation der Software nachgewiesen.

Extrempunkte des Gültigkeitsbereiches Die Extrempunkte im (s,Θ)-Konfigurationraum ergeben sich aus der Kombination der Extremwerte für s und Θ (vgl. Abb. 7 links). Zur Berechnung des Schutzfeldes für den Gültigkeitsbereich $[s_{min}, s_{max}] \times [\Theta_{min}, \Theta_{max}]$ wird zunächst das Schutzfeld für jeden der vier Extrempunkte ermittelt und die konvexe Hülle der Vereinigung dieser Schutzfelder $\Re(V_1 \cup V_2 \cup V_3 \cup V_4)$ gebildet.

$$V_1 = \texttt{SCHUTZFELD\_STheta}(s_{min}, \Theta_{min}, P_1 \ldots P_n) \tag{23}$$
$$V_2 = \texttt{SCHUTZFELD\_STheta}(s_{max}, \Theta_{min}, P_1 \ldots P_n) \tag{24}$$
$$V_3 = \texttt{SCHUTZFELD\_STheta}(s_{max}, \Theta_{max}, P_1 \ldots P_n) \tag{25}$$
$$V_4 = \texttt{SCHUTZFELD\_STheta}(s_{min}, \Theta_{max}, P_1 \ldots P_n) \tag{26}$$

Ränder des Gültigkeitsbereiches Abschließend werden die vier Trajektorien $B^{(12)}$, $B^{(23)}$, $B^{(43)}$, $B^{(14)}$ betrachtet, die die Veränderung der Stoppposition für den Verlauf der Konfiguration $T(s,\Theta)$ entlang der vier Grenzen des Gültigkeitsbereiches $\overrightarrow{E_1E_2}$, $\overrightarrow{E_2E_3}$, $\overrightarrow{E_4E_3}$, $\overrightarrow{E_1E_4}$ beschreiben. Diese Trajektorien müssen für jeden Eckpunkt der Roboterkontur betrachtet werden. Sie werden alle gemeinsam durch Festlegung eines Pufferradius q behandelt, der die bis hierhin berechnete konvexe Hülle $W = V_1 \cup V_2 \cup V_3 \cup V_4$ derart erweitern soll, dass das Schutzfeld aus allen Punkten besteht, die von der konvexen Hülle höchstens den Abstand q haben.

Der notwendige Pufferradius q hat die Größe:

$$\begin{aligned} q = &\frac{1}{6}\left(\frac{\Theta_{max}}{2} - \frac{\Theta_{min}}{2}\right)^2 \max\{|s_{max}|;|s_{min}|\} \\ &+ \left(1 - \cos\frac{\Theta_{max}-\Theta_{min}}{2}\right) \max_{1\le i\le n}\{|P_i|\} \end{aligned} \tag{27}$$

Dieser kann aus der Formel für die Trajektorien $B^{(12)}$, $B^{(23)}$, $B^{(43)}$, $B^{(14)}$ abgeleitet werden:

$$B(\mathrm{s},\Theta,P_i) = \mathrm{s}\,\mathrm{sinc}\frac{\Theta}{2}\begin{pmatrix}\cos\frac{\Theta}{2}\\ \sin\frac{\Theta}{2}\end{pmatrix} + \begin{pmatrix}\cos\Theta & -\sin\Theta\\ \sin\Theta & \cos\Theta\end{pmatrix} P_i \tag{28}$$

Die Trajektorien $B^{(12)}$ und $B^{(43)}$ ergeben sich aus $\mathrm{s} \in [s_{min}, s_{max}]$ für konstantes $\Theta = \Theta_{min}$ bzw. konstantes $\Theta = \Theta_{max}$. $B^{(12)}$ und $B^{(43)}$ sind Geraden, die bereits für alle Konturpunkte in der konvexen Hülle $\mathfrak{K}(V_1 \cup V_2 \cup V_3 \cup V_4)$ enthalten sind. Für sie wäre daher ein Pufferradius von $q = 0$ ausreichend.
Die Trajektorie $B^{(23)}$ ergibt sich aus $\Theta \in [\Theta_{min}, \Theta_{max}]$ für konstantes $\mathrm{s} = \mathrm{s}_{max}$. Man betrachtet diese Trajektorie zunächst im Referenzpunkt des Fahrzeuges und ermittelt den Abstand von der Geraden zwischen ihren Endpunkten Q_2 und Q_3 (Abbildung 8):

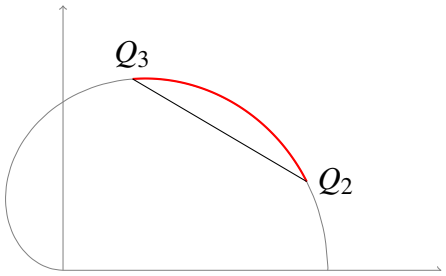


Abbildung 8: Approximation der Trajektorie $B_R(s,\Theta)$ durch die Verbindungsgerade ihrer Endpunkte Q_2 und Q_3 sowie den Abstand $dR \leq \frac{1}{6}\left(\frac{\Theta_{max}}{2} - \frac{\Theta_{min}}{2}\right)^2 \max\{|\mathrm{s}_{max}|;|\mathrm{s}_{min}|\}$ von deren Verbindungsgeraden $\overline{Q_2Q_3}$.

$$B_R(s,\Theta) = B(s,\Theta,\begin{pmatrix}0\\0\end{pmatrix}) \tag{29}$$

$$Q_2 = B_R(s_{max},\Theta_{min}) \tag{30}$$

$$Q_3 = B_R(s_{max},\Theta_{max}) \tag{31}$$

Es lässt sich zeigen, dass jeder Punkt der Trajektorie $B_R^{(23)}$ von der Geraden $\overline{Q_2Q_3}$ höchstens den Abstand $\frac{1}{6}\left(\frac{\Theta_{max}}{2} - \frac{\Theta_{min}}{2}\right)^2 \max\{|\mathrm{s}_{max}|;|\mathrm{s}_{min}|\}$ hat. Dieser Abstand aus der Betrachtung im Referenzpunkt des Fahrzeuges vergrößert sich für die Trajektorie eines Konturpunktes P_i höchstens um den Aufschlag $\left(1-\cos\frac{\Theta_{max}-\Theta_{min}}{2}\right) max\{|P_i|\}$, der die Drehung des Konturpunktes repräsentiert.
Die Gerade $\overline{Q_2Q_3}$ bzw. ihr Äquivalent für einen Konturpunkt P_i liegt in der konvexen Hülle $\mathfrak{K}(V_1 \cup V_2 \cup V_3 \cup V_4)$. Damit ist die Trajektorie $B^{(23)}$ vollständig in dem durch die konvexe Hülle

und den Pufferradius definierten Schutzfeld enthalten. Die Argumentation für $B^{(14)}$ ist entsprechend. Damit entsteht die in (27) angegebene Schranke für den benötigten Pufferradius. Der Beweis dieser Schranke wird als Teil der formalen Verifikation der Implementation geführt. Aus diesem Grund wird an dieser Stelle auf einer ausführliche Begründung verzichtet.

Gemeinsam mit einem zusätzlichen, konfigurierbaren Pufferradiusaufschlag q_{nutzer} ergibt sich damit insgesamt der in Abb. 9 dargestellte Algorithmus zur Berechnung des Schutzfeldes für den Gültigkeitsbereich $[v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$.

Name:	`SCHUTZFELD`
Input:	v_{min},v_{max},ω_{min},ω_{max}, Eckpunkte des konvexen Konturpolygons $P_1, \ldots, P_n$
Output:	Punktmenge W, Pufferradius q (Das Schutzfeld ist die konvexe Hülle der Punkte in W zzgl. des Pufferradius q)

Berechnung:

(i) Berechne Gültigkeitsbereich $[s_{min}, s_{max}] \times [\Theta_{min}, \Theta_{max}]$ aus $[v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$ enstprechend Tab. 1 und Gl. (2)-(4)

(ii) Für $k = 1 \ldots 4$ entsprechend Gl. (23)-(26):

$$V_k = \texttt{SCHUTZFELD\_STheta}(\mathrm{s}_{\{min|max\}}, \Theta_{\{min|max\}}, P_1 \ldots P_n)$$

(iii) Bilde Vereinigung der Schutzfelder

$$W = V_1 \cup V_2 \cup V_3 \cup V_4$$

(iv) Berechne Pufferradius als Approximation der Trajektorien $B^{(12)}$, $B^{(23)}$, $B^{(43)}$, $B^{(14)}$

$$\begin{aligned} q = & \frac{1}{6}\left(\frac{\Theta_{max}}{2} - \frac{\Theta_{min}}{2}\right)^2 \max\{|\mathrm{s}_{max}|; |\mathrm{s}_{min}|\} \\ & + \left(1 - \cos\frac{\Theta_{max} - \Theta_{min}}{2}\right) \max_{1 \le i \le n}\{|P_i|\} \\ & + q_{nutzer} \end{aligned}$$

Abbildung 9: Algorithmus `SCHUTZFELD`

3.3 Schutzfeldaufschläge (Schutzfelderweiterung)

3.3.1 Späterer Bremsbeginn

Die bisher in diesem Konzept beschriebenen Algorithmen haben als Startzeitpunkt der durch das Schutzfeld abzusichernden Bewegung den Startzeitpunkt des Bremsvorganges betrachtet. Dies ist allerdings nicht ausreichend. Vom Zeitpunkt der Anwendung des Schutzfeldes bis zum tatsächlichen Startzeitpunkt des betrachteten Bremsvorganges vergeht ein aus mehreren unterschiedlich begründeten Zeitintervallen zusammengesetzter Zeitraum t_R, in dem sich das Fahrzeug ungebremst mit seiner aktuellen Geschwindigkeit bewegt. (Abb. 10).

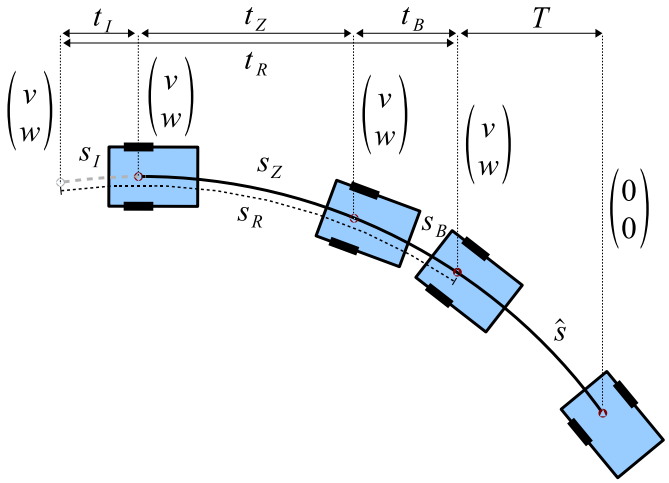


Abbildung 10: Schutzfeldaufschläge

Der Zeitraum t_R ergibt sich aus den folgend aufgeführten Einzelzeiträumen. In der Definition der Zeiträume ist zu beachten, dass die Schutzfeldprüfung bzw. die Bearbeitung eines Zyklus in der Sicherungskomponente zwar einen Zeitraum in Anspruch nimmt, dieser Zeitraum aber nicht gesondert aufgeführt wird. Stattdessen wird die Schutzfeldprüfung, die einmal pro Rechenzyklus ausgeführt wird, mit dem Zeitpunkt zu Beginn des Zyklus identifiziert. Verzögerungen, die sich aus der Dauer bis zum Vorliegen eines Berechnungsergebnisses ergeben, werden in die Reaktionszeiten der aus den Rechenergebnissen entstehenden Signale übernommen/integriert.

t_Z *Zykluszeit*: Zeitraum vom Zeitpunkt der jetztigen Schutzfeldprüfung bis zum Zeitpunkt der nächsten Schutzfeldprüfung

t_B *Reaktionszeit Bremse*: Zeitraum zwischen Zeitpunkt der Schutzfeldprüfung und bis zum (physikalischen) Start der Bremsung durch das Notbremssystem. t_B umfasst sowohl die benötigte Rechenzeit als auch die Reaktionszeit der Notbremseinrichtung.

t_I *Alterung der Inputdaten*: Sowohl Laserscan als auch Geschwindigkeitsvektor werden vor Beginn der Schutzfeldprüfung gemessen. Das Alter dieser Daten wird den Zeitraum t_i beachtet. t_I ist das Maximum der Alter aller Inputgrößen.

Es gilt

$$t_R = t_I + t_Z + t_B \tag{32}$$

Unter der Annahme, dass sich während des Zeitraumes t_R der Geschwindigkeitsvektor nicht ändert, können die Schutzfeldaufschläge durch Ersetzen des Weges s durch $\hat{s} = s_R + s$ in Gleichung (3) in die Schutzfeldberechnung integriert werden. Das heißt, dass sich die bisher vorgestellten Algorithmen nicht verändern mit der Ausnahme, dass in die Gleichungen (4) und (5) ein verlängerter Bremsweg $\hat{s}$ eingesetzt werden muss.

3.3.2 Messungenauigkeit Laserscanner

Der maximale Messfehler des Laserscanners wird nach vollständiger Berechnung des Schutzfeldes auf die Längenangabe für jede Strahlrichtung aufgeschlagen (vgl. auch Darstellung des Schutzfeldes als Laserscan).

3.3.3 Keine weiteren Aufschläge

Über die in den vorangegangenen Abschnitten vorgestellten Schutzfeldaufschläge hinaus werden keine weiteren Schutzfeldaufschläge benötigt. Grund dafür ist die Beachtung der verschiedenen Reaktionszeiten bzw. Fehlerquellen an anderer Stelle. Hier sei speziell noch einmal darauf hingewiesen, dass Meßfehler für v und ω behandelt werden, indem das ermittelte Geschwindigkeitsintervall um die möglichen Messfehler vergrößert wird. Tabelle 2 enthält eine Übersicht über die Behandlung von Reaktionszeiten und Fehlern.

Typ	Bezeichnung	Behandlung
RZ	Zykluszeit	t_R (t_Z)
RZ	Reaktionszeit Aktorik	t_R (t_B)
RZ	Rechenzeit für Überprüfung des Schutzfeldes	t_R (t_B)
RZ	veraltete Laserscannmessung	t_R (t_I)
RZ	veraltete Geschwindigkeitsmessung	t_R (t_I)
ERR	Bremsverschleiß	s in Bremsmodell
ERR	Rechenfehler / Rundungsfehler im Algorithmus	in Implementation behandelt, daher nicht explizit aufgeführt
ERR	Messfehler $(v, \omega)^T$	Erweiterung des Geschwindigkeitsbereiches
ERR	Messfehler Laserscan	Schutzfeldaufschlag (vgl. Abschnitt 3.3.2)
ERR	Ungenauigkeit der Roboterkontur	nicht betrachtet, da der Nutzer diese korrekt eingeben, d.h. evtl. um den Messfehler erweitern muss

Tabelle 2: Übersicht über die Behandlung von Fehler (ERR) und Reaktionszeiten (RZ)

3.4 Abtastung des Schutzfeldes

3.4.1 Überblick

Die Fläche des Schutzfeldes ist vor der Abtastung definiert durch die konvexe Hülle einer Punktmenge zzgl. eines Pufferradius q (Frese u. Täubig, 2009a,b). Dabei umfasst das Schutzfeld alle die Punkte der Ebene, die von der konvexen Hülle höchstens die Entfernung q besitzen. Formal ist die Fläche damit wie folgt als Punktmenge definiert:

$$F\left(q;(p_i)_{i=1}^n\right) = \left\{ p_q + \sum_{i=1}^{n} \lambda_i p_i \,\middle|\, \lambda_i \geq 0 \forall i, \quad \sum_{i=1}^{n} \lambda_i = 1, \quad |p_q| \leq q \right\} \tag{33}$$
$$q > 0, \quad p_i \in \mathbb{R}^2 \forall i$$

Eine geometrische Deutung ergibt sich als Minkowski-Summe der Fläche eines konvexen Polygons P (Repräsentant der konvexen Hülle) mit einem Kreis des Radius q (siehe Abbildung 11). Im Folgenden wird die sich ergebende Fläche des konvexen Polygons P zzgl. Pufferradius q auch als erweitertes Polygon P^+ bezeichnet. Der Rand des erweiterten Polygons P^+ besteht aus den Kanten $\left\{k_i^+\right\}$ und den Bögen $\left\{b_i^+\right\}$, die sich aus der Erweiterung der Kanten $\{k_i\}$ und der Eckpunkte $\{P_i\}$ des Polygons P ergeben.

Die Fläche des erweiterten Polygons P^+ wird mittels des Abtastalgorithmus in eine Laserscan-Darstellung umgewandelt, welche im Folgenden als minimaler Scan bezeichnet wird. Der minimale Scan definiert eine Fläche, die auf den Öffnungswinkel (Sichtbereich) des Laserscanners beschränkt ist. Der minimale Laserscan überdeckt die Fläche des Schutzfeldes innerhalb des Öffnungswinkels vollständig. Dazu wird die in Abschnitt 3.4.3 erläuterte Repräsentation verwendet.

Der Algorithmus zur Abtastung des Schutzfeldes in einen minimalen Scan arbeitet in zwei Schritten: zunächst wird die konvexe Hülle der Punktmenge durch ein konvexes Polygon repräsentiert. Dazu wird der Graham-Scan-Algorithmus verwendet (siehe Abschnitt 3.4.4). Danach wird das konvexe Polygon zzgl. Pufferradius von einem festen Punkt aus (Scanzentrum) mit einer festen Winkelauflösung in Sektoren geteilt und in jedem dieser Sektoren die maximale radiale Ausdehnung des Schutzfeldes ermittelt (vgl. Abb. 12 links). Der dazu verwendete Algorithmus ist in Abschnitt 3.4.5 beschrieben.

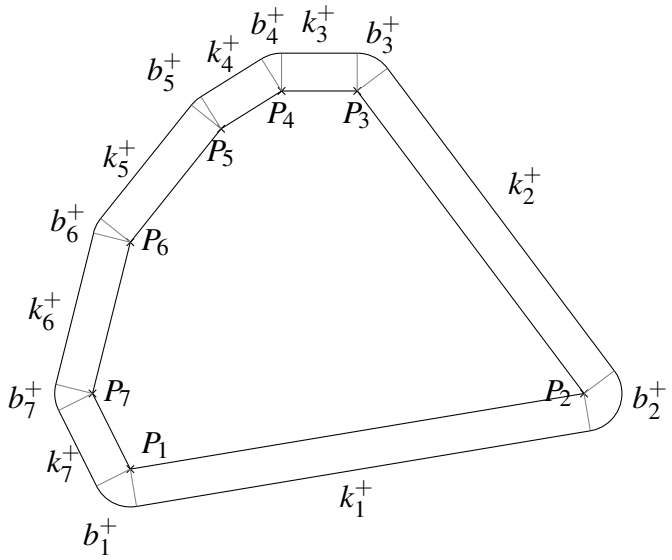


Abbildung 11: Das erweiterte Polygon P^+ stellt das Schutzfeld als konvexe Hülle zzgl. Pufferradius dar. Sein Rand besteht aus den Kanten $\{k_i^+\}$ und den Bögen $\{b_i^+\}$, die sich aus der Erweiterung der Kanten $\{k_i\}$ und der Eckpunkte $\{P_i\}$ des Polygons P ergeben.

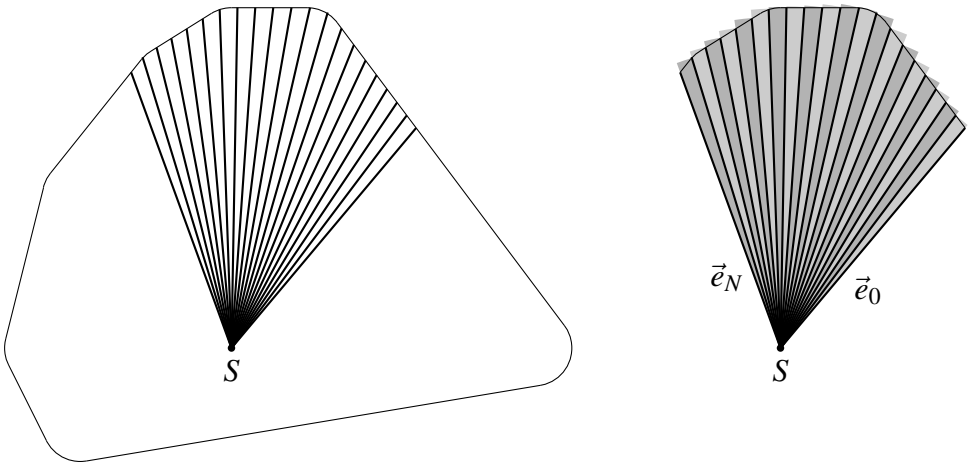


Abbildung 12: Der minimale Scan (*rechts*) gibt in jedem Sektor die maximale radiale Ausdehnung des Schutzfeldes (*links*) an. Der minimale Scan ist eine Obermenge des Teils des Schutzfeldes, der innerhalb des Öffnungswinkels des Laserscanners liegt.

3.4.2 Bezeichnungen

$\mathfrak{K}(Z_1,\ldots,Z_m)$	konvexe Hülle der Punkte $Z_1,\ldots,Z_m$
Polygon $P = P_1..P_n$	konvexes Polygon mit den Eckpunkten $P_1,\ldots,P_n$ (Eckpunkte im mathematischen Drehsinn (gegen Uhrzeigersinn) mit beliebigem beginnend geordnet)
k_i	Kante $\overline{P_iP_{i+1}}$ des Polygon P
erweitertes Polygon $P^+ = P_1..P_n \oplus q$	das konvexe Polygon P zzgl. Pufferradius q. Entsteht durch Minkowski-Plus der Fläche von P mit der Fläche eines Kreises von Radius q.
k_i^+	Kante des erweiterten Polygons P^+. k_i^+ entsteht aus k_i durch Verschiebung um Entfernung q in Richtung der negativen Normalen von k_i (normierter Richtungsvektor von k_i um -90° gedreht).
b_i^+	Bogen des erweiterten Polygons P^+. b_i^+ entsteht als Erweiterung der Ecke P_i.
minimaler Scan	Schutzfeld in Laserscan-Darstellung
Richtung $\vec{e}_j$	normierter Richtungsvektor, der die Grenze zwischen den benachbarten Sektoren $j-1$ und j des minimalen Laserscan angibt

3.4.3 Repräsentation Minimaler Laserscan

Die Repräsentation des minimalen Laserscan basiert auf der Interpretation eines Laserscanners als Flächensensor. Damit wird ein einzelner Eintrag eines Laserscans ausgehend vom Scanzentrum nicht einem Strahl sondern einem Kreissektor zugeordnet. Diese Kreissektoren teilen den Öffnungswinkel des Laserscanners in Winkelbereiche, die sich nicht überscheiden und in ihrer Summe den gesamten Öffnungswinkel abdecken. Die Winkelbereiche der Sektoren sind außerdem alle gleich groß, da von einer gleichmäßigen Winkelauflösung des Laserscanners ausgegangen wird.

Der minimale Laserscan ist nun ein Array, in dem Entfernungen gespeichert sind. Jeder Eintrag des Arrays ist dem entsprechenden Kreissektor zugeordnet und beschreibt den minimalen Freiraum, der in diesem Sektor gemessen werden muss, um Kollisionsfreiheit zu garantieren. Der minimale Laserscan ist damit eine Darstellung des Anteils des Schutzfeldes, der sich im Öffnungswinkel des Laserscanners befindet (vgl. Abb. 12 rechts). Um die mathematische Korrektheit dieser Darstellung des Schutzfeldes zu gewährleisten, stellt sie im Öffnungswinkel des Laserscanners eine Obermenge des erweiterten Polygones P^+ (ebenfalls ein Repräsentation des Schutzfeldes) dar.

Ein Algorithmus zur Detektion von Schutzfeldverletzungen könnte also direkt die Werte des minimalen Laserscan mit den Meßwerten des Laserscanners vergleichen. Eine Schutzfeldverletzung würde vorliegen, sobald einer oder mehrere Meßwerte des Laserscan kleiner oder gleich des zugehörigen Eintrages im minimalen Scan sind. Dieser Algorithmus ist aber nicht Teil der zertifizierten Software zur Schutzfeldberechnung.

Die Nummerierung der Sektoren erfolgt aufsteigend im positiven Drehsinn und jeder Sektor j ist

begrenzt durch die Richtungsvektoren $\vec{e}_j$ und $\vec{e}_{j+1}$, die die Richtung der rechten und linken Sektorgrenze darstellen. Die Liste der Sektorgrenzen hat damit $N+1$ Einträge und ist ebenfalls im positven Drehsinn sortiert. Die erste Sektorgrenze $\vec{e}_0$ ist gleichzeitig die rechte Grenze des Öffnungswinkels und die letzte Sektorgrenze $\vec{e}_N$ gleichzeitig die linke Grenze des Öffnungswinkels. Alle Sektorgrenzen $\{\vec{e}_j\}$ sind aus den Parametern des Laserscanners eindeutig berechenbar.

3.4.4 Konvexe Hülle als Polygon

Die Repräsentation der konvexen Hülle in einem konvexen Polygon wird durch Anwendung des Graham-Scan-Algorithmus erzeugt. Diesem wird eine Liste von Punkten übergeben. Das Ergebnis ist eine geordnete Liste von Punkten, die die Eckpunkte des Polygons P sortiert im geometrischen Drehsinn beinhaltet. Die Fläche dieses Polygons P ist die konvexe Hülle der Punkte, die dem Algorithmus übergeben wurden.
Die wichtigsten Eigenschaften:

- Konvexe Hülle der Ergebnispunkte ist gleich der konvexen Hülle der Eingabepunkte

$$\mathfrak{K}(Z_1,\ldots,Z_m) = \mathfrak{K}(P_1,\ldots,P_n) \tag{34}$$

- Es werden keine neuen Punkte konstruiert

$$\{P_1,\ldots,P_n\} \subseteq \{Z_1,\ldots,Z_m\} \tag{35}$$

- Fläche von P ist die konvexe Hülle der Eckpunkte $P_1,\ldots,P_n$. Die Eckpunkte sind im geometrischen Drehsinn geordnet.

$$P = \mathfrak{K}(P_1,\ldots,P_n) \tag{36}$$

- *nicht benötigt:* Ergebnismenge ist minimal

$$\forall j : \mathfrak{K}(\bigcup_{i\neq j} P_i) \subset \mathfrak{K}(\bigcup_{i} P_i) \tag{37}$$

Nach Ausführung des Graham-Scan ist das erweiterte Polygon P^+ definiert durch die Ergebnispunkte $P_1,\ldots,P_n$ sowie den gesondert zu ermittelnden Pufferradius q.

3.4.5 Abtastung des konvexen Polygons zzgl. Pufferradius

Grundprinzip Der Algorithmus folgt einem Sweepline-Prinzip. Die Sweepline ist ein Strahl, der, immer vom Scanzentrum S ausgehend, die Orientierungen zwischen rechtem und linkem Rand des Öffnungswinkels im positiven Drehsinn durchläuft. Dabei werden alle Sektoren des Laserscan durchlaufen. Gleichzeitig werden die Kanten und Bögen des erweiterten Polygons P^+ überstrichen. Beim Durchlaufen jedes einzelnen Sektors wird unter Behandlung der dabei überstrichenen Kanten k_i^+ und Bögen b_i^+ die maximale radiale Ausdehnung innerhalb dieses Sektors ermittelt und bei Erreichen des linken Sektorrandes gespeichert.

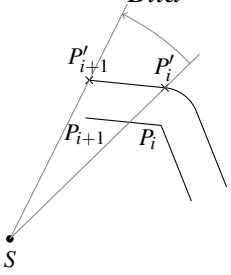
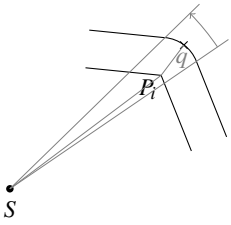
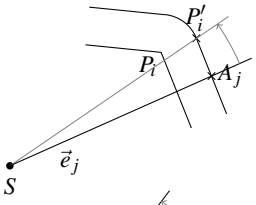
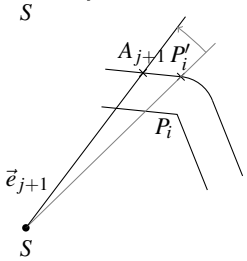
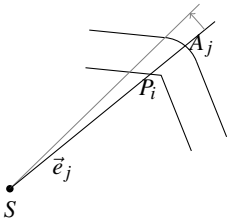
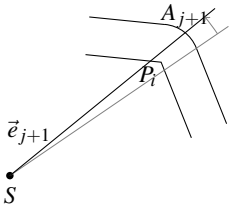
Beschreibung	*Bild*	*Abschätzung*
vollständig überstrichene Kante k_i^+		$d_{max} \leq \max\left(\lvert P'_i - S\rvert, \lvert P'_{i+1} - S\rvert\right)$
vollständig überstrichener Bogen b_i^+		$d_{max} \leq \lvert P_i - S\rvert + q$
von rechtem Sektorrand geschnittene Kante		$d_{max} \leq \max\left(\lvert A_j - S\rvert, \lvert P'_i - S\rvert\right)$
von linkem Sektorrand geschnittene Kante		$d_{max} \leq \max\left(\lvert P'_i - S\rvert, \lvert A_{j+1} - S\rvert\right)$
von rechtem Sektorrand geschnittener Bogen		$d_{max} \leq \lvert P_i - S\rvert + q$ (wie vollständig überstrichener Bogen)
von linkem Sektorrand geschnittener Bogen		$d_{max} \leq \lvert P_i - S\rvert + q$ (wie vollständig überstrichener Bogen)

Tabelle 3: Überabschätzung der maximalen radialen Entfernung einzelner Teile des erweiterten Polygons P^+

Beschreibung des Algorithmus

1. Initialisiere i

   Finde den Bogen b_i^+ oder die verschobene Kante k_i^+, die von der Richung $\vec{e}_0$ geschnitten wird.

2. Durchlaufe die Sektoren $j = 0, \ldots, N-1$ und bestimme die maximale radiale Ausdehnung innerhalb des Sektors j

   Bei Eintritt in den Sektor j ist $i = i_1$ und entweder $b_{i_1}^+$ oder $k_{i_1}^+$ werden von der Richtung des rechten Sektorrandes $\vec{e}_j$ geschnitten. Nun werden abwechselnd Kanten und Bögen, die vollständig innerhalb des Sektors liegen, überstrichen. Dies endet bei $i = i_2$, wenn der Bogen $b_{i_2}^+$ oder die verschobene Kante $k_{i_2}^+$ erreicht wurde, die vom linken Sektorrand $\vec{e}_{j+1}$ geschnitten wird. Die maximale radiale Ausdehnung in Sektor j ergibt sich nun aus dem Maximum der radialen Entfernungen der vollständig überstrichenen Bögen b_i^+ und Kanten k_i^+ $(i_1 < i < i_2)$ sowie den maximalen Entfernungen der zum Sektor gehörigen Teile der Bögen oder Kanten an den Sektorrändern ($b_{i_1}^+$ bzw. $k_{i_1}^+$ und $b_{i_2}^+$ bzw. $k_{i_2}^+$).

   Die Abschätzungen der maximalen radialen Entfernungen der einzelnen Teile von P^+ werden dabei unter Verwendung der Überabschätzungen in Tabelle 3 ermittelt. Die Abschätzung der vollständig überstrichenen Kanten k_i^+ braucht allerdings nicht durchgeführt zu werden, da deren maximale Entfernung an einem der Endpunkte P_i' und P_{i+1}' erreicht und diese in jedem Fall durch die angrenzenden Bögen mit abgeschätzt werden.

In Abb. 13 ist der vollständige Algorithmus zur Abtastung des Schutzfeldes dargestellt. Darin werden für die Entscheidung, ob eine Sektorgrenze $\vec{e}_j$ die Kante k_i^+ schneidet, die Ungleichungen

$$|P_{i+1} - P_i| d_1 < -q(s_1 - s_2) \tag{38}$$

$$|P_{i+1} - P_i| d_2 \geq -q(s_1 - s_2) \tag{39}$$

verwendet (Erklärung folgt). Sind beide Ungleichungen erfüllt, so schneidet die Richtung $\vec{e}_j$ die Kante k_i^+. Diese Ungleichungen sowie die Berechnung der Entfernung des Schnittpunktes zwischen $\vec{e}_j$ und k_i^+ werden im Folgenden begründet.

Schnitt mit der Kante k_i^+ Numerische Basis des Streckenschnittalgorithmus der erweiterten Kante k_i^+ mit der Richtung $\vec{e}_j$ bilden die Projektionen der Endpunkte P_i und P_{i+1} der Kante k_i des nicht erweiterten Polygons P auf die Richtung $\vec{e}_j$ sowie deren Normalen $\vec{e}_j^{+90}$ (Abbildung 14).

$$\vec{e}_j^{+90} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \vec{e}_j \tag{40}$$

$$s_1 = \left\langle P_i, \vec{e}_j \right\rangle \tag{41}$$

$$s_2 = \left\langle P_{i+1}, \vec{e}_j \right\rangle \tag{42}$$

$$d_1 = \left\langle P_i, \vec{e}_j^{+90} \right\rangle \tag{43}$$

$$d_2 = \left\langle P_{i+1}, \vec{e}_j^{+90} \right\rangle \tag{44}$$

Name: `SCHUTZFELD_ABTASTEN`

Input: Punktmenge W, Puferradius q
Output: minimaler Laserscan (Distanzliste D)

Berechnung:

(i) $P_1 \ldots P_n = \texttt{GRAHAM\_SCAN}(W)$

(ii) Initialisiere i

(a) $i = 0$

(b) Solange die Anfangspunkte von Bogen b_i^+ und Kante k_i^+ links von $\vec{e}_0$ liegen:

$$i = (i-1) \mod n$$

(c) Solange die Endpunkte von Bogen b_i^+ und Kante k_i^+ rechts von $\vec{e}_0$ liegen:

$$i = (i+1) \mod n$$

(d) Falls $\vec{e}_0$ schneidet b_i^+:

$$a_0 = |P_i| + q$$

Sonst $\vec{e}_0$ schneidet k_i^+:

$$a_0 = \texttt{Schnittpunktentfernung}(\vec{e}_0, k_i)$$

(iii) Berechne Distanz für jeden Sektor

Für $j = 0, \ldots, n-1$:

(a) Entfernung am rechten Sektorrand:

$$d_{max} = a_j$$

(b) Vollständige Bögen und Kanten überstreichen:

Solange die Endpunkte von Bogen b_i^+ und Kante k_i^+ rechts von $\vec{e}_j$ liegen:

$$i = (i+1) \mod n$$
$$d_{max} = \max(d_{max}, |P_i| + q)$$

(c) Entfernung am linken Sektorrand:

Falls $\vec{e}_{j+1}$ schneidet b_i^+:

$$a_{j+1} = |P_i| + q$$

Sonst $\vec{e}_{j+1}$ schneidet k_i^+:

$$a_{j+1} = \texttt{Schnittpunktentfernung}(\vec{e}_j, k_i)$$

(d)

$$D_j = \max(a_j, d_{max}, a_{j+1})$$

Abbildung 13: Algorithmus `SCHUTZFELD_ABTASTEN`. Das Scannzentrum befindet sich bei $S = (0,0)^T$.

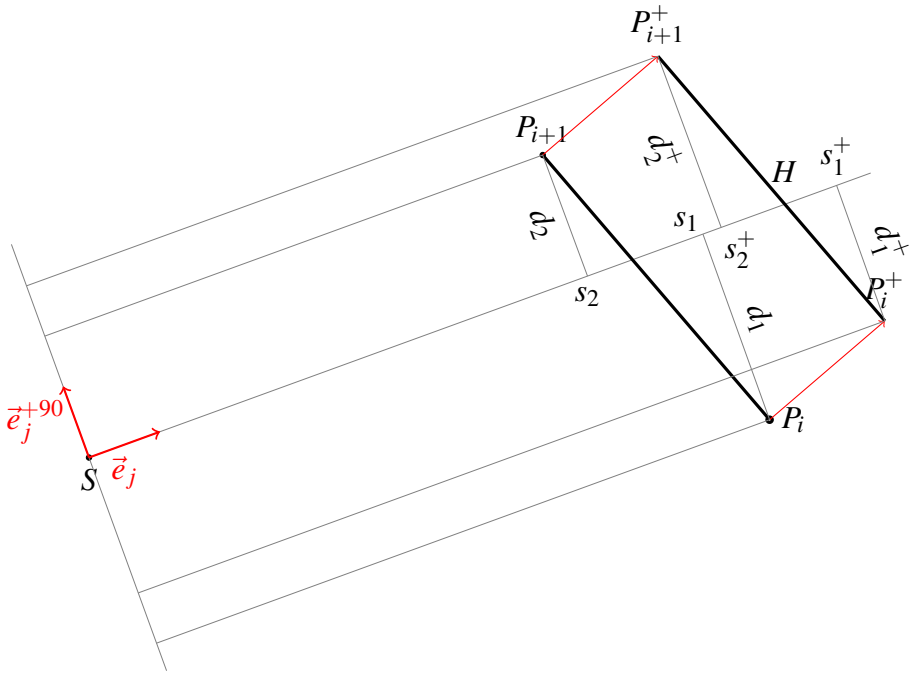


Abbildung 14: Schnitt einer Richtung mit einer Kante k^+ des erweiterten Polygons P^+.

Die Kante k_i^+ des erweiterten Polygons P^+ ergibt sich aus k_i durch eine Verschiebung um den Pufferradius q in Richtung $\vec{n}$. $\vec{n}$ ist dabei der um -90° gedrehte normierte Richtungsvektor der Kante k_i.

$$\vec{n} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \frac{P_{i+1} - P_i}{|P_{i+1} - P_i|} \tag{45}$$

$$P_i^+ = P_i + q \cdot \vec{n} \tag{46}$$

$$P_{i+1}^+ = P_{i+1} + q \cdot \vec{n} \tag{47}$$

Der Richtungsvektor $\vec{n}$ wird nur dann berechnet, wenn die Kante k_i eine gewisse Mindestlänge besitzt und die Berechnung dadurch numerisch ohne Probleme möglich ist. Eine Sonderbehandlung für kürzere Kanten wird im letzten Absatz dieses Abschnittes beschrieben.

Für die Projektionen der Kante k_i^+ des erweiterten Polygons P^+ auf die Abtastrichtung $\vec{e}_j$ sowie

deren Normale $\vec{e}_j^{+90}$ gilt dann

$$s_1^+ = \langle P_i^+, \vec{e}_j \rangle \tag{48}$$
$$= \langle P_i, \vec{e}_j \rangle + q \langle \vec{n}, \vec{e}_j \rangle \tag{49}$$
$$s_2^+ = \langle P_{i+1}, \vec{e}_j \rangle + q \langle \vec{n}, \vec{e}_j \rangle \tag{50}$$
$$d_1^+ = \left\langle P_i, \vec{e}_j^{+90} \right\rangle + q \left\langle \vec{n}, \vec{e}_j^{+90} \right\rangle \tag{51}$$
$$d_2^+ = \left\langle P_{i+1}, \vec{e}_j^{+90} \right\rangle + q \left\langle \vec{n}, \vec{e}_j^{+90} \right\rangle \tag{52}$$
$$\tag{53}$$

Mit Hilfe der so definierten Projektionen lässt sich nun sowohl beschreiben, wann eine Abtastrichtung $\vec{e}_j$ eine Kante k_i^+ des erweiterten Polygons P^+ schneidet, als auch die Entfernung berechnen, in der sie dies tut.

Die Kante k_i^+ wird von der Verlängerung der Richtung $\vec{e}_j$ geschnitten, wenn gilt

$$d_1^+ < 0 \qquad \text{und} \tag{54}$$
$$d_2^+ \geq 0 \tag{55}$$

Diese Ungleichungen sind äquivalent zu den verwendeten Ungleichungen

$$|P_{i+1} - P_i| d_1 < -q(s_1 - s_2) \qquad \text{und} \tag{56}$$
$$|P_{i+1} - P_i| d_2 \geq -q(s_1 - s_2) \tag{57}$$

Entsprechende Ungleichungen für aufeinanderfolgende Kanten können auch genutzt werden, um festzustellen, ob die Richtung $\vec{e}_j$ zwischen dem Anfangspunkt einer Kante k_i^+ und dem Endpunkt der vorhergehenden Kante k_{i-1}^+ hindurchführt, also ob sie den Bogen b_i^+ schneidet.

Die Entfernung des Schnittpunktes H der Kante k_i^+ mit der Verlängerung von $\vec{e}_j$ ist

$$\overline{SH} = s_2^+ + \frac{d_2^+}{d_2^+ - d_1^+}(s_1^+ - s_2^+) \tag{58}$$
$$= s_2^+ + \frac{d_2^+}{d_2 - d_1}(s_1 - s_2) \tag{59}$$

Eine *Sonderbehandlung sehr kurzer Kanten* k_i wird durchgeführt, falls deren Länge eine numerisch bedingte Mindestlänge `MINIMALE_KANTENLAENGE` unterschreitet. Diese Sonderbehandlung ist nur für die Entfernungsberechnung des Schnittpunktes notwendig. Die Ungleichungen (56) und (57) zum Test der Lage einer verschobenen Kante k_i^+ bezüglich der Richtung $\vec{e}_j$ sind so formuliert, dass sie ohne numerische Probleme für beliebige $P_i \neq P_{i+1}$ verwendet werden dürfen. Die Entfernung des Schnittpunktes wird für sehr kurze Kanten statt der Verwendung von Gleichung (59) durch

$$\overline{SH} \leq \max\left(|P_i^+|, |P_{i+1}^+|\right) \tag{60}$$
$$\leq \max\left(|P_i| + q, |P_{i+1}| + q\right) \tag{61}$$

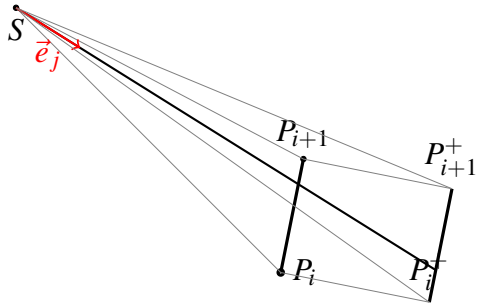


Abbildung 15: Obere Schranke der Entfernung des Schnittpunktes einer Richtung $\vec{e}_j$ mit einer Kante k_i^+ des erweiterten Polygons P^+.

überabgeschätzt (vgl. Abb. 15). Diese Abschätzung ist immer dann möglich, wenn bereits bekannt ist, dass die Richtung $\vec{e}_j$ die erweiterte Kante k_i^+ schneidet. Diese Voraussetzung ist bei Aufruf des Teilalgorithmus immer erfüllt.

Eine weitere numerische Sonderbehandlung erfolgt, falls $d_2 - d_1$ kleiner als `MINIMALE_KANTENLAENGE` ist. In diesem Fall ist die Richtung $\vec{e}_j$ fast parallel zur Kante k_i^+ und es wird die obere Grenze

$$\overline{SH} \leq \max\left(s_1^+, s_2^+\right) \tag{62}$$

verwendet.

Der vollständige Algorithmus zur Berechnung der Entfernung des Schnittpunktes zwischen $\vec{e}_j$ und k_i^+ ist in Abb. 16 beschrieben.

Literatur

[Frese u. Täubig 2009a] FRESE, U. ; TÄUBIG, H.: *Verfahren zur Vermeidung von Kollisionen gesteuert beweglicher Teile einer Anlage*. Patentanmeldung beim Deutschen Patentamt unter 102009006256.4-32, 2009. – (eingereicht)

[Frese u. Täubig 2009b] FRESE, U. ; TÄUBIG, H.: Verfahren zur Vermeidung von Kollisionen gesteuert beweglicher Teile einer Anlage / Deutsches Forschungszentrum für Künstliche Intelligenz; Robert-Hooke-Strasse 5, 28359 Bremen; http://www.dfki.de. 2009 (RR-09-01). – Technischer Report. – ISSN 0946–008X

[Täubig 2009] TÄUBIG, Holger: *Konzeptpapier Bremsmodell*. SAMS Projektdokumentation, 2009. – DFKI, Forschungsbereich Sichere Kognitive Systeme

Name:	`Schnittpunktentfernung`
Input:	Richtung $\vec{e}_j$, Kante k_i von P
Output:	Obere Schranke d_{max} der Entfernung des Schnittpunktes von $\vec{e}_j$ mit der verschobenen Kante k_i^+

Berechnung:

(i) Falls $|P_{i+1} - P_i| <$`MINIMALE_KANTENLAENGE`

$$d_{max} = \max\left(|P_i| + q, |P_{i+1}| + q\right)$$

Sonst falls $d_2 - d_1 <$`MINIMALE_KANTENLAENGE`

$$d_{max} = \max\left(s_1^+, s_2^+\right)$$

Sonst

$$d_{max} = s_2^+ + \frac{d_2^+}{d_2 - d_1}(s_1 - s_2)$$

Abbildung 16: Algorithmus `Schnittpunktentfernung`

Sicherungskomponente für
Autonome Mobile Systeme

Eine Kooperation zwischen
DFKI-Labor Bremen • Leuze lumiflex • Universität Bremen

Anwenderhandbuch

Zusammenfassung

Das SAMS-Schutzfeldberechnungsmodul ist ein Softwaremodul zur Berechnung von geschwindigkeitsabhängigen Schutzfeldern. Es ist für den Einsatz in Sicherheitssystemen bis zu SIL-3 zertifiziert.

Dieses Benutzerhandbuch enthält die für Endanwender und Systemintegratoren nötigen Informationen für den Einsatz des Moduls. Es beschreibt die Einsatzmöglichkeiten, die notwendigen Voraussetzugen für seinen Einsatz, die Definition der benötigten Programmierschnittstellen und die bei der Softwareintegration zu beachtenden Punkte.

Projektbezeichnung	SAMS
Verantwortlich	Alle
Erstellt am	17.02.2009
Version	1.5
Bearbeitungszustand	fg.
Revision	4326
Letzte Änderung	08.10.2009
Dokumentablage	Projektdokumente/Anwenderhandbuch/Anwenderhandbuch.tex

Änderungsliste

17.02.09 – *Version:* 0.1 – *Bearbeiter:* Alle

Initiale Version.

25.03.09 – *Version:* 1.0 – *Bearbeiter:* Alle

Bearbeitungen der einzelnen Abschnitte durch die jeweils Verantwortlichen abgeschlossen und zur Prüfung bereit.

07.04.09 – *Version:* 1.1 – *Bearbeiter:* C. Lüth, H. Täubig

Pseudo-Code eingefügt, wie in Prüfung vom 26.03.09 angeregt; Datenstruktur `Einstellungen` erweitert

24.06.09 – *Version:* 1.2 – *Bearbeiter:* D. Walter

Änderungen (3., 5., 6.) aus Prüfung vom 23.06.09

05.07.09 – *Version:* 1.3 – *Bearbeiter:* D. Walter

Datenbeschreibung Funktion schutzfeld_berechnen konkretisiert.

06.07.09 – *Version:* 1.4 – *Bearbeiter:* C. Lüth

Abschnitt “Sicherheitshinweise” eingefügt.

– *Version:* 1.5 – *Bearbeiter:* H. Täubig

Abschnitt mit weiteren “Anforderungen an die Nutzerdaten” eingefügt. (Es handelt sich um eine Tabelle, die zusätzliche während der Konfiguration geprüfte Anforderungen an die vom Nutzer übergebenen Parameter dokumentiert.); Nicht verwendeten Scannerparameter *ausrichtung* abgeschafft; Nutzerparameter protokoll_level korrigiert

21.08.09 – *Version:* 1.5 – *Bearbeiter:* C. Hertzberg

Anpassung der int32-Definition als mögliche Änderung hinzugefügt.

Prüfverzeichnis

26.03.09 – *Version:* 1.0 – *Prüfer:* D. Walter
Neuer Produktzustand: **i. B.**

Korrekturgelesen; kleine Änderungen an Formulierungen; Einfügen von Pseudo-Code / Codeausschnitten einer realen Verwendung angeregt.

23.06.09 – *Version:* 1.1 – *Prüfer: Alle* – $\{U.Frese\}$
Neuer Produktzustand: **i. B.**

Anmerkungen aus Review des TÜV einzupflegen:

1. Abschnitt “Sicherheit” vervollständigen
2. SIL-3 Hinweis in Abschnitt “Sicherheit”
3. 4.6.3: Auf grafische Prüfungsmöglichkeit hinweisen; wird explizit nicht vom SW-Modul geleistet
4. Spezifikation (tabellarisch) Ein-/Ausgabedaten der Funktionen
5. Verweis auf Laserscannerhandbuch als Referenz einfügen
6. Abb. 1: Hinweis auf Beweglichkeit einfügen

06.07.09 – *Version:* 1.3 – *Prüfer:* C. Lüth
Neuer Produktzustand: **i. B.**

Durchführung der Änderungen (3.)– (6.) von Liste oben geprüft.

06.07.09 – *Version:* 1.4 – *Prüfer:* D. Walter
Neuer Produktzustand: **vg.(TÜV)**

Neuen Abschnitt “Sicherheitshinweise” korrekturgelesen. Anmerkungen über Codeänderungen ausgelagert in Abschnitt “Softwareintegration” (4.6)

23.08.09 – *Version:* 1.5 – *Prüfer:* C. Lüth
Neuer Produktzustand: **vg. (TÜV)**

Änderungen geprüft, keine Anmerkungen.

25.09.2009 – *Version:* 1.2 – *Prüfer:* TÜV (P. Supavatanakul)
Neuer Produktzustand: **fg.**

Siehe Prüfbericht (Techical Report no. LF82764T)

Inhaltsverzeichnis

1 Einleitung

Dieses Handbuch enthält die für den Einsatz des SAMS-Schutzfeldberechnungsmoduls nötigen Informationen. Das Handbuch ist wie folgt gegliedert:

- Abschnitt 2: Hinweise zur Sicherheit und Normenkonformität;
- Abschnitt 3: mögliche Applikationen und Voraussetzungen für den Einsatz;
- Abschnitt 4: Details der Systemintegration (Programmierschnittstellen, Spezifikation der Funktionalität);
- Anhang: Index der benutzten Begriffe, mit Referenz auf das definierende Auftreten.

2 Sicherheitshinweise

2.1 Normen

Beim Einsatz der in diesem Anwenderhandbuch beschriebenen Software sind unter anderem folgende Normen und Richtlinien relevant:

- IEC 61508 — Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/-programmierbar elektronischer Systeme
- IEC 61496 — Sicherheit von Maschinen — Berührungslos wirkende Schutzeinrichtungen
- 98/37/EG — Maschinenrichtlinie

2.2 Bestimmungsgemäße Verwendung

Die SAMS-Schutzfeldberechung ist ein Softwaremodul zur Berechnung von geschwindigkeitsabhängigen Schutzfeldern, die mit einem Sicherheitslaserscanner berührungslos überwacht werden können.
Die SAMS-Schutzfeldberechnung ist ein reines Softwaremodul. Dieses kann in eine Applikation zur Fahrzeugsicherung bis zum Sicherheitslevel SIL-3 nach IEC 61508 integriert werden, wenn die genutzte Hardware, die Systemumgebung und die Systemintegration den Anforderungen der Abschnitte 4.2 und 4.6 entsprechen.

2.3 Sachkundige Person: Systemintegrator

Der Systemintegrator ist die für die Integration der Schutzfeldberechnung in eine Applikation (Endprodukt; Fahrzeugsicherung) verantwortliche Person. Er oder sie muss mit dem Stand der

Technik, insbesondere im Sicherheitsbereich, und den relevanten Normen vertraut sein, und eine geeignete technische Ausbildung oder hinreichende einschlägige Erfahrung besitzen.

Diese Handbuch richtet sich im wesentlichen an den Systemintegrator, enthält aber auch für den Endanwender relevante Informationen. Der Systemintegrator hat dafür Sorge zu tragen, dass sich diese Informationen im Nutzerhandbuch der integrierten Applikation wiederfinden.

Die Verantwortung für die Normenkonformität liegt beim Systemintegrator; das SAMS-Schutzfeldberechnungsmodul ist kein zertifizierungsfähiges oder zertifiziertes Produkt.

2.4 Grenzen der Anwendung

Anforderungen an die Hardware, auf der diese Software betrieben werden soll, sind in Abschnitt 4.2.1 aufgeführt.

Das Bremsverhalten des Fahrzeuges, das mithilfe dieser Software gesichert wird, muss die in Abschnitt 3.2.2 dargelegten Annahmen erfüllen. Abschnitt 3.2 beschreibt die Voraussetzungen an Fahrzeuge genauer, und legt insbesondere die Fahrzeugtypen dar, für die prinzipiell eine Fahrwegsicherung mit der SAMS-Schutzfeldberechnung möglich ist.

3 Applikationsteil

3.1 Beschreibung der Applikation

Das Haupteinsatzgebiet des Moduls ist im Bereich der mobilen *Gefahrbereichssicherung* eines sich autonom bewegenden Fahrzeugs, beispielsweise eines fahrerlosen Transportsystems (FTS).

3.1.1 Mobile Gefahrbereichssicherung

Die mobile Gefahrbereichssicherung schützt Personen, die sich im Fahrweg eines Fahrzeugs befinden. Ein *Schutzfeld* wird durch einen Sicherheitslaserscanner überwacht; die Distanz zwischen Schutzfeldvorderkante und Fahrzeugfront muss für jede Geschwindigkeit größer sein als der Anhalteweg des Fahrzeugs. Abb. 1 zeigt ein mögliches Einsatzszenario für ein freibewegliches fahrerloses Transportfahrzeug. Stand der Technik ist hier eine kleine, feste Anzahl von Schutzfeldern. Dies ist für Geradeausfahrt mit geringen Geschwindigkeiten ausreichend, aber nicht bei Kurvenfahrten oder höheren Geschwindigkeiten. Um die Sicherheit zu gewährleisten müssten die Schutzfelder hier derart überabgeschätzt werden, dass die Geschwindigkeit und Manövrierfähigkeit des Fahrzeugs zu stark eingeschränkt werden. Die geschwindigkeitsabhängigen Schutzfelder sind dagegen jederzeit so klein wie möglich, aber so groß wie nötig.

Der *Anhalteweg* D_A eines Fahrzeugs ist die Strecke, die das Fahrzeug vom Auslösen eines Bremssignals bis zum völligen Stillstand zurücklegt, und berechnet sich im einfachen Fall der Geradeausfahrt zu

$$D_A = v_x \cdot (T_1 + T_2) + D_B \cdot L_1 \cdot L_2, \tag{1}$$

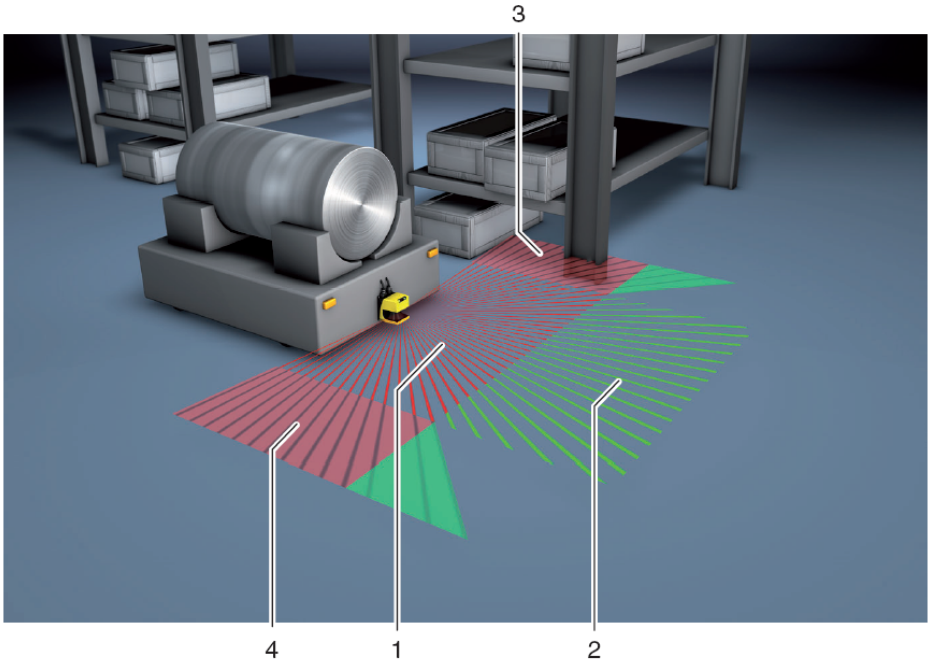


Abbildung 1: Veranschaulichung der geschwindigkeitsabhängigen Schutzfeldberechnung zur mobilen Gefahrbereichssicherung am Beispiel eines fahrerlosen Transportsystems. (1), (2) Schutz- und Warnfeld für Geradeausfahrt, (3), (4) Schutz- und Warnfelder für Kurvenfahrten.

wobei v_x die momentante Geschwindigkeit des Fahrzeugs ist, T_1 die Ansprechzeit des Sicherheits-Sensors, T_2 die Ansprechzeit des Bremssystems, D_B der Bremsweg bei der Geschwindigkeit v_x, L_1 ein Faktor für Bremsenverschleiß und L_2 ein Faktor für ungünstige Bodenbeschaffenheit, z.B. Schmutz oder Nässe.

3.1.2 Schutzfelder und Sicherheitsdistanz

Die vom Modul SAMS-Schutzfeldberechnung berechneten *Schutzfelder* beinhalten den Anhalteweg zuzüglich eines gerätespezifischen Zuschlags Z_{GES} für den Laserscanner.

Die Formel (1) beschreibt das Bremsen bei Geradeausfahrt. In der *Schutzfeldberechnung* wird auch der Fall der Bremsung während der Kurvenfahrt behandelt, welches ein kurvenförmiges Schutzfeld ergibt. Das Programm rechnet für jede *Translationsgeschwindigkeit v* und jede sich aus dem *Lenkwinkel* ergebende *Winkelgeschwindigkeit* ω das passende Schutzfeld aus, welches die bei einer sofort eingeleiteten Bremsung bis zum Stillstand überstrichene Fläche beschreibt. Der Berechnung der Schutzfelder liegt eine formale Beschreibung des Bremsverhaltens zugrunde, das sogenannte *Bremsmodell* des Fahrzeugs. Das Bremsverhalten in der Kurve wird dabei aus dem Bremsverhalten in der Gerade unter bestimmten, plausiblen Annahmen (Abschnitt 3.2) extrapoliert.

Die Parameter aus (1) werden wie folgt bestimmt: D_B wird aus dem Bremsmodell berechnet (Ab-

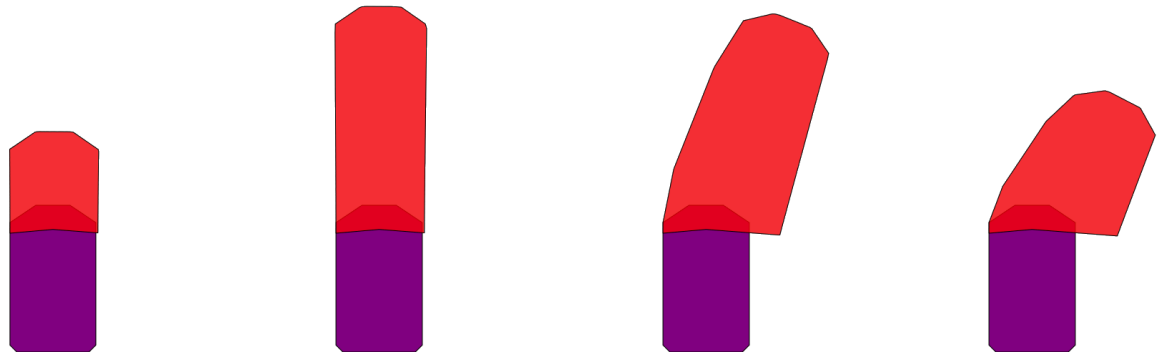


Abbildung 2: Beispiel für geschwindigkeitsabängige Schutzfelder bei Geradeausfahrt, zunehmender Geschwindigkeit und Kurvenfahrt (von links nach rechts).

schnitt 3.2.4), v_x muss gemessen werden (Abschnitt 3.2.3), und die anderen Parameter werden vom Systemintegrator parametriert.

Zu beachten ist folgender Unterschied zum gängigen Verfahren bei Sicherheitslaserscannern: während früher der Anhalteweg vom Benutzer berechnet und zuzüglich des Sicherheitszuschlags als Schutzfeld konfiguriert wurde, berechnet das SAMS-Modul, bei korrekter Parametrierung, Schutzfelder, welche sämtliche Sicherheitszuschläge bereits beinhalten.

3.2 Voraussetzung an das Fahrzeug

3.2.1 Fahrzeugtyp

Die Schutzfeldberechnung kann mit allen Fahrzeugen verwendet werden, die sich in einer Ebene fortbewegen, und die mindestens zwei ungelenkte Räder haben, so dass sich das Fahrzeug nicht seitwärts bewegen kann.

Beispiele für geeignete Fahrzeuge sind: *Fahrzeuge mit Differentialantrieb* (Abb. 3a-d), wenn diese die Lenkwinkel entsprechend steuern, und *Fahrzeuge mit einer gelenkten und einer ungelenkten Achse* (Abb. 3f-h), wenn der Lenkwinkel während des Bremsvorganges fix ist.

3.2.2 Bremsverhalten

Damit die Schutzfeldberechnung verwendet werden kann, muss das Bremsverhalten des Fahrzeugs die folgenden *Anforderungen* erfüllen:

(1) Das Bremsverhalten des Fahrzeugs muss reproduzierbar sein. Gibt es begrenzte Schwankungen, z.B. auf Grund der Umgebungsbedingungen, müssen diese durch einen geeignet gewählten Bremsaufschlag abgedeckt werden.

(2) Das Fahrzeug muss beim Bremsen auf der Kurve bleiben, die es zu Beginn des Bremsvorgangs befährt. Fährt es geradeaus, muss es geradeaus bremsen; fährt es

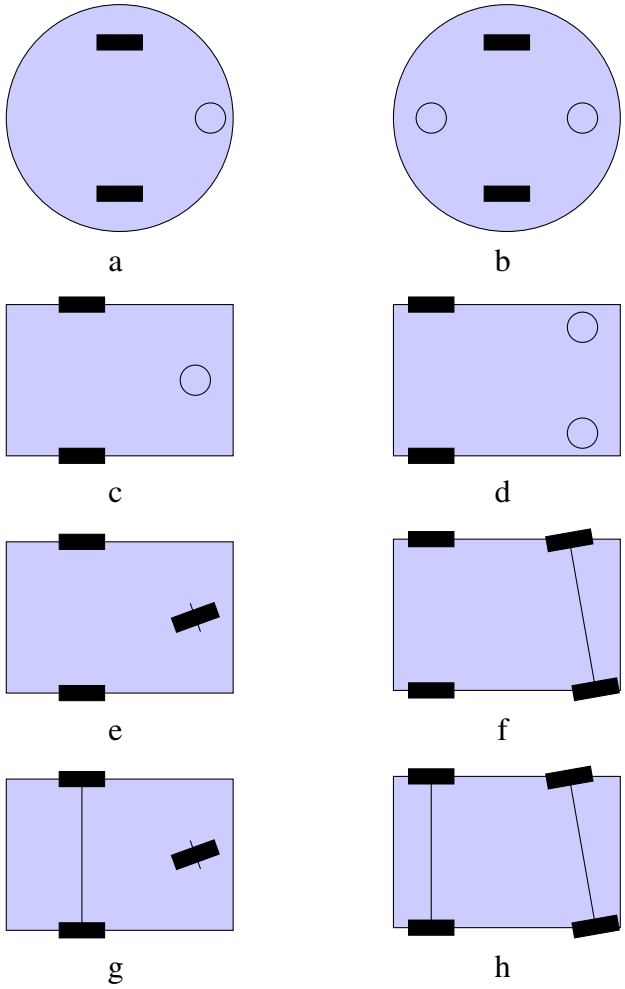


Abbildung 3: Fahrzeuge mit (a)-(b): Differentialantrieb und ein oder zwei zusätzlichen Kontaktpunkten, (c)-(d): zwei unabhängig angetriebenen Rädern und ein oder zwei omnidirektionalen Rädern, (e)-(f): zwei Antriebsrädern hinten und einer gelenkten Vorderachse, (g)-(h): zwei freien Hinterrädern und einer gelenkten und angetriebenen Vorderachse.

auf einem Kreisbogen, muss es auf demselben Kreisbogen bremsen.

(3) Die Bremsverzögerung darf höchstens proportional zur Geschwindigkeit sein. Dies ist erfüllt für Bremsen, die auf Coulombreibung basieren, entsprechend dem Bauprinzip der Bremsen die üblicherweise in Kraftfahrzeugen verwendet werden. Es ist weiter erfüllt für Bremsen durch einen (über einen Widerstand) kurzgeschlossen Elektromotor. Nicht zulässig sind Bremsen, die auf viskoser Reibung basieren, z.B. Bremsen durch Luftreibung, Bremsfallschirme, oder Bremsen in Flüssigkeiten. (Derartige Bremsen sind allerdings im Bereich der Fabrikautomatisierung ohnehin unüblich.)

(4) Erfolgt das Bremsen mit Hilfe einer sicheren Antriebssteuerung, müssen die Bremsrampen so konfiguriert sein, dass die Verzögerung nur von der momentanen Geschwindigkeit abhängt, und höchstens proportional zur Geschwindigkeit ist. Dies ist automatisch erfüllt für Bremsen mit konstanter Verzögerung.

(5) Beherrscht das Fahrzeug Drehbewegungen auf der Stelle, so muss die Bremse auch reine Drehbewegungen in vergleichbarer Zeit wie entsprechende Vorwärtsbewegungen zum Stillstand bringen (höchstens ein Drittel länger). Dies ist automatisch erfüllt, wenn zwei Räder gebremst werden, deren Abstand mindestens die Hälfte der Breite und Länge des Fahrzeugs beträgt. Es ist außerdem automatisch erfüllt, wenn ein Rad gebremst wird, welches mindestens die Hälfte der Fahrzeuglänge von der Verbindungslinie zweier ungelenkter Räder entfernt ist.

3.2.3 Sensorik

Translationsgeschwindigkeit v und Drehgeschwindigkeit ω, die zusammengenommen den *Geschwindigkeitsvektor* bilden, müssen sicher gemessen werden. Die Translationsgeschwindigkeit bezieht sich auf den Referenzpunkt in der Mitte der beiden ungelenkten Räder und ist für positive Geschwindigkeiten nach vorn, also senkrecht zur Verbindungslinie der Räder gerichtet. Die Messung kann beispielsweise durch Inkrementalencoder an den ungelenkten Rädern oder durch einen Encoder und einen Lenkwinkelsensor an einem gelenkten Rad bestimmt werden. Für eine sichere Messung können zwei unabhängige Messverfahren benutzt werden.

3.2.4 Bestimmung des Bremsmodells

Das Bremsmodell wird durch ein oder mehrere Messungen von Bremswegen D_B des Fahrzeuges bei Geradeausfahrten parametriert. Die Bremswege des Fahrzeuges bei Kurvenfahrt werden durch die Schutzfeldberechnungssoftware automatisch berechnet und müssen damit nicht als Parameter übergeben werden.

Jede Messung ist durch ein Paar (v_i, s_i) repräsentiert, wobei s_i den Bremsweg des Fahrzeuges aus der Geschwindigkeit v_i bis zum Stillstand darstellt. Der an die Schutzfeldberechnugssoftware zu

übergebende Bremsweg s_i besteht dabei bereits aus dem gemessenen Bremsweg D_{Bi} multipliziert mit den Sicherheitsfaktoren L_1 (Bremsverschleiß) und L_2 (ungünstige Bodenbeschaffenheit).

Es ist sinnvoll, eine der Bremswegmessungen bei Maximalgeschwindigkeit v_{max} des Fahrzeuges durchzuführen, da die Schutzfeldberechnungssoftware für Geschwindigkeiten überhalb des Messwertes mit der größten Geschwindigkeit eine sehr grobe Überabschätzung der Bremswege verwendet.

Grundsätzlich ist eine vollständige Parametrierung des Bremsmodells bereits mit einer Messung möglich. Weitere Messungen verbessern allerdings die Genauigkeit des Bremsmodells und führen damit zu kleineren Schutzfeldern. Bei Parametrierung mit mehreren Messwerten ist eine Gleichverteilung der Messgeschwindigkeiten zwischen 0 und v_{max} sinnvoll. Beispielsweise sollten bei Parametrierung mit zwei Messwerten die Messungen für die maximale Geschwindigkeit v_{max} und die Hälfte der maximalen Geschwindkeit $\frac{v_{max}}{2}$ durchgeführt werden.

3.2.5 Konfigurationsparameter

Die Schutzfeldberechnung muss mit folgenden Parametern konfiguriert werden (Abschnitt 4.4):

(1) Für das Fahrzeug (Abschnitt 4.4.1): die *Fahrzeugkontur* als konvexes Polygon in der Ebene, welches das ganze Fahrzeug inklusive aller überstehenden Aufbauten enthält, sowie die Position der ungelenkten Räder am Fahrzeug;

(2) Für den Laserscanner (Abschnitt 4.4.2): Position und Orientierung am Fahrzeug, sowie technische Daten wie Anzahl Strahlen, Winkelbereich und maximale Reichweite;

(3) Die Daten aus der Messung des Bremsmodells (Abschnitt 4.4.3);

(4) Die *Latenzzeit* und Sicherheitsabstände (Abschnitt 4.4.4).

Die Parametrierung erfolgt programmatisch über die Funktion `nuterzkonf_vorverarbeiten` (Abschnitt 4.3.3). Zusätzliche Anforderungen an die vom Nutzer zu übergebenden Parameter sind in Abschnitt 4.4.6 dokumentiert.

4 Integrationshandbuch

4.1 Beschreibung der Funktionalität

4.1.1 Überblick

Eine korrekte Konfiguration vorausgesetzt, wird für die berechneten *Schutzfelder* garantiert, dass das Fahrzeug bei einer Bremsung mit Geschwindigkeit v und Winkelgeschwindigkeit ω vollständig innerhalb des für (v, ω) berechneten Schutzfeldes zum Stillstand kommt. Wenn gewährleistet wird, dass das zur momentanen Geschwindigkeit korrespondierende Schutzfeld stets frei von Hindernissen ist, garantiert dies, dass das Fahrzeug nicht mit stationären Hindernissen kollidiert.

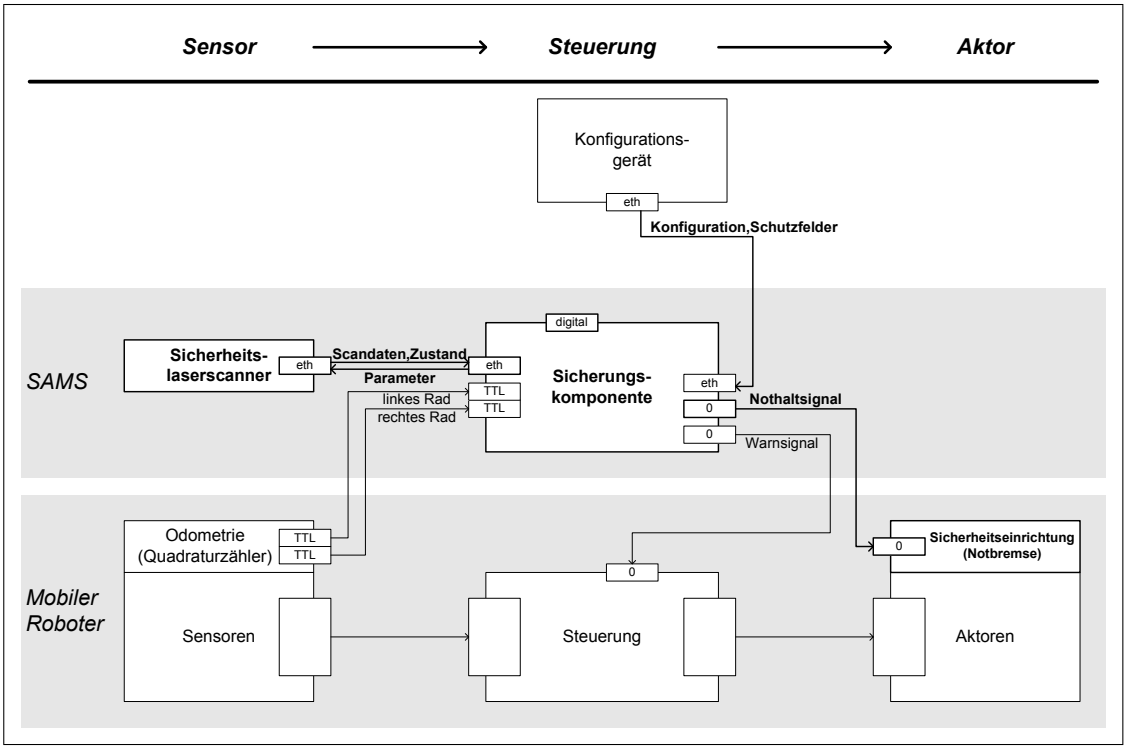


Abbildung 4: Einsatzszenario für eine mit der Schutzfeldberechnung implementierte Sicherungskomponente

4.1.2 Einsatzszenarien

Der Einsatz der Schutzfeldberechnung erfolgt im wesentlichen in zwei Phasen: während der *Konfigurationsphase* werden die *Konfigurationsdaten* vorverarbeitet, und zur Laufzeit wird aus den vorverarbeiteten Konfigurationsdaten das Schutzfeld für einen Bereich von Geschwindigkeitsvektoren berechnet (vgl. Abb. 5).

Die Schutzfeldberechnung kann zur Implementation einer *Sicherungskomponente* verwendet werden. Dieses ist eine in Hardware (als eingebettetes Gerät) oder Software implementierte Komponente, welche als Eingaben die *Odometrie* (Geschwindigkeitsmessung) und die Messdaten des Sicherheitslaserscanners liest, und als Ausgabe ein Nothaltsignal produziert. Abb. 4 zeigt ein typisches Einsatzszenario für eine solche Sicherungskomponente.

Eine Sicherungskomponente ist typischerweise getaktet, d.h. durchläuft in einem festen Systemtakt einen Zyklus, in dem Messdaten gelesen, verarbeitet, und wenn nötig ein Nothaltsignal generiert wird. Hierbei können wir zwischen zwei Szenarien unterscheiden: In der *online-Berechnung* wird das Schutzfeld aktuell für jede Messung berechnet, während in der *offline-Berechnung* die Schutzfelder für gegebene Intervalle von Geschwindigkeitsvektoren vorab berechnet und in einer Tabelle abgelegt werden, auf die zur Laufzeit nur noch zugegriffen wird. Das zweite Verfahren ist für Systeme mit geringer Rechenleistung zu bevorzugen, benötigt allerdings mehr Speicher. Für die eigentliche Schutzfeldberechnung gibt es keinen Unterschied zwischen online und offline.

4.1.3 Spezifikation

Die Schutzfeldberechnung wird initialisiert durch den Aufruf der Initialisierungsfunktion `init_memory` (Abschnitt 4.3.2), und konfiguriert durch den Aufruf der Funktion `nutzerkonf_vorverarbeiten` (Abschnitt 4.3.3). Für diese Funktionen ist Termination zugesichert. Die Argumente der zweiten Funktion beschreiben Bremsmodell und Fahrzeug.

Die Hauptfunktion für die Schutzfeldberechnung ist die Funktion `schutzfeld_berechnen` (Abschnitt 4.3.4). Die Schutzfeldberechnung erhält als Argument einen Geschwindigkeitsbereich, und liefert als Rückgabe eine Statusmeldung (Abschnitt 4.3.1), sowie ein Schutzfeld repräsentiert als Feld von Entfernungsmessdaten.

Unter der Annahme, dass Initialisierung und Konfiguration aufgerufen worden sind und erfolgreich waren, wird folgendes für die Schutzfeldberechnung zugesichert:

(1) Wenn die Schutzfeldberechnung erfolgreich ist, dann schließt das Schutzfeld die beim Bremsen mit jedem Geschwindigkeitsvektor des angebenen Bereichs bis zum Stillstand vom Fahrzeug überstrichene Fläche vollständig ein.

(2) Die Berechnung terminiert immer, und die Berechnungszeit ist deterministisch (für gleiche Eingabedaten wird die gleiche Zeit benötigt).

(3) Die Berechnung hat konstanten Speicherplatzverbrauch.

(4) Die Berechnung erzeugt keine Ausnahmen.

(5) Es findet eine elementare Ablaufkontrolle statt.

Abb. 5 zeigt beispielhaft einen schematischen Programmablaufplan mit einer online-Berechung (die Berechung erfolgt in der Hauptkontrollschleife).

4.2 Systemanforderungen

Die hier aufgeführten Anforderungen müssen erfüllt sein, damit eine korrekte Verwendung der SAMS-Schutzfeldberechnung gewährleistet werden kann.

4.2.1 Hardware-Anforderungen

Schutzfeldüberwachung Die Überwachung der berechneten Schutzfelder kann durch einen Sicherheits-Laserscanner erfolgen, der die einschlägigen Sicherheitsanforderungen erfüllt, etwa das Modell RS-4 der Fa. Leuze electronics (Leuze electronics).

Speicheranforderungen In der Datei `sams_konfig.h` können drei Makro-Definitionen angepasst werden, die Einfluss auf die Speicheranforderungen des Softwaremoduls haben (s. Abschnitt 4.5). Die Speicherkomplexität des Moduls ist $O(K \cdot L + N)$, wobei in dieser Formel die

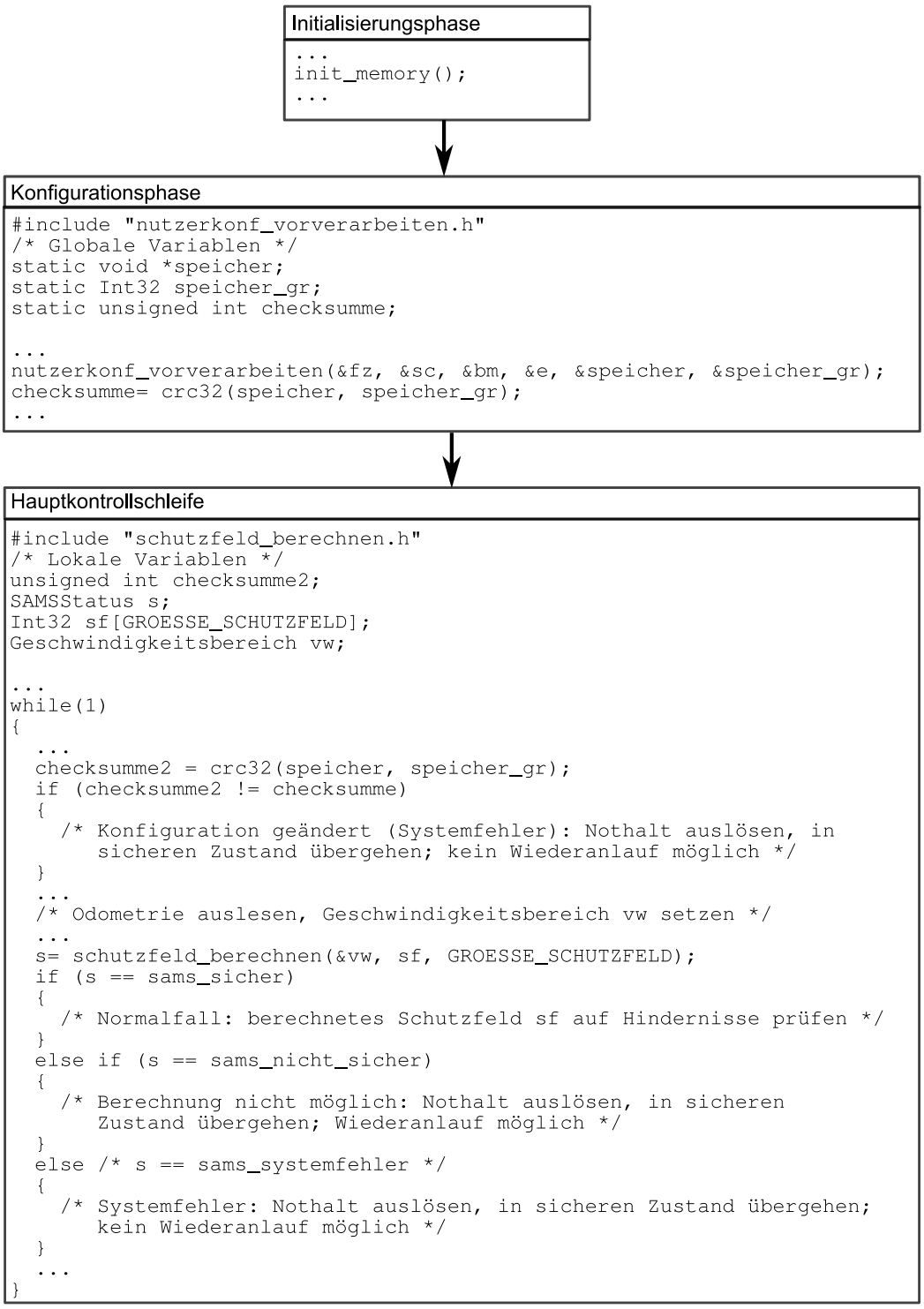


Abbildung 5: Skizze für einen Programmablauf bei Verwendung der Schutzfeldberechung. Die *Initialiserungsphase* findet zum Systemstart (Boot) statt, die *Konfigurationsphase* zu Inbetriebnahme oder Programmstart, und die *Hauptkontrollschleife* wird wiederholt durchlaufen. Die Funktion `crc32` steht exemplarisch für eine Prüfsummenberechnung; sie wird nicht durch die Schutzfeldberechnung zur Verfügung gestellt.

in der nachfolgenden Tabelle definierten Abkürzungen verwendet werden. Diese Tabelle gibt die resultierende Größe des Datensegments für verschiedene typische Parameterwerte an (Kenndaten: `gcc` Version 4.2.1, AMD Opteron Dual Core, 64 Bit).

N	`MAXIMALE_ANZAHL_STRAHLEN`	529	529	529	529
K	`MAXIMALE_ANZAHL_KONTURPUNKTE`	50	100	200	800
L	`APPROXIMATIONSPUNKTE_JE_BOGEN`	6	8	8	8
	Größe Datensegment (*data*+*common*+*bss*) (in *kB*)	**39**	**97**	**194**	**775**

Diese Werte sind keine Garantien bezüglich des Speicherbedarfs des Softwaremoduls auf der eingesetzten Hardware, sondern nur informative Rahmenwerte. Bei der SW-Integration muss individuell der konkrete Speicherbedarf ermittelt und sichergestellt werden, dass dieser gedeckt werden kann. Die Werte wurden mithilfe des Programms `size` der GNU binutils ermittelt.

Fließkommaarithmetik Die SAMS-Schutzfeldberechnung verwendet Fließkommaarithmetik zur Berechnung der Schutzfelder. Daher muss die Hardwarearchitektur, auf der die Software eingesetzt wird, eine Fließkommaeinheit besitzen. Diese muss dem internationalen Standard IEEE 754 genügen und das dort definierte *Single Format* vollständig unterstützen (d. h. Fließkommazahlen der Größe 32 Bit).

Bitbreite ganzzahliger arithmetischer Typen Ganzzahlige arithmetische Werte werden grundsätzlich als Worte der Größe 32 Bit gespeichert. Die Hardwarearchitektur muss daher diese Wortgröße unterstützen. Der entsprechende Datentyp wird mit `Int32` bezeichnet. Dessen Definition (`typedef`) ist in der Datei `sams_typen.h` entsprechend anzupassen, sofern die voreingestellte Definition als `int` für die Kombination aus Compiler und Hardwarearchitektur nicht zutreffend ist.

4.2.2 Software-Anforderungen

Anforderungen an den Compiler Die Entwicklung der Software wurde mit dem GNU C Compiler (Version 4.3.2) auf gängigen Industrie-PC-Hardwarearchitekturen (z. B. Prozessor Intel CPU T2500) durchgeführt. Die Software wurde gemäß ISO-Standard 9899:1990 (sogenanntes C90) unter Verwendung einer Sprachteilmenge entwickelt. Daher lässt sie sich mit jedem zu diesem Standard kompatiblen Compiler übersetzen. Für den Einsatz der Software in der Sicherungskomponente muss zur Übersetzung des Codes ein Compiler verwendet werden, der den Anforderungen aus Abschnitt 7.4.4 der DIN/EN 61508-3 genügt. Insbesondere muss dieser nach einer anerkannten nationalen oder internationalen Norm zertifiziert sein oder dessen Eignung für den Einsatz beurteilt worden sein.

Ein-/Ausgaberoutinen Die SAMS-Schutzfeldberechnung ist ein rein funktionales Modul. Mechanismen wie Ein-/Ausgabe in Dateien, Standardeingabe/-ausgabe und Lesen/Schreiben auf Kommunikationskanälen (Netzwerk, I/O-Controller, etc.) werden nicht verwendet. Stattdessen

erfolgt aller Datenfluss über die Schnittstelle der bereitgestellten öffentlichen Funktionen in Form von Rückgabewerten und Ausgabeparametern. Diagnosemeldungen werden im Speicher abgelegt und können durch die entsprechenden Funktionen gelesen werden. Es bestehen somit keine Anforderungen an Ein-/Ausgaberoutinen.

Multithreading Innerhalb der SAMS-Schutzfeldberechnung wird kein Multithreading verwendet. Die Schutzfeldberechnung ist nicht thread-sicher; bei einem Einsatz in einer Multithreading-Umgebung sollte die Komponente nur von genau einem Thread verwendet werden, um *Race Conditions* auszuschließen und eine sichere und einfachstmögliche Synchronisation bezüglich dieses Teils der Gesamtsoftware zu gewährleisten.

Laufzeitverhalten Die *Laufzeitkomplexität* der Software lässt sich in Abhängigkeit der Anzahl der Punkte der konfigurierten Fahrzeugkontur beschreiben. Bezeichnet man diese mit N, so ist die Laufzeitkomplexität der Software $O(N \log N)$. Der $\log N$-Faktor entsteht durch die Verwendung eines Sortieralgorithmus. Eine Laufzeitangabe für eine feste Konfiguration wird nicht gegeben, da diese sehr stark von der verwendeten Hardwarearchitektur abhängig ist.

4.3 Schnittstellendefinition (pro Funktion)

4.3.1 Rückgabetyp SAMSStatus

```
typedef enum SAMSStatusEnum
{
  sams_nicht_sicher      = 0,
  sams_sicher            = 1,
  sams_systemfehler      = 2
} SAMSStatus;
```

Beschreibung Jede Funktion liefert ein Ergebnis dieses Types zurück. Die Funktion war *erfolgreich*, wenn das Ergebnis sams_sicher zurückgegeben wird, anderenfalls ist ein Fehler aufgetreten.

Nach jedem Aufruf einer der Funktionen aus Abschnitt 4.3 muss der Rückgabewert geprüft werden.

Wenn der Rückgabewert sams_sicher ist, dann war die Ausführung erfolgreich, und die als Rückgabeparameter zurückgelieferten Ergebnisse der Funktion sind vertrauenswürdig.

Wenn der Rückgabewert sams_nicht_sicher ist, dann liegt durch Konfiguration oder Zustand des Systems bedingte Situation vor, in der die Sicherheit der Ergebnisdaten nicht gewährleistet werden kann. Die Ergebnisse einer mit sams_nicht_sicher beendeten Funktion sind nicht zuverlässig und dürfen nicht weiterverarbeitet werden.

Wenn das Ergebnis sams_systemfehler ist, dann ist ein schwerwiegender Systemfehler aufgetreten, der die Integrität der Schutzfeldberechnungssoftware gefährdet. Das Ergebnis deutet auf einen fehlerhafte Systemintegration hin, und darf nur während der Integrationsphase auftreten. Tritt der Fehler während des laufenden Betriebs auf, darf die Komponente erst nach Analyse und Beseitigung des Fehlers durch dazu befugte Personen wieder in Betrieb genommen werden.

Die Ursache für die Ergebnisse sams_nicht_sicher oder sams_systemfehler können aus dem Ereignisprotokoll ausgelesen werden (Abschnitt 4.3.5).

4.3.2 Initialisierung

```
#include "speicher.h"

void init_memory( void  );
```

Beschreibung Die Funktion init_memory dient der Initialisierung. Es ist ein einmaliger Aufruf nach Laden des Softwaremoduls erforderlich, danach keine weitere Verwendung.

4.3.3 Konfigurationsfunktion

```
#include "nutzerkonf_vorverarbeiten.h"

SAMSStatus nutzerkonf_vorverarbeiten(
              const FahrzeugdatenNutzer *FZ,
              const ScannerdatenNutzer  *SC,
              const BremsdatenNutzer    *BM,
              const Einstellungen       *E,
              void **speicher_zeiger,
              Int32 *speicher_groesse );
```

Beschreibung nutzerkonf_vorverarbeiten führt die Konfiguration der über die Einsatzdauer festen Parameter der Schutzfeldberechnung durch. Diese Funktion muss vor Aufruf der Funktion schutzfeld_berechnen erfolgreich ausgeführt worden sein.

Die Funktion erhält Fahrzeugparameter FZ, Scannerparameter SC, Bremsdaten des Fahrzeuges BM und spezielle Einstellungen E als Eingabeparameter. Die Elemente dieser Strukturen werden in Abschnitt 4.4 beschrieben. speicher_zeiger und speicher_groesse sind Rückgabeparameter.

Rückgabe Nach erfolgreicher Ausführung der Funktion nutzerkonf_vorverarbeiten sind im internen Konfigurationsspeicher alle festen Parameter der Schutzfeldkonfiguration korrekt abgelegt. Position speicher_zeiger und Größe (in Bytes) speicher_groesse des internen Konfigurationsspeichers werden von nutzerkonf_vorverarbeiten zurückgeliefert. Der Inhalt dieses

Speicherbereiches ändert sich nur durch erneutes Ausführen der Funktion nutzerkonf_vorverarbeiten. Dies sollte durch Überwachung des Speicherbereiches im laufenden Betrieb (z.B. vor jedem Aufruf der Funktion schutzfeld_berechnen) regelmäßig überprüft werden.

Wenn die Funktion nutzerkonf_vorverarbeiten einen Fehler zurückgibt, dann liegt eine ungültige interne Konfiguration vor, und es dürfen solange keine Schutzfelder berechnet werden, bis die Funktion nutzerkonf_vorverarbeiten erneut erfolgreich ausgeführt worden ist.

4.3.4 Schutzfeldberechnungsfunktion

```
#include "schutzfeld_berechnen.h"

SAMSStatus schutzfeld_berechnen(
              const Geschwindigkeitsbereich * VW,
              Int32 * schutzfeld_daten,
              Int32 schutzfeld_laenge_max,
              Float32 t_l );
```

Beschreibung Die Funktion schutzfeld_berechnen berechnet ein Schutzfeld für den als VW übergebenen Geschwindigkeitsbereich. Dieses Schutzfeld kann elementweise mit einem Laserscan verglichen werden um festzustellen, ob der Fahrweg mit einer Geschwindigkeiten aus dem Geschwindigkeitsbereich VW ohne Kollisionen befahren werden kann.

1. Struktur von VW: siehe Abschnitt 4.4.

2. schutzfeld_daten enthält das berechnete Schutzfeld für VW bezüglich letztmalig durchgeführter Konfiguration (durch nutzerkonf_vorverarbeiten) in Form von *erforderlichen Entfernungsmesswerten* jedes Laserstrahls, unterhalb derer eine Schutzfeldverletzung vorliegt.

3. Der Kürze halber seien für folgende Punkte $M \stackrel{def}{=}$ nutzerkonf_scanner->strahlenanzahl und $\beta \stackrel{def}{=}$ nutzerkonf_scanner->winkelbereich, jeweils bezogen auf die aktuell gültige Konfiguration.

4. $M \leq$ schutzfeld_laenge_max

5. Die Distanz schutzfeld_daten[i] (für $0 \leq$ i $< A$) bezieht sich auf den Winkelsektor der Breite $\frac{\beta}{M}$, der sich als $[(\frac{i}{M}-\frac{1}{2})\beta;(\frac{i+1}{M}-\frac{1}{2})\beta]$ darstellen lässt. Hierbei entspricht der Winkel 0 gerade der Ausrichtung des Laserscanners, so dass folglich der Winkelsektor $[-\frac{\beta}{2};\frac{\beta}{2}]$ der vom Laserscanner erfasste ist.

6. t_l beschreibt einen optionalen Aufschlag auf die Latenzzeit (die Verzögerung bis zum Einsetzen des Bremsvorganges), zusätzlich zu den vorgegebenen Zeiten T_1, T_2 (Abschnitt 3.1.1).

Der Wert muss ≥ 0 (nicht-negativ) sein. Durch diesen Parameter können größere Schutzfelder berechnet werden als für die Sicherung des Fahrzeuges nötig. Diese können als *Warnfelder* verwendet werden. Siehe dazu auch Abschnitt 4.7.

7. Siehe auch die formale Spezifikation der Funktion im Quellcode (Dateien `hol_dekl.h`, `schutzfeld_berechnen.h`)

Vorbedingung Die Funktion schutzfeld_berechnen darf nur ausgeführt werden, wenn eine für den aktuellen Fahrzeugzustand gültige interne Konfiguration vorliegt, die durch die Funktion nutzerkonf_vorverarbeiten erstellt wurde.

Rückgabe Nach erfolgreicher Ausführung der Funktion schutzfeld_berechnen befindet sich in schutzfeld_daten $[0..N-1]$ ein Schutzfeld, das für alle Geschwindigkeitsvektoren (v, ω) mit

$$\begin{aligned} \text{VW}->\text{v\_min} \leq v \leq \text{VW}->\text{v\_max} \\ \text{VW}->\text{omega\_min} \leq \omega \leq \text{VW}->\text{omega\_max} \end{aligned}$$

ein gültiges Schutzfeld ist. Die Anzahl N der relevanten Einträge in schutzfeld_daten ist gleich der Anzahl der Strahlen des Laserscanners.

Wenn die Funktion schutzfeld_berechnen einen Fehler zurückgibt, so ist das zurückgelieferte Schutzfeld ungültig und kann nicht zur Fahrwegsicherung verwendet werden.

4.3.5 Diagnosefunktion

```
#include "ereignisprotokoll.h"

Bool        neue_ereignisse( void  );
EPEreignis  lies_ereignis(
               Int32* zeit,
               Int32 *daten,
               Int32 *laenge,
               Int32 laenge_max );

const Char* ausgabetext_ereignis( EPEreignis typ );
```

Beschreibung Die Funktion neue_ereignisse liefert, ob weitere unbearbeitete Ereignisse vorliegen. Die Funktion lies_ereignis darf nur aufgerufen werden, wenn die Funktion neue_ereignisse wahr geliefert hat.

Die Funktion lies_ereignis liest das nächste unbearbeitete Ereignis aus dem Diagnosepuffer. Ereignisse beinhalten nicht ausschließlich Fehler, so dass im Fehlerfall der gesamte Puffer über wiederholte Aufrufe dieser Funktion auszulesen ist, um die Ursache eines Fehlers zu klären. In

daten wird der Funktion der Zeiger auf ein existierendes Int32 Array übergeben. Dorthinein schreibt die Funktion zusätzliche Daten falls das ausgelesene Ereignis solche beinhaltet. Die Größe des Arrays wird in laenge_max übergeben. zeit und laenge sind Rückgabeparameter.
Die Funktion ausgabetext_ereignis übersetzt den übergebenen Ereignistyp typ in den dazugehörigen Beschreibungstext.

Rückgabe Die Funktion lies_ereignis liefert den Typ des gelesenen Ereignisses zurück. Nach Ausführung befindet sich in zeit der Zeitpunkt an dem das Ereignis protokolliert wurde, in daten zusätzliche Daten des Ereignisses und in laenge die Anzahl der in daten beschriebenen Einträge.

4.4 Nutzerdatenstrukturen (Datenspezifikation)

Das Nutzerkoordinatensystem kann frei gewählt werden. Alle Koordinaten müssen aber in der *Einheit mm* angegeben werden.
Zeiten werden in der *Einheit Sekunden* angegeben.

4.4.1 Fahrzeugparameter

```
typedef struct FahrzeugdatenNutzerStruct
{
  Vektor2D kontur_daten[ROBOTERKONTUR__ARRSZ];
  Int32    kontur_laenge;

  Vektor2D starre_achse_links;
  Vektor2D starre_achse_rechts;

  Float32 faktor_ueberschreitung_v_max;
  Geschwindigkeit v_max_schleichfahrt;
  Geschwindigkeit v_min;
  Laenge          r_min;
  WinkelgeschwindigkeitRad omega_max;
} FahrzeugdatenNutzer;
```

`kontur_daten`	Alle Eckpunkte der Außenkontur des Fahrzeuges in Nutzerkoordinaten
`kontur_laenge`	Anzahl der Konturpunkte in `kontur_daten`
`starre_achse_links`	Position des linken Rades der starren Achse in Nutzerkoordinaten
`starre_achse_rechts`	Position des rechten Rades der starren Achse in Nutzerkoordinaten
`faktor_ueberschreitung_v_max`	Dieser Faktor definiert, wie weit die maximale Geschwindigkeit, für die das Bremsmodell konfiguriert wurde, überschritten werden darf, bevor eine Überschreitung der Maximalgeschwindigkeit vorliegt. In dem sich dadurch ergebenden (erlaubten) Geschwindigkeitsbereich oberhalb der größten Geschwindigkeit für die ein Bremsweg gemessen wurde, wird der Bremsweg kubisch überapproximiert (siehe auch Abschnitt 4.4.6)
`v_max_schleichfahrt`	— *unbenutzt* —
`v_min`	minimale Vorwärtsgeschwindigkeit des Fahrzeuges
`r_min`	minimaler Kurvenradius des Fahrzeuges
`omega_max`	maximale Winkelgeschwindigkeit in Bogenmaß pro Sekunde (rad/s). Sie muss im Fall `r_min == 0` (Drehung auf der Stelle möglich) angegeben werden, anderenfalls wird sie ignoriert.

4.4.2 Scannerparameter

```
typedef struct ScannerdatenNutzerStruct
{
  Vektor2D           position;
  WinkelGrad         orientierung_alpha;

  Int32      strahlenanzahl;
  WinkelGrad winkelbereich;
  Int32      maximaler_scan;
} ScannerdatenNutzer;
```

`position`	Position des Scanzentrums des Scanners in Nutzerkoordinaten.
`orientierung_alpha`	Orientierung α des Scanners
`strahlenanzahl`	Anzahl der Laserstrahlen pro Scan
`winkelbereich`	Winkelbereich β des Scanners
`maximaler_scan`	maximale Reichweite des Scanners in *mm*

4.4.3 Bremsdaten des Fahrzeuges

```
typedef struct BremsmessungStruct
{
  Geschwindigkeit v;
  Laenge          s;
} Bremsmessung;
```

v	Geschwindigkeit einer Geradeausfahrt
s	Bremsweg bei Geschwindigkeit v

```
typedef struct BremsdatenNutzerStruct
{
  Int32 anzahl;
  Bremsmessung messungen[BREMSDATEN__ARRSZ];
} BremsdatenNutzer;
```

anzahl	Anzahl der Bremsmessungen
messungen	Bremsmessungen absteigend nach Geschwindigkeiten sortiert (beginnend mit der Höchsten).

4.4.4 Spezielle Einstellungen

```
typedef struct EinstellungenStruct
{
  Int32 ueberwachung_start;
  Int32 ueberwachung_ende;

  Int32 protokoll_level;

  Float32 t_r;
  Int32 e_scanner;
  Float32 e_radius;

  Vektor2D vektor_sicherheitsdistanz;
} Einstellungen;
```

`ueberwachung_start`	— *unbenutzt* —
`ueberwachung_ende`	— *unbenutzt* —
`protokoll_level`	Muss immer auf 1 gesetzt werden.
`t_r`	*Latenzzeit*: Summe aller Zeitverzögerungen bis Bremsstart in Sekunden ($T_0 + T_1 + T_2$) T_0 maximale Zeitdifferenz zwischen zwei Aufrufen der Schutzfeldberechnung T_1 Ansprechzeit des Sicherheitssensors T_2 Ansprechzeit des Bremssystems
`e_scanner`	Messfehler des Laserscanners in mm
`e_radius`	zusätzlicher Aufschlag auf den Pufferradius in mm
`vektor_sicherheitsdistanz`	gerichtete Sicherheitsdistanz. Die Schutzfelder werden so konstruiert, dass das Fahrzeug in der Richtung dieses Vektors (relativ zum Fahrzeug) immer in einer Distanz vor dem Hindernis anhält, die der Länge dieses Vektors entspricht. Der Vektor wird in Nutzerkoordinaten übergeben.

4.4.5 Geschwindigkeitsbereich

```
typedef struct GeschwindigkeitsbereichStruct
{
  Geschwindigkeit v_min;
  Geschwindigkeit v_max;
  WinkelgeschwindigkeitRad omega_min;
  WinkelgeschwindigkeitRad omega_max;
} Geschwindigkeitsbereich;
```

`v_min`	untere Grenze der Vorwärtsgeschwindigkeit
`v_max`	obere Grenze der Vorwärtsgeschwindigkeit
`omega_min`	untere Grenze der Winkelgeschwindigkeit in Bogenmaß pro Sekunde
`omega_max`	obere Grenze der Winkelgeschwindigkeit in Bogenmaß pro Sekunde

4.4.6 Anforderungen an die Nutzerdaten

Im Folgenden verwendete Konstanten:

`mindist_raeder`	$1mm$
`mindist_bremsmessung`	$1mm$
`maxdiff_alpha__faktor2pi`	2.0

Die folgenden Anforderungen müssen durch die vom Nutzer übergebenen Parameter erfüllt sein. Sind diese Anforderungen nicht erfüllt, wird die Konfiguration mittels nutzerkonf_vorverarbeiten nicht erfolgreich durchgeführt. Die Funktion signalisiert dies durch Rückgabe des Fehlers sams_nicht_sicher.

`starre_achse_links` `starre_achse_rechts`	Die Distanz zwischen Position des linken und rechten Rades muss mindestens `mindist_raeder` betragen.
`BremsdatenNutzer`	Die Geschwindigkeiten der Bremsmessungen müssen sich aus numerischen Gründen (paarweise) mindestens um `mindist_bremsmessung` unterscheiden. Außerdem muss die kleinste der Geschwindigkeiten mindestens diesen Wert aufweisen.

Folgende Anforderungen müssen ebenfalls erfüllt sein. Eine Verletzung der Anforderungen wird aber erst während der Berechnung eines Schutzfeldes festgestellt. In diesem Fall liefert die Funktion schutzfeld_berechnen den Fehler sams_nicht_sicher und es liegt nach ihrer Ausführung damit kein gültiges Schutzfeld vor.

`BremsdatenNutzer` `FahrzeugdatenNutzer`	Es ist normalerweise nicht möglich eine Bremsmessung mit der tatsächlichen Maximalgeschwindigkeit des Fahrzeuges durchzuführen. Durch Ungenauigkeit in Steuerung und Messung werden im laufenden Betrieb immer wieder Geschwindigkeiten gemessen, die größer als die Geschwindigkeit der größten Bremsmessung sind. Außerdem werden Bremsmessungen nur für Geradeausfahrten durchgeführt. Für Kurvenfahrten $(v,\omega)^T$ bestimmt die Schutzfeldberechnung eine zu $(v,\omega)^T$ äquivalente Geradeausgeschwindigkeit, die zur Berechnung des Schutzfeldes verwendet wird. Diese Geradeausgeschwindigkeit ist größer als v und kann bei schnellen Kurvenfahrten ebenfalls oberhalb der Geschwindigkeit der größten Bremsmessung liegen. Aus diesem Grund darf die Geradeausgeschwindigkeit größer sein als die Maximalgeschwindigkeit der Bremsmessungen, aber maximal um den Faktor `faktor_ueberschreitung_v_max`. Wird dieser Faktor überschritten, so schlägt die Schutzfeldberechnung fehl. Um das Fehlschlagen der Schutzfeldberechnung und einen damit verbunden Halt des Fahrzeuges zu vermeiden, sollte deshalb eine der Bremsmessungen annähernd mit der erreichbaren Maximalgeschwindigkeit des Fahrzeuges durchgeführt werden. Der Faktor `faktor_ueberschreitung_v_max` sollte nicht zu groß gewählt werden

`Geschwindigkeitsbereich`	Die Größe des Geschwindigkeitsbereiches, für den ein Schutzfeld berechnet werden kann, ist durch die Konstante `maxdiff_alpha__faktor2pi` beschränkt. Diese Beschränkung ist aber nicht in v oder ω definiert, sondern ergibt sich aus einer abgeleiteten, im Verlauf der Schutzfeldberechnung ermittelten Größe. Beschränkt wird die maximale Größe der Winkeldifferenz zwischen den Orientierungen zweier Stoppposen, die sich für zwei beliebige Geschwindigkeitsvektoren aus dem Geschwindigkeitsbereich ergeben können. Diese Winkeldifferenz darf höchstens `maxdiff_alpha__faktor2pi` vollständige Kreisumdrehungen betragen.

4.5 Parametrierung des Speicherbedarfs

Die Datei `sams_konfig.h` enthält drei Parameter in Form von Makrodefinitionen für den C-Präprozessor. Über diese kann das Softwaremodul in gewissem Umfang an die praktischen Einsatzgegebenheiten angepasst werden. Die Werte können frei *innerhalb der hier angegebenen Intervalle* gewählt werden.

`MAXIMALE_ANZAHL_STRAHLEN`	Bestimmt die Elementanzahl des Arrays, in dem das Schutzfeld abgelegt wird. Sollte identisch zur maximalen Strahlenanzahl des verwendeten Laserscanners sein. *Voreingestellter Wert: 529. Gültigkeits-Intervall: [4, 65535]*
`MAXIMALE_ANZAHL_KONTURPUNKTE`	Bestimmt die maximale Anzahl an Punkten, mittels derer die Roboterkontur modelliert wird. *Voreingestellter Wert: 100. Gültigkeits-Intervall: [4, 4095]*
`APPROXIMATIONSPUNKTE_JE_BOGEN`	Bestimmt die Anzahl der Hilfspunkte, die verwendet werden, um einen Kreisbogen als regelmäßiges N-Eck zu approximieren. Es wird nur im Falle mangelnden Speichers empfohlen, diesen Wert anzupassen. *Voreingestellter Wert: 8. Gültigkeits-Intervall: [4, 360].*

Der Speicherbedarf der Komponente wird durch die Werte dieser Parameter bestimmt, siehe Abschnitt 4.2. Im Auslieferungszustand enthält die Datei empfohlene Standardwerte für diese Größen, die in den meisten Fällen nicht angepasst werden müssen.

4.6 Softwareintegration

Dieser Abschnitt beschreibt die Maßnahmen und Voraussetzungen, die der SW-Integrator durchführen bzw. sicherstellen muss um einen sicheren Betrieb der Schutzfeldberechnungskomponente zu gewährleisten.

4.6.1 Code-Integrität und mögliche Änderungen

Die Software muss bis auf die in Abschnitt 4.5 geschilderten möglichen Parametrierungen und die gegebenenfalls nötige Anpassung der `Int32`-Definition (siehe auch Abschnitt 4.2.1) *ohne Änderungen am Quellcode* integriert werden. Die Verwendung der von der Schutzfeldberechnung zur Verfügung gestellten Funktionen (Schnittstellen) muss gemäß der in Abschnitt 4.3 definierten Schnittstellenanforderungen erfolgen.

4.6.2 Schutz der Konfigurationsdaten

Die Funktion `nutzerkonf_vorverarbeiten`, die im Abschnitt 4.3.3 beschrieben ist, erzeugt die internen Konfigurationsdaten. Diese werden in einem modul-internen Speicherbereich abgelegt, dessen Adresse und Größe nach Aufruf von `nutzerkonf_vorverarbeiten` durch die Werte `*speicher_zeiger` und `*speicher_groesse` spezifiziert sind. Der SW-Integrator muss die Speicherkonsistenz dieses Bereichs sicherstellen, d. h. es muss gewährleistet sein, dass dieser Speicherbereich während der gesamten Laufzeit unverändert bleibt und fehlerfrei ist. Dies ist etwa dadurch zu realisieren, dass eine betriebsbewährte CRC-Funktion einmal vor dem ersten Aufruf der Schutzfeldberechnungsfunktion und danach regelmäßig zur Laufzeit eine Prüfsumme von dem geschützten Speicherbereich erstellt und auf Konstanz überprüft.

4.6.3 Sicherheitsprüfungen

Die folgenden Prüfungen müssen einmalig während des Integrationsprozesses durchgeführt werden:

- **Rechenzeit.** Die Rechenzeit der Schutzfeldberechnungsfunktion ist endlich (terminiert stets) und deterministisch. Eine Schranke für die maximale Rechenzeit einer Schutzfeldberechnung wird nicht garantiert, u.a. weil sie hardwareabhängig ist; eine Laufzeitüberwachung für die Gesamtapplikation bleibt hiervon unberührt, d.h. die Reaktionsfähigkeit der Applikation muss extern sichergestellt werden.

  Um die Verfügbarkeit der Applikation sicherzustellen, ist es zweckmäßig, einen zielhardwarespezifischen Richtwert für die Rechenzeit zu bestimmen. Hierbei wird exemplarisch für eine Kontur mit M Punkten gemessen, wobei M die Maximalzahl N der Konturpunkte in der Applikation deutlich überabschatzen sollte. Kann eine aussagekräftige Anzahl von Berechnungen mit M Punkten klar in der geforderten Zeit durchgeführt werden, kann davon ausgegangen werden, dass die Laufzeit ausreichend ist. Die Aufgabe des SW-Integrators besteht in der Erstellung der Konfiguration für den Test und der Zeitnahme.

- **Speicher.** Die Schutzfeldberechnungsfunktion benötigt einen konstanten Speicherplatz (siehe Abschnitt 4.2). Der SW-Integrator muss überpüfen, dass der vom Zielsystem bereitgestellte Speicher ausreichend ist.

4.6.4 Validation

Beispielkonfiguration Anhand der mitgelieferten Beispielkonfiguration kann das Laufzeitverhalten der Software zur Schutzfeldberechnung im Betrieb auf der Sicherungskomponente evaluiert werden. Die Beispielkonfiguration gibt eine Fahrzeugkontur und weitere Parameter vor, sowie die Ergebnisse von typischen Aufrufen der Schutzfeldberechnung (Funktion schutzfeld_berechnen). Diese können dazu verwendet werden, die Schutzfeldberechnung nach der Integration im System zu validieren.

Die Vorgehensweise zur Verwendung der Beispielkonfiguration ist in der dazugehörigen `README`-Datei beschrieben. Diese findet sich im Verzeichnis `Beispielkonfiguration`.

Validation durch Inaugenscheinnahme Die Korrektheit der berechneten Schutzfelder bezüglich der vom Anwender durch Aufruf der Funktion `nutzerkonf_vorverarbeiten` eingestellten Konfiguration ist durch die formale Verifikation des Softwaremoduls sichergestellt. Es muss also diesbezüglich *keine* Validation durchgeführt werden.

Um Fehler bei der Eingabe von Konfigurationsdaten (Vertausch von Parametern, falsche Größeneinheiten, usw.) zu entdecken, können die Schutzfelder visualisiert und vom Anwender auf Sinnhaftigkeit geprüft werden. Die Visualisierung wird vom Softwaremodul nicht geleistet; die Daten für eine Visualisierung können berechnet werden, indem für verschiedene Geschwindigkeitsintervalle die Funktion `schutzfeld_berechnen` aufgerufen wird.

4.7 Sonstiges

Der SW-Integrator muss die Möglichkeit für eine tägliche Funktionsprüfung zur Verfügung stellen. Dazu kann z.B. auf Tastendruck das Schutzfeld zur maximalen Geschwindigkeit aktiviert und vom Laserscanner überprüft werden.

Neben dem sicherheitsgerichteten Schutzfeld ist häufig auch ein Warnfeld sinnvoll, um frühzeitig die Geschwindigkeit des Fahrzeuges zu reduzieren. Dieses kann durch den Aufruf der Funktion `schutzfeld_berechnen` mit verschiedenen Latenzzeitaufschlägen (Funktionsparameter `t_l`) erreicht werden. Warnfelder sind nicht Bestandteil der für den Einsatz nach SIL-3 zertifizierten Sicherheitsfunktion der Schutzfeldberechnung. Sie stellen lediglich ein Zusatzmerkmal des Softwaremoduls dar.

Literatur

[Leuze electronics] LEUZE ELECTRONICS: *ROTOSCAN RS4/PROFIsafe*. `http://www.leuze.de/products/las/slsc/rs4-p/p_01_de.html`. – Produktübersicht Sicherheits-

Laserscanner

Verzeichnis der Begriffe

Sicherungskomponente für
Autonome Mobile Systeme

Eine Kooperation zwischen
DFKI-Labor Bremen • Leuze lumiflex • Universität Bremen

SAMS Verifikationsumgebung – Referenzhandbuch

Zusammenfassung

Dieses Dokument beschreibt die Funktionalität der im Rahmen des SAMS-Projektes entwickelten und verwendeten Verifikationsumgebung (SAMS-VU). Der Arbeitsfluss bei der Verifikation von Programmen wird dargestellt und es werden der unterstützte Sprachumfang (eine Untermenge der Programmiersprache C) sowie die verwendete Spezifikationssprache definiert.

Projektbezeichnung	SAMS
Verantwortlich	Christoph Lüth, Dennis Walter
Erstellt am	25.01.2008
Version	2.3
Bearbeitungszustand	fg.
Revision	4326
Letzte Änderung	08.10.2009
Dokumentablage	Projektdokumente/Verifikationsumgebung/svw-spec.tex

Änderungsliste

29.01.08 – *Version:* 1.0 – *Bearbeiter:* C. Lüth

Initiale Version.

30.01.08 – *Version:* 1.1 – *Bearbeiter:* D. Walter

Einschränkungen des unterstützten Sprachumfangs aus dem Wiki übertragen.

31.01.08 – *Version:* 1.2 – *Bearbeiter:* D. Walter

Korrektur gelesen und nachgebessert

01.02.08 – *Version:* 1.3 – *Bearbeiter:* C. Lüth

Semantik der Zustandsprädikate genauer erläutert, weitere kleinere Änderungen.

05.02.08 – *Version:* 1.4 – *Bearbeiter:* D. Walter

Abschnitt “Zeiger und Felder” hinzugefügt

08.02.08 – *Version:* 1.5 – *Bearbeiter:* D. Walter

Abschnitt “Erlaubte Typumwandlungen” hinzugefügt. Abschnitt “Einschränkungen” erweitert.

10.11.08 – *Version:* 1.6 – *Bearbeiter:* C. Lüth

Neue Operatoren hinzugefügt: Isabelle-Funktionen, Quote/Antiquote, `@`.

10.02.09 – *Version:* 1.7 – *Bearbeiter:* D. Walter

Neue Sprachelemente für Repräsentationsfunktionen hinzugefügt. Interpretation der Spezifikationssprache in Isabelle detailliert.

18.02.09 – *Version:* 2.0 – *Bearbeiter:* D. Walter

Abschnitte “L-Werte”, “Normierung von Schleifen”, “Spezifikationssprache”, “Interpretation der Annotationselemente in Isabelle” eingefügt bzw. erweitert und überarbeitet. Technische Aufräumarbeiten.

04.06.09 – *Version:* 2.1 – *Bearbeiter:* D. Walter

Kapitel 4 neu geschrieben und inhaltlich erweitert. Aktualisierung aller Abschnitte auf derzeitigen Stand der Sprachteilmenge und Spezifikationssprache (neue Sprachelemente `@define`, `:: abbreviation`). Anpassung der Grammatik in Kap. A.

05.06.09 – *Version:* 2.2 – *Bearbeiter:* D. Walter

Feinarbeiten an allen Kapiteln. Kapitel 4 vorläufig fertiggestellt.

30.06.09 – *Version:* 2.3 – *Bearbeiter:* D. Walter

Anhang D eingefügt; Anhänge A und B aktualisiert;

Prüfverzeichnis

31.01.08 – *Version:* 1.5 – *Prüfer:* D. Walter
Neuer Produktzustand: **i.B.**

Korrektur gelesen und nachgebessert.

12.02.08 – *Version:* 1.5 – *Prüfer:* C. Lüth
Neuer Produktzustand: **vg (TÜV)**

Kleinere Rechtschreibkorrekturen und Klarstellungen direkt eingearbeitet.

05.06.09 – *Version:* 2.2 – *Prüfer:* C. Lüth
Neuer Produktzustand: **vg (TÜV)**

Änderungen an Kapiteln 1 und 2 direkt eingearbeitet.

30.06.09 – *Version:* 2.3 – *Prüfer:* M. Bortin
Neuer Produktzustand: **vg (intern)**

Korrektur gelesen und nachgebessert. Der Auzählung der Spezifikationsklauseln in Kapitel 3 @assigns hinzugefügt und erwähnt das @assigns bzw. @modifies auch vor Schleifen verwendet werden können.

25.09.2009 – *Version:* 2.3 – *Prüfer:* TÜV (P. Supavatanakul)
Neuer Produktzustand: **fg.**

Siehe Prüfbericht (Techical Report no. LF82764T)

Inhaltsverzeichnis

1 Funktionsweise

1.1 Begrifflichkeiten

- Isabelle (Nipkow u. a., 2002) ist ein generischer Theorembeweiser, in dem diverse Logiken formalisiert wurden. Innerhalb dieses Dokuments ist mit **Isabelle** immer die Formalisierung von einfach getypter Logik höherer Stufe (*Isabelle/HOL*) —mit der dazugehörigen Syntax und den dazugehörigen Theorien— gemeint. Die für die Entwicklung der Verifikationsumgebung verwendete Isabelle-Version ist *Isabelle 2009*.

- Mit **C** ist die Programmiersprache C gemäß dem Standard ISO/IEC 9899 in der ersten Ausgabe (1990) gemeint. Ist eine Unterscheidung zur Sprache wie sie im neueren Standard ISO/IEC 9899:1999 definiert wird nötig, werden die Sprachen jeweils als C90 bzw. C99 bezeichnet.

1.2 Einführung

Die in diesem Dokument beschriebene und in der SAMS-VU implementierte formale Verifikation von C-Programmen dient dem Nachweis der funktionalen Korrektheit von Programmen. Es können dabei folgende Programmeigenschaften nachgewiesen werden:

1. die Erfüllung von Vor- und Nachbedingungen von Programmfunktionen;

2. die Termination von Programmen, insbesondere also der darin enthaltenen Schleifen;

3. die Effekte jeder Funktion — d.h. die nach Aufruf sichtbaren möglichen Änderungen von Objekten im Speicher.

4. Die *Ausführungssicherheit*, d. h. die Garantie, dass

   - Arrayzugriffe nicht außerhalb der erlaubten Grenzen vorgenommen werden
   - Zeigerdereferenzierungen nur auf gültigen Zeigern ausgeführt werden
   - keine Division durch 0 durchgeführt wird.

Diese Eigenschaften werden gesondert für jede Funktion nachgewiesen.

Vor- und Nachbedingungen werden dabei als zustandsabhängige Prädikate — formuliert über Programmvariablen — spezifiziert. Eine Funktion erfüllt Vor- und Nachbedingung P und Q, wenn für alle Zustände, die der Vorbedingung P genügen, die Funktion in einem Zustand endet, der Q erfüllt (totale Korrektheit (Loeckx u. a., 1987)). Effekte werden als eine Menge von Speicherstellen spezifiziert, die eine Funktion verändern kann.

Dieses Dokument richtet sich an Benutzer der SAMS-Verifikationsumgebung, also zum einen an die *Programmierer* und zum anderen an die *Verifizierer*.

Programmierer finden in diesem Dokument die Spezifikation der Sprachteilmenge von C, welches die Verifikationsumgebung abdeckt (es handelt sich um eine Teilmenge des MISRA-Standards (MISRA, 2004)). Allein die Einhaltung dieser Sprachteilmenge bedeutet eine höhere Software-Qualität. Programmierer können das syntaktische Frontend daher als ein über den MISRA-Standard hinausgehendes Prüfwerkzeug benutzen; sie kommen mit Isabelle direkt nicht notwendigerweise in Berührung.

Verifizierern obliegt die Aufgabe, das gewünschte Programmverhalten formal zu spezifizieren (meist in Zusammenarbeit mit dem Programmierer), und die durch die Spezifikation erzeugten Beweisverpflichtungen in Isabelle zu beweisen. Sie müssen Isabelle kennen und benutzen können. Verifizierer finden in diesem Dokument die Spezifikationssprache, sowie die für die Beweisführung in Isabelle nötigen Informationen über die Übersetzung von Spezifikationen und Programmen nach Isabelle.

1.3 Annotationen

```
/*@
  @ensures \result >= x && \result >= y
       && (\forall int z; z >= x && z >= y --> \result <= z)
  @*/
int max(int x, int y)
{
  int r = 0;
  if (x <= y) { r = y; }
  else { r = x; }
  return r;
}
```

Abbildung 1: Beispiel für annotierten Quellcode

In der angewandten Spezifikationsmethode wird der Quellcode mit Korrektheitsbedingungen, d. h. prädikatenlogischen Aussagen über den Programmzustand, versehen (sog. *annotierter Quellcode*).

Die Verwendung von annotiertem Quellcode hat dabei folgende Vorteile:

- Es besteht eine enge Bindung zwischen Spezifikation und Quellcode, was die Zusammenarbeit von Programmierer und Spezifikateur fördert.
- Der Quellcode muss (v.a. nach Änderungen) nicht in eine Spezifikationsumgebung überführt werden.
- Die Konsistenz zwischen Quellcode und Verifikation wird automatisch sichergestellt, da der Quellcode sowohl für Verifikation als auch Übersetzung zu einer ausführbaren Datei verarbeitet wird (siehe Abb. 3).

Abb. 1 zeigt ein einfaches Beispiel für eine annotierte Funktion, die das Maximum der beiden Parameter berechnet. Die Spezifikation einer Funktion wird innerhalb eines speziell formatierten Kommentars /*@ ... @*/ angegeben und ist damit für herkömmliche Compiler nicht sichtbar. Diese Technik ist u.a. aus der Java Modelling Language (JML) (Burdy u. a., 2005) bekannt. Die Verifikationswerkzeuge Why und Caduceus (Filliâtre u. Marché, 2004, 2007) sowie das diese Werkzeuge seit Anfang 2009 subsumierende Rahmenwerk FRAMA-C mit der Spezifikationssprache ACSL (**?**) verwenden eine ähnliche Syntax.

1.4 Architektur

Das Kernstück der SAMS Verifikationsumgebung ist ein in dem Theorembeweiser Isabelle (Nipkow u. a., 2002) implementierter Verifikationsbedingungsgenerator (VCG, *verification condition generator*). Dieser besteht aus einem Satz von Regeln (ähnlich denen aus dem klassischen Hoare-Kalkül (Winskel, 1993; Loeckx u. a., 1987)), deren Korrektheit gegenüber der Semantik der Programmiersprache in Isabelle bewiesen wurde.

Weitere Komponenten der Verifikationsumgebung sind ein syntaktisches Frontend, welches Programme in annotierter C-Syntax einliest, eine statische Analyse (Typprüfung sowie Konformität zur vorgegebenen Sprachteilmenge) durchführt, und in die abstrakte Syntax der internen semantischen Repräsentation konvertiert, sowie die Domänenmodellierung, welche die in dem Programm und in der Spezifikation verwendeten Datentypen, mathematischen Sätze und Beweiswerkzeuge bereitstellt. In diesem Fall beinhaltet die Domänenmodellierung insbesondere die Mechanik bewegter Objekte in der Ebene.

Abb. 2 zeigt die Architektur der Verifikationsumgebung. Programme in annotierter C-Syntax (links) werden von dem syntaktischen Frontend gelesen, und in die semantische Repräsentation umgewandelt. Der regelbasierte VCG reduziert die annotierten Korrektheitsbedingungen zu einer Menge von Verifikationsbedingungen. Einige von diesen sind triviale Aussagen, die automatisch bewiesen werden können. Andere sind nicht automatisch herleitbare Aussagen, die sich insbesondere auf die Domänenmodellierung beziehen. Aussagen dieser Art sind der Kern der Korrektheitsargumentation. [1] Solche nicht mehr automatisch zu beweisenden Aussagen kennzeichnen die reiche Domänenmodellierung, die für Robotik-Systeme benötigt wird; hier kann man nicht mehr erwarten, mit automatischen Beweiswerkzeugen allein alle Bedingungen herleiten und beweisen zu können.

1.5 Datenfluss und Arbeitsweise

Die SAMS-Verifikationsumgebung baut auf der Arbeitsweise des Theorembeweisers Isabelle auf. Dieser kennt für den halbautomatischen Beweis zwei Modi: im *interaktiven* Modus werden Theorien und Beweise interaktiv schrittweise erstellt; der Theorembeweiser gibt direktes Feed-

[1] Beispiele wären numerische Abschätzungen der Genauigkeit von Ergebnissen, oder eine auf anderen nichttrivialen mathematischen Zusammenhängen beruhende Berechnung. Nicht-triviale Arrayindizes können häufig ebenfalls nicht vollautomatisch als gültig bewiesen werden.

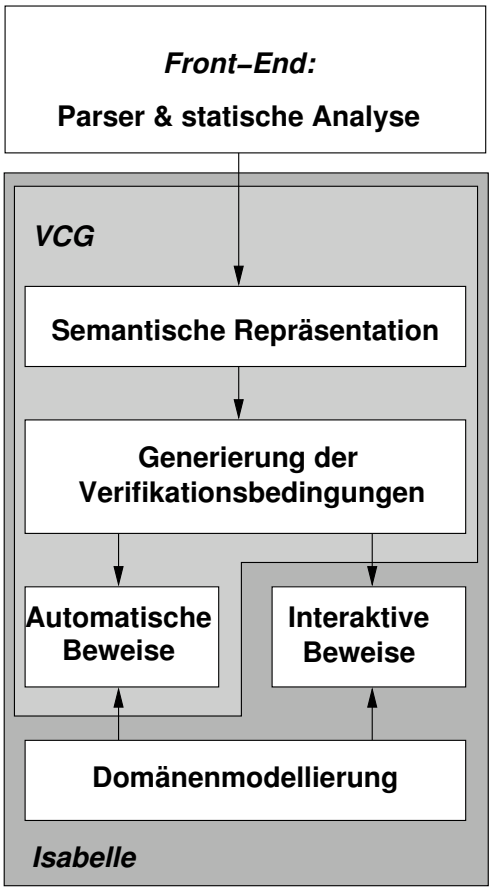


Abbildung 2: Architektur der Verifikationsumgebung

back für jeden Beweisschritt. Im *Dokumenten-Modus* wird eine (vorher im interaktiven Modus erstellte) Bücherei von Theorien eingelesen, auf Korrektheit geprüft, und über die Zwischenstufe von generiertem LaTeX-Code in ein PDF-Dokument verwandelt. Das PDF-Dokument kann als Zertifikat der Korrektheit der Theorien betrachtet werden.

Die Arbeitsweise und der Datenfluss der SAMS-Verifikationsumgebung ist in Abb. 3 skizziert. Ausgangspunkt ist immer ein annotiertes Quellprogramm (links oben). Dieses kann entweder direkt zu einem ausführbaren Programm übersetzt werden (linke Seite), oder die Korrektheit des Programms bezüglich seiner Spezifikation kann verifiziert werden (rechte Seite). Im zweiten Fall liest das Frontend das Programm ein, überprüft die Typkorrektheit (sowohl des ausführbaren Programmes als auch der Annotationen) und stellt sicher, dass der Quellcode innerhalb der definierten Sprachuntermenge liegt. Danach wird eine Isabelle-Theorie generiert, welche die Repräsentation des zu verifizierenden Programmes und der Spezifikationen enthält. Für jede Funktion in der Quelldatei muss es jetzt eine Isabelle-Theorie geben, welche den Korrektheitsbeweis für diese Funktion enthält; das Frontend kann *Stubs* für diese Beweise erzeugen, aber den Hauptteil der Beweise muss der Verifizierer interaktiv erstellen. Ferner erzeugt das Frontend eine Protokoll-Datei, welche verschiedenen Informationen über den übersetzten Quellcode (Revision,

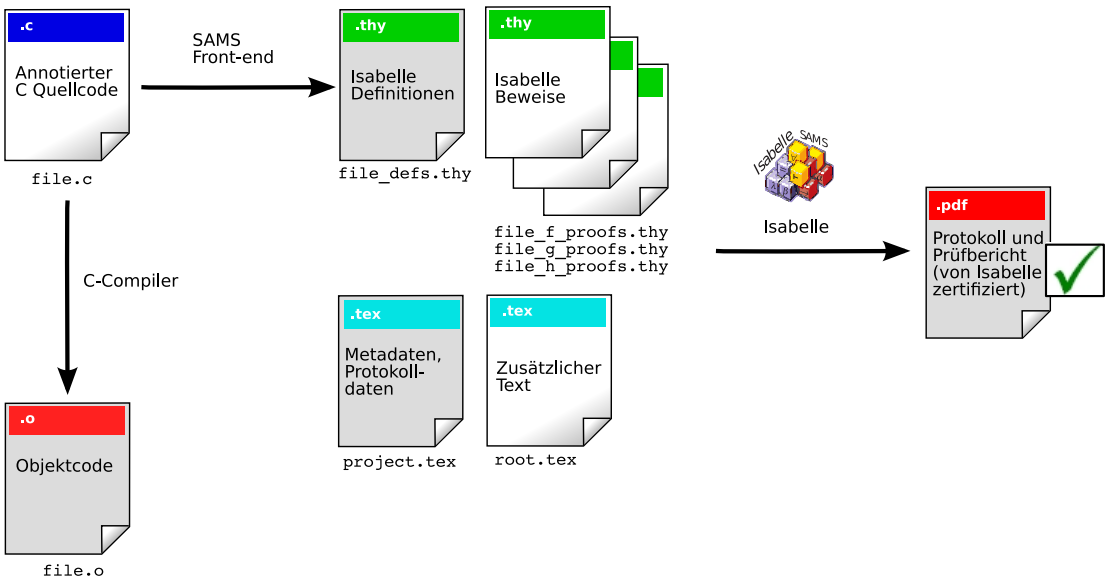


Abbildung 3: Datenfluss in der SAMS-Verifkationsumgebung. Generierte Dateien sind grau. Die Namen sind beispielhaft.

Datum, Größe, auf Wunsch MD5-Hash) enthält; die Protokoll-Datei dient dazu, rekonstruieren zu können, aus welcher Quelldatei die Theorien erzeugt wurden.

Sind alle Beweise beendet, wird Isabelle im Dokumentenmodus aufgerufen (durch das Werkzeug `make`), und erzeugt (via LaTeX) eine PDF-Datei, welche die übersetzten Theorien und Korrektheitsbeweise enthält. Diese PDF-Datei dient als Nachweis der Korrektheit. Die relevanten Daten für die Zertifikation sind also die Isabelle-Theorien; ihre Korrektheit kann mit Isabelle automatisch überprüft werden.

1.6 Grenzen, Korrektheit und Validation

Die Grenzen der hier vorgestellten Herangehensweise werden in Abschnitt 5 diskutiert.

Die Frage der Korrektheit und Validation — warum können wir den Ergebnissen der Verifikation vertrauen, und wie stellen wir das sicher — werden in einem separaten Dokument (Lüth, 2009) behandelt.

2 Programmiersprache

Die SAMS Verifikationsumgebung dient der Verifikation von Programmen, die in der Sprache C gemäß ISO/IEC 9899:1990 (kurz: C90) verfasst sind. In diesem Kapitel werden die normativ geforderten und technisch notwendigen Einschränkungen des Sprachumfangs beschrieben.

Die unterstützte Sprachteilmenge orientiert sich eng an den MISRA-Programmierrichtlinien (MISRA, 2004), der viele aus sicherheitstechnischer Sicht bedenklichen Konstrukte der Programmier-

sprache C einschränkt, und bereits einen anerkannten Industriestandard im Bereich der Programmierung eingebetteter sicherheitskritischer Systeme darstellt. Die hierdurch definierte Sprachteilmenge muss allerdings für die Zwecke der formalen Verifikation weiter eingeschränkt werden. Anhang C legt die Abweichungen vom MISRA-Standard im Detail dar.

2.1 Typen und Deklarationen

Folgende Typen und Typkonstruktoren werden gar nicht unterstützt. Diese können weder als `typedef` vereinbart werden, noch dürfen Bezeichner dieser Typen deklariert werden, noch als Werte mit diesen Typen im Quelltext verwendet werden:

(1) Vereinigungstypen (`union`);

(2) Bitfelder (Felder von Strukturen mit angegebener Breite in Bits);

(3) Zeiger auf Funktionen;

(4) Zeiger auf Aufzählungstypen (`enum`);

(5) der Typqualifikator `volatile`;

(6) unvollständige Datentypen;

(7) als `static` deklarierte lokale Objekte.

Eine Vorwärtsdeklaration von Strukturen und ihre zwischenzeitliche Verwendung als unvollständiger Typ (auf den dann z.B. schon Zeiger verweisen dürfen) wird nicht unterstützt.

2.1.1 Erlaubte Typumwandlungen

Arithmetische Konversionen Bezüglich impliziter und expliziter arithmetischer Konversionen gelten die Regeln des Abschnitts 6.10 des MISRA-Standards. Grob gesprochen werden nur bezüglich der Zielarchitektur werterhaltende Konversionen unterstützt.

Zeigerkonversionen Die einzige erlaubte Zeigerkonversion ist diejenige von einem nicht qualifizierten Zeigertypen in denselben, zusätzlich mit dem `const`-Attribut versehenen Typen. Der Nullzeiger ist ein ausgezeichneter Wert (CStandard, 1999, §6.3.2.3 (3)); er kann in Zeiger auf beliebige andere Werte konvertiert werden. *Nicht* erlaubt sind die Konversionen

- implizit oder explizit von oder nach `void` $*$ für Zeiger außer dem Nullzeiger;
- eines ganzzahligen Wertes in einen Zeiger oder umgekehrt für Werte ungleich Null;
- zwischen Zeigern verschiedenen Typs (außer dem Nullzeiger);
- eines mit dem `const`-Attribut versehenen Typen in einen unqualifizierte Typ.

Kompatible Typen (CStandard, 1999, §6.2.7) Zwei Typen gelten als kompatibel, wenn sie nach Anwendung aller Typdefinitionen (`typedef`) gleich sind. Die Reihenfolge der Typattribute sowie das implizite `int` spielen hierbei keine Rolle; `const long int` und `long const` sind kompatibel.

2.1.2 Deklarationen

Typqualifikatoren und Speicherklassen Die Verwendung von `static` für globale Bezeichner, die lokal zu einer Übersetzungseinheit deklariert werden, ist zulässig. Bei der Übersetzung nach Isabelle werden diese globalen Bezeichner durch Umbennenung global eindeutig gemacht.

Der Typqualifikator `register` ist semantisch transparent, und wird von der SAMS-Verifikationsumgebung ignoriert.

Initialisierer Initialisierer müssen die Struktur des initialiserten Deklarators berücksichtigen, und dürfen nicht "flachgeklopft" werden. Die Verwendung partieller Initialisierungen ist zulässig:

```
struct p { int x, int y};
struct q { struct p a; float len; };

struct q q1 = {{4, 5}, 3.0}; /* OK */
struct q q2 = {1, 2, 3.0};   /* Falsch */
struct p p  = {1};           /* OK */
int a[10]   = {1, 2, 3};     /* OK */
```

Funktionen Die Typen der Funktionsparameter sowie der Rückgabetyp von Funktionen darf nur ein skalarer Typ sein (arithmetischer Typ oder Zeiger, keine Strukturen oder Felder).

Jede Funktion muss vor Benutzung mit einem Prototypen deklariert werden.

Es sind nur Parameterlisten definierter Länge zulässig; die Deklaration von Parametern mittels Ellipsis (...) ist nicht zulässig.

2.2 Ausdrücke

Nicht unterstützt werden folgende Operatoren:

- Der Komma-Operator,
- die Schiebe-Operatoren (`x << y`, `a >> b`), und
- die Bit-Operatoren (`a & b`, `a | b`, `a ^ b`).

Der Komma-Operator ist ohne Seiteneffekte ohnehin zweckfrei; Operationen auf Bit-Ebene sind in unserer Anwendungsdomäne weniger typisch und werden daher zur Zeit nicht unterstützt.

2.2.1 Auswertung von Ausdrücken

Das Frontend stellt sicher, dass alle Ausdrücke in jeder Auswertungsreihenfolge zu demselben Ergebniswert, der nicht undefiniert sein darf, führen. Konkret sind folgende Ausdrücke nur auf oberster Ebene (als eine aus einem Ausdruck dieser Art bestehende Anweisung) erlaubt, und dürfen nicht innerhalb eines Ausdrucks auftreten:

(1) Zuweisungen,

(2) Inkrement- und Dekrement-Operatoren, und

(3) Aufrufe von nicht seiteneffektfreien Funktionen.

Dies schließt Ausdrücke wie $(x = 1) + (x = 2)$ oder $a[i{+}{+}] = i$ aus, deren Wert gemäß C90-Standard undefiniert ist. Es schließt jedoch auch Ausdrücke aus, deren Wert aufgrund der unspezifizierten Auswertungsreihenfolge uneindeutig sein kann, wie z. B. $g() - f()$. (Es ist möglich, dass sowohl der Rückgabewert von f, als auch der von g von einer globalen Variablen abhängen, die wiederum von beiden Funktionen modifiziert werden kann.)

Reine, d. h. seiteneffektfreie Funktionen wie beispielsweise die trigonometrischen Funktionen sin und cos dürfen in verschachtelten Ausdrücken an beliebiger Stelle verwendet werden. Die Seiteneffektfreiheit muss durch die Annotation @modifies \nothing (Abschn. 3.5) spezifiziert sein.

2.2.2 L-Werte

Die Definition von L-Werten in (CStandard, 1999, §6.3.2) als Ausdrücke von Objekttyp oder einem unvollständigen Typ außer void ist sehr weit gefasst. Im ANSI-Standard von 1989 werden allgemein L-Werte und veränderliche L-Werte unterschieden, um den Konflikt innerhalb des Standardgremiums beizulegen, ob L-Werte generell Ausdrücke seien, die Speicherstellen bezeichnen, oder solche, die auf der linken Seite einer Zuweisung vorkommen dürfen(CStandard Rationale, 1989, §3.2.2.1). Die Bezeichnung "veränderlicher L-Wert" wurde für das zweitgenannte Konzept eingeführt. Die von der SAMS-Verifikationsumgebung unterstützte Sprachteilmenge gibt eine einfachere Definition von L-Wert als *Bezeichner einer Speicherstelle* vor.

Ein *L-Wert* ist ein Ausdruck, der über Variablen- und Feldnamen durch die Operatoren Dereferenzierung ($*L$), Arrayzugriff ($L[E]$), Feldzugriff ($L.f$ und $L{-}{>}f$) gebildet wird. Hierbei steht L für einen Unterausdruck, der ein L-Wert ist, und E für einen seiteneffektfreien Ausdruck von ganzzahligem Typ. Siehe hierzu auch Abb. 4.

2.2.3 Ausdrücke strukturierten Typs

Ausdrücke strukturierten Typs (struct) werden nicht unterstützt. Variablen strukturierten Typs dürfen deklariert und verwendet werden, doch können über sie keine Ausdrücke formuliert werden. Im Einzelnen bedeutet dies:

Beispiel 1: Die folgenden Ausdrücke sind L-Werte (/* *OK* */), bzw. keine L-Werte (/* *Falsch* */)

```
*x                  /* OK */
a.f.g[i + 1]        /* OK */
n->p[1].*a          /* OK */
&x.f                /* Falsch */
*f(x, y)            /* Falsch */
*(p + 1)            /* Falsch */
```

Abbildung 4: Beispiele für L-Werte nach Definition in 2.2.2

1. Zuweisungen an Objekte strukturierten Typs müssen komponentenweise erfolgen.
2. Objekte strukturierten Typs können nicht direkt durch die Operatoren == bzw. != verglichen werden, sondern ebenfalls nur komponentenweise.
3. Funktionen können keine Strukturen als Argumente erwarten oder solche zurückgeben.
4. Es können Arrays von Strukturen gebildet werden.
5. Strukturen können Bestandteil anderer Strukturen sein.
6. Die Adresse von Strukturen kann berechnet werden, d. h. `&s` ist ein gültiger Ausdruck von *Zeigertyp* für einen L-Wert `s` von strukturiertem Typ.
7. Der Typ "Zeiger auf T" für einen strukturierten Typen T unterliegt diesen Einschränkungen nicht. Insbesondere können Werte dieses Typs zugewiesen werden und Funktionen können derartige Werte als Argumente und Rückgabewerte verwenden.

2.2.4 Zeiger und Felder

Die Verwendung von Zeigern und Feldern ist erlaubt. Sie unterliegt jedoch einigen Einschränkungen im Vergleich zum vollen Sprachumfang von C, insbesondere die Adressarithmetik betreffend.

Ein Feldtyp `T[]` und ein Zeigertyp `T *` werden *nur dann* gleichgesetzt, wenn diese als Parameter einer Funktion auftauchen. Ein Feldname wertet zur Adresse des ersten Elements des Feldes aus. Die in C typische Wertgleichheit von `v` und `&v` für Feldnamen besteht nicht, vielmehr ist der zweite Ausdruck nicht zulässig.

Adress-Arithmetik wird auf Zeigern und Feldern nur in Form der Auswahl eines Feldindexes (`v[e]`, wobei `e` ein Integer-Ausdruck ist) unterstützt. Weitergehende Zeigerarithmetik (Subtraktion von Zeigern, die in dasselbe Feld zeigen: `p - q`; Addition von ganzzahligen Werten zu einem Zeiger: `*(p + 3)`) wird vom Frontend zurückgewiesen; Abb. 5 zeigt einige Beispiele.

```
int a[5];
int *p;

a[3];    /* OK */
p[3];    /* Falsch */
*a(+3); /* Falsch */
a;       /* Nicht zulaessig */
*p;      /* OK */
*(p+1); /* Falsch */
a[10];   /* Nicht zu verifizieren */
```

Abbildung 5: Zulässige und unzulässige Zeigerarithmetik

2.2.5 Funktionsparameter

Bei Funktionsparametern werden Zeiger und Felder gleichgesetzt. Folgende Deklarationen sind äquivalent:

```
int f(int *a);
int f(int a[]);
```

Falls im Rumpf der Funktion f auf a als Feld zugegriffen wird, muss a als ein Feld bestimmter Größe spezifiziert werden, um f verifizieren zu können. Dazu dient die Annotation \array(a, len), wobei len die Größe des Feldes ist. Die Größe kann eine Konstante oder ein anderer Parameter sein (siehe Abschn. 3.1.3):

```
/*@
  @requires \array(a, 5)
  @ */
```

Im Rahmen der Verifikation werden alle Zugriffe auf Zulässigkeit geprüft, d.h. ob sie bei Feldern innerhalb der deklarierten Größe liegen, oder bei Parametern innerhalb der spezifizierten Größe. Deshalb führt in Abb. 5 der letzte Ausdruck zu der unbeweisbaren Bedingung $10 < 5$.

Für Parameter, die natürlicherweise Strukturen oder Felder sind, kann ein Zeiger auf den Eingabewert übergeben werden. Dies simuliert einen *call by reference*. Derartige Zeiger sind als Zeiger auf Konstanten (const Typattribut) zu deklarieren.

2.2.6 Adress- und Dereferenzierungsoperator

Der Adress-Operator & und der Dereferenzierungsoperator $*$ darf lediglich auf L-Werte angewandt werden.

Das Frontend prüft und stellt sicher, dass die Adresse von Objekten mit automatischer Speicherklasse keinen Objekten zugewiesen wird, deren Lebensdauer die des Objekts, dessen Adresse es beinhaltet, übersteigt. Gültig, wenn auch wenig sinnvoll, ist ein Vorgehen wie das folgende:

```
short f(const short *ip_1, const short *ip_2)
{
  const short *aux = ip_1;    /* Zuweisungen an Objekt mit */
  if (*ip_2 < *aux) {         /* k\"urzerer Lebensdauer    */
    aux = ip_2;
  }
  return *aux * 2;
}
```

Nicht erlaubt ist es dagegen, die Funktionsparameter `ip_i` an eine globale Variable zuzuweisen (wobei dies nicht die einzige Art darstellt, Adresswerte länger zu speichern als erlaubt).

2.3 Anweisungen

Folgende Anweisungen werden nicht unterstützt:

(1) Zusammengesetzte Zuweisungen (z. B. +=);

(2) Fallunterscheidungen mit **switch**;

(3) Kontrollflußänderungen mit **break** oder **continue**;

(4) Sprünge mittels **goto** oder longjmp();

(5) Rekursion, direkt oder indirekt.

2.3.1 Einfache Zuweisungen

Bei einer Zuweisung muss der Typ des rechten Operanden kompatibel mit dem Typ des linken Operanden sein. Dies gilt —im Gegensatz zu (CStandard, 1999, §6.5.16.1 (1)) — insbesondere auch für arithmetische Typen.

Ferner muss der Typ des linken Operanden ein skalarer Typ (arithmetischer Typ oder Zeigertyp) sein; Strukturen oder Felder dürfen nicht zugewiesen werden.

2.3.2 Funktionsargumente/-rückgabewerte

Die zulässigen Typen von Funktionsargumenten und Funktionsrückgabewerten (Argumenten der **return**-Anweisung) sind identisch zu (CStandard, 1999, §6.5.2.2) festgelegt: Jedes Argument soll einen Typ haben, so dass dessen Wert einem Objekt mit der nicht-qualifizierten Version des Typs des entsprechenden Parameters (bzw. Rückgabetyps) zugewiesen werden kann.

2.3.3 Austrittspunkte von Funktionen

Jede Funktion hat einen ausgezeichneten Austrittspunkt, an dem die Funktion verlassen wird. Dieser befindet sich immer am Ende des Funktionskörpers. Hat eine Funktion logisch gesehen zwei Austrittspunkte, z. B. in den Zweigen einer **if**-Anweisung, so ist eine lokale Variable zu verwenden, über die beide Pfade wieder zusammengeführt werden können, wie in diesem Beispiel:

```
int foo(int a, int b)
{
 int result;
  if (a < b) { result = a; }
  else { result = b; }
  return result;
}
```

Genauso zulässig wäre die kürzere Formulierung

```
  return a < b ? a : b;
```

2.3.4 Funktionen der Standardbücherei

Viele Funktionen der Standardbücherei dürfen nicht verwendet werden, da sie semantisch nicht modellierbare Effekte haben, beispielsweise dynamische Speicherverwaltung, Ein- und Ausgabe, Signale und Unterbrechungen, Zufallszahlen, Zeit. Abschn. B enthält eine detaillierte Kategorisierung.

2.3.5 Normierte Repräsentation von Schleifen

Während C drei Arten von Schleifen kennt, verwendet die Programmrepräsentation der Verifikationsumgebung lediglich ein Konstrukt für alle Schleifen. Dieses entspricht der gewöhnlichen **while**-Schleife. Die notwendigen Übersetzungen lassen sich auf Ebene des Quellcodes wie folgt beschreiben:

Darstellung im Quellcode	*wird übersetzt zu*
for (E; B; I) { S }	E; **while** (B) { S; I }
do { S } **while** (B);	S; **while** (B) { S }

Für den Anwender ergibt sich hierbei keine weitere Einschränkung, allerdings ist der MISRA-Standard restriktiv, was Schleifen betrifft: die Kontrollanweisung *B* darf keine Seiteneffekte haben, bei **for**-Schleifen darf die Initialisierung *E* und das Inkrement *I* nur die Laufvariable initialisieren resp. inkrementieren, und die Laufvariable darf im Rumpf der Schleife nicht modifiziert werden (MISRA, 2004, Regel 13.5, 13.6).

2.3.6 Präprozessor

Die Verwendung des C-Präprozessors und damit von Makros ist erlaubt und unterliegt lediglich den Einschränkungen, wie sie vom MISRA-Standard vorgegeben werden.

Symbolische Konstanten Eine wichtige Aufgabe von Makros ist zur Verwendung als symbolische Konstanten, etwa für Arraygrößen:

```
#define V_ARRSZ 100
/* ... */
double v[V_ARRSZ];
```

Das Frontend wendet den GNU C-Präprozessor `cpp` auf jede Übersetzungseinheit an, bevor es selbst die Eingabe parst und verarbeitet. [2] Hierdurch gehen jedoch derartige symbolische Konstanten verloren, da sie bereits durch ihre Definition ersetzt wurden. Um es in den speziellen Fällen, in denen Makros als symbolische Konstanten (von Integer- oder Fließkommatyp) definiert werden, zu ermöglichen die Symbolnamen nach Ausführung des Präprozessors zu erhalten, wurde das Konzept der bedingten Übersetzung angewandt.

Das Makro VERIFY_MODE erlaubt eine Unterscheidung zwischen der Erzeugung von ausführbarem Code und der Übersetzung nach Isabelle mittels des Frontends. Dabei gilt:

1. Das Makro ist ausschließlich im Fall der Übersetzung nach Isabelle definiert.

2. Der Quellcode enthält keine Definition des Makros und wird diesbezüglich auch niemals modifiziert. Die Definition wird nur mittels Kommandozeilenargument `-DVERIFY_MODE` des `cpp` vorgenommen.

3. Die ausschließliche Verwendung der bedingten Übersetzung anhand der Definition dieses Makros — d. h. die einzige Verwendung der Bedingung #ifdef VERIFY_MODE — dient der Einführung symbolischer Konstanten wie im folgenden Beispiel

   ```
   #ifdef VERIFY_MODE
   /*@
     @define int X = Expr;
     @*/
   #else
   #define X Expr
   #endif
   ```

4. Zur Verwendung des @define-Konstrukts siehe 3.9.

[2] Durch Verwendung der Option `-C` wird sichergestellt, dass Kommentare, insbesondere also Annotationen, in der Ausgabe erhalten bleiben.

3 Spezifikationssprache

Die Spezifikationssprache wird über speziell formatierte Kommentare (*Annotationen*) in den Quellcode eingebettet. Annotationen werden mit der Zeichenfolge /*@ eingeleitet und durch @*/ beendet; der dazwischenstehende Text ist eine Sequenz von Annotationselementen. Jedes Element wird durch ein mit dem Zeichen @ beginnendes Schlüsselwort eingeleitet. Annotationen können an mehreren Stellen im Quellcode auftauchen, etwa als Theorie-Importe auf Dateiebene, als Schnittstellenspezifikation von Funktionen vor deren Deklaration, oder innerhalb von Funktionsblöcken als Invarianten vor Schleifen. Welche Annotationselemente an welcher Stelle erlaubt bzw. gefordert sind, wird in den Abschnitten 3.2 bis 3.7 definiert.

Anhang A beschreibt die zum Einlesen von Annotationen nötigen Modifikationen und Erweiterungen der Sprachgrammatik.

Die folgende Aufzählung gibt einen kurzen Überblick über alle Annotationselemente:

- @requires leitet die Spezifikation der Vorbedingung einer Funktion ein. Eine Vorbedingung ist ein *Zustandsprädikat*, das bei jedem Aufruf der Funktion erfüllt sein muss. ⇒ 3.2

- @ensures leitet die Spezifikation der Nachbedingung einer Funktion ein. Eine Nachbedingung ist eine *Zustandsrelation*, die den Programmzustand nach Ausführung der vorliegenden Funktion beschreibt. ⇒ 3.2

- @invariant dient der Spezifikation von Invarianten von Schleifen im Körper von Funktionen. Eine Invariante ist ein *Zustandsprädikat*, dessen Erfülltheit vor jeder Ausführung des Schleifenkörpers behauptet wird. Invarianten sind nicht Bestandteil der Schnittstellenspezifikation einer Funktion und dienen vor allem der Unterstützung von manuellen Beweisen in Isabelle. Die Angabe einer Schleifeninvariante ermöglicht nicht den Korrektheitsbeweis einer fehlerhaften Funktion. ⇒ 3.3

- @variant Ganzzahliger *Programmausdruck* (Typ int), der unter Annahme der zugehörigen Invariante vor jeder Iteration nicht-negativ ist und bei jeder Schleifeniteration verringert wird. Wird für den Beweis der Termination der Schleife benötigt. ⇒ 3.4

- @modifies (oder synonym @assigns) leitet eine durch Kommas getrennte Liste von außerhalb einer Funktion oder einer Schleife sichtbaren Speicherstellen ein (*Modifikationsliste*), die bei Ausführung der Funktion oder Iteration der Schleife modifiziert werden können. Diese Liste hat notwendigerweise approximativen Charakter, stellt aber in jedem Fall eine Obermenge aller möglicherweise modifizierten Speicherstellen dar. Der Nachweis der Korrektheit dieser Annotation ist Bestandteil des formalen Beweises. ⇒ 3.5

- @theory spezifiziert Abhängigkeiten der Spezifikation von Isabelle-Theorien, in denen die in der vorliegenden Datei verwendeten Isabelle-Prädikate und -Funktionen definiert sind. ⇒ 3.6

- ^function, $function, $!function dienen der Typdeklaration von Isabelle-Funktionen, die in Spezifikationsausdrücken verwendet werden. Die Deklarationssyntax folgt der für Funktionsdeklarationen von gewöhnlichen C-Deklarationen. ⇒ 3.7

- :: abbreviation dient der *Benennung* von wiederholt benötigten Spezifikationsausdrücken. Die Syntax zur Einführung einer derartigen Abkürzung folgt wie bei $function-Deklarationen der Syntax für Funktionsdeklarationen. Mittels eines gewissen Missbrauchs der C-Syntax wird der definierende Ausdruck hierbei als Initialisierungswert der Deklaration angegeben. 3.8

- @define dient der Einführung von Bezeichnern für numerische Konstanten (von Integer-Typ oder Fließkomma-Typ). Einerseits kann so eine Verbindung zu in Isabelle existierenden (insb. irrationalen) Konstanten hergestellt werden, die sich nicht als Literale notieren lassen. Andererseits dient @define als Ersatz für symbolische Konstanten, die in der zu kompilierenden Übersetzungseinheit als Präprozessor-Makros definiert sind. ⇒ 3.9

- @assert – *nicht verwendet* – Vorbehalten für künftige Erweiterungen.

3.1 Zustandsprädikate und Zustandsrelationen

Dieser Abschnitt führt die Spezifikationssprache für Zustandsprädikate und -relationen ein. Diese werden unter dem Begriff *Spezifikationsausdrücke* zusammengefasst. Weiterhin wird eine abstrakte Syntax für Spezifikationsausdrücke definiert. Anhang A enthält die konkrete Syntax der Spezifikationssprache, wie sie z. B. von einem Parsergenerator verwendet werden kann.

Zustandsprädikate und Zustandsrelationen sind das fundamentale Sprachmittel zur Formulierung von Schnittstellenspezifikationen von Funktionen. Dahingegen dienen Modifikationslisten vor allem der Begrenzung von Effekten von Funktionen, wodurch eine knappe und lokale Formulierung der Vor-/Nachbedingungen ermöglicht wird.

Ein Zustandsprädikat ist ein boolescher Ausdruck über den Programmzustand an einer Stelle der Programmausführung (*Ausführungspunkt*). Dieses kann, im Falle der Verwendung in Vor-/Nachbedingungen zusätzlich von den Argumenten der Funktion sowie ihrem Rückgabewert abhängig sein. Eine Zustandsrelation ist ein boolescher Ausdruck über die Programmzustände an zwei unterschiedlichen Ausführungspunkten, ebenfalls auf analoge Weise parametrisiert. Mögliche Ausführungspunkte, die durch Zustandsprädikate/-relationen spezifiziert werden können, sind:

- Vor Ausführung einer Funktion: derjenige Programmzustand, der direkt vor dem Aufruf der Funktion besteht. Es sind keine lokalen Variablen der Funktion sichtbar, das Zustandsprädikat ist parametrisiert über die Funktionsargumente, die über die Namen der formalen Funktionsparameter referenziert werden können.

- Nach Ausführung einer Funktion: derjenige Programmzustand, der nach Ausführung der Funktion besteht. Lokale Variablen der Funktion wurden bereits deallociert und sind nicht

mehr sichtbar; der Rückgabewert der Funktion ist an die spezielle Variable \result gebunden; Funktionsargumente können über die Namen der formalen Funktionsparameter referenziert werden.

- Vor jeder Iteration einer Schleife, nach Auswertung der Schleifenbedingung: jeder Programmzustand, der besteht, nachdem die Scheifenbedingung b in der normierten Form der Schleife (als "while (b) {s}", siehe 2.3.5) ausgewertet wurde. An dieser Stelle sind alle Parameter und lokalen Variablen der Funktion sichtbar und können referenziert werden.

3.1.1 Einbettung von Isabelle in die Spezifikation

Programmfunktionen und andere Annotationselemente werden in Isabelle-Terme der entsprechenden Datentypen (*Expr, Stmt, Decl*, etc.) übersetzt. Die Semantikdefinition erfolgt hierbei innerhalb von Isabelle. Aus technischen Gründen werden dahingegen Zustandsprädikate und -relationen direkt vom Frontend in Isabelle-Prädikate übersetzt, d. h. ihre semantische Interpretation wird vom Frontend generiert. Diese Entwurfsentscheidung liegt darin begründet, dass es so ermöglicht wird, Isabelle-Ausdrücke direkt in die Spezifikationssprache einzubetten. Dadurch erhält die Spezifikationssprache auf einfache Weise eine angemessene Ausdrucksmächtigkeit. Wir unterscheiden im Folgenden die *äußere Spezifikationssprache* und in diese *eingebettete Isabelle-Terme*.

In modellbasierten Spezifikationsansätzen wie Z (Z-Spec, 2002) oder anderen vom Design-by-Contract (Meyer, 1992) inspirierten Sprachen wie JML (Burdy u. a., 2005) enthält die Spezifikationssprache selbst Abstraktionsmechanismen und Datentypen für Konzepte wie Mengen, Stacks, Listen usw. Im Gegensatz dazu wird hier der Ansatz verfolgt, eine grundsätzlich primitive Spezifikationssprache zu verwenden, mittels derer vornehmlich technische Spezifikationen wie Arraygrößen, gültige Zeiger oder arithmetische Beziehungen zwischen numerischen Variablen angegeben werden. Spezifikationen, die auf der Interpretation der Daten in abstrakteren Konzepten basieren, werden direkt in der Logik und den bereits vorhandenen Theorien von Isabelle notiert und über einen Quoting-Mechanismus in die Spezifikation eingefügt. Siehe hierzu Abschnitt 3.1.3; siehe auch Abschnitt 4 für eine Definition der Übersetzung von Annotationselementen.

3.1.2 Erweiterung des Typsystems

Die Annotationselemente @requires, @ensures, @invariant und @variant sind getypte Ausdrücke. Das hierfür verwendete Typsystem ist eine vereinfachte Form des Typsystems von C. Folgende Modifikationen und Erweiterungen wurden vorgenommen:

- Der Typ _Bool ist der Typ für boolesche Ausdrücke. (Im Programmtext haben diese Ausdrücke gemäß C-Standard den Typen int.) Alle Vergleichsoperatoren (<, >, <=, >=, ==, !=) liefern Ausdrücke vom Typ _Bool. Alle booleschen Operatoren erwarten Argumente vom Typ _Bool und liefern Ausdrücke desselben Typs. Siehe 3.1.3 für weitere Ausdrücke dieses Typs.

- Typqualifikatoren (const, volatile) entfallen
- Ausdrücke des Typs void kommen nicht vor.
- Der Typ _Any bezeichnet Teilausdrücke eines Typs, der in C keine Entsprechung findet. Derartige Ausdrücke entstehen durch die Einbettung von Isabelle-Termen über die Quoting-Mechanismen. Da das Typsystem von Isabelle inkompatibel mit dem Typsystem von C ist, haben alle Isabelle-Terme innerhalb einer Spezifikation diesen Typ, sofern nicht durch die in 3.1.7 angegebenen Typ-Regeln ein spezifischerer Typ abgeleitet werden kann.

3.1.3 Sprachelemente

Zustandsprädikate und -relationen werden als Ausdrücke über die Programmvariablen formuliert. Die Syntax hierfür ist eine Erweiterung der C-Syntax für zuweisungsfreie Ausdrücke (Grammatikregel *conditional-expression*, s. (CStandard, 1999, §A.2.1)).
Die folgenden zusätzlichen Sprachelemente können bei der Formulierung von Zustandsprädikaten/-relationen verwendet werden. Sie sind zu Referenzzwecken in Tabelle 1 zusammengefasst.

Äußere Spezifikationssprache

1. Die Konstante *NULL* für den Wert des Nullzeigers. Sie kann überall verwendet werden, wo ein Ausdruck vom Typ "Zeiger auf *T*" erwartet wird.
2. Die Wahrheitswerte \ *true* und \ *false* . Der Typ dieser Variablen ist _Bool.
3. [Nur @ensures] Der Rückgabewert einer Funktion in der Variablen \ *result* . Der Typ dieser Variablen ist der Typ des Rückgabewerts der Funktion.
4. Die booleschen Operatoren "−−>" und "<−>" für logische Implikation und Äquivalenz. Ihre Argumente müssen vom Typ _Bool sein. Der Ausdruck hat den Typen _Bool.
5. [Nur @ensures] Der unäre Operator \old zur Auswertung eines Ausdrucks im Zustand, in dem die Vorbedingung ausgewertet wird. Der Ausdruck muss ein L-Wert sein. Der Typ dieses Ausdrucks entspricht dem Typen des Operanden.
6. Der unäre Operator \valid. Sein Argument muss ein L-Wert vom Typ $\mathsf{T} *$ (*Zeiger auf T*) sein. Ein Ausdruck der Form "\valid p" wertet genau dann zu *True* aus, wenn der Wert des Zeigers p eine gültige Adresse für den Typen T enthält, sonst zu *False*. Eine Adresse ist gültig für T, wenn sich an dieser Stelle ein Objekt dieses Typs befindet, und damit eine Dereferenzierung des Zeigers p erlaubt ist. Der Operator liefert einen Ausdruck vom Typ _Bool.
7. \array(a, n) ist ein zweistelliges Prädikat (Ausdruck vom Typ _Bool). Das Argument a muss ein L-Wert vom Typ *Zeiger auf T* oder *Array von T* sein, das zweite Argument n ein seiteneffektfreier Ausdruck von ganzzahligem Typ. Der Ausdruck wertet genau dann zu

True aus, wenn der Wert von a die Adresse eines Elements eines Arrays vom Typ *Array von T* ist, dessen Größe mindestens derart ist, dass über Zugriffe p[0] bis p[n−1] Elemente dieses Arrays erreicht werden, und also hierdurch kein Zugriff über die Arraygrenzen hinaus erfolgt.

8. \separated(a, m, b, n) ist ein vierstelliges Prädikat (Typ _Bool) über zwei Zeiger a, b (die L-Werte sein müssen) und zwei seiteneffektfreie Ausdrücke ganzzahligen Typs m, n. Die Typen von a und b müssen nicht zwingend kompatibel sein. Das Prädikat wertet genau dann zu *True* aus, wenn die Speicherbereiche &a[0] ...&a[m] und &b[0] ...&b[n] sich nicht überlappen. Es wird dabei nicht als Überlappung angesehen, wenn &a[m] == &b[0] (oder analog &b[n] == &a[0]) gilt.

Beispiel 2: Unterschiedliche Arrays sind immer separiert: Im Kontext

```
int as[10];
double ds[3];
```

gelten

(a) \separated(as, 10, ds, 3)

(b) \separated(&as[1], 3, &as[4], 6)

Es gelten hingegen nicht

(a) \separated(ds, 3, ds, 3)

(b) \separated(ds, 2, &ds[1], 2) (*Überlappung um 1 Element*)

(c) \separated(as, 20, ds, 3) (*Offset 20 zu groß*)

9. [Nur @invariant] Der Infixoperator @ erlaubt die Referenzierung des Werts einer Variablen an einem mit einem Label versehenen Ausführungspunkt. Der linke Operand muss ein L-Wert sein, der rechte Operand der Name des Labels. Dieses Konstrukt wird äußerst selten benötigt. Das referenzierte Label muss an der Spezifikationsstelle sichtbar sein, d. h. vereinfacht: das Label muss sich auf ein Statement beziehen, das auf jedem Pfad vom Funktionsbeginn zur Spezifikationsstelle im Kontrollflussgraphen durchlaufen wird. Abbildung 6 gibt ein Beispiel. Der Typ dieses Ausdrucks entspricht dem Typen des linken Operanden.

10. Quantifizierung durch "\forall <*parameter-list*>; pred" und "\exists <*parameter-list*>; pred". Hierbei ist pred ein Ausdruck (Typ _Bool), in dem die in <*parameter-list*> deklarierten Variablen vorkommen können. Sie sind die durch den Quantor *gebundenen* Variablen. <*parameter-list*> folgt hierbei der Syntax der Deklaration von Funktionsparametern. Es kann auf diese Weise nur über skalare Typen quantifiziert werden. [3] Die wichtigste Verwendung dieser Quantoren ist die zur Quantifizierung über Arrayindizes.

[3] Quantifizierungen über andere Datentypen müssen innerhalb von eingebetteten Isabelle-Termen vorgenommen werden.

Beispiel 3: (a) spezifiziert, dass das Array a von 0 bis len − 1 aufsteigend sortiert ist. (b) besagt, dass im Array a ein Element pivot vorhanden ist.

(a)
```
\forall int i, int j; 0 <= i & i <= j & j < len -->
        a[i] <= a[j]
```
(b)
```
\exists int i; 0 <= i & i < len & a[i] == pivot
```

Der Typ dieser Ausdrücke ist `_Bool`.

11. Einbettung von Isabelle-Termen (*Quoting*). Es werden drei Arten von Quoting unterscheiden. Siehe hierzu 3.1.4.

3.1.4 Quoting von Isabelle-Termen

1. Verwendung von in Isabelle definierten Funktionen und Variablen, die ausschließlich von skalaren (Programm-)*Werten* abhängen (und nicht vom gesamten Programmzustand). Die Typen der Parameter derart referenzierter Funktionen müssen den zur Interpretation der Typen der konkreten Argumente verwendeten entsprechen. Der Rückgabetyp dieser Funktionen im Typsystem von Spezifikationsausdrücken wird durch seine Deklaration bestimmt; siehe 3.7. Aufseiten von Isabelle kann die Funktion einen beliebigen Rückgabetyp besitzen. In allen Fällen, in denen dieser Typ keine Entsprechung im Typsystem der Spezifikationsausdrücke hat, wird `_Any` verwendet.

Beispiel 4: Eine Isabelle-Funktion

$$foo :: DomInt \Rightarrow DomDouble \Rightarrow DomInt \Rightarrow \tau$$

deren Argumenttypen die Interpretationen von `int` bzw. `double` sind, und die einen Rückgabewert eines in C nicht existierenden Typen τ liefert, kann in Spezifikationsausdrücken mit der (über `$function` angegebenen) Typdeklaration `_Any foo(int a, double d, int c)` verwendet werden. Der Aufruf erfolgt als

```
$foo(a, b, c)
```

Dieser Mechanismus erlaubt auch einen Rückbezug auf in Isabelle definierte bzw. gebundene Konstanten und Variablen. In diesem Fall werden keine Klammern benötigt, so dass die Variable *x* als `$x` referenziert werden kann. Ist der Typ von `$x` nicht durch eine entsprechende `$function` Deklaration festgelegt, so wird `int` angenommen. [4]

2. Verwendung von in Isabelle definierten Funktionen, die zusätzlich zur vorigen Variante als erstes Argument den Programmzustand (Isabelle-Typ *State*) erwarten. Dieser Mechanismus findet vornehmlich Anwendung zur Referenzierung von Repräsentationsfunktionen,

[4] Der Grund hierfür ist, dass gebundene Variablen —die nicht deklariert werden *können*— in praxi ausschließlich als Feldindizes auftreten.

die nicht lediglich auf skalaren Werten arbeiten (können), z. B. einer Funktion zur Repräsentation eines Arrays von Integers als Menge, etwa mit der Signatur

$$arrayset :: State \Rightarrow DomRef \Rightarrow DomInt \Rightarrow int\ set$$

die dann in der Form

```
$!arrayset(a, len)
```

referenziert werden kann. In Spezifikationsausdrücken hätte der Ausdruck den Typ `_Any`, da Mengen im C-Typsystem nicht existieren. Siehe bezüglich Repräsentationsfunktionen auch 3.1.5.

3. Durch *Quoting* von Isabelle-Termen können beliebige Isabelle-Terme vom Typ *bool* eingebunden werden (die dann im Typsystem der Spezifikationsausdrücke den Typ `_Bool` annehmen). Ein derartiger Isabelle-Term *Q* wird eingebunden über den Spezifikationsausdruck

   ```
   ${ Q }
   ```

   Der Term *Q* kann wiederum *Antiquotations* enthalten, die einen Rückbezug auf Programmausdrücke (in der Syntax der äußeren Spezifikationssprache) ermöglichen. Dieses Verfahren ist in 3.1.5 beschrieben.

Sichtbare Variablen Innerhalb von eingebetteten Isabelle-Termen sind alle Definitionen sichtbar, die über `@import`-Anweisungen der sie enthaltenden Theorien eingeführt wurden, sowie all solche, die in der Basistheorie *SAMS* sichtbar sind. Darüberhinaus sind die folgenden Variablen innerhalb von Quotations gebunden und damit sichtbar:

Name	Token	*Beschreibung*	*Typ in Isabelle*
Γ	`\<Gamma>`	Variablenumgebung	*Env*
Σ	`\<Sigma>`	Programmzustand	*State*
Σ'	`\<Sigma>'`	Programmvorzustand	*State*
Result	`Result`	Resultatwert der Funktion	*Dom{Int,Double,Ref}*

Da diese Variablen auch in der Interpretation von Antiquotations verwendet werden, dürfen sie nicht redefiniert werden. Sie sollten innerhalb der Isabelle-Terme als Schlüsselworte aufgefasst werden. Bis auf *Result*, das auf Isabelle-Seite dem Schlüsselwort $\backslash$*result* der äußeren Spezifikationssprache entspricht, sollten diese Variablen nicht explizit verwendet werden, da sich ihre Tokendarstellung in späteren Versionen ändern kann. (Auch sollte statt *Result* vorzugsweise `'{\ result }` verwendet werden, s. 3.1.5.)

3.1.5 Eingebettete Isabelle-Terme: Antiquotations

1. *Allgemeines Antiquoting*: Ein eingebetteter Isabelle-Term *Q* kann mittels des Antiquotation-Mechanismus wiederum Spezifikationsausdrücke enthalten. Einfache Variablen des

Spezifikationsausdrucks werden durch Voranstellen eines Backquote-Zeichens referenziert, komplexe Ausdrücke werden zudem in geschweifte Klammern eingefasst:

```
`ident
`{ expr }
```

Die Verschachtelungstiefe von Quotes/Antiquotes ist theoretisch unbegrenzt; die Verschachtelung sollte jedoch eine Tiefe von drei (Quotes in Antiquotes in Quotes) nicht überschreiten.

Beispiel 5:

```
/*@
  @ensures
    ${ ALL i p. Q1 p --> Q2 i `len -->
         Q3 p `{a[$i] + d} }
  @*/
void quz(struct point *a, int len, int d);
```

2. *Antiquotations für Repräsentationsfunktionen*: Da sich die Modellierung von C-Werten in Isabelle auf skalare Werte beschränkt und insbesondere für strukturierte Werte (**struct**) keine generische Representation vorhanden ist, kann über obigen Antiquoting-Mechanismus kein Wert von Strukturtyp in einen Isabelle-Term eingebunden werden, etwa der Form `{a[0]}: es gäbe für diesen Term schlicht keine semantische Interpretation. Repräsentationsfunktionen umgehen dieses Problem, indem sie die *Adresse* der jeweiligen Struktur zusammen mit dem Programmzustand als Argumente erwarten. So kann eine beliebige Repräsentation der Struktur aus dem Speicher gelesen werden. Dieses Paradigma wird durch eine spezielle Syntax unterstützt. Eine Isabelle-Funktion mit der Signatur

$$\textit{point-rep} :: \textit{State} \Rightarrow \textit{Loc} \Rightarrow \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \tau$$

wobei die *Loc* der Isabelle-Typ für Speicheradressen, σ_i Interpretationen von skalaren C-Typen und τ beliebig sind, kann innerhalb einer Quotation über die spezielle Antiquotation-Syntax

```
^point_rep{s, a1, ..., aN}
```

referenziert werden. Die exakte Auswertung ist in Abschnitt 4 definiert; intuitiv erfolgt diese, indem an *point-rep* als erstes Argument der aktuelle Programmzustand, als zweites Argument die *Adresse* des Objekts `s` und als weitere Argumente die *Werte* der Ausdrücke `a1` bis `aN` übergeben werden. Ein Antiquoting der Form `^f{s, ...}` ist nahezu äquivalent zu `{ $!f(&s, ...) }`, mit den zwei Unterschieden, dass jener noch einen geringen Anspruch auf Lesbarkeit erheben darf, und zusätzlich in ihm das erste Argument als konkrete Adresse (*Loc*) ausgewertet wird, während es in diesem als optionale Adresse (*DomRef*) interpretiert würde, also auch der Nullzeiger sein könnte.

```
int f(int a, int b) {
  int i = 0;
  ...
lab0:
  i = g(a, b);
  ...
  /*@invariant a * b - z = i@lab0 && ..
    @variant ..
    @*/
  while (..) { ... }
  ...
}
```

Abbildung 6: Beispiel für die Verwendung des Label-Operators @

3. *Auswertung im Vorzustand*: Eine Abkürzung erleichtert den Verweis auf Werte von Programmausdrücken im Vorzustand innerhalb von Nachbedingungen. Die folgenden beiden Ausdrücke sind vollständig äquivalent.

```
@{ expr }

`{ \old( expr ) }
```

3.1.6 Abstrakte Syntax

Innerhalb von Spezifikationen werden verschärfte Anforderungen an Ausdrücke gestellt, die der C-Standard nicht vorsieht. Es ist eindeutiger und verständlicher, eine neue Definition der erlaubten Ausdrücke in Zustandsprädikaten und -relationen zu geben, als durch Auflistung von Einschränkungen die C-Syntax für Ausdrücke auf das gewünschte Maß zurechtzustutzen. Dennoch sei an dieser Stelle kurz auf die wichtigsten Einschränkungen hingewiesen:

1. Verbot von Zustandsmodifikationen. Der Zuweisungsoperator $=$ sowie die Dekrement-/Inkrementoperatoren $--$ und $++$ dürfen nicht verwendet werden.
2. Funktionsaufrufe sind ebenfalls nicht erlaubt. Die einzigen Funktionen, die in Spezifikationen verwendet werden können, sind über den Quoting-Mechanismus referenzierte Isabelle-Funktionen.
3. Keine Bit-Operationen
4. Keine bedingten Ausdrücke (“`e ? a : b`”)
5. Keine expliziten Typkonversionen (cast)

Konstanten	
true	Wahrheitswert
\ *false*	Wahrheitswert
NULL	NULL-Zeiger
\ *result*	Rückgabewert der spezifizierten Funktion
Boolesche Operatoren	
`-->`	Logische Implikation
`<->`	Logische Äquivalenz
Spezielle Funktionen und Operatoren	
`\old e`	Wert des Ausdrucks e im Zustand vor Ausführung der Funktion
`\valid p`	L-Wert p vom Typ T $*$ enthält gültige Adresse auf Objekt vom Typ T
`\array(a, n)`	L-Wert a enthält Adresse eines Arrays der Größe m $\geq$ n
`\separated(a, m, b, n)`	Speicherbereiche ab a mit Breite m $*$ `sizeof(a[0])` und ab b mit Breite n $*$ `sizeof(b[0])` überlappen sich nicht
`lv@L`	Wert des L-Wert lv am Label L
Quantoren	
`\forall T v; P`	Allquantor (z. B. `\forall int x; x < x + 1`) zur Quantifizierung über arithmetische Werte (**int**, **double**)
`\exists T v; P`	Existenzquantor (z. B. `\exists int x, y; x + 1 == y`)
Einbettung von Isabelle-Termen	
`$f(x1 ,..., xN)`	Verweis auf in Isabelle definierte N-stellige Funktion f
`$!f(x1 ,..., xN)`	Verweis auf in Isabelle definierte (N+1)-stellige zustandsbehaftete Funktion f
`${` *Isabelle-Term* `}`	Erlaubt die Einbettung von beliebigen Isabelle-Termen in die Spezifikation (*Quoting*)
`‘{` *Ausdruck* `}`	Erlaubt den Rückbezug auf Spezifikationsausdrücke (insbesondere L-Werte) in eingebettetem Isabelle-Text (*Antiquoting*). Kurzform für Variablen: ‘var
`@{` *Ausdruck* `}`	Kurzform für ‘{\old(*Ausdruck*)}
`^f{lv,x1 ,..., xN}`	Antiquoting von Funktionsargumenten der Isabelle-Funktion f. Das erste Argument lv ist ein L-Wert, dessen Speicheradresse (*nicht* sein Wert) ausgewertet wird. Eingeführt für Repräsentationsfunktionen.

Tabelle 1: Zusätzliche Operatoren in Annotationselementen

6. Bitoperatoren (|, &, ~) werden nur für boolesche Ausdrücke verwendet und entsprechen dort den logischen Operatoren (||, &&, !)

7. Keine Verwendung von Präprozessor-Makros

Abbildung 7 zeigt die abstrakte Syntax für Ausdrücke in Zustandsprädikaten und -relationen. *ident* steht dabei für die syntaktische Kategorie der Bezeichner.

3.1.7 Typ-Regeln

Eine Formalisierung des Typsystems ist derzeit nicht vorgesehen. Die Typkorrektheit der übersetzten Spezifikationsausdrücke wird durch den Typ-Check in Isabelle sichergestellt.

3.2 Vor- und Nachbedingungen

Syntax:	@requires P @ensures Q wobei P ein Zustandsprädikat und Q eine Zustandsrelation ist

1. Stehen unmittelbar vor der Funktionsdeklaration oder -definition, auf die sie sich beziehen.

2. Sichtbare Variablen sind die bis zur Stelle des Auftretens deklarierten globalen Variablen und alle Funktionsparameter.

3. Als Zustandsrelation können @ensures-Klauseln die Sprachelemente \old und \ *result* enthalten.

4. Es können mehrere @requires- und @ensures-Klauseln in derselben Annotation angegeben werden. Die Konjunktion aller jeweiligen Klauseln stellt dann die gesamte Vor- bzw. Nachbedingung dar.

5. Die Vor- und Nachbedingung kann entweder bei der Deklaration oder bei der Definition einer Funktion angegeben werden.

6. Eine fehlende Vor- oder Nachbedigungung wird als *True* interpretiert.

Semantik:	Eine Funktion f erfüllt eine Vorbedingung P und eine Nachbedingung Q, wenn die Ausführung von f in jedem Zustand σ, der das Zustandsprädikat P erfüllt, *terminiert* und in einen Zustand σ' führt, der zusammen mit σ die Zustandsrelation Q erfüllt.

```
lval ::= ident
       | lval.ident
       | *lval
       | lval[expr]
       | &lval

expr ::= number
       | lval
       | expr1 OP expr2      (* OP ist +, -, *, /, &&, ||, -->,
                                <->, <, <=, ==, !=, >=, > *)
       | - expr | ! expr | ~ expr
       | \forall bindexpr; expr
       | \exists bindexpr; expr
       | \old lval
       | lval @ ident
       | \valid lval
       | \array(lval, expr)
       | \separated(lval, expr1, lval, expr2)
       | $ident
       | $ident(expr1, ..., exprN)
       | $!ident(expr1, ..., exprN)
       | ${ isaterm1 ... isatermN }

bindexpr ::= TYP_1 ident1, ..., TYP_n identN
         (* TYP_i ist unqualifizierter skalarer Typ *)

isaterm ::= raw-isabelle-text
          | `ident
          | `{expr}
          | @{ expr }
          | ^ident{expr1, ..., exprN}
```

Abbildung 7: Abstrakte Syntax von Zustandsprädikaten und Zustandsrelationen

3.3 Invarianten

Syntax:	@invariant *I*, wobei *I* ein Zustandsprädikat ist

1. Steht unmittelbar vor der Schleife (while / do−while / for), auf die sie sich bezieht.
2. *I* kann über den Operator @ auf Werte von L-Werten verweisen, die zu einem Ausführungspunkt bestanden, der vor dem Auftreten von *I* liegt.
3. Invarianten erleichtern die Herleitung der Verifikationsbedingungen, haben aber keinen normativen Charakter. Der Beweis ihrer Gültigkeit ist lediglich ein Hilfslemma für den Beweis der Gesamtkorrektheit.

Semantik:	Eine Schleife erfüllt ihre Invariante *I*, wenn die Auswertung von *I* in jedem Zustand *True* ergibt, der nach der Ausführung der Schleifenbedingung oder nach der Ausführung des Schleifenrumpfs erreichbar ist. Dies schließt also alle Zustände ein, die während der Iteration der Schleife (jeweils nach Ausführung der Schleifenbedingung) erreichbar sind.

3.4 Varianten

Syntax:	@variant *V*

1. Steht vor der Schleife (while / do−while / for), auf die sie sich bezieht.
2. *V* ist ein seiteneffektfreier Programmausdruck von ganzzahligem Typ. Der Wert dieses Ausdrucks muss bei jedem Schleifendurchlauf strikt abnehmen und in jedem Fall nicht-negativ sein.
3. Ist ein Hinweis an die Routine zur Berechnung der Verifikationsbedingungen, wie die Termination der Schleife bewiesen werden kann.
4. Die typische Variante für eine for-Schleife, deren Zähler i die Werte von 0 bis k durchläuft, ist die Differenz k − i; zu beachten ist hierbei, dass durch geeignete Wahl der Invariante sichergestellt werden kann, dass diese Differenz nicht-negativ ist:

Beispiel 6:

```
/*@
  @invariant 0 <= i && i <= k && ...
  @variant k - i
  @*/
for (i = 0; i < k; ++i) { ... }
```

Semantik: Eine Schleife erfüllt eine Variante V unter Annahme einer Invariante I, wenn für jeden Zustand, der I erfüllt, die Auswertung von V vor dem Eintritt in den Schleifenrumpf positiv (> 0) ist und einen größeren Wert ergibt als die Auswertung in dem Zustand nach dem Durchlaufen des Schleifenrumpfes.

3.5 Modifikationslisten

Syntax:

	@modifies	$m_1, m_2, \ldots, m_n$	(m_i Änderungsdeskriptoren)
		$m_i ::= n^i$	
		$\mid n^i \mid\!\!-\!\!> t^1 \mid \ldots \mid t^m$	(n_i,t^i M-Werte)
	@modifies	\nothing	

1. Kann in Funktionsspezifikationen und Schleifenannotationen verwendet werden.

2. Liefert eine konservative Überapproximation der möglichen Effekte der Ausführung des Programmcodes, auf den sich die Annotation bezieht. Als *Effekt* bezeichnet man in diesem Zusammenhang die Modifikation einer Speicherstelle.

3. Die Angabe der Effekte erfolgt in Form von *Änderungsdeskriptoren*. Es werden beschränkte und unbeschränkte Änderungsdeskriptoren unterschieden. In Anlehnung an ihre inhaltliche und syntaktische Ähnlichkeit zu L-Werten werden unbeschränkte Änderungsdeskriptoren im Folgenden als *M-Werte* bezeichnet. Syntaktisch stellen M-Werte eine Erweiterung der L-Werte aus Abschnitt 2.2.2 dar, indem für sie zusätzlich die Muster `L[i:j]` (Intervallangabe) und `L[*]` (Erfassung des gesamten Arrays, d. h. all seiner Elemente) zugelassen werden. Diese Muster heißen *Bereichsangaben*. `i` und `j` sind hierbei einfache Ausdrücke ganzzahligen Typs, die über ganzzahlige Konstanten/Variablen und den arithmetischen Operatoren $+$, $-$, $*$, `/` und `%` gebildet werden.

   M-Werte bezeichnen wie L-Werte Objekte im Speicher. Jedoch können durch einen einzigen M-Wert vermittels der Verwendung von Bereichsangaben mehrere Objekte bezeichnet werden. Der Ausdruck `a[*]` bezeichnet alle Elemente eines Arrays. Zugriffe (sowohl Struktur- als auch Feldzugriffe), die eine Bereichsangabe fortsetzen, wie etwa in `a[*].x`, beziehen sich automatisch auf alle durch die Bereichsangabe bezeichneten Objekte. Im voranstehenden Beispiel werden also alle `x`-Strukturelemente in einem Array `a` von Strukturen beschrieben.

   Präzisere Angaben können durch eine Intervallangabe vorgenommen werden. Eine Intervallangabe `a[i:j]` bezeichnet die Elemente `i` bis `j`-1 des Arrays `a`, d. h. Bereichsangaben umfassen den rechten Index *nicht*. Die abkürzende Schreibweise `a[:i]` steht für `a[0:i]`.

   Da strukturierte Werte nicht unterstützt werden (s. 2.2.3), bezeichnen L-Werte gemeinhin Objekte skalaren Typs. Diese Einschränkung gilt für M-Werte nicht, d. h. ein M-Wert strukturierten Typs bezeichnet automatisch all seine Komponenten.

Beispiel 7: Die folgende Funktion ist spezifiziert mit dem Effekt, in jedem Arrayelement von p bis zum Index len−1 *beide* Komponenten x und y zu verändern, sowie q[2].a.x und q[3].a.x.

```
struct s1 { int x; float y; };
struct s2 { struct s1 a; };
/*@
  @modifies p[:len], q[2:4].a.x
  @*/
int f(struct s1 * p, int len, struct s2 * q);
```

4. Für die formale Verifikationsumgebung wären unbeschränkte Änderungsdeskriptoren völlig ausreichend, da die Spezifikation der Art der Änderung in den Vor-/Nachbedingungen angegeben werden kann. Für eine statische Prüfung oder Berechnung der Änderungsdeskriptoren ist es jedoch hilfreich, gewisse Änderungen zu begrenzen, d. h. die möglichen zugewiesenen Werte anzugeben. Hierzu dienen beschränkte Änderungsdeskriptoren.

   Syntaktisch werden diese aus M-Werten gebildet, indem für den eigentlichen Änderungsdeskriptor eine Sequenz von M-Werten angegeben wird, welche die möglichen neuen Werte der geänderten Objekte bezeichnen. Zur Verdeutlichung ein Beispiel.

Beispiel 8:

```
int a;
int b[10];
/*@
  @modifies *pp |-> a | b[*], *q
  @*/
void f(int **pp, int *q);
```

   Die Funktion f ändert den durch ∗pp bezeichneten Zeiger so, dass er nach Ausführung der Funktion entweder die Adresse der globalen Variablen a oder aber die Adresse eines der Elemente des Arrays b enthält.

5. Beschränkte Änderungsdeskriptoren sind nur für M-Werte von Zeigertypen möglich. Die angegebenen neuen Ziele des Zeigers werden dabei ausgewertet, als stünden sie auf der linken Seite einer Zuweisung. Die Angabe p |−> a bedeutet also, dass p die *Adresse* von a enthalten wird (p == &a), und nicht dessen Wert (so dass p == a gölte).

6. Änderungslisten für Funktionen werden grundsätzlich im Vorzustand ausgewertet.

7. Änderungslisten für Schleifen werden grundsätzlich im Zustand vor Eintritt in die Schleife, d. h. vor der ersten Auswertung der Schleifenbedingung ausgewertet.

8. \nothing steht für die leere Modifikationsliste. Mit der Annotation "@modifies \nothing" spezifizierter Programmcode ist seiteneffektfrei. Er kann somit lediglich lokalen Speicher verändern, dessen Lebendauer auf die Ausführung des spezifizierten Programmcodes begrenzt ist.

9. *Alias:* @assigns kann synomym zu @modifies verwendet werden.

10. Die formale Semantik von Änderungsdeskriptoren ist in Abschnitt 4 definiert.

Semantik: Eine Funktion f erfüllt die Modifikationsliste $m_1, \ldots, m_n$ wenn

(i) jedes Objekt, dessen Lebenszeit die Ausführungdauer von f überschreitet und dessen Wert sich durch die Ausführung von f ändern kann, durch einen M-Wert in der Modifikationsliste bezeichnet wird und

(ii) die (Zeiger-)Objekte, die durch beschränkte Änderungsdeskriptoren beschrieben werden, nach Ausführung von f entweder *NULL* sind oder die Adresse einer durch die entsprechenden M-Werte angegebenen Adressen enthält.

3.6 Theorieimporte

Syntax: @theory T, wobei T ein Theoriename ist

1. Steht nur auf der Ebene der Übersetzungseinheit.

2. Kann für unterschiedliche Theorieimporte beliebig oft auftreten, jedoch für jede Theorie nur genau einmal pro Übersetzungseinheit.

3. Das Vorkommen des Annotationselements "@theory T" führt dazu, dass die vom Frontend für die Repräsentation der Programmfunktionen und ihrer Spezifikationen erzeugte Isabelle-Theorie die Theorie T importiert. Hierdurch werden die in T eingeführten Definitionen und Theoreme in der erzeugten Theorie sichtbar und damit verwendbar gemacht. Typischerweise wird man in einer Übersetzungseinheit diejenigen Theorien importieren, die das für die Korrektheitsbeweise benötigte Domänenwissen beinhalten.

4. Um die Theorie zu importieren, in der gängige Repräsentationsfunktionen definiert sind, muss die folgende Annotation (an beliebiger Stelle) auf Dateiebene in die Übersetzungseinheit eingefügt werden:

Beispiel 9:

```
/*@
  @theory SAMSRepr
  @*/
```

3.7 Deklarationen von Isabelle-Funktionen

Syntax:	`@$function` *declaration* `@$!function` *declaration* `@^function` *declaration*	(*declaration* Funktionsdeklaration)

1. Steht nur auf der Ebene der Übersetzungseinheit.

2. Deklariert eine Isabelle-Funktion mit den dazugehörigen Parametertypen und dem Rückgabetyp. Diese Funktion kann dann innerhalb von Spezifikationsausdrücken verwendet werden. Die drei möglichen Deklarationsarten sind gegenseitig ausschließend, d. h. dass keine Isabelle-Funktion auf mehr als eine Art deklariert werden kann. (Die Art der Übersetzung bedingt, dass jede Deklarationsart unterschiedliche Funktionstypen erfordert.)

   - `$function`: die N-stellige deklarierte Funktion $f :: \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_N \Rightarrow \tau$ kann in der Form `$f(x1 ,..., xN)` referenziert werden.
   - `$!` — die deklarierte Funktion $f :: State \Rightarrow \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_N \Rightarrow \tau$ ist zustandsbehaftet und kann in der Form `$!f(x1 ,..., xN)` referenziert werden.
   - `^` — die deklarierte Funktion $f :: State \Rightarrow Loc \Rightarrow \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_N$ ist eine Repräsentationsfunktion und kann als Antiquoting der Form `^f{|v,x1 ,..., xN}` referenziert werden.
   - In allen Fällen gilt: $\sigma_i \in Dom\{Int, Double, Ref\}$

3. Folgt syntaktisch einer C Deklaration. Es muss sich bei *declaration* um eine einzelne Funktionsdeklaration handeln; Speicherklassen und Typqualifikatoren sind nicht erlaubt; alle Funktionsparameter müssen benannt werden.

4. *Beachte:* eine *declaration* wird mit einem `;` (Semikolon) beendet.

5. `_Any` und `_Bool` können als Typen in Deklarationen verwendet werden.

6. Argumente der Funktion sind von skalarem Typ. Einzige Ausnahme: das erste Argument einer Repräsentationsfunktion, die mittels `@^function` deklariert wird.

7. Im folgenden Beispiel wird jeweils ein Isabelle-Prädikat *dvd* (Teiler-Relation), *ist-zyklenfrei*, und eine Repräsentationsfunktion *array-als-liste* deklariert, die hernach in Spezifikationen verwendet werden können. Das Frontend nimmt an, dass diese Funktionen eine entsprechende Definition in Isabelle besitzen.

Beispiel 10:

```
/*@
  $function
    _Bool dvd(int a, int b);
  $!function
    _Bool ist_zyklenfrei(struct list *hd);
  ^function
    _Any array_als_liste(int hd, int len);
@*/
```

3.8 Parametrisierte Abkürzungen

Syntax: :: abbreviation *declaration*
(*declaration* Funktionsdeklaration mit Initialisierungsausdruck)

1. Steht nur auf der Ebene der Übersetzungseinheit.
2. Deklariert eine parametrisierte Abkürzung. Hierdurch wird ein Name für einen Spezifikationsausdruck eingeführt.
3. Der Spezifikationsausdruck kann über die Argumente der Abkürzung, d. h. über Variablen skalaren Typs, parametrisiert werden.
4. Es bestehen alle Anforderungen an die *declaration* wie in 3.7. Zusätzlich muss (syntaktisch betrachtet) ein *initializer* angegeben werden, welcher den definierenden Ausdruck des eingeführten Namens darstellt.
5. Wird verwendet, um wiederholt auftretende Spezifikationsausdrücke zu benennen und damit Spezifikationen lesbarer zu halten.
6. Eine K-stellige Abkürzung `A(p_1, .., p_K)` kann nach ihrer Deklaration in Vor-/Nachbedingungen und Invarianten verwendet werden. Sie wird dort notationell mit vorangestellten Doppel-Doppelpunkten, also als `::A(a_1, .., a_K)` referenziert.
7. Einer K-stelligen parametrisierten Abkürzung müssen immer alle K Argumente übergeben werden. Diese müssen (analog zu regulären Funktionen) typkompatible zu den Parametertypen sein.
8. Für jede Abkürzung wird vom Frontend eine entsprechende Definition in der ausgegebenen Isabelle-Theorie hinzugefügt. Eine Abkürzung wird vom Frontend also nicht einfach "aufgefaltet", sondern besteht als Konzept in Isabelle fort.
9. Ein Beispiel: Die folgende Spezifikation drückt aus, dass eine Ordnung zwischen zwei globalen Variablen besteht und eine andere durch einen Parameter der Abkürzung absolut beschränkt ist.

Beispiel 11:

```
double gx, gy;
int ga;
/* Weiterer Code ... */

/*@
::abbreviation
  _Bool konfig_OK(int a) =
    (gx <= gy && $abs(ga) <= a)
  @*/
```

10. Abkürzungen sollten den Typ `_Bool` haben —also zur Definition von Prädikaten verwendet werden— auch wenn alle anderen skalaren Typen erlaubt sind.

11. Der definierende Ausdruck (*initializer*) soll in Klammern eingefasst sein.

Semantik: Ein Spezifikationsausdruck `::A(x)` wertet im Zustand σ genau dann zu *True* aus, wenn der Definitionsterm `E` von `A`, in dem alle Vorkommen der formalen Parameter durch die Aktualparameterausdrücke ersetzt wurden, in σ zu *True* auswertet. Diese Definition ist in der offensichtlichen Weise auf mehrere/keine Argumente zu erweitern.

3.9 Definition symbolischer Konstanten

Syntax: `@define` *declaration*
(*declaration* Variablendeklaration von arithmetischem Typ)

1. Dient der Definition *symbolischer Konstanten*, die aufseiten der Übersetzung nach Isabelle durch das Frontend dieselbe Rolle spielen wie über `#define` definierte numerische Konstanten aufseiten der Übersetzung in ausführbaren Code durch einen Compiler.

2. Symbolische Konstanten (wie sie hier definiert werden) sind ein technisches Hilfsmittel, um die Problematik zu umgehen, dass Makros vom Präprozessor aufgefaltet werden, und damit deren symbolische Namen nicht mehr sichtbar sind. C kennt das Konzept von symbolischen Konstanten nicht: der `const` Typqualifikator hat keine hinreichend scharfe Semantik, um diese Rolle zu spielen. Daher sind Makros das einzige Mittel, um symbolische Namen für Konstanten einzuführen, die auch als Arraygrößen verwendet werden können.

   Um bei der Verifikation nicht auf die symbolischen Namen verzichten zu müssen, wurde das Konzept der symbolischen Konstanten eingeführt. Diese erlauben die Definition symbolischer Arraygrößen und numerischer Konstanten. Eine über `@define` deklarierte symbolische Konstante wird innerhalb des Frontends als (speziell markierte) Programmkonstante angesehen, und darf als solche in Programmausdrücken und insbesondere auch Arraygrößen verwendet werden.

3. Folgt syntaktisch der Variablendeklaration; Speicherklassen und Typqualifikatoren sind nicht erlaubt; nur arithmetische Typen dürfen deklariert werden.

4. Die *declaration* muss einen *initializer* enthalten, der den Wert der definierten symbolischen Konstanten festlegt.

5. Der *initializer* für symbolische Konstanten von Integer-Typ muss entweder eine einzelne über $ referenzierte Isabelle-Konstante oder ein einfacher Ausdruck ($+$, $-$, $*$, $/$, $\%$) über Integerkonstanten und bereits definierten symbolischen Konstanten sein.

6. Der *initializer* für symbolische Konstanten von Fließkomma-Typ muss entweder eine einzelne über $ referenzierte Isabelle-Konstante oder eine Fließkommankonstante sein.

7. Für jede symbolische Konstante, die im Code (im Ggs. zu Spezifikationen) verwendet wird, muss exakt eine Makrodefinition desselben Namens vorliegen. Die Definition der symbolischen Konstanten darf nur innerhalb eines bedingten Codeabschnitts liegen, in dem VERIFY_MODE definiert ist. Die korrespondierende Makrodefinition muss außerhalb dieses Blocks liegen und direkt auf diesen folgen, bzw. im #else-Zweig des bedingten Codeabschnitts liegen. Die intendierte Verwendung symbolischer Konstanten folgt dem folgenden Schema:

```
/*@ $function double pi; @*/

#ifdef VERIFY_MODE
/*@
  @define int X = 9;
  @define double M_PI = $pi;
  @*/
#else
#define X 9
#define M_PI 3.14159265358979323846
#endif
```

8. Für jede symbolische Konstante X wird vom Frontend in der erzeugten Isabelle-Theorie eine Definition *sams-X* eingefügt.

4 Interpretation der Annotationselemente in Isabelle

Für den formalen Nachweis, dass annotierte Funktionen ihre Spezifikation erfüllen, ist es erforderlich, ein Verständnis davon zu haben, wie Spezifikationsausdrücke in Isabelle-Terme übersetzt werden. Die nachfolgenden Abschnitte geben hierüber Aufschluss.

4.1 Übersetzung von Spezifikationsausdrücken

Spezifikationsausdrücke werden in Isabelle flach eingebettet, d. h. sie werden als Funktionen nach *bool* dargestellt und nicht über einen dedizierten Datentypen repräsentiert. Das Frontend übernimmt diese Übersetzung des abstrakten Syntaxbaums für Spezifikationsausdrücke in den entsprechenden Isabelle-Term. Die Funktionen im Frontend, die diese Übersetzungen vornehmen, liefern daher einfach Zeichenfolgen, die von Isabelle als Terme entsprechenden Typs geparst werden. Die von den Übersetzungsfunktionen für Vorbedingungen (*Pre*), Nachbedingungen (*Post*) und Invarianten (*Inv*) erzeugten Zeichenketten werden in Isabelle als Terme der folgenden Typen eingelesen:

$$Pre : Env \Rightarrow Val\ list \Rightarrow State \Rightarrow bool \tag{1}$$

$$Post : Env \Rightarrow Val\ list \Rightarrow State \Rightarrow (Val \times State) \Rightarrow bool \tag{2}$$

$$Inv : Env \Rightarrow State \Rightarrow bool \tag{3}$$

Hierbei repräsentiert *Val list* für Vor-/Nachbedingungen jeweils die Argumentwerte beim Funktionsaufruf und das Argument vom Typ *Val* in (2) den Resultatwert der Funktion. Invarianten hängen nicht von den Funktionsparametern ab, da deren Initialwerte zum Zeitpunkt der Auswertung der Invariante keine Bedeutung haben (Funktionsparameter werden im Code wie lokale Variablen behandelt, können also insbesondere modifiziert werden).

Wir beschreiben die Übersetzung als eine Operation # auf der abstrakten Syntax der Spezifikationsausdrücke. In (Abb. 7) wird diese abstrakte Syntax beschrieben; sie sollte jedoch auch ohne formale Definition intuitiv verständlich sein.

Wie bereits in 3.1.5 (S. 26) angedeutet, werden Spezifikationsausdrücke in einem *Kontext* übersetzt, der Zustände, Argumentwerte, etc. an Variablen bindet. Konkret werden Vor-/Nachbedingungen und Invarianten zu Lambda-Abstraktionen übersetzt, die jeweils alle Argumente der Funktionen aus (1)-(3) binden. Die Operation # erzeugt nun den Körper dieser Lambda-Abstraktionen. Es entstehen die folgenden Termstrukturen für die drei Arten von Spezifikationsausdrücken:

$$\lambda\ \Gamma\ [v_1, \ldots, v_n]\ \Sigma \bullet \#(Pre) \tag{4}$$

$$\lambda\ \Gamma\ [v_1, \ldots, v_n]\ \Sigma'\ (Result, \Sigma) \bullet \#(Post) \tag{5}$$

$$\lambda\ \Gamma\ \Sigma \bullet \#(Inv) \tag{6}$$

Die Variablennamen sind hierbei nicht schematisch, sondern literal zu interpretieren: Die Übersetzung durch # findet in einem Kontext statt, der $\Gamma : Env$ als Variablenumgebung, $\Sigma : State$ als Programmzustand, $\Sigma' : State$ als Zustand vor Ausführung der Funktion, $Result : Val$ als Resultatwert und für jeden Funktionsparameter v_i eine Variable $v_i : Dom\{Int, Double, Ref\}$ enthält. [5]

4.1.1 Übersetzungsregeln

Abb. 8 zeigt eine repräsentative Auswahl von Übersetzungsregeln, denen # folgt. Die Regeln sind syntaxgerichtet definiert, d. h. dass die Übersetzung eines Knotens im abstrakten Syntaxbaum

[5] Isabelle erlaubt die Verwendung eines gegenüber ASCII wesentlich erweiterten Zeichensatzes, so dass insbesondere griechische Buchstaben und diverse Sonderzeichen verwendet werden können.

Prädikate:	#(!A)	$\leadsto$	$(\neg\ \#(\mathsf{A}))$
	#(A && B)	$\leadsto$	$(\#(\mathsf{A}) \wedge \#(\mathsf{B}))$
	#(\valid(p))	$\leadsto$	$(\mathit{valid\text{-}ref}\ \#(\mathsf{p})\ \mathsf{type}(\mathsf{p})\ \Sigma)$
	#(\array(p, n))	$\leadsto$	$(\mathit{valid\text{-}array\text{-}acc}\ \#(\mathsf{p})\ \mathsf{type}(\mathsf{*p})\ \#(\mathsf{n})\ \Sigma)$
	#(\forall int i; P)	$\leadsto$	$(\forall\ (i :: \mathit{DomInt}).\ \#(\mathsf{P}))$
	#(${$s_1$ `{a1} $s_2$ `{a2} $\cdots$ s_k })	$\leadsto$	$(s_1\ \#(\mathsf{a1})\ s_2\ \#(\mathsf{a2})\ \cdots s_k)$
Ausdrücke:	#(*NULL*)	$\leadsto$	*None*
	#(\result)	$\leadsto$	*Result*
	#(\old(e))	$\leadsto$	$(\#(\mathsf{e}))[\Sigma'/\Sigma]$
	#(\true || \false)	$\leadsto$	$(\mathit{True} \vee \mathit{False})$
	#(E1 + E2)	$\leadsto$	#(E1) + #(E2)
	#(E1 < E2)	$\leadsto$	#(E1) < #(E2)
	#($i)	$\leadsto$	i
	#(x)	$\leadsto$	x x ist durch Quantoren gebundener Bezeichner
	#(|val|)	$\leadsto$	$[\![\|\mathsf{val}\|]\!]\ \Gamma\ \Sigma$

Abbildung 8: Regeln für die Übersetzung von abstrakter Syntax nach Isabelle

direkt aus der Übersetzung seiner Kinderknoten konstruiert werden kann. Die Regeln werden nachfolgend erläutert.

1. Alle unären und binären logischen Operatoren (etwa Konjunktion: &&) werden direkt in die korrespondierenden Isabelle-Operatoren übersetzt (etwa $\wedge$).

2. Für jedes Zeigerprädikat (\separated, \valid, \array) existiert ein zustandsbehaftetes Prädikat in Isabelle. Diese sind in den Theorien `State.thy` und `Selection.thy` definiert.

$$\mathit{valid\text{-}ref} : \mathit{DomRef} \Rightarrow \mathit{RTT} \Rightarrow \mathit{State} \Rightarrow \mathit{bool} \tag{7}$$

$$\mathit{valid\text{-}array\text{-}acc} : \mathit{DomRef} \Rightarrow \mathit{RTT} \Rightarrow \mathit{int} \Rightarrow \mathit{State} \Rightarrow \mathit{bool} \tag{8}$$

$$\mathit{separated\text{-}arrays} : \mathit{State} \Rightarrow \mathit{DomRef} \Rightarrow \mathit{RTT} \Rightarrow \mathit{int} \Rightarrow \mathit{DomRef} \Rightarrow \mathit{RTT} \Rightarrow \mathit{int} \Rightarrow \mathit{bool} \tag{9}$$

3. Quantoren (\forall, \exists) werden direkt in Quantoren in Isabelle übersetzt. Da nur über arithmetische Typen quantifiziert werden darf, haben die so eingeführten Variablen in Isabelle den Typen *DomInt* oder *DomDouble*.

4. Quotations werden generell uninterpretiert übersetzt; die Ausnahme bilden darin vorkommende Antiquotations, die durch ihre Übersetzung durch # an der Stelle ihres Vorkommens ersetzt werden.

5. Die Übersetzung von Ausdrücken (die immer einen skalaren Typen besitzen) erfolgt für binäre Operatoren und Konstanten auf sehr strukturierte Weise.

6. Ein Ausdruck \old(e) wird zunächst übersetzt wie der Ausdruck e selbst. Im so entstehenden Term t werden dann alle Vorkommen des Zustands Σ durch den Vorzustand Σ' ersetzt.

7. Isabelle-Konstanten ($\$i$) werden zu ihrem Namen (i) übersetzt. Somit ist klar, dass die Verwendung von in Isabelle gebundenen Variablen innerhalb von Antiquotations möglich ist.

8. Einfache Bezeichner, die durch einen Quantor der Spezifikationssprache eingeführt und von diesem gebunden wurden, werden ebenfalls zu ihrem Namen übersetzt. Dies ist bedingt dadurch, dass der zugehörige Quantor ebenfalls zu einem Isabelle-Quantor über denselben Namen übersetzt wurde.

9. Andere L-Werte lval werden innerhalb von Isabelle über eine entsprechende Semantikfunktion $[\![\cdot]\!]$ ausgewertet, so dass das Frontend lediglich die Anwendung dieser Funktion ausgeben muss.

4.2 Übersetzung von Modifikationslisten

Modifikationslisten (s. 3.5) werden in Isabelle als Datentyp repräsentiert. Die Übersetzung des Frontends ist also eine triviale Datentyp-Übersetzung. Die Semantik innerhalb von Isabelle wird über eine Funktion $[\![\cdot]\!]_m$ gegeben.

$$[\![\mathsf{mlist}]\!]_m : \Gamma \rightarrow \mathit{State} \rightarrow \mathit{Loc\ set}$$

5 Einschränkungen

Die SAMS Verifikationsumgebung dient dem Nachweis der funktionalen Korrektheit von C-Programmen. Damit kann die Verifikation der zwei Ebenen Modultest und Integrationstest (Module) im Entwicklungslebenszyklus durchgeführt werden. Im Folgenden wird dargelegt, welche sicherheitsrelevanten Verifikationsaufgaben der Softwareentwicklung explizit *nicht* durchgeführt werden können, bzw. welchen Einschränkungen die Reichweite der Verifikation in bestimmten Bereichen unterliegt.

5.1 Laufzeitgarantien/-analysen

Die Laufzeit des Programms und einzelner im Programm verwendeter Funktionen wird vom Begriff der funktionalen Korrektheit nicht berücksichtigt. Die hier vorgestellte Methode kann also abgesehen von der Terminationsgarantie keine näheren Laufzeitaussagen treffen.

5.2 Ein-/Ausgabe

Die Ein-/Ausgabe von Daten (Lesen aus/Schreiben in Dateien, Netzwerkkommunikation, spezielle Hardware-Register) wird über einfache Programmvariablen nachgebildet. Anfänglich vorhandene Eingaben müssen in Eingabevariablen abgespeichert sein. Berechnete Ausgaben werden in die dafür vorgesehenen Ausgabevariablen geschrieben.

Programme, deren Ablauf nicht in der Form {(1) Lesen aller Eingaben; (2) Berechnungen durchführen; (3) Ergebnisse ausgeben} erfolgt, können daher bezüglich ihres Ein-/Ausgabeverhaltens nicht verifiziert werden.

5.3 Nicht-terminierende Programme

Software, die permanent eine Sicherheitsfunktion ausübt, wird typischerweise dauerhaft innerhalb eines Regelkreises ausgeführt. Üblicherweise terminieren derlei *reaktive Programme* nicht, sondern laufen während des gesamten Betriebs der Maschine, bezüglich derer die Sicherheitsfunktion ausgeführt wird.

Die Verifikation der funktionalen Korrektheit von Funktionen mittels der SAMS-VU ist ausdrücklich nur für terminierende Funktionen möglich. Es kann für ein reaktives Programm also lediglich der terminierende, üblicherweise die Berechnungen eines Regelzyklus durchführende Teil verifiziert werden. Dies stellt für sehr viele reaktive Programme keine starke Einschränkung dar, da derlei Programme gemeinhin aus einer recht simplen äußeren Schleife bestehen, und die funktionale Korrektheit ein entscheidender Faktor der funktionalen Sicherheit ist.

Literatur

[Burdy u. a. 2005] BURDY, Lilian ; CHEON, Yoonsik ; COK, David ; ERNST, Michael ; KINIRY, Joe ; LEAVENS, Gary T. ; LEINO, K. Rustan M. ; POLL, Erik: An overview of JML tools and applications. In: *International Journal on Software Tools for Technology Transfer* 7 (2005), June, Nr. 3, S. 212– 232

[CStandard 1999] *Programming langauges — C*. ISO/IEC Standard 9899:1999(E), 1999. – Second Edition

[CStandard Rationale 1989] *Rationale for American National Standard for Information Systems – Programming Language – C*. 1989. – Available at `http://www.lysator.liu.se/c/rat/title.html`

[Filliâtre u. Marché 2004] FILLIÂTRE, Jean-Christophe ; MARCHÉ, Claude: Multi-Prover Verification of C Programs. In: *Sixth International Conference on Formal Engineering Methods (ICFEM)* Bd. 3308. Seattle : Springer, November 2004 (Lecture Notes in Computer Science), S. 15–29

[Filliâtre u. Marché 2007] FILLIÂTRE, Jean-Christophe ; MARCHÉ, Claude: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: DAMM, Werner (Hrsg.) ; HERMANNS, Holger (Hrsg.): *19th International Conference on Computer Aided Verification*, Springer, Juli 2007 (Lecture Notes in Computer Science)

[Kernighan u. Ritchie 1988] KERNIGHAN, Brian W. ; RITCHIE, Dennis M.: *The C Programming Language*. Prentice Hall, 1988

[Loeckx u. a. 1987] LOECKX, Jacques ; SIEBER, Kurt ; STANSIFER, Ryan: *The Foundations of Program Verification*. Second Edition. John Wiley & Sons, 1987 (Wiley-Teubner Series in Computer Science)

[Lüth 2009] LÜTH, Christoph: *Validation der SAMS Verifikationsumgebung*. SAMS Projektdokumentation, 2009. – DFKI, Forschungsbereich Sichere Kognitive Systeme

[Meyer 1992] MEYER, B.: Applying 'design by contract'. In: *Computer* 25 (1992), Oct, Nr. 10, S. 40–51. `http://dx.doi.org/10.1109/2.161279`. – DOI 10.1109/2.161279. – ISSN 0018–9162

[MISRA 2004] *MISRA-C: 2004. Guidelines for the use of the C language in critical systems.* 2004

[Nipkow u. a. 2002] NIPKOW, Tobias ; PAULSON, Lawrence C. ; WENZEL, Markus: *Lecture Notes in Computer Science*. Bd. 2283: *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002

[Winskel 1993] WINSKEL, Glynn: *The Formal Semantics of Programming Langauges*. The MIT Press, 1993 (Foundations of Computing Series)

[Z-Spec 2002] *ISO/IEC 13568: Information technology – Z formal specification notation – Syntax, type systems and semantics*. 2002. – ISO/IEC, Geneva, Switzerland

A Syntax der Annotationen

A.1 Neue Terminalsymbole

Konkrete Syntax	Symbolname
_Any	anytype
_Bool	booltype
\true	true
\false	false
NULL	null
\result	result
\old	old
\valid	valid
\array	validindex
\separated	separated
\forall	forall
\exists	exists
@invariant	invariant
@modifies	modifies
@assigns	modifies
@requires	requires
@ensures	ensures
@variant	variant
@assert	assert
@join	join
@theory	theory
@define	define
$var	var1
$function	function1
$! function	function2
^ function	function3
:: abbreviation	function4

Des weiteren wurden die binären Operatoren $<->$, $-->$ (bzw. synonym $<==>$ und $==>$), @ und $|->$ hinzugefügt, die in Spezifikationsausdrücken vorkommen können. Diese werden i. F. nicht speziell benannt, sondern literal verwendet.

A.2 Grammatik

Wir geben im folgenden eine Erweiterung der C-Grammatik nach (CStandard, 1999, §Anhang A) oder (Kernighan u. Ritchie, 1988, Anhang A.13) an, welche annotierte Programme beschreibt. Da nur neue Alternativen oder Regeln hinzugefügt werden, ist diese Änderung konservativ, d.h.

alle vorher zulässigen Programme werden weiter durch diese Grammatik parsiert. Die Grammatik erlaubt viele unzulässige Ausdrücke, insbesondere können die neuen Konstanten auch innerhalb von Programm-Quellcode auftreten. Dieses wird sowohl von der statischen Analyse geprüft, als auch von jedem standardkonformen C-Compiler zurückgewiesen.

A.2.1 Neue Kategorien von Bezeichnern

Zusätzlich zur Kategorie der C-Bezeichner (`identifier`) kommen fünf Kategorien von Bezeichnern hinzu. Diese unterscheiden sich von C-Bezeichnern durch ein vorangestelltes symbolisches Präfix. Aufgrund der beschränkten Möglichkeiten ihres Auftretens sind Mehrdeutigkeiten zwischen diesen Präfixen und etwa unären Operatoren ausgeschlossen.

Kategorie	*Präfix*	*Beispiel*
`HOLIdent`	$	$foo
`HOLAbbrev`	::	::konfig_OK
`HOLStateFun`	$!	$!reach
`HOLAntiqIdent`	‘	‘len
`HOLReprIdent`	^	^Matrix

A.2.2 Erweiterte Regeln

Folgende Regeln werden erweitert (die Erweiterungen sind durch ein << gekenzeichnet):

```
ExternalDeclaration
  : FunctionDefinition
  | AnnDeclaration      <<
  | FunctionAnnoList    <<
  | ThyAnnoList         <<

FunctionDefinition
  : DeclarationSpecifiers Declarator CompoundStatement
  | Declarator CompoundStatement
  | Spec DeclarationSpecifiers Declarator CompoundStatement  <<
  | Spec Declarator CompoundStatement                        <<

Statement
 : LabelledStatement
 | ExpressionStatement
 | CompoundStatement
 | SelectionStatement
 | IterationStatement
 | JumpStatement
 | AssertStatement    <<
```

```
 | SpecStmt           <<

ConditionalExpression
 : BindExpr           <<
 | LogicalOrExpression '?' Expression ':' ConditionalExpression

UnaryExpression
 : SpecExpression                  <<
 | '++' UnaryExpression
 | '--' UnaryExpression
 | UnaryOperator CastExpression
 | sizeof UnaryExpression
 | sizeof '(' TypeName ')'

UnaryOperator
 : '*'
 | '+'
 | '-'
 | '~'
 | '!'
 | '&'
 | old    <<
 | '@' Id <<
```

A.2.3 Neue Regeln

Die folgenden Regeln werden neu hinzugefügt:

```
AnnoEndMarker
 : '*/'
 | '@*/'

AnnDeclaration
 : Declaration
 | Spec Declaration

AssertStmt
 : '/*@' assert Expression AnnoEndMarker

SpecStmt
 : Spec Statement

ConstExpr
```

```
 : true
 | false
 | null
 | result

BindExpr
 : BindOp ParameterList ';' BindExpr
 | ImpliesExpression

BindOp
 : forall
 | exists

ImpliesExpression
 : LogicalOrExpression
 | ImpliesExpression '-->'  LogicalOrExpression
 | ImpliesExpression '<->'  LogicalOrExpression
 | ImpliesExpression '==>' LogicalOrExpression
 | ImpliesExpression '<==>' LogicalOrExpression

SpecExpression
 : PostfixExpression
 | array '(' AssignmentExpression ',', AssignmentExpression ')'
 | separated '(' ArgumentExpressionList ')'
 | HOLIdent
 | HOLIdent '(' ArgumentExpressionList ')'
 | HOLAbbrev '(' ')'
 | HOLAbbrev '(' ArgumentExpressionList ')'
 | HOLStateFun '(' ArgumentExpressionList ')'
 | LitHOL

LitHOL
 : '${' QuoteContents '}'

QuoteContents
 :
 | QuoteContents StringOrAntiquote

StringOrAntiquote
 : '`{' Expression '}'
 | QuotedText
 | HOLAntiqIdent
 | HOLReprIdent '(' ArgumentExpressionList '}'
```

```
FunctionAnnoList
 : '/*@' FunctionAnnoItemList AnnoEndMarker

FunctionAnnoItemList
 : FunctionAnnoItem
 | FunctionAnnoItemList FunctionAnnoItem

FunctionAnnoItem
 : function1 Declaration
 | function2 Declaration
 | function3 Declaration
 | function4 Declaration
 | var1 Declaration
 | define Declaration
 | function1 '{' DeclarationList '}'
 | function2 '{' DeclarationList '}'
 | function3 '{' DeclarationList '}'
 | function4 '{' DeclarationList '}'
 | var1 '{' DeclarationList '}'

ThyAnnoList
 : ThyAnnoItem
 | ThyAnnoItemList ThyAnnoItem

ThyAnnoItem :: { ThyDecl }
  : theory identifier

MValueList
 :
 | MValue
 | MValueList '|' MValue

MValue
  : PrimMValue
  | '*' MValue

PrimMValue
  : Id
  | '(' MValue ')'
  | PrimMValue '[' MIndexExpr ']'
  | PrimMValue '.' Id
  | PrimMValue '->' Id
```

```
MIndexExpr
  : Expression
  | Expression ':' Expression
  | ':' Expression
  | '*'

BoundedMValueList
  :
  | BoundedMValue
  | BoundedMValueList ',' BoundedMValue

BoundedMValue
  : MValue
  | MValue '|->' MValueList


Annotation
  : invariant Expression
  | assigns   BoundedMValueList
  | assigns   nothing
  | modifies  BoundedMValueList
  | modifies  nothing
  | requires  Expression
  | ensures   Expression
  | join      Expression
  | variant   Expression

AnnotationList
  : Annotation
  | AnnotationList Annotation

Spec
  : '/*@' AnnotationList AnnoEndMarker
```

B Kategorisierung der Standardbibliothek

Funktionen aus der Standardbücherei (CStandard, 1999) werden in der ersten Ausbaustufe der Verifikationsumgebung zunächst nur in relativ geringer Anzahl unterstützt. Generell zerfallen die Standardbüchereien in vier Kategorien:

- Funktionen der Kategorie **A** dürfen verwendet werden, müssen aber spezifiziert werden

In (CStandard, 1999)	Name	Kat.	Anmerkung
7.2	`<assert.h>`	B	Zusicherungen als Annotation verfügbar
7.4	`<ctype.h>`	A	Bis auf Ein-/Ausgabefunktionen
7.5	`<errno.h>`	A	Modellierung als globale Variable
7.7	`<float.h>`	A	Fließkommaarithmetik in SAMS nicht vorgesehen
7.10	`<limits.h>`	A	
7.11	`<locale.h>`	B	
7.12	`<math.h>`	A	
7.13	`<setjmp.h>`	X	Keine nicht-lokalen Sprünge erlaubt
7.14	`<signal.h>`	X	Signalbehandlung wird nicht modelliert
7.15	`<stdarg.h>`	X	Variable Argument-Listen werden nicht modelliert
7.17	`<stddef.h>`	A	
7.19	`<stdio.h>`	B/X	Ausgabe: kein Effekt; Eingabe: nicht modelliert
7.20.1	`<stdlib.h>`	C	Numerische Konversionen (`atoi` etc)
7.20.2	`<stdlib.h>`	X	Zufallszahlen (`rand`)
7.20.3	`<stdlib.h>`	X	Dynamische Speicherverwaltung (`malloc`, `free`)
7.20.4	`<stdlib.h>`	X	Kommunikation mit der Prozessumgebung (`abort`, `exit`, `getenv`, `system`)
7.20.5	`<stdlib.h>`	A	Suchen und sortieren
7.20.6	`<stdlib.h>`	A	Ganzzahlarithmetik
7.20.7	`<stdlib.h>`	C	Multibyte-Character-Konversion
7.20.8	`<stdlib.h>`	C	Multibyte-String-Konversion
7.21	`<string.h>`	C	Zeichenketten als Integer-Arrays modelliert
7.23	`<time.h>`	X	Zeit wird nicht modelliert

Tabelle 2: Übersicht über die C-Standardbücherei gemäß ISO/IEC 9899:1990. Kapitelverweise beziehen sich jedoch auf Version 1999.

(z.B. mathematische Funktionen);

- Funktionen der Kategorie **B** dürfen verwendet werden, haben aber keinen modellierten Effekt (z.B. Ausgabe);
- Funktionen der Kategorie **C** dürfen nicht verwendet werden, weil sie nicht modellierte Datentypen verwenden oder strukturierte Werte als Argumente erwarten bzw. diese zurückgeben (z.B. div_t);
- Funktionen der Kategorie **X** dürfen nicht verwendet werden, da sie Seiteneffekte, Zustandsübergänge, oder Kontrollflussänderungen, die in der Semantik nicht modelliert werden.

Die im Rahmen des Projektes entwickelten Spezifikationen können als formaler Anhang von Abschnitt 7 des C-Standards (CStandard, 1999) verstanden werden. Ihr Beweis ist nicht Teil des Projektes SAMS; es soll jedoch im Rahmen der Zertifizierung validiert werden, dass die Spezifikationen die im Standard informell spezifizierte Semantik korrekt formalisieren.

Funktionen der Kategorie **C** könnten verwendet werden, wenn vorher die Datentypen auf semantischer Seite modelliert werden (beispielsweise Zeichenketten als Listen von Zeichen). Dieses ist im Rahmen des SAMS-Projektes nicht vorgesehen.

Tabelle 2 kategorisiert die Funktionen der C-Standardbücherei nach diesen Kriterien.

C MISRA-Standard – Anpassungen/Erweiterungen

Dieser Anhang führt die Verschärungen und Erweiterungen des von der SAMS-VU unterstützten Sprachteilmenge gegenüber den MISRA-Programmierrichtlinien auf:

ad 1.2 Unspezifiziertes und undefiniertes Verhalten:

- Auswertung von verschachtelten Ausdrücken, siehe Abschnitt 2.2.1.

ad 1.5 Fließkommaarithmetik muss dem IEEE 754 Standard folgen, sofern sie vorhanden ist.

ad 12.13 Die Inkrement- und Dekrementoperatoren ++ und −− besitzen intern keine Repräsentation, sondern werden durch explizite Zuweisungen kodiert. Daher dürfen diese Operatoren lediglich als unverschachtelte Anweisungen der Form ++x,x++ bzw. −−x oder x−− verwendet werden.

ad 14.6 Der MISRA-Standard erlaubt die Verwendung einer einzelnen `break`-Anweisung in einer Schleife. Es werden keine unbedingten Sprunganweisungen (außer eines `return` am Ende jeder Funktion) unterstützt.

ad 14.7 Siehe Abschnitt 2.3.3.

ad 15.* `switch`-Anweisungen werden nicht unterstützt.

ad 17.1 Beliebige Zeigerarithmetik wird nicht unterstützt. Zugriffe auf Arrays können nur über die explizite Zugriffssyntax `a[expr]` vorgenommen werden (wobei allerdings Zeiger auf T und Felder unbestimmter Größe vom Typ T äquivalent sind). Der Dereferenzierungsoperator $*$ darf sich lediglich auf *L-Werte* beziehen. Er wandelt also explizit *nicht* einen Ausdruck in einen L-Wert um, wie dies in C90 z. B. im Ausdruck $*(x + 2)$ der Fall ist.

ad 17.6 Das C-Frontend prüft und stellt sicher, dass die Adresse von Objekten mit automatischer Speicherklasse keinen Objekten zugewiesen wird, deren Lebensdauer die des Objekts, dessen Adresse es beinhaltet, übersteigt. Gültig, wenn auch wenig sinnvoll, ist ein Vorgehen wie das folgende:

```
short f(const short *ip_1, const short *ip_2)
{
  const short *aux = ip_1;    /* Zuweisungen an Objekt mit */
  if (*ip_2 < *aux) {         /* k\"urzerer Lebensdauer     */
    aux = ip_2;
  }
  return *aux * 2;
}
```

Nicht erlaubt ist es dagegen, die Funktionsparameter `ip_i` an eine globale Variable zuzuweisen (wobei dies nicht die einzige Art darstellt, Adresswerte länger zu speichern als erlaubt).

ad 18.4 Unionen (**union**) werden nicht verwendet. Das Frontend weist daher alle Programme zurück, die Unionen verwenden. Aufseiten von Isabelle finden Unionen keine Repräsentation.

D Übersicht über Isabelle-Theorien

Alle eigens entwickelten Theorien leiten sich von der Isabelle/HOL Basistheorie *Complex_Main* ab. Die arithmetischen Typen *nat*, *int*, *rat*, *real* und *complex* sowie etliche diesbezügliche Theoreme stehen damit direkt zur Verfügung. Die eigens entwickelten Theorien gliedern sich in zwei Stränge: *Programmverifikation und Modellierung der Programmiersprache* und *Domänenmodellierung*. Die folgende Tabelle gibt eine kurze Übersicht über den Inhalt jeder Theorie.

Detaillierte Informationen finden sich in den Theoriedateien selbst; aus diesen kann durch den Aufruf von `isabelle make ilc-documents` im Verzeichnis `/trunk/tooldev/src/theories` das PDF-Dokument `/trunk/tooldev/src/theories/ilc/output/HOL/ilc/outline.pdf` erzeugt werden, welches die Dokumentation und allen technischen Inhalt (Datentypen, Definitionen, Theoreme, Syntax, usw.) der Theorien in lesbarer Form enthält.

Theoriename	*Beschreibung*

—Programmverifikation und Modellierung der Programmiersprache—	
Auxiliaries	Technische Hilfstheoreme, die für Beweise in anderen Theorien benötigt werden.
State	Definition des Zustandsmodells; Vereinfachungsregeln für Zustands-Updates; Gültigkeit von Zeigern.
Selection	Array-Element und Struktur-Komponenten-Zugriffe
Modifies	Die `@modifies`-Zustandsrelation
ILC	Datentypen zur Repräsentation von C-Übersetzungseinheiten (generisch bezüglich der verwendeten Spezifikationen)
ST	Zustandstransformationen: Zuweisung, Alternierung, Iteration
Spec	Zustandsprädikate; Erfülltheit von Spezifikationen
Env	Umgebungen: Funktionen (Semantik und Spezifikationen) und Variablen
SpecILC	Instanziierung von ILC für konkret verwendete Spezifikationstypen
Parameters	Typkorrektheit von Funktionsparametern
Sem	Formalisierung der Semantik der Programmiersprache C
SemLemmas	Lemmas bezüglich der Semantik: Seiteneffekfreie Auswertung
ModularCorrectness	Definition der Korrektheitsaussage
ProofRulesSetup	Definition der Hoare-Tripel
ProofRules	Beweisregeln für Hoare-Tripel
TacticLemmas	Für automatisiertes Beweisen benötigte Hilfslemmata
SAMS	Abschließende Theorie: Einbindung der Taktiken aus `sams_tactics.ML` und Einführung von Beweismethoden
RecordRepr	(experimentell) Repräsentation von Datentypen als Records
SAMSRepr	Repräsentationsfunktionen für Datentypen
—Domänenmodellierung—	
SAMSKonvex	Konvexe Mengen
Trigonometrie	Zusätzliche Theoreme bezüglich der trigonometrischen Funktionen
SAMSDomain	Definitionen und Theoreme aller in Programmspezifikationen verwendeten Operationen

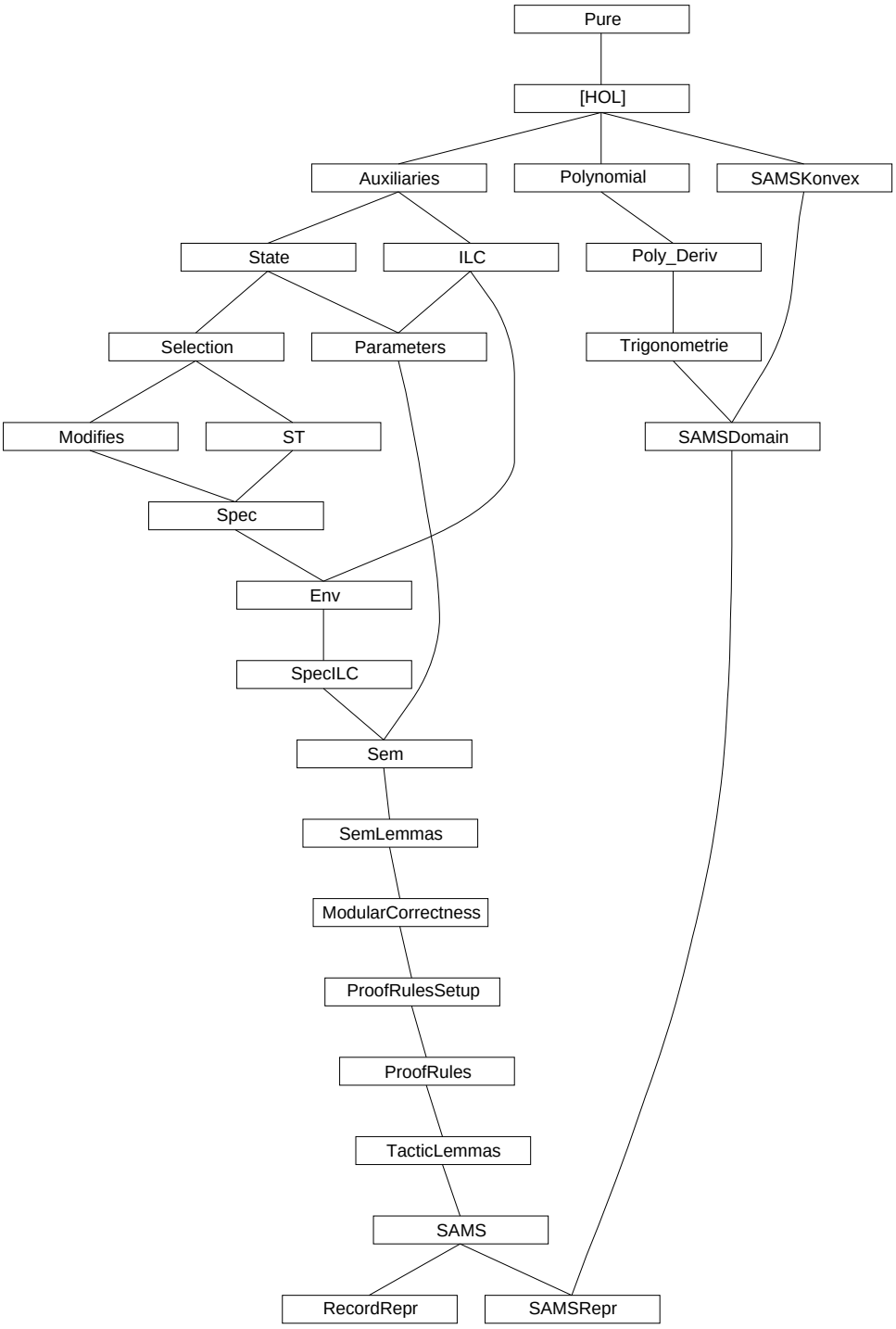


Abbildung 9: Theorie-Graph der SAMS-Verifikationsumgebung. Eine Verbindung zwischen Theorien bezeichnet eine direkte Abhängigkeit der tiefer gelegenen zur höher gelegenen Theorie. Die Theorien *Polynomial* und *Poly_Deriv* gehören zum Umfang der Isabelle-Distribution.

Index

-10-01
ch Report