AG Vernetzte Systeme
Fachbereich Informatik
Technische Universität Kaiserslautern

Masterarbeit

Spezifikation und automatisierte Implementierung zeitkritischer Systeme mit TC-SDL

Dennis Christmann

15. September 2010

Spezifikation und automatisierte Implementierung zeitkritischer Systeme mit TC-SDL

Masterarbeit

AG Vernetzte Systeme
Fachbereich Informatik
Technische Universität Kaiserslautern

Dennis Christmann

Tag der Ausgabe : 20.04.2010
Tag der Abgabe : 15.09.2010

Erstgutachter : Prof. Dr. Reinhard Gotzhein
Zweitgutachter : Dipl.-Inf. Philipp Becker

Ich erkläre hiermit, die vorliegende Masterarbeit selbständig verfasst zu haben. Die verwendeten Quellen und Hilfsmittel sind im Text kenntlich gemacht und im Literaturverzeichnis vollständig aufgeführt.

Kaiserslautern, 15. September 2010

(Dennis Christmann)

Zusammenfassung

Diese Masterarbeit behandelt die Entwicklung zeitkritischer Systeme mit ITU-T's *Specification and Description Language* (SDL). Die Sprache SDL, die mittlerweile auf eine über 30jährige Geschichte zurückblickt, eignet sich durch hierarchische Strukturierungsmöglichkeiten für die modellgetriebene Entwicklung komplexer nebenläufiger Systeme und hat dies bereits in vielen Applikationen aus den Bereichen Telekommunikation und Netzwerkprotokolle nachgewiesen. An vielen Stellen, beispielsweise bei der Ausführungsreihenfolge von Agenten eines SDL-Systems, fehlt es allerdings an Ausdrucksstärke, um in ausreichendem Ausmaß Einfluss auf das Laufzeitverhalten von zeitkritischen Systemen zu nehmen.

Mit *Time-critical*-SDL (TC-SDL) führt diese Masterarbeit eine Erweiterung von SDL ein, die es anhand von Annotationen in der SDL-Spezifikation ermöglicht, das Laufzeitverhalten entsprechend den Anforderungen eines Szenarios zu beeinflussen. Im Konkreten wird dem Entwickler eines SDL-Systems die Möglichkeit gegeben, zwischen verschiedenen Planungsverfahren zu wählen und zeitkritische Systemkomponenten durch die Vergabe von Prioritäten in der Ausführung zu bevorzugen. Hierzu wird das SDL-Laufzeitsystem *SdlRE* um ein Framework für Planungsverfahren erweitert, in das ein prioritätsbasiertes Planungsverfahren eingegliedert wird.

Des Weiteren wird in dieser Arbeit mit *Black Burst Synchronization* (BBS) ein Synchronisationsprotokoll behandelt, dessen Realisierung mit SDL aufgrund der hohen zeitlichen Anforderungen ausscheidet. Die Lösung dieser Einschränkung besteht aus einem Kompromiss, der einerseits auf einer effizienten Implementierung des Protokolls in der Programmiersprache C beruht, andererseits aber ein komfortables Konfigurationsinterface in SDL zur Verfügung stellt.

Anschließend wird die funktionale und quantitative Evaluation sowohl des prioritätsbasierten Planungsverfahrens als auch der Realisierung von BBS beschrieben. Die Evaluationen wurden experimentell auf der Imote2-Plattform durchgeführt. Die Ergebnisse des prioritätsbasierten Planungsverfahrens zeigen dabei, dass in der Integration adäquater Planungsverfahren ein großes Potential hinsichtlich der Vorhersagbarkeit von SDL-Systemen und deren Eignung für zeitkritische Systeme steckt. Ferner weisen die Ergebnisse der Evaluation von BBS die Anwendbarkeit des Protokolls in Single- und Multi-Hop-Topologien nach.

Abstract

This master's thesis covers the development of time-critical systems with ITU-T's *Specification and Description Language* (SDL). Due to options for hierarchical structuring, SDL, a language looking back on it's history more than 30 years by now, is qualified for the model-driven development of complex concurrent systems and demonstrated its ability in applications in the areas of telecommunication and network protocols. However, in many cases, e.g., the execution order of agents of SDL systems, SDL is missing expressiveness to influence the runtime behavior of time-critical systems in a sufficient way.

This thesis introduces *Time-critical*-SDL (TC-SDL), an extension of SDL, which enables the manipulation of the runtime behavior by means of annotations in the SDL specification according to the scenario's demands. In detail, the developer is given the possibility to choose between several scheduling strategies and to prefer the execution of time-critical components by assigning priorities. For this, the SDL runtime environment *SdlRE* is extended by a framework for scheduling strategies, in which a priority-based scheduling strategy is integrated.

In addition, the thesis presents *Black Burst Synchronization* (BBS), a synchronization protocol, whose realization with SDL is ruled out due to the high time requirements. The solution to this limitation consists of a trade-off, which is on the one hand based on an efficient implementation of the protocol in the programming language C, on the other hand, it provides a convenient configuration interface in SDL.

Thereafter, the thesis describes the functional and quantitative evaluation of both the priority-based scheduling strategy and the realization of BBS. The evaluations were done on the Imote2 platform experimentally. The results of the priority-based scheduling strategy show that the integration of appropriate scheduling strategies offers a high potential according to the predictability of SDL systems and their qualification for time-critical systems. In addition, the evaluation of BBS shows the applicability of the protocol in a single as well as a multi hop topology.

Inhaltsverzeichnis

1. KAPITEL

Einleitung

Nahezu täglich erscheinen Newsticker-Meldungen, in denen von neuen Produkten und Entwicklungen in echtzeitfähigen Systemen berichtet wird [The10, Ele10, Pre10]. Dies zeigt, dass Echtzeitsysteme trotz ihrer langen Geschichte, die bereits in den 60er-Jahren des 20 Jahrhunderts begann, noch immer keine Lösungen für alle aktuellen Problemstellungen liefern und der Bedarf an neuen Methoden und Werkzeugen auch heute noch vorhanden ist. Insbesondere die Anforderung der Integration in bewährte (Software-)Entwicklungsprozesse und die Möglichkeiten, die durch Mehrkernprozessoren entstanden sind, zeigen die Notwendigkeit, die Entwicklung im Bereich der Echtzeitsysteme weiterhin voranzutreiben.

Dabei ist der Begriff der *Echtzeitfähigkeit* vermutlich einer der am häufigsten irrtümlich verwendeten Begriffe der Informatik: Er beschreibt nicht etwa, dass ein System besonders schnell auf Stimuli reagiert, sondern vielmehr, dass die Antwort bis zu einer vorgegebenen Frist vorliegt [Sta88]. Diese Basisdefinition lässt bereits vermuten, dass Echtzeitsysteme Vorhersagbarkeit voraussetzen und insbesondere in zeitkritischen Systemen – beispielsweise in Systemen der Regelungstechnik – ihre Anwendung finden.

Die vorliegende Masterarbeit behandelt die Eignung von ITU-T's *Specification and Description Language* (SDL, [Int99]) hinsichtlich der Spezifikation zeitkritischer Systeme. Hierzu zählen zum Beispiel Kommunikationsprotokolle, wobei die in der vorliegenden Arbeit vorgestellten Ergebnisse stark von Szenarien in drahtlosen Sensornetzwerken (*Wireless Sensor Networks*, WSNs [ASSC02]) motiviert wurden. Auch wenn diese Arbeit auf die Verbesserung der Echtzeitfähigkeit von SDL abzielt, stellt sie nur einen ersten Schritt in diese Richtung dar, da strikte Echtzeitfähigkeit die Vergabe von deterministischen Garantien bezüglich Ausführungszeiten impliziert, die auch mit den vorgestellten Maßnahmen nicht gegeben werden können. Einerseits liegt dies an aufwendigen SDL-Sprachkonstrukten, die zusätzlich ein indeterministisches Verhalten beschreiben können. Andererseits lassen sich in drahtlosen Ad-Hoc-Netzwerken nur sehr schwer Annahmen über das störanfällige Übertragungsmedium treffen, so dass strikte Garantien in Drahtlosnetzwerken prinzipiell kaum möglich sind [Kop91].

Die Sprache SDL steht mit ihrem aktuellen SDL2000-Standard vor dem Problem, dass sie ursprünglich nicht als Implementierungssprache vorgesehen war, sich aber heute vorrangig in modellgetriebenen Protokollentwicklungen etabliert hat [Got07]. Dies liegt unter Anderem daran, dass SDL viele Vorteile für die Software-Entwicklung mit sich bringt, zum Beispiel Möglichkeiten zur hierarchischen Strukturierung eines Systems in nebenläufige Komponenten oder die Definition von neuen Datentypen. Des Weiteren hat sich die formale Semantik als Vorteil erwiesen, da sie eine präzise und eindeutige Beschreibung des Verhaltens ermöglicht. Nicht zuletzt hat

auch die Einführung der Objektorientierung in SDL-92 zur steigenden Verwendung von SDL als Implementierungssprache beigetragen [EGG+01]. Mit SDL-2000 wurde diese Entwicklung fortgeführt, indem der Fokus weiter auf Design und Implementierung gelegt wurde [Ree00], zum Beispiel durch die Einführung von Exceptions. Auch mit dem nächsten SDL-Standard (SDL-2010) soll dieser Trend durch die Verwendung von Konzepten aus Programmiersprachen (Java, C, ...) zur Definition von Datentypen weitergeführt werden [Ree09].

An vielen Stellen des SDL-Standards ist erkennbar, dass die Sprache ursprünglich als high-level Spezifikationssprache zur Validierung und Verifikation funktionaler Eigenschaften entwickelt wurde und nicht als Implementierungssprache für Systeme auf realer Hardware. Insbesondere wird dies bei dem Zeitbegriff deutlich, der im SDL-Standard allgemein und abstrakt gehalten wurde, so dass Simulatoren und Verifikationswerkzeuge von Laufzeiten abstrahieren und die Zeit anhalten, solange noch aktive Transitionen im System sind. Dies hat allerdings zur Folge, dass eine erfolgreich validierte Spezifikation auf realer Hardware mit vorhandenen Laufzeiten ein völlig unerwartetes Verhalten zeigen kann.

Um dieses Problem zu lösen, sind Maßnahmen nötig, um bereits auf Spezifikationsebene Implementierungs- und Hardwaredetails einfließen zu lassen und dadurch die Lücke zwischen Spezifikations- und Implementierungsverhalten zu schließen. In dieser Arbeit werden erste solche Maßnahmen vorgestellt. In einem ersten Schritt wird auf die Reihenfolge eingegangen, in der einzelne Agenten eines SDL-Systems ausgeführt werden, und Möglichkeiten vorgestellt, diese Reihenfolge zu beeinflussen. Obwohl dies eine klare Einschränkung der SDL-Semantik von asynchronen *Abstract State Machines* (ASMs) darstellt, widerspricht diese Interpretation keineswegs der SDL-Semantik, sondern beschreibt nur eine mögliche Interpretation. Eine solche Interpretation ist ohnehin notwendig, wenn ein SDL-System auf beschränkter Hardware, zum Beispiel einem Einkernprozessor, zur Ausführung kommt.

Die Beeinflussung der Ausführungsreihenfolge erlaubt zwar das Priorisieren wichtiger Transitionen, kann aber ohne Präemption oder exakte Bestimmung maximaler Laufzeiten keine Garantien bezüglich der Einhaltung von Fristen geben. Die Realisierung von Protokollen, die hierauf angewiesen sind, benötigt demnach eine andere Vorgehensweise. Mit der Umsetzung von *Black Burst Synchronization* (BBS, [GK08]), welches ein Protokoll mit hohen Anforderungen bezüglich der Einhaltung von Fristen darstellt, wird in dieser Arbeit gezeigt, wie mit solchen Problemen als Kompromiss zwischen effizienter/hardwarenaher Implementierung und modellgetriebener Konfiguration umgegangen werden kann.

Diese Arbeit ist wie folgt gegliedert: Kapitel 2 fasst die Eigenschaften und den Stand der Technik von echtzeitfähigen Systemen zusammen. In Kapitel 3 wird SDL in einer Übersicht vorgestellt. Dabei werden neben den Schlüsselaspekten sowohl die Probleme von SDL bei der Verwendung in zeitkritischen Anwendungen als auch einschlägige Lösungsansätze behandelt. Kapitel 4 stellt mit *Time-critical-SDL* (TC-SDL) eine Erweiterung von SDL vor, die es ermöglicht, das Verhalten des Gesamtsystems zur Laufzeit zu beeinflussen, wobei die Erweiterung den SDL-Standard einschränkend interpretiert, also ohne die eigentliche Semantik zu verletzen. Da TC-SDL nur die Chancen zur Einhaltung von Fristen erhöht, ohne feste Garantien geben zu können, wird in Kapitel 5 am Beispiel von *Black Burst Synchronization* (BBS) die Integration einer stark zeitkritischen Komponente mit deterministischen Anforderungen bezüglich Ausführungszeit in ein SDL-System behandelt. Anhand der Evaluationen in Kapitel 6 sollen anschließend die Verbesserungen durch TC-SDL und das korrekte Verhalten der BBS-Integration experimentell bestätigt und quantifiziert werden. Abschließend gibt Kapitel 7 eine Zusammenfassung und einen Ausblick.

2. KAPITEL

Grundlagen von Echtzeitsystemen

Auch wenn das Thema Echtzeitfähigkeit nicht neu ist, ist die anhaltende Präsenz in wissenschaftlichen Publikationen ein Indiz dafür, dass dieses Thema noch lange nicht vollständig erforscht wurde. Obwohl sich an der Problemstellung in der jahrzehntelangen Geschichte nicht viel verändert hat, zwingen neuartige Technologien und moderne Anwendungsgebiete zu neuen und angepassten Lösungen.

In diesem Kapitel werden grundlegende Konzepte zur Realisierung von Echtzeitsystemen vorgestellt. Zunächst wird in Abschnitt 2.1 eine Definition des Begriffs Echtzeitsystem gegeben und die Merkmale echtzeitfähiger Systeme diskutiert. Anschließend behandelt und vergleicht Abschnitt 2.2 die bekanntesten Planungsverfahren, welche die zentrale Rolle in einem Echtzeitsystem einnehmen. Das durch prioritätsbasiertes Scheduling entstehende Problem der Prioritätsinvertierung wird in Abschnitt 2.3 mit gängigen Lösungsansätzen vorgestellt.

2.1 Echtzeitsysteme: Definition und Eigenschaften

Echtzeitsysteme sind bereits seit vielen Jahrzehnten ein Thema in der Forschung und auch längst in der Wirtschaft kein Neuland. So sind Echtzeitsysteme zum Beispiel im Automobilbereich, in Flugzeugüberwachungssystemen, in der Telekommunikation oder in industriellen Produktionslinien erfolgreich im Einsatz. Ihr häufigstes Einsatzgebiet sind sicherheitsrelevante Funktionen, zum Beispiel die Airbagsteuerung oder das Anti-Blockier-System (ABS) eines Kraftfahrzeugs. Neben diesen Anwendungen in eingebetteten Systemen findet Echtzeitfähigkeit auch zunehmend Relevanz in anderen Teilen der Informatik, etwa in den Gebieten Datenbanken und Künstliche Intelligenz [Sta92].

2.1.1 Definition

Unter einem Echtzeitsystem versteht man ein System, bei dem die Korrektheit nicht nur von dem logischen Ergebnis abhängt, sondern ebenfalls von dem Zeitpunkt, zu dem das Ergebnis vorliegt [Sta88, But05]. Dabei ist die primäre Eigenschaft nicht die Geschwindigkeit, in der die Aufgabe ausgeführt wird, sondern setzt sich zusammen aus den Kriterien Rechtzeitigkeit, Vorhersagbarkeit, Sicherheit und Zuverlässigkeit [Zöb08]. Allerdings ist die Ausführungsgeschwindigkeit eines Echtzeitsystems auch nicht vollständig vernachlässigbar. Stattdessen muss sie als

notwendige, aber nicht hinreichende, Bedingung gesehen werden, um die vorgegebenen Anforderungen zu erfüllen [Sta92]. Vorhersagbarkeit und Rechtzeitigkeit beziehen sich demnach auf das Einhalten von Fristen, was nicht zwingend eine zügige Bearbeitung und eine schnelle Hardware verlangt.

Verglichen mit einem herkömmlichen System unterscheidet sich ein Echtzeitsystem insbesondere in dem starken Zusammenspiel zwischen Korrektheit und Performanz und den Auswirkungen bei Nichterfüllung der Anforderungen. Denn je nach Einsatzgebiet kann in einem Echtzeitsystem die Verletzung einer Frist hohe finanzielle oder personelle Schäden nach sich ziehen. Aus diesem Grund erlaubt eine Aussage zur Vorhersagbarkeit, d.h. ob die Frist einer bestimmten Aufgabe eingehalten wird, nicht die Einschränkung der Analyse auf den durchschnittlichen Fall (*average case*-Analyse), sondern verlangt die Betrachtung des schlimmstmöglichen Falls (*Worst Case*-Analyse) [Sta92]. Dieser schlimmstmögliche Fall besteht aus der ungünstigsten Konstellation, die sich zusammensetzt aus der schlechtesten Ressourcensituation, dem Vorhandensein weiterer Aufgaben, die möglicherweise zuvor abgearbeitet werden, und der maximalen Bearbeitungszeit der betrachteten Aufgabe selbst. Im Allgemeinen führt ein starkes Abweichen des schlimmstmöglichen Falls zu einer nötigen Überdimensionierung der verwendeten Hardware, was allerdings im Regelfall eine Ressourcenunternutzung zur Folge hat [Kop91].

Bereits bei kleinen Systemen sind aufgrund von übermäßigem Wachstum des Zustandsraums (*state explosion* [Sta88]) aufwändige Analysen notwendig, um die Echtzeitfähigkeit des Systems nachzuweisen. Bei größeren Systemen werden hingegen die Analysen häufig zu komplex, wodurch die fehlende Skalierbarkeit eines der größten Probleme von Echtzeitsystemen wird. Auch werden Aussagen über das Zeitverhalten durch moderne Technologien, wie zum Beispiel *Pipelining*, *Direct Memory Access (DMA)*, die Abbildung von virtuellem Speicher auf realen Speicher und der Vielzahl an Caches, immer schwieriger [Sta92, But05]. Zwar konnte die durchschnittliche Ausführungszeit durch diese Maßnahmen über die Jahre hinweg immer weiter gesenkt werden, allerdings erschweren die mitgebrachten indeterministischen Verzögerungen eine Analyse deutlich [But05]. Die zahllosen Abhängigkeiten der Ausführungszeit einer Komponente von dem zugrunde liegenden Betriebssystem, der Systemhardware, der Programmiersprache und dem Compiler lassen eine Analyse des Worst Cases wie ein „Fass ohne Boden“ erscheinen.

Hinzu kommt der Trend zu komplexen verteilten Systemen, welcher neben den Anforderungen an einzelne lokale Systeme zusätzliche Anforderungen an das Kommunikationsmedium stellt, um realistische und gültige Annahmen über das Zeitverhalten kommunizierender Systemkomponenten treffen zu können. Dieses Teilgebiet der Echtzeitsysteme ist auch unter dem Begriff der „Echtzeitkommunikation“ bekannt. Das größte Problem verteilter Systeme ist die im Allgemeinen sinkende Kontrollierbarkeit der Umgebung, die sich unmittelbar sowohl auf die lokalen Systeme selbst als auch auf das Übertragungsmedium und damit die Kommunikation innerhalb des Gesamtsystems auswirkt. Zum Beispiel bieten drahtlose Übertragungen keine räumliche Eingrenzung, wodurch äußere Störeinflüsse schwer einzuschätzen sind. Gerade in diesen Szenarien ist offen, was als realistischer Worst Case angesehen werden muss. Allgemein besteht daher die Auffassung, dass eine drahtlose Übertragung in sicherheitsrelevanten Szenarien ausscheidet, da die fehlende Kontrollierbarkeit keinerlei Garantien bezüglich der Einhaltung von Fristen zulässt [Kop91]. Andere Autoren – wie zum Beispiel Zöbel [Zöb08] – gehen allerdings davon aus, dass drahtlose Echtzeitnetze unter definierten Voraussetzungen zu realisieren sind und die steigende Nachfrage nach drahtlosen Netzen den Reifeprozess in diesem Teilgebiet der Echtzeitsysteme auch zukünftig vorantreiben wird.

Die vorgestellten Probleme zeigen, dass Echtzeitsysteme auch heute noch ein wichtiges Thema mit vielen offenen Fragen sind. Durch immer komplexer werdende Systemarchitekturen und den Trend zu verteilten Systemen wird dieses Thema auch in Zukunft nicht an Bedeutung verlieren. Mit steigender Rechenleistung lässt sich Geschwindigkeit auch in Zukunft nicht mit Vorhersagbarkeit gleichsetzen und Fristen werden auch dann nicht automatisch eingehalten [Sta88]. Des Weiteren fehlen Konzepte und Methoden, um modellgetriebene Entwicklungen von Echtzeitsystemen und eine Automatisierung des Entwicklungsprozesses zu ermöglichen.

2.1.2 Eigenschaften und Unterscheidungsmerkmale

In der jahrzehntelangen Geschichte von Echtzeitsystemen entstanden einige Erkenntnisse, Konzepte und Verfahren, die heute vielfach im Einsatz sind. Dabei lässt sich das Thema „Echtzeitsysteme“ nicht auf ein einzelnes Teilgebiet einschränken, sondern umfasst unter Anderem die Gebiete Systemarchitektur, Kommunikationsprotokolle, Scheduling und Fehlertoleranz. Aufgrund der Breite dieses Themas liegt der Fokus im Folgenden auf dem Teilgebiet der Planungsverfahren, d.h. auf dem Erstellen von Ablaufplänen (engl.: *Schedules*), die festlegen, zu welchen Zeitpunkten einzelne Aufgaben beginnen und spätestens enden müssen. In einem SDL-System wird diese Aufgabe von der SDL-Laufzeitumgebung erledigt, so dass die folgende Zusammenfassung in direktem Bezug zu den in Abschnitt 4.2.5 vorgestellten Maßnahmen zu sehen ist.

Echtzeitsysteme, wie sie in eingebetteten Systemen zu finden sind, sind häufig auf spezielle Szenarien zugeschnitten. Häufig profitieren sie von einem detaillierten a priori-Wissen und werden in einer Umgebung eingesetzt, in welcher der Stimulus der Umgebung bekannt ist oder stark eingegrenzt werden kann. Dadurch können (und müssen) sie sehr genau auf ihren Einsatzzweck zugeschnitten werden. Generische Lösungen sind kaum möglich. Dadurch ist die Entwicklung echtzeitfähiger Systeme häufig kostenintensiv, benötigt viel manuellen Programmieraufwand und kann kaum auf vorgefertigte Komponenten oder Automatisierungstechniken zurückgreifen [Sta88]. Verglichen mit Desktop- oder Serversystemen dürfen Echtzeitsysteme nicht als Systeme angesehen werden, bei denen verschiedene Aktivitäten untereinander konkurrieren. Stattdessen ist bei einem Echtzeitsystem der Code jeder Komponente bekannt und bildet einen kooperierenden Teil einer gemeinsamen Aufgabe [But05].

Echtzeitsysteme lassen sich nach Kopetz in zwei Kategorien unterteilen [Kop97, Kop91]: *Harte Echtzeitsysteme* sind insbesondere in sicherheitsrelevanten Szenarien zu finden und stellen die größte Herausforderung für die Entwicklung dar, da Fristen unter allen Umständen eingehalten werden müssen, um personelle und finanzielle Schäden zu verhindern. Deutlich einfacher zu realisieren sind *weiche Echtzeitsysteme*, da sie keine strikte Einhaltung der Fristen benötigen. Bei dieser Kategorie ist bereits die Einhaltung des durchschnittlichen Falls ausreichend.

In [Zöb08] werden Schäden, die aus aus einem Fehlverhalten des Systems resultieren können, in drei Stufen untergliedert. Diese drei Stufen ziehen absteigend einen unterschiedlichen Aufwand bei der Entwicklung nach sich: Bei möglichen Schäden für Leib und Leben, zum Beispiel bei ABS, ist der Aufwand am größten und aufwendige Qualitätskontrollen werden nötig. Schäden finanzieller Art, zum Beispiel bei einer Produktionsstraße, sind weniger sicherheitsrelevant. Hier hat aber die Industrie ein berechtigtes Interesse, solche Schäden zu vermeiden. Am unkritischsten sind Qualitätseinbußen, beispielsweise die Verschlechterung der Sprachqualität eines Telefongesprächs, als mögliche Folge der Nichteinhaltung von Fristen.

Echtzeitsysteme können mit zwei Ausführungsmodellen realisiert werden: *Zeitgesteuerte Ausführung* (time-triggered) vs. *ereignisgesteuerte Ausführung* (event-triggered) [Kop97, Kop91]. Bei

der zeitgesteuerten Ausführung kann bereits zur Übersetzungszeit (*compile time*) festgelegt werden, wann welche Aktivität startet und endet, wobei auch die Vermeidung von Verklemmungen (*deadlocks* oder *livelocks*) zur Übersetzungszeit verhindert werden muss. Sie hat demnach den Vorteil eines geringen Overheads zur Laufzeit, da die Bestimmung der rechnenden Aktivität einem einfachen Nachschlagen in einer Tabelle (*table look up*) entspricht. Die ereignisgesteuerte Ausführung ist hingegen deutlich flexibler, da es Aktivitäten mit dynamischer Priorität erlaubt, und kann auf Ereignisse von außen zum Beispiel in Form eines Interrupts schneller reagieren. Der Nachteil ist der erhöhte Overhead zur Laufzeit, da die rechnenden Aktivitäten dynamisch bestimmt werden müssen. Zusätzlich sind Analysen bezüglich der Einhaltung von Fristen schwieriger, da sie in Form von ausgiebigen Simulationen durchgeführt werden müssen. Dabei müssen in den Tests die schlimmstmöglichen Lastsituationen als Stimuli generiert werden, was dem Auftreten von *Event-Bursts* entspricht und möglicherweise kaum mit der realen Lastsituation übereinstimmt. Für Regelungssysteme und sicherheitskritische Applikationen allgemein hat sich daher das zeitgesteuerte Ausführungsmodell durchgesetzt [Kop91].

2.2 Häufig verwendete Planungsverfahren

Dieser Abschnitt fasst die wichtigsten Planungsverfahren aus dem Bereich der Echtzeitsysteme zusammen. Unter einem Planungsverfahren, oft auch als Schedulingstrategie bezeichnet, versteht man eine Menge von Regeln, die vorschreiben, in welcher Reihenfolge einzelne Aktivitäten eines Systems ausgeführt werden. Aktivitäten findet man häufig auch unter den Begriffen Prozess oder Aufgabe (*task*). Die Literatur unterscheidet zwischen *Schedulern*, die eine Sortierung der Aktivitäten gemäß ihrer Ausführungsreihenfolge vornehmen, und *Dispatchern*, welche den eigentlichen Kontextwechsel durchführen.

Verglichen mit gebräuchlichen Planungsverfahren aus dem Desktopbereich, bei denen Interaktivität und daher die Minimierung der durchschnittlichen Antwortzeit im Mittelpunkt steht, müssen Strategien für Echtzeitsysteme weitaus höhere Anforderungen erfüllen [Sta88]. In [Zöb08] wurde daher der Begriff „Echtzeitplanung“ (*real-time scheduling*) definiert, deren Aufgabe es ist, einzelne Aktivitäten einem Prozessor so zuzuteilen, dass die von der Anwendung herrührenden Zeitbedingungen unter allen Umständen eingehalten werden. Im Unterschied zu Strategien im Desktopbereich steht demnach das Einhalten von Fristen im Fokus der Echtzeitplanung. Beispiele für Planungsverfahren, deren Optimierungsziel hingegen die Minimierung der Wartezeit ist, sind *Round Robin* oder *Shortest-Job-First* [Tan09]. Da diese Verfahren jedoch für Echtzeitsysteme ungeeignet sind, finden sie im Folgenden keine weitere Beachtung.

Obwohl die Anzahl an Strategien, die in der Literatur zu finden sind, gewaltig ist, finden nur wenige Strategien in realen Produkten ihre Verwendung. Gründe hierfür sind die teilweise exotischen Anwendungsszenarien, aber andererseits auch der zu große Overhead, den die Verfahren zur Laufzeit erzeugen und sie damit unpraktikabel macht. Allgemein stehen Schedulingstrategien vor der Herausforderung, sowohl optimal zu sein, d.h. einen brauchbaren Plan zu finden, falls ein solcher existiert, aber andererseits auch eine niedrige Komplexität bezüglich Rechenzeit und Speicherverbrauch zu besitzen. Allerdings wurde nachgewiesen, dass viele Probleme der Echtzeitplanung NP-hart sind, d.h. dass es keine Lösungen geben kann, die unter allen Umständen effizient einen gültigen Plan liefern. Insbesondere wurde gezeigt, dass das Finden eines Schedules für eine Menge von Aufgaben, die sowohl Fristen als auch dynamische Prioritäten besitzen, NP-hart ist [But05]. Auch sind die meisten Planungsverfahren, die zusätzlich gemeinsamen Speicher und Semaphore oder ähnliche Sperr-Mechanismen beachten müssen, NP-

hart [SRL90]. Für hoch dynamische Systeme bedeutet dies, dass es die benötigten effizienten und adaptiven Schedulingalgorithmen nicht geben kann und stattdessen Heuristiken verwendet werden müssen, die keine Garantie bieten, dass ein möglicher Plan tatsächlich gefunden wird [Sta88].

Da die speziellen Aktionen einer Aktivität zum Finden eines Planes nicht von Belang sind, arbeiten Schedulingverfahren auf Modellen, die von konkreten Aktivitäten abstrahieren. Für jede Aktivität interessieren hierbei lediglich die Ausführungszeit, die Bereitzeit (Zeitpunkt, zu dem eine Aktivität ausführbar wird) sowie die Frist (Zeitpunkt, zu dem die Ausführung einer Aktivität spätestens abgeschlossen sein muss) [Zöb08]. Diese Parameter müssen zur Erstellung eines Planes bekannt sein, wobei sie nicht notwendigerweise zur Übersetzungszeit vorliegen müssen, sondern auch dynamisch zur Laufzeit bestimmt werden können. Insbesondere ist die Ausführungszeit in vielen Applikationen ein Parameter, der erst zur Laufzeit bekannt ist. Die Besonderheit von Echtzeitsystemen ist allerdings auch, dass viele dieser Charakteristiken häufig bereits a priori bekannt sind [Sta92]. Zur weiteren Vereinfachung findet bei der Modellbildung zusätzlich ein Diskretisieren der Zeit statt.

Viele Verfahren beschränken sich auf sehr einfaches Modelle, bei denen alle Aktivitäten periodisch mit bekannter Periodendauer ausgeführt werden und die Frist implizit der Periodendauer entspricht. Bei solchen Modellen kann eine zeitgesteuerte Ausführung stattfinden, die eine effiziente offline-Implementierung und -Analyse des Systems erlaubt. Erweiterte und realistischere Modelle unterteilen die Aktivitäten hingegen in drei Kategorien: Periodische Aktivitäten mit konstanter Periodendauer, aperiodische Aktivitäten ohne Angaben bezüglich der Ausführungshäufigkeit und sporadische Aktivitäten, die eine Untergruppe der schwer kontrollierbaren aperiodischen Aktivitäten bilden, indem sie eine Mindestdauer zwischen zwei aufeinanderfolgenden Ausführungen fordern [Zöb08, But05]. Für aperiodische und sporadische Aktivitäten müssen zur Analyse Hilfsmittel der Stochastik verwendet werden, wobei die Warteschlangentheorie (*queueing theory*) die theoretische Grundlage bildet. Als weitere Vereinfachung gehen viele Verfahren von voneinander unabhängigen Aktivitäten aus, d.h. die Aktivierung einer Aktivität ist nicht von dem Zustand oder den Aktionen einer anderen Aktivität abhängig [But05].

2.2.1 Unterscheidungskriterien von Planungsverfahren

Die Gemeinsamkeit aller Planungsverfahren ist deren Ziel, eine Menge von Prozessen gemäß einer Priorität zu sortieren und die Aktivität mit der höchsten Priorität zur Ausführung auszuwählen. Unterschiede der Verfahren bestehen unter Anderem in der Vergabe der Priorität an einzelne Aufgaben und dem Zeitpunkt, zu dem ein Plan erstellt wird, d.h. der Zeitpunkt, zu dem die Ausführungszeiten einzelner Aktivitäten bestimmt werden.

Planungsverfahren lassen sich in statische und dynamische Strategien unterteilen [LL73]. Statische Schedulingstrategien ordnen Aktivitäten einmalig Prioritäten zu, d.h. jede Aktivität besitzt eine feste Priorität. Bei dynamischen Verfahren ist hingegen die Priorität einer Aktivität von der Zeit abhängig. Verglichen mit festen Prioritäten sind diese Verfahren zwar flexibler, allerdings auch deutlich ineffizienter zur Laufzeit. Aus diesem Grund versuchen viele gängige Planungsverfahren (vgl. Rate Monotonic in Kapitel 2.2.3.1), dynamische Prioritäten zu vermeiden.

Bezüglich des Zeitpunkts der Planerstellung differenziert man zwischen *expliziten* Plänen und *impliziten* Plänen [Zöb08]. Bei expliziten Plänen wird genau festgelegt, wann welche Aktivität zur Ausführung kommt. Voraussetzung hierfür ist das a priori-Wissen über die Menge an Prozessen und deren Priorität, Ankunftszeit, Frist und Laufzeit. Unter diesen Voraussetzungen sind

explizite Pläne sehr gut mit einer zeitgesteuerten Ausführung vereinbar. In diesem Kontext wird häufig auch von *Offline-Schedulern* gesprochen, die sich über die Planerstellung zur Übersetzungszeit definieren [Sta88]. Insbesondere terminierende Systeme oder Systeme mit ausschließlich periodischen Aktivitäten sind für explizite Pläne prädestiniert.

Bei Systemen mit aperiodischen oder sporadischen Aktivitäten scheidet eine Anwendung von expliziten Plänen in den meisten Fällen aus. In diesen Fällen müssen Pläne implizit anhand von Regeln zur Ausführungszeit erstellt werden. Obwohl diese *Online-Scheduler* deutlich ineffizienter sind, ist deren Anwendung in dynamischen Szenarien, in denen unter Umständen neue Aktivitäten hinzukommen, unumgänglich.

Ein weiteres Unterscheidungsmerkmal von Planungsverfahren ist durch die Präemption der verwalteten Aktivitäten gegeben. Bei präemptiven Verfahren können Aktivitäten in ihrer Ausführung unterbrochen werden, um anderen Aktivitäten Vorrang zu geben. Nicht-präemptive Verfahren benötigen hingegen eine Kooperation der beteiligten Aktivitäten, da ihnen das Betriebsmittel, d.h. der Prozessor, nicht von außen entzogen werden kann. In modernen Desktop-Betriebssystemen sind vorrangig präemptive Planungsverfahren zu finden. Der bekannteste Vertreter eines präemptiven Verfahrens mit nicht-kooperierenden Aktivitäten ist Round Robin. Präemptive Verfahren finden häufig auch Pläne, wenn nicht-präemptive Verfahren keine gültige Ausführungsreihenfolge finden können. Falls allerdings alle Aktivitäten zum gleichen Zeitpunkt bereit werden, sind präemptive Verfahren äquivalent zu nicht-präemptiven. Der Overhead, der durch die Umschaltung zwischen den Aktivitäten entsteht, wird im Regelfall von den Planungsverfahren ignoriert. Besonders bei präemptiven Verfahren kann dieser Overhead allerdings deutlich ins Gewicht fallen und zu einer Verfälschung der Ergebnisse führen.

Zusammenfassend lässt sich festhalten, dass präemptive Planungsverfahren mit periodischen Aktivitäten bereits gut handhabbar sind. Es besteht allerdings ein Mangel an ausgereiften Methoden für nicht-präemptive Aktivitäten [Zöb08]. Die für Anwendungsszenarien „richtige“ Strategie lässt sich nur ermitteln, wenn genügend Vorwissen über die Anzahl und Art der Aktivitäten (statische vs. dynamische Prioritäten, Periodizität, maximale Laufzeit,...) vorhanden ist.

2.2.2 Bewertungskriterien

Die Vielzahl der Planungsverfahren entstand unter Anderem auch durch deren unterschiedliche Optimierungsziele [Tan09]. Die folgende Auflistung gibt eine Auswahl dieser Ziele:

- **Optimalität:** Ein Planungsverfahren ist optimal, wenn es für eine Menge von Aktivitäten mit Fristen, für die ein gültiger Plan existiert, auch einen solchen findet [Zöb08].
- **Fairness:** Ein Planungsverfahren ist fair, wenn keine Aktivität existiert, die nie zur Ausführung kommt.
- **Durchsatz:** Ein Planungsverfahren besitzt einen hohen Durchsatz, wenn der Prozessor möglichst viele Aktivitäten pro Zeiteinheit abarbeitet und nicht durch den Overhead des Planungsverfahrens oder durch häufige Wechsel der ausgeführten Aktivität bei präemptiven Planungsverfahren ausgelastet wird.
- **Antwortzeit:** Ein Planungsverfahren besitzt eine geringe Antwortzeit, wenn das System schnell eine von außen (zum Beispiel von einem Benutzer vor einem Terminal) erkennbare Reaktion zeigt. Insbesondere in interaktiven Systemen ist dieses Optimierungsziel am bedeutendsten. Es ist in der Regel aber widersprüchlich zu dem Ziel eines hohen Durchsatzes.

- **Komplexität:** Die Komplexität eines Planungsverfahrens ist ein Maß für den Overhead, den das Verfahren an Rechenzeit und Speicherverbrauch sowohl zur Analyse der Planbarkeit einer Menge von Aktivitäten benötigt, als auch für die Erstellung des Planes selbst. Bei Offline-Schedulern ist die Komplexität in der Regel zweitrangig, bei Online-Schedulern hingegen eines der primären Optimierungsziele.

2.2.3 Statische Planungsverfahren

Die statischen Planungsverfahren, d.h. das Planen nach festen Prioritäten, bilden eine bedeutende Klasse von Planungsverfahren im Bereich der Echtzeitsysteme. Auch wenn diese Klasse von Verfahren nicht die gleiche Auslastung erreicht wie das in Abschnitt 2.2.4 vorgestellte Planen nach dynamischen Prioritäten [Zöb08], besitzt das Planen nach festen Prioritäten einige Vorteile, die deren Anwendung in vielen Szenarien begünstigen. Der größte Vorteil ist die niedrige Rechenkomplexität von $O(1)$, mit denen ein Planen nach festen Prioritäten implementiert werden kann [But05]. Im Folgenden werden mit *Rate Monotonic* und *Deadline Monotonic* zwei der bekanntesten statischen Planungsverfahren vorgestellt [Zöb08, But05]:

2.2.3.1 Rate Monotonic

Das Planungsverfahren Rate Monotonic, welches auch heute durch seine Einfachheit und Effizienz nicht an Relevanz verloren hat, wurde bereits 1973 in der wegweisenden Publikation von Liu und Layland vorgestellt [LL73]. Es existieren sowohl präemptive als auch nicht-präemptive Versionen dieses Verfahrens.

Rate Monotonic arbeitet auf einer Menge von periodischen Aktivitäten, bei denen die Priorität so an die Aktivitäten vergeben wird, dass Aktivitäten mit kürzerer Periodendauer eine höhere Priorität erhalten. Da alle Aktivitäten periodisch sind, wiederholt sich ein gefundener Plan nach dem kleinsten gemeinsamen Vielfachen aller Perioden. Pläne können sowohl implizit zur Laufzeit als auch explizit festgelegt werden. Falls auch sporadische Aktivitäten mit Rate Monotonic verplant werden müssen, muss für diese Aktivitäten eine schlimmstmögliche Periodendauer ermittelt werden. In diesem Fall lohnt es sich auch, Rate Monotonic als implizites Planungsverfahren zu realisieren, da somit zur Laufzeit bestimmt werden kann, ob sporadische Aktivitäten tatsächlich zur Ausführung kommen müssen.

Für die präemptive Version von Rate Monotonic konnte gezeigt werden, dass dieses Planungsverfahren optimal innerhalb der statischen Planungsverfahren ist [LL73, Zöb08, But05]. D.h. wenn eine Strategie aus der Menge der Planungsverfahren mit fester Priorität einen gültigen Plan liefert, dann findet auch Rate Monotonic einen Plan. Ebenso existiert für die präemptive Version von Rate Monotonic ein einfacher Schedulability-Test, d.h. ein einfacher Nachweis, dass eine Menge von Aktivitäten mit Rate Monotonic verplant werden kann. Im Konkreten besagt der Test, dass bei einer Auslastung $\leq ln(2)$, immer ein gültiger Plan gefunden wird, wobei die Auslastung bei einer Menge von N Aktivitäten definiert ist als

$$\sum_{i=0}^{N-1} \frac{e_i}{p_i} \tag{2.1}$$

mit der Berechnungszeit e_i und der Periodendauer p_i einer Aktivität i. Da dieser Test ein pessimistischer Test ist, kann in vielen Fällen auch ein gültiger Plan gefunden werden, obwohl der einfache Schedulability-Test ein negatives Ergebnis liefert.

2.2.3.2 Deadline Monotonic

Deadline Monotonic stellt eine Weiterentwicklung von Rate Monotonic dar und beachtet Fälle, in denen die Frist einer Aktivität nicht unmittelbar mit der Periodendauer übereinstimmen muss [But05]. Anstatt in Abhängigkeit zu der Periodendauer werden Prioritäten bei Deadline Monotonic invers proportional zu der relativen Frist der einzelnen Aktivitäten vergeben, d.h. die Aktivität mit der kürzesten Frist erhält die höchste Priorität.

Für die präemptive Version wurde ebenso Optimalität nachgewiesen, d.h. wenn ein Planungsverfahren mit Aktivitäten, deren Periodendauer nicht der Frist entspricht, einen Plan findet, dann findet auch Deadline Monotonic einen gültigen Plan. Der größte Nachteil des Verfahrens ist allerdings der aufwändige Schedulability-Test.

2.2.4 Dynamische Planungsverfahren

Dieser Abschnitt behandelt dynamische Planungsverfahren, bei welchen die Priorität einer Aktivität abhängig von der Zeit ist. Im Gegensatz zu den statischen Planungsverfahren verursachen diese Verfahren einen höheren Berechnungsaufwand zur Laufzeit, da der Scheduler die Prioritäten einzelner Prozesse zur Laufzeit aktualisieren muss [But05]. Sie haben allerdings den bedeutenden Vorteil einer größeren Flexibilität und können eine höheren Systemauslastung als statische Verfahren erreichen, da sie in Situationen Pläne finden können, in denen statische Verfahren versagen. Zwei der bekanntesten Vertreter dieser Klasse von Planungsverfahren sind *Earliest-Deadline-First* (EDF) und *Least-Laxity-First* (LLF), welche im Folgenden vorgestellt werden.

2.2.4.1 Earliest-Deadline-First (EDF)

Der bekannteste Vertreter der dynamischen Planungsverfahren ist Earliest-Deadline-First (EDF), welches bereits 1973 von Liu und Layland vorgestellt wurde [LL73]. Die zentrale Idee hinter EDF ist das Sortieren von Aktivitäten hinsichtlich ihrer Fristen, so dass die Aktivität mit der frühesten Frist die höchste Priorität erhält. Bei identischer Bereitzeit aller Aktivitäten wird das Verfahren, das einen Plan nach den gleichen Regeln erstellt, auch mit *Earliest Due Date (EDD)* bezeichnet.

Für die nicht-präemptive Version von EDF kann anhand einfacher Beispiele Nicht-Optimalität nachgewiesen werden. Grund hierfür ist, dass eine Aktivität mit später Frist und einer frühen Bereitzeit andere Aktivitäten mit späterer Bereitzeit aber früherer Frist blockieren kann. Hingegen ist die präemptive Version optimal in der Menge aller Planungsverfahren [Zöb08, But05], so dass eine Auslastung des Prozessors bis zu 100% möglich ist. Dementsprechend entspricht der Schedulability-Test der Prüfung, ob die Auslastung ≤ 1 ist, wobei der Test den Overhead durch Kontextwechsel außer Acht lässt.

Obwohl EDF nicht mit konstantem Berechnungsaufwand zu implementieren ist, ist eine effiziente Implementierung mit einem balancierten binären Baum möglich, was in einer Komplexität bezüglich des Rechenaufwands von $O(log\, n)$ resultiert [But05].

2.2.4.2 Least-Laxity-First (LLF)

Ein weiteres bekanntes dynamisches Planungsverfahren ist Least-Laxity-First (LLF), welches im Deutschen auch mit „Planen nach Spielräumen“ bezeichnet wird [Zöb08]. Der *Spielraum* definiert hierbei die Zeitspanne, die eine Aktivität innerhalb des Zeitintervalls zwischen ihrer Bereitzeit und Frist abzüglich ihrer Ausführungszeit ungenutzt lässt. Die Vergabe der Prioritäten

erfolgt invers zu den Spielräumen, d.h. die Aktivität mit dem geringsten Spielraum erhält die höchste Priorität. Bei gleichen Spielräumen erhält die Aktivität mit der früheren Frist Vorrang.

Das Verfahren existiert in einer präemptiven und einer nicht-präemptiven Version. Verglichen mit EDF hat die präemptive Version den Nachteil, dass es zu häufigen Umschaltvorgängen zwischen verschiedenen Aktivitäten kommen kann. Grund hierfür sind die Spielräume der wartenden Aktivitäten, die sich permanent während der Ausführung einer anderen Aktivität verringern können. Wegen der Abhängigkeit der Spielräume zu den Prioritäten kann sich somit die ausgeführte Aktivität nach jeder Zeiteinheit ändern, was im ungünstigen Fall zu einem „Flattern“ von Aktivitäten führt.

Trotz dieses Nachteils ist die präemptive Version von LLF optimal und damit identisch mit der Optimalität von EDF[1]. Für die nicht-präemptive Version gilt ebenso wie für EDF, dass das Verfahren nicht optimal ist. LLF ist in diesem Fall sogar schwächer als EDF, da es selbst bei gleichen Bereitzeiten aller Aktivitäten nicht optimal ist [Zöb08].

In Multiprozessor-Systemen ist allerdings weder LLF noch EDF optimal [Kop91], wobei es in diesen Fällen Szenarien gibt, in denen LLF gültige Pläne findet, EDF jedoch nicht.

2.3 Prioritätsinvertierung

Die bisher betrachteten Planungsverfahren gingen implizit von unabhängigen Aktivitäten aus. Dieser Abschnitt geht nun auf die Probleme ein, die bei der Nutzung von gemeinsamem Speicher auftreten, und fasst die einschlägigen Lösungsansätze zusammen.

2.3.1 Problemstellung

In nahezu allen Systemen dürfen Aktivitäten nicht als unabhängige Entitäten angesehen werden, sondern sind Teil eines komplexen Gesamtsystems. Hieraus ergibt sich eine notwendige Interaktion zwischen Aktivitäten, die in einer kontrollierten und synchronisierten Weise geschehen muss. Häufig findet die Interaktion auf gemeinsamem Speicher statt und wird mit Hilfe von kritischen Abschnitten geregelt [Tan09]. Zur Realisierung von kritischen Abschnitten wurden zahlreiche Konzepte vorgestellt, die sich in nebenläufigen Softwaresystemen etabliert haben. Eine Auswahl dieser bekannten Konzepte sind Mutexe, Semaphore [Dij65] und Monitore [Hoa74].

Ein Planungsverfahren für interagierende Aktivitäten hat demnach nicht nur die Aufgabe, die Aktivitäten gemäß ihren Prioritäten bzw. Fristen zu sortieren, sondern muss ebenso die Interaktion berücksichtigen. Im Konkreten ergeben sich folgende Ziele:

- Beachtung von Prioritäten: Die Ausführung einer ausführbaren, hochprioren Aktivität muss bevorzugt werden.
- Wahrung der Datenkonsistenz: Eine Aktivität, die sich in einem kritischen Abschnitt befindet, muss den kritischen Abschnitt beenden, bevor eine andere Aktivität diesen betritt.

Die Ziele zeigen, dass die Ausführung höherpriorer Aktivitäten unter Umständen zurückgestellt werden muss, falls eine nierderpriore Aktivität im Besitz eines kritischen Abschnittes ist, den die höherpriore Aktivität betreten muss. Allerdings sind Maßnahmen erforderlich, welche die Blockierungszeit der hochprioren Aktivität gering halten. Ohne zusätzliche Maßnahmen wären

[1] Trotz identischer Optimalität liefern EDF und LLF im Allgemeinen unterschiedliche Pläne.

auch Konstellationen möglich, in welchen höherpriore Aktivitäten von niederprioren Aktivitäten für unbegrenzte Zeit blockiert werden [SRL90].

2.3.2 Lösungsansätze

Im Folgenden werden zwei Lösungen zur Vermeidung von unbegrenzter Blockierung hochpriorer Aktivitäten zusammengefasst:

2.3.2.1 Das Priority Inheritance Protocol (PIP)

Die bekannteste Lösung ist das *Priority Inheritance Protocol* [SRL87, SRL90], für welches im Laufe der Zeit auch viele Erweiterungen und Variationen entstanden sind. Die zentrale Idee des Protokolls ist die Vererbung von Prioritäten hochpriorer Aktivitäten an niederpriore Aktivitäten [Zöb08, But05]. Sobald eine Aktivität einen kritischen Abschnitt betreten will und dieser bereits durch eine andere Aktivität blockiert ist, wird die Priorität der blockierenden Aktivität auf die Priorität der blockierten Aktivität angehoben. Die Priorität einer Aktivität, die einen kritischen Abschnitt besitzt, entspricht damit dem Maximum der Prioritäten aller Aktivitäten, die auf Freigabe desselben kritischen Abschnittes warten.
Obwohl das Priority Inheritance Protocol ein unbegrenztes Blockieren hochpriorer Aktivitäten verhindert, besitzt es zwei gravierende Nachteile: Erstens können Blockierungsketten entstehen, die zwar in ihrer Länge begrenzt sind, aber dennoch eine lange Wartezeit für hochpriore Aktivitäten zur Folge haben. Der zweite Nachteil des Protokolls sind Deadlocks, die während der Ausführung auftreten können. Das Protokoll fördert zwar keine Deadlocks, d.h. die Wahrscheinlichkeit für einen Deadlock wird durch Verwendung des Protokolls nicht unmittelbar höher, trägt aber auch nicht zu deren Vermeidung bei.

2.3.2.2 Das Priority Ceiling Protocol (PCP)

Motiviert von den Nachteilen des Priority Inheritance Protocols entstand das *Priority Ceiling Protocol* [GS88]. Durch eine Offline-Analyse der Programme einzelner Aktivitäten gelingt es dem Protokoll sowohl Deadlocks als auch transitive Blockierungen zu verhindern.
Im Gegensatz zu der Vererbung von Prioritäten zwischen Aktivitäten, ordnet das Priority Inheritance Protocol jedem Semaphor eine Priorität zu [But05]. Diese Priorität entspricht dem Maximum der Prioritäten aller Aktivitäten, die das Semaphor in ihrer Ausführung verwenden bzw. verwenden könnten. Eine Aktivität darf einen kritischen Abschnitt nur betreten, wenn ihre Priorität höher ist als das Maximum der Prioritäten aller zur Zeit betretenen kritischen Abschnitte. Dadurch wird eine an einem kritischen Abschnitt blockierte Aktivität maximal einmal von einer niederprioreren Aktivität blockiert. Blockierungsketten werden demnach verhindert und auch Deadlocks sind aufgrund einer unglücklichen Verschachtelung kritischer Abschnitte nicht mehr möglich, da zirkuläre Abhängigkeiten erst gar nicht entstehen können.
Der Nachteil des Priority Ceiling Protocols ist der Aufwand, der durch die notwendige Analyse der Aktivitätenprogramme entsteht [Zöb08]. In der Analyse muss untersucht werden, welche Aktivität welchen kritischen Abschnitt betreten kann, um die Prioritäten der Semaphore zu ermitteln. Hierbei kann es auch zu einer *Verschiebung* von Prioritäten kommen, wenn hochpriore Aktivitäten existieren, die nur in Ausnahmefällen einen kritischen Abschnitt betreten. In diesem Fall würde der kritische Abschnitt die hohe Priorität erhalten, obwohl die Priorität der Aktivitäten, die diesen Abschnitt in der Regel betreten, geringer wäre.

3. KAPITEL

SDL – Eine erweiterte Übersicht

Die *Specification and Description Language*, kurz SDL, ist eine von der ITU-T standardisierte Spezifikations- und Beschreibungssprache für reaktive Systeme, die sich insbesondere in den Bereichen Telekommunikation und Netzwerkprotokolle etabliert hat [Int99]. Durch hierarchische Strukturierungsmöglichkeiten bietet sich SDL für die Entwicklung komplexer nebenläufiger Systeme an, deren Implementierungen automatisiert mit Hilfe von Werkzeugen anhand von SDL-Modellen erstellt werden können.

Dieses Kapitel gibt eine Einführung in die Konzepte von SDL. Nach einer Vorstellung der Schlüsselaspekte in Abschnitt 3.1 wird eine Übersicht über Syntax und Semantik der Sprache gegeben. In Abschnitt 3.2 werden anschließend Probleme bezüglich der Handhabung zeitkritischer Aufgaben und die Eignung von SDL für Echtzeitsysteme diskutiert, indem die Anwendbarkeit bekannter Konzepte der „Echtzeitwelt“ auf SDL-Modelle untersucht wird. Abschließend werden in Abschnitt 3.3 existierende Ansätze zur Erhöhung der Vorhersagbarkeit und Echtzeitfähigkeit von SDL-Spezifikationen vorgestellt.

3.1 Die Sprache SDL

SDL ist keine Sprache, die erst in den letzten Jahren entstand [Ree00]. Sie blickt mittlerweile auf eine über 30jährige Geschichte zurück, in welcher insgesamt sechs Standards veröffentlicht wurden. Die erste Version wurde 1976 in Form einer 23-seitigen Empfehlung publiziert. Mit SDL-88 entstand der erste SDL-Standard mit einer formalen Semantik [EGG+01]. Die aktuelle Version des Standards ist SDL-2000 [Int99] und beinhaltet in seinem Anhang eine formale statische und dynamische Semantik, die auf der mathematischen Grundlage von *Abstract State Machines* (ASMs) basiert [Int00]. Insbesondere die formale Semantik, die das Verhalten eines SDL-Systems bis auf explizit eingeführten Indeterminismus vorhersagbar und eindeutig beschreibt, wird als Vorteil im Vergleich zu der Unified Modelling Language (UML, [OMG10]) angesehen[2]. Die voraussichtlich nächste Version des SDL-Standards ist SDL-2010 [Ree09], welche die Kompatibilität von SDL zu gebräuchlichen Programmiersprachen, wie zum Beispiel Java oder C, erhöhen soll und die Bildung einer Untermenge des mittlerweile sehr umfangreichen Sprachstandards vorschlägt.

[2] Seit UML 2.0 und mit der Einführung von UML-Profilen gibt es anhaltende Bestrebungen, eine formale Semantik für UML-Modelle zu definieren.

Während der Entwicklung von SDL stellte man sich der Herausforderung, eine Sprache zu entwickeln, die sowohl visuell lesbar (*human readable*) und einfach zu erlernen ist als auch von Werkzeugen effizient und eindeutig verarbeitet werden kann [Int99]. Insbesondere durch die formale Syntax und Semantik ist die Eindeutigkeit des Verhaltens gewährleistet [PST07]. Durch Möglichkeiten zur Strukturierung sollte die Sprache auch für komplexe Systeme geeignet sein und Wiederverwendbarkeit und Modularisierung fördern. Daraus resultierte eine Sprache, die dem modellgetriebenen Ansatz folgend durch den kompletten Entwicklungsprozess führt [Got07]. SDL erlaubt die plattformunabhängige Spezifikation von Systemen, für welche anschließend automatisierte Implementierungen für unterschiedliche Plattformen erzeugt werden können. Dadurch können die gleichen SDL-Systeme als Grundlage zur funktionalen Simulation und zur Codegenerierung für den Einsatz auf realer Hardware verwendet werden. Ermöglicht und unterstützt wird der Entwicklungsprozess mit SDL von zahlreichen Werkzeugen wie den kommerziellen Programmen IBM Tau [IBMar], Pragmadev RTDS [Praar] und Cinderella [Cinar] oder dem Transpiler *ConTraST* aus dem akademischem Umfeld [FGW06]. Die für diese Masterarbeit relevanten Werkzeuge sind in Anhang A zusammengefasst.

3.1.1 Schlüsselaspekte

SDL-Systeme basieren auf asynchron kommunizierenden, erweiterten endlichen Zustandsautomaten [San00]. Explizit definierte Zustände und Werte von Variablen bilden den aktuellen Zustand eines Automaten. Zustandstransitionen werden durch den Empfang von Signalen ausgelöst[3], die der asynchronen Kommunikation zwischen verschiedenen Komponenten und der Umgebung eines SDL-Systems dienen.

In einem SDL-System finden sich fünf Schlüsselkonzepte [PvL03a, PvL03b]:

- **Agenten** stellen die aktiven Objekte in einem SDL-System dar, die ein Programm in Form eines Zustandsautomaten abarbeiten. Sie haben definierte Schnittstellen, die angeben, welche Signale ein Agent empfangen und senden kann.
- **Kanäle** verbinden Agenten und legen fest, welche Agenten mit welchen Signalen miteinander kommunizieren können.
- **Signale** dienen der Kommunikation zwischen Agenten und werden in den Kanälen transportiert. Bevor Signale von einem Agenten konsumiert werden und einen Zustandsübergang bewirken, werden sie in der Warteschlange (*input port*) des empfangenden Agenten gespeichert. Die Kommunikation zwischen Agenten erfolgt asynchron.
- **Daten** werden in Variablen gespeichert und in Signalen transportiert. SDL besitzt vordefinierte Datentypen (zum Beispiel Integer), Generatoren (zum Beispiel Arrays) und erlaubt außerdem die Definition neuer Datentypen.
- **Aktionen** können die Werte von Variablen ändern, Signale versenden, Timer verwalten oder neue Agenten erzeugen. Sie stellen den „Rumpf“ von den Transitionen in den Zustandsautomaten eines Agenten dar.

Zusätzlich sind im Sprachumfang von SDL noch weitere Funktionen enthalten, von denen allerdings viele auf diese fünf Schlüsselkonzepte abgebildet werden.

[3] Signale sind in den meisten SDL-Systemen der häufigste Grund für Zustandsübergänge. Neben Signalen bietet SDL noch weitere Sprachkonstrukte (zum Beispiel spontaneous transitions), die einen Zustandswechsel bewirken können.

3.1.2 Syntax

Unter der Syntax einer Spezifikationssprache versteht man die Menge an Regeln und Randbedingungen, die zur Formulierung einer Spezifikation, welche dieser Sprache folgt, eingehalten werden müssen [Pri00]. Damit gibt die Syntax einer Sprache keine Informationen über die Bedeutung einer Wortfolge, legt aber fest, welche Wortfolgen in einer Sprache erlaubt sind. Zur Definition von Syntax wird in der Regel auf formale Grammatiken zurückgegriffen.

SDL-Spezifikationen lassen sich sowohl graphisch mit SDL/GR als auch textuell mit SDL/PR beschreiben, welche in dem SDL-Standard Z.100 festgelegt sind [Int99]. Da der Sprachumfang von SDL/PR und SDL/GR (nahezu) äquivalent ist, lassen sich beide Repräsentationen ineinander überführen. SDL/GR wurde entwickelt, um den Entwicklern eine anschauliche Möglichkeit zur Spezifikation zu geben. Vor einer automatisierten Weiterverarbeitung, zum Beispiel vor einer syntaktischen Analyse oder der Erzeugung von Programmcode, findet eine Umwandlung in SDL/PR statt. Die Syntax von SDL/PR wurde unter Verwendung einer Backus-Naur-Form definiert, welche einer Typ-2-Grammatik (kontextfreie Grammatik) gemäß der Chomsky-Hierarchie entspricht [SSH92, PvL03b]. Neben SDL/PR und SDL/GR existiert seit SDL-96 ein weiteres Darstellungsformat für SDL, das *Common Interchange Format* (CIF), welches in Z.106 des Standards definiert ist. Aufgabe dieses Formates ist es, den Austausch von SDL-Spezifikationen zwischen unterschiedlichen Werkzeugen zu ermöglichen, ohne dabei Informationen über die graphische Darstellung (exakte Lage von Strichen/Rechtecken, etc.) zu verlieren. Dies ist nötig, da SDL/GR zwar geometrische Formen einführt, allerdings nicht auf die Größe oder Position eingeht.

Die Syntax von SDL enthält viele Sprachelemente, die ausschließlich Abkürzungen darstellen und auch mit Hilfe anderer Sprachelemente ausgedrückt werden können. In diesem Zusammenhang spricht man auch von „syntaktischem Zucker“ (*syntactic sugar*). Beispiele solcher Abkürzungen sind Signallisten oder entfernte Prozeduren (*remote procedures*). Die im folgenden beschriebene Semantik bildet diese Abkürzungen auf Kernelemente der Sprache ab.

3.1.3 Semantik

Eine der am häufigsten hervorgehobenen Vorteile von SDL im Vergleich zu anderen Spezifikationssprachen ist durch das mathematisch-formale Modell gegeben, welches die Semantik einer SDL-Spezifikation definiert und der Ausführung eines SDL-Systems zugrunde liegt. Hierbei ist die Semantik einer Spezifikationssprache definiert als die Menge von Randbedingungen, welche sie beschreibt, und setzt eine syntaktisch korrekte Spezifikation voraus [Pri00, PvL03a, PvL03b, EGG+01].

Die Semantik von SDL besteht aus zwei Teilen: Die *statische Semantik* und die *dynamische Semantik*, welche in Anhang F.2 bzw. Anhang F.3 des Z.100 Standards definiert sind [Int00].

Ausgangspunkt der statischen Semantik ist die syntaktisch korrekte SDL-Spezifikation, die auch als *konkrete Syntax* bezeichnet wird [Pri00, PvL03a, PvL03b, EGG+01]. Darauf aufbauend gibt die statische Semantik Bedingungen in der *Prädikatenlogik erster Ordnung* vor, die aussagen, ob die Spezifikation wohlgeformt und damit gültig ist (*well-formedness conditions*). Hierunter fällt zum Beispiel die Prüfung, ob für einen Signalbezeichner auch eine Signaldefinition vorliegt, d.h. ob ein verwendetes Signal zuvor auch deklariert wurde, oder die Verträglichkeit von Operatoren. Diese Bedingungen sagen zwar nichts über das Verhalten des Systems aus und können ohne Interpretation überprüft werden, lassen sich aber nicht mit einer kontextfreien Grammatik beschreiben und sind daher nicht der Syntax der Sprache zuordenbar. Ein zweiter Teil der

statischen Semantik beschreibt Transformationen anhand von Ersetzungsregeln (*rewrite rules* [BK86]), die eine Ersetzung von abkürzenden Sprachelementen (*short hands*) in Kernelemente der Sprache vornehmen. Diese Ersetzungen haben das Ziel, die Menge an Sprachkonstrukten, für welche anschließend eine dynamische Semantik definiert wird, klein zu halten.

Das Resultat der statischen Semantik ist die *abstrakte Syntax*, welche aus der konkreten Syntax mit den entsprechenden Transformationen und dem Entfernen irrelevanter Zeichen (Separatoren, überflüssige Leerzeichen, ...) entsteht. Die abstrakte Syntax ist die Eingabe für die dynamische Semantik, die auf der statischen Semantik aufbauend das Verhalten einer Spezifikation zur Laufzeit definiert und damit das Ziel der Ausführbarkeit verfolgt, d.h. das Schaffen eines operationalen Formalismus [EGG+01]. Zur weiteren Beschreibung der Semantik unterteilt man die abstrakte Syntax in drei Teile [Pri00]:

- Die *Struktur* beschreibt den Zusammenhang zwischen Kanälen, Signalen und Agenten und wird bei Systemstart bei dem SDL-System ausgehend rekursiv entfaltet.
- Das *Verhalten* – zum Beispiel die Aktionen innerhalb einer SDL-Transition – wird in Primitiven der *SDL Abstract Machine* (SAM), welche eine logische Hardwareplattform darstellt, übersetzt.
- Die *Daten* werden gekapselt, damit das Daten-Modell bei Bedarf ausgetauscht werden kann.

Die theoretische Grundlage der Beschreibung der dynamischen Semantik basiert auf verteilten Abtract State Machines (ASMs) [BS03]. Das Verhalten jedes ASM-Agenten wird durch ein ASM-Programm beschrieben, das aus Transitionsregeln besteht. Eine Transitionsregel ändert in einem atomaren Schritt (in ASM-Terminologie: *move*) den Wert von Variablen (in ASM-Terminologie: *locations*) [GP05]. Die Ausführung unterschiedlicher ASM-Agenten erfolgt nebenläufig, wobei durch Interaktion entstehende Konflikte durch die Definition von partiell-geordneten Abläufen (*partially ordered runs*) und der Gültigkeit einer Kohärenzbedingung gelöst werden [EGG+01, GP05]. Die nebenläufige Ausführung von Agenten resultiert daher in einem *run*, d.h. in einer Menge von *moves*, die partiell angeordnet sind. Die Auswahl und Ausführung von Agenten wird von der *SDL Virtual Machine* (SVM) vorgenommen, welche die Laufzeitumgebung bereitstellt und die übersetzte und initialisierte SDL-Spezifikation auf der SDL Abstract Machine (SAM) ausführt.

Der enorme Aufwand zur Definition einer formalen Semantik resultiert in einigen Vorteilen, die SDL aufweisen kann. Zum Beispiel erlaubt die formale Semantik das Prüfen von Eigenschaften der Spezifikation (Deadlocks, Livelocks, ...) ohne das Vorliegen einer konkreten Implementierung, wobei sich an der Unentscheidbarkeit mancher Probleme selbstverständlich nichts ändert [Pri00]. Des Weiteren lassen sich aufbauend auf der Semantik SDL-Systeme in ASM-Regeln übersetzen, die anschließend mit Hilfe von entsprechenden ASM-Compilern ausgeführt werden können [PvL03a, Resar]. Ein weiterer Vorteil ist das vereinfachte Testen der Korrektheit von Werkzeugen und der Implementierung, da die Konsistenz zwischen Standard und der zu untersuchenden Implementierung einfach nachzuvollziehen ist. Transformationen von SDL-Modellen in eine Programmiersprache werden dadurch vereinfacht und – bis auf explizite Ausnahmen von Indeterminismus – eindeutig [San00].

Diese – im Standard zunächst formal und abstrakt dargelegte Vorgehensweise – erfolgt auch für Implementierungen auf realer Hardware. Die dabei entstehenden Eigenheiten und Probleme sind Thema des folgenden Abschnittes.

3.2 SDL – Eine Sprache für Echtzeitsysteme?

Die bisherige Betrachtung von SDL beschränkt sich auf den formalen Standard, der die theoretische Grundlage zu einer Validierung und Verifikation bildet. Allerdings gibt es heutzutage viele Entwicklungsprozesse, in denen SDL-Systeme nicht nur zur Analyse von Eigenschaften der entwickelten Systeme eingesetzt werden, sondern in denen SDL durch den kompletten Entwicklungsprozess leitet und Implementierungen für unterschiedliche Hardware-Plattformen aus SDL-Spezifikationen automatisiert generiert werden [Got07]. SDL wurde jedoch als Spezifikations- und Beschreibungssprache entworfen, nicht als Programmiersprache [MT00]. An vielen Stellen des SDL-Standards ist erkennbar, dass Implementierungen von SDL-Systemen nicht das vorrangige Ziel bei der Entwicklung der Sprache waren und die Semantik weniger auf die Codegenerierung abzielt, sondern auf Simulation und Verifikation [BGK+00]. Die Probleme bei automatisierten Implementierungen und die Eignung von SDL für Echtzeitsysteme sind Gegenstand dieses Abschnittes.

3.2.1 Formale Modelle als Grundlage für Echtzeitsysteme?

Ein wissenschaftliches Schlüsselproblem im Bereich der Echtzeitsysteme ist für Stankovic die Problematik, dass viele (formale) Spezifizierungs- und Verifikationstechniken auf der Abstraktion von Implementierungsdetails basieren [Sta88, Sta92]. Stankovic kritisiert, dass unter dem Begriff „Korrektheit“ oft nur funktionale Aspekte verstanden werden und sich die Verifikation auf eine rein qualitative Verifikation beschränkt. In Echtzeitsystemen definieren aber gerade die nicht-funktionalen Aspekte, wie Implementierungsdetails und die Umgebung des Systems, die zeitlichen Randbedingungen und sind ausschlaggebend für die Korrektheit eines Systems.

Die Aussagen von Stankovic sind auf allgemeine Entwurfstechniken bezogen. In [ÁDL+03] und [Ger97] werden diese Probleme SDL betreffend konkretisiert. Insbesondere wird der Mangel an fehlenden Analysemöglichkeiten bezüglich des Echtzeitverhaltens einer SDL-Spezifikation aufgeführt. Ohne diese Möglichkeiten und die Beachtung von Performanzaspekten ist eine Überprüfung der Korrektheit allerdings nur bedingt sinnvoll, da sich das Verhalten eines Systems in einer realen Umgebung grundlegend unterscheiden kann [Ger97].

Beispiele, dass SDL primär nicht als Implementierungssprache entwickelt wurde, finden sich vielfach in der formalen Semantik der Sprache. Listing 3.1 zeigt ein Prädikat aus dem SDL-Standard [Int00], das bei der Ausgabe eines Signals innerhalb einer SDL-Transition und bei jeder Weiterleitung des Signals entlang des Signalpfades ausgewertet wird. Die Vielzahl an Quantoren, die für eine Implementierung in der SDL-Laufzeitumgebung zu realisieren und zu optimieren sind, ist ein Indiz dafür, dass die funktionale Korrektheit und nicht die Effizienz bei der Entstehung des SDL-Standards im Vordergrund stand.

```
1  Applicable(s: SIGNAL, toArg: TOARG, viaArg: VIAARG, g: GATE, l: LINK): BOOLEAN =def
2    ∃ commPath ∈ { lSeq ∈ LINK*: (lSeq ≠ empty ∧ Connected(lSeq.head, lSeq.tail))}:
3      (∀ l in commPath: s ∈ l.with ∧ l.owner ≠ undefined ∧ commPath.head.from = g ∧
4      if l ≠ undefined then commPath.head = l endif ∧
5      ¬ ∃ l ∈ LINK: (l.from = commPath.last.to ∧ s ∈ l.with) ∧ // the path is complete
6      viaArg ⊆ commPath.commPathIds ∧
7      if toArg ∈ Agent−identifier then
8        commPath.last.to.myAgent.nodeAS1.nodeAS1ToId = toArg endif ∧
9      if toArg ∈ PID ∧ toArg ≠ null then
10       ∃ sa ∈ AGENT: (sa.owner = commPath.last.to.myAgent ∧ sa = toArg.s–AGENT) endif )
```

Listing 3.1: Beispiel eines Prädikates aus dem SDL-Standard [Int00].

Die Beschreibung der dynamischen Semantik von SDL in Form von *Abstract State Machines* hat zwar den Vorteil, dass sie das Verhalten eines SDL-Systems formal definiert, jedoch kommt es bei dem Einsatz eines SDL-Systems auf einer konkreten Hardware-Plattform immer zu einem Bruch zwischen der formalen Semantik und der Implementierung. Insbesondere die unterschiedlichen Ziele von SDL-Standard (formale Semantik und funktionale Korrektheit) und Implementierung (Effizienz bei beschränkten Ressourcen und Vorhersagbarkeit) sind die Hauptgründe für diesen Bruch.

3.2.2 Planungsverfahren für SDL

Für die Ausführung von SDL-Systemen werden die strukturellen Elemente einer SDL-Spezifikation, wie SDL-Prozesse und Signalrouten, durch ASM-Agenten repräsentiert. Der SDL-Standard unterscheidet zwischen drei Typen von Agenten: *Linkagenten*, *SDL-Agenten* und *SDL-Agentenmengen* [Int00]. Ihr Verhalten ist anhand von ASM-Programmen beschrieben. Gemäß der Semantik von verteilten ASMs (*distributed* ASMs) entscheidet jeder ASM-Agent selbst und unabhängig von anderen ASM-Agenten über seine Ausführung und die Abarbeitung seines Programms. Das Ausführungsmodell des SDL-Standards geht demzufolge von einer asynchronen parallelen Ausführung aller ASM-Agenten aus.
In der Realität ist es unüblich und aus Kostengründen nahezu unmöglich, dass alle ASM-Agenten parallel auf getrennten physikalischen Objekten abgearbeitet werden [BH93]. Bei der Abbildung dieses Ausführungsmodells auf reale Hardware muss die vollständige Parallelität in der Regel aufgegeben werden. Eine Serialisierung der Ausführung von ASM-Agenten wird notwendig, um die Agenten beispielsweise auf einem Einkernprozessor auszuführen. Bezüglich der Reihenfolge, in der einzelne Agenten ausgeführt werden, lässt das ASM-Ausführungsmodell sehr viele Freiräume. Prinzipiell gilt hier die Regel, dass ein ASM-Agent ausgeführt werden *kann*, aber nicht *muss*. Diese Eigenschaft gibt der Laufzeitumgebung viele Freiheiten in Bezug auf das angewandte Planungsverfahren. Die Kehrseite davon ist jedoch ein indeterministisches Verhalten bei dem Einsatz des Systems auf realer Hardware, das stark von der Laufzeitumgebung und dem verwendeten Planungsverfahren abhängt. So wird in [ÁDL+03] beispielsweise kritisiert, dass es SDL an Möglichkeiten fehlt, Prioritäten auf struktureller Ebene an Prozesse oder Signale zu vergeben, um damit zur Laufzeit die Ausführungsreihenfolge der Agenten zu beeinflussen.
Die Freiheiten in der Ausführung von ASM-Agenten beziehen sich nicht nur auf die Reihenfolgen der Agenten, sondern ebenfalls auf die Unterbrechbarkeit. Es gilt zwar eine *run-to-completion*-Semantik für Transitionen, d.h. ein Agent kann die Ausführung einer Transition nicht unterbrechen und mit einer anderen Transition fortfahren, allerdings gibt es keine Regeln, die das temporäre Aussetzen einer Transition und das zwischenzeitliche Ausführen eines anderen Agenten verbietet. Diese Freiheit basiert auf zwei Eigenschaften von SDL:

1. **Agenten haben keine gemeinsamen Variablen:** Durch die Unterbrechung eines Agenten und die Ausführung eines anderen Agenten kann der Zustand des unterbrochenen Agenten nicht beeinflusst werden[4].
2. **Eine Aktion *kann* beliebige Zeit verbrauchen:** Es gibt keine Vorschriften über die Zeitdauer zwischen Start und Ende einer Transition. Im Standard ist dies folgendermaßen formuliert:

[4] Die Formulierung ist nicht vollständig korrekt. Der unterbrechende Agent kann zum Beispiel durch Senden eines Signals den *input port* bzw. das *gate* des unterbrochenen Agenten um Signale erweitern. Er kann allerdings keinen Wert ändern, auf den der unterbrochene Agent innerhalb seiner Transition zugreifen könnte.

> *„An undefined amount of time may pass while an action is interpreted. It is valid for the time taken to vary each time the action is interpreted. “* [Int99]
> Hier ist insbesondere der zweite Satz ausschlaggebend, denn bei einer Serialisierung von Agenten ist die Unterbrechung eines Agenten aus Agentensicht äquivalent zu dem Ausführen einer Aktion, die „länger dauert“. D.h ein Agent kann zunächst nicht unterscheiden, ob er durch einen anderen Agenten unterbrochen wurde oder ob eine Aktion entsprechend lange gedauert hat.

Entsprechend der Ausführungsreihenfolge können auch Unterbrechungen von Transitionen zu einem unterschiedlichen Systemverhalten führen, d.h. auch hier behindern die Freiheiten des Ausführungsmodells eine Vorhersagbarkeit des Laufzeitverhaltens basierend auf der SDL-Spezifikation.

Bei der Implementierung der SDL-Laufzeitumgebung wird die Frage aufgeworfen, ob das Modell der asynchron kommunizierenden ASMs tatsächlich eins zu eins umgesetzt werden muss oder ob es effizientere Möglichkeiten gibt, SDL standardkonform zu implementieren. Beispielsweise fällt auf, dass die parallele Ausführung aller Agenten in der ASM-Semantik selbst unter Beibehaltung der Fairness keineswegs eine Serialisierung *aller* Agenten in der Implementierung erfordert, da viele Agenten oft gar keine Aktionen zur Ausführung haben. Gemäß den Grundlagen von Echtzeitsystemen ist folglich eine ereignisgesteuerte Ausführung der Agenten eine naheliegende Lösung (vgl. Kapitel 2.1), die auch verträglich mit den Anwendungsbereichen von SDL in reaktiven Systemen ist [Int99]. Insbesondere durch die Anbindung von SDL an eine potentiell kaum zu kontrollierende Umgebung, wie sie zum Beispiel ein Drahtlosnetzwerk darstellt, ist eine zeitbasierte Ausführung ohnehin schwer vorstellbar. Kopetz schreibt hierzu, dass die ereignisgesteuerte Ausführung in Szenarien, in denen keine genauen Angaben über Zeitintervalle von Zustandsänderungen (und damit auch von externen Ereignissen) vorliegen, die bessere Wahl darstellt [Kop91]. Die ereignisgesteuerte Ausführung ist bei vollständiger Beachtung des SDL-Sprachumfangs nicht nur eine naheliegende und sinnvolle Lösung, sondern die einzig mögliche, denn das dynamische Hinzufügen von Agenten, wie es zum Beispiel mit der Instanziierung von SDL-Prozessen zur Laufzeit möglich ist, macht eine zeitgesteuerte Ausführung unmöglich [Kop91].

Im Unterschied zu der Modellbildung bei der Planung von Aktivitäten in Echtzeitsystemen, die von vielen Aktivitätseigenschaften abstrahiert, lohnt es sich, in SDL zwischen der Granularität der zu planenden Entitäten zu unterscheiden[5]. Basierend auf der Semantik von ASMs entsprechen einerseits SDL-Agenten den auszuführenden Entitäten, so dass eine Planung von SDL-Agenten offensichtlich erscheint. Andererseits wird bei der Ausführung eines SDL-Agenten nicht der Agent als Ganzes ausgeführt, sondern einzelne Transitionen. Die Transitionen konkretisieren also die auszuführenden Aufgaben (*tasks*), sodass ebenso eine (feingranularere) Planung von aktivierten Transitionen denkbar wäre. Die Aktivierung einer Transition ist dabei über die Feuerbereitschaft des entsprechenden Zustandsüberganges definiert, d.h. eine Transition ist aktiviert, sobald die Bedingung zur Abarbeitung der Transition erfüllt ist.

Die Planung von Transitionen hat allerdings das Problem, dass eine aktivierte Transition wieder deaktiviert werden kann, d.h. eine bereits aktivierte Transition muss nicht zwingend weiterhin feuerbereit sein, wenn der zugehörige SDL-Agent ausgeführt wird. Der Hintergrund dieser Eigenschaft ist unter Anderem in den Charakteristiken der *input ports* zu finden, die nicht

[5] Diese Unterscheidung gilt nur für SDL-Agenten. Bei SDL-Agentenmengen und Linkagenten kann nicht zwischen der Granularität unterschieden werden. Oft werden diese beiden Typen von ASM-Agenten aber gar nicht explizit von dem Planungsverfahren in der Implementierung berücksichtigt, so dass die Fragestellung gar nicht aufkommt. Abschnitt 4.2.2 geht hierauf näher ein.

zwingend einer FIFO-Strategie folgen muss. Beispielsweise existieren mit priority inputs Möglichkeiten, mit welcher später ankommende Signale vor allen anderen Signalen des *input ports* konsumiert werden können. Dadurch kann es passieren, dass eine aktivierte Transition, dessen *trigger* ein reguläres Signal ist, nach Eintreffen eines Signals, welches im aktuellen SDL-Zustand als priority input definiert ist, wieder deaktiviert wird. Ist jedoch ein SDL-Agent ausführbar, indem irgendein *trigger* (zum Beispiel ein Signal in dem *input port*) des Agenten gesetzt ist, kann sich hieran bis nach der Ausführung des Agenten nichts ändern[6]. Priority inputs, enabling conditions oder continuous signals könnten zwar auch bei SDL-Agenten dazu führen, dass aktivierte Transitionen wieder deaktiviert werden, im Gegenzug sorgen diese Sprachkonstrukte aber für die Aktivierung von anderen Transitionen. Die Ausführbarkeit des Agenten als Ganzes ändert sich erst nach dessen Abarbeitung. Ohne Einschränkung des SDL-Sprachumfangs, zum Beispiel durch das Verbieten von priority inputs, oder Änderung der Arbeitsweise von *input ports* (vgl. [ÁDL+03]) ist infolgedessen die Planung von SDL-Agenten sinnvoller als die feingranulare Planung von Transitionen, da bei der Planung von Transitionen eine häufige und aufwendige Transitionsauswahl notwendig wird, um die Ausführbarkeit von Transitionen festzustellen.

Die „Unplanbarkeit“ von Transitionen resultiert allerdings in einem in Bezug auf die Echtzeitfähigkeit enorm störenden Problem: Die Prioritäten, die bei der Planung berücksichtigt werden können, beziehen sich auf den Agenten, da er die zu planende Entität repräsentiert, und nicht auf die Transition, welche die eigentliche Aktion durchführt[7]. Man hat also die Einschränkung, dass transitionsabhängige Dringlichkeiten oder Fristen in der Planung nur sehr schwer beachtet werden können. Ebenso schwierig gestaltet sich daher die Abschätzung der Ausführungsdauer eines Agenten, die sich abhängig von den auszuführenden Transitionen stark unterscheiden kann. Bezogen auf die Möglichkeiten der verwendbaren Planungsverfahren stellen diese Punkte eine klare Beschränkung dar.

3.2.3 Zeitbegriff in SDL

Neben SDL-Timern und den Datentypen Time und Duration bietet die Sprache SDL ebenfalls den Ausdruck now, der systemweiten Zugriff auf die Systemzeit ermöglicht [Int99]. Zeit ist daher ein zentraler Bestandteil in SDL, der die Sprache im Vergleich zu vielen (Implementierungs-)Sprachen auszeichnet.

Bei genauerem Hinsehen erkennt man allerdings, dass dieses Ausmaß an Unterstützung des Zeitbegriffs nicht ausreichend für die adäquate Spezifikation von Echtzeitverhalten ist. So kann das Verhalten nur *be*schreibend anstelle von *vor*schreibend oder regulierend spezifiziert werden. Es fehlen also die Möglichkeiten, um ein gewünschtes Zeitverhalten zu erzwingen [EGG+01]. Dieses Problem fällt unter Anderem bei SDL-Timern auf, die nach Ablauf für unbestimmte Zeit in dem *input port* des Agenten verweilen können. SDL-Timer bieten demnach ausschließlich die Möglichkeit, eine minimale Zeitspanne zu warten [BGM+01]. Der für Echtzeitsysteme eher relevante späteste Zeitpunkt, zu dem eine Transition feuern muss, kann mit SDL folglich nicht spezifiziert werden. SDL-Timern fehlen demnach sinnvolle Aspekte aus dem Bereich der Programmierung von Echtzeitsystemen, um die Spezifikation echter Notfall-Timeout-Prozeduren zu ermöglichen [BGK+00]. Bildet man die Möglichkeiten von SDL-Timern auf die Terminologie

[6] Werden enabling conditions oder continuous signals betrachtet, in deren Prädikat ein zeitabhängiger Parameter (beispielsweise now) verwendet wird, kann sich auch die Ausführbarkeit eines SDL-Agenten verändern. Auf solche Konstrukte sollte allgemein verzichtet werden, da sie kaum effizient zu implementieren sind.

[7] Als Zwischenschritt zwischen den Prioritäten von Transition und SDL-Agenten wäre noch eine Planung nach Prioritäten denkbar, die von dem Zustand des SDL-Agenten abhängen.

von Echtzeitsystemen ab, so können SDL-Timer ausschließlich zur Angabe der Bereitzeit einer Aktivität verwendet werden; zur Spezifikation der Frist oder Ausführungsdauer gibt es keine Möglichkeit (siehe Abschnitt 2.2).

Ein weiteres häufig kritisiertes Problem von SDL hinsichtlich der Realisierung von Echtzeitsystemen ist die Abstraktion von Laufzeiten. Wie oben bereits genannt, können Aktionen und die Auswertung von Prädikaten innerhalb einer Transition beliebige Zeit in Anspruch nehmen [San00, BGM+01, GP05]. In SDL ist es demnach ebenso möglich, dass eine Aktion in Nullzeit ausgeführt wird; auf realer Hardware wird hingegen immer Zeit zur Abarbeitung einer Aktion beansprucht [MT00]. Gleiches gilt für die Kommunikation zwischen Agenten eines SDL-Systems, die auch von Laufzeiten abstrahiert, wobei Kanäle eine der wenigen Ausnahmen darstellen, da eine Unterscheidung zwischen verzögernden und nicht-verzögernden Kanälen möglich ist [EGG+01]. Allerdings wissen weder sendende Agenten, wann ein gesendetes Signal bei dem Empfänger konsumiert wird, noch wissen empfangende Agenten, wie lange sich das Signal bereits in ihrem *input port* befand oder auf dem Signalpfad verzögert wurde [ÁDL+03]. Auch wenn SDL keine Möglichkeiten zur Bestimmung von Signallaufzeiten durch die sendenden/-empfangenden Agenten vorsieht, könnte diese Information nachträglich anhand von expliziten Signalparametern oder Signalen ermittelt werden. Zur Beeinflussung der Signallaufzeiten, zum Beispiel durch die Bevorzugung bestimmter Signale, wäre allerdings eine Unterstützung seitens SDL erforderlich, die SDL nicht vorsieht und in dem formalen Ausführungsmodell auch nicht notwendig ist. Die SDL-Signale in Form von asynchronen Nachrichten erlauben daher a priori keine Aussagen über die Zeit, die für eine Kommunikation benötigt wird. Es bleibt die Aufgabe der Implementierung, eine „ausreichend“ effiziente und schnelle Umsetzung und Hardware zu wählen, welche den Anforderungen, die der Entwickler an das SDL-System während der Ausführung stellt, gerecht wird [BH93].

Der Indeterminismus, den SDL bezüglich Zeit an vielen Stellen lässt, ist tief im formalen Ausführungsmodell der Sprache verankert. Das Modell sieht zur Erfüllung der Kohärenzbedingung lediglich vor, dass die Ausführung des nächsten Agenten dem minimalen *move* bezüglich des partiell geordneten *runs* entspricht [GP05]. Dabei findet die Zeit in der Auswahl des nächsten Agenten keine Beachtung. Da allerdings alle Agenten eines SDL-Systems die gleiche globale Zeitbasis besitzen, findet bei der Ausführung auf realer Hardware eine Anordnung der Agenten gemäß einer totalen Ordnung statt, die die Möglichkeiten der Partialordnung einschränkt [GP05].

3.2.4 Datentypen und unbeschränkte Speicherressourcen

SDL unterstützt viele vordefinierte Datentypen wie Integer, Charstring oder Boolean. Viele dieser Datentypen sind unbeschränkt, d.h. es gibt beispielsweise für Integer kein Minimum oder Maximum. Ebenso unbeschränkt sind die *input ports*, die in der Theorie beliebig viele Signale aufnehmen können. Bei der Implementierung der Datentypen für die Ausführung von SDL auf einer realen Hardware-Plattform müssen die abstrakten Daten in die nicht-ideale reale Welt der Programmiersprachen und Computer übertragen werden [San00], in welcher Daten in die von der Hardware-Plattform vorgegebenen Wortgrenzen passen müssen [MT00]. Insbesondere erfordert dies eine Beschränkung der Datentypen. Außerdem zwingen die eingeschränkten Speicherressourcen zu einer Limitierung der Anzahl an Agenten und der Größe der *input ports* und *gates*. Selbst wenn diese nicht explizit beschränkt werden, entstehen spätestens zur Laufzeit Grenzen, wenn die endlichen Hardware-Ressourcen nicht mehr ausreichen.

3.2.5 Vorhersagbarkeit von Laufzeiten

Bei der Verwendung von SDL als Implementierungssprache fallen verschiedene Eigenschaften des Sprachstandards auf, die der Vorhersagbarkeit eines SDL-Systems schaden [But05]:

- **Dynamische (Daten-)Strukturen:** SDL ermöglicht die Erzeugung von neuen Objekten zur Laufzeit. Hierunter fallen sowohl Agenten als auch Signale und Datenobjekte. Erschwerend ist hierbei die Unbeschränktheit vieler Datentypen. Beispiele hierfür sind die primitiven Datentypen Integer und Natural oder Generatoren (Array, String, ...).
- **Rekursion:** Prozeduren bieten die Möglichkeit zu einer unbeschränkten Rekursion.
- **Zeitlich unbegrenzte Schleifen:** Es ist nicht möglich, eine Obergrenze an Schleifendurchläufen zu spezifizieren.

Im Allgemeinen können diese Eigenschaften von SDL zu einer unbeschränkten Laufzeit oder gar zur Nicht-Terminierung von Transitionen eines SDL-Agenten führen. Ohne entsprechende Einschränkungen oder Annahmen setzen sie somit der Bestimmung der Laufzeit von Transitionen Grenzen.

Aber nicht nur die Bestimmung der Laufzeit eines Agenten selbst stellt sich als äußerst schwierig heraus. Ebenso problematisch ist die Tatsache, dass die durch einen Agenten verursachte Kommunikation je nach eingesetztem Kontext innerhalb eines SDL-Systems einen unterschiedlichen Aufwand zur Laufzeit verursacht. Ist beispielsweise der Pfad eines Signals „lang“ im Sinne von vielen Blockgrenzen, die das Signal bei der Zustellung zwischen Agenten passieren muss, dann ist die Zustellung aufwendiger als bei einer Zustellung zwischen zwei Agenten, deren Prozessdefinition im gleichen SDL-Block instanziiert wird[8]. Die Zustellung von Signalen ist demnach abhängig von dem Szenario, was eine allgemeine Aussage bezüglich der Ausführungsdauer verbietet. Da die Zustellung von Signalen, die als Primitive der SDL-Laufzeitumgebung angesehen werden kann, vergleichbar ist mit der Interprozess-Kommunikation in Betriebssystemen, deren zeitliche Beschränkung Voraussetzung zur Vorhersagbarkeit ist [Sta92], leidet auch SDL unter einer fehlenden Vorhersagbarkeit.

3.2.6 Vorteile von SDL

Im Standard wird SDL als Sprache zur Spezifikation von Aspekten von Echtzeitsystemen umworben [Int99]. De facto besitzt die Sprache auch viele Eigenschaften, die zur Realisierung von Echtzeitsystemen förderlich sind.

Ein wichtiger Vorteil besteht in der asynchronen Kommunikation zwischen den Agenten eines SDL-Systems. Diese verhindert zwar einerseits eine Bestimmung der Laufzeiten von SDL-Signalen, andererseits macht sie den Sendevorgang unabhängig von dem Zustand des Empfängers [But05]. Es muss demnach kein *rendezvous* zwischen Sender und Empfänger an Synchronisationspunkten stattfinden, wodurch die Betrachtung der Ausführungszeit einer SDL-Transition zusätzlich erschwert werden würde. Des Weiteren hat die Unbeschränktheit der *input ports* bzw. *gates* neben den zahlreichen oben genannten Nachteilen den Vorteil, dass bei vollständig vorliegendem Signalpfad gemäß der Semantik keine Signale verloren gehen. Wenn also von der dynamischen Zerstörung von Agenten abgesehen wird, kann ein sendender Agent ohne explizite Überprüfung davon ausgehen, dass ein gesendetes Signal den *input port* des Empfängers erreicht. Eine spezielle Bestätigung über den Erhalt eines Signals ist nicht nötig.

[8] Der Standard sieht die explizite Weiterleitung an jeder Blockgrenze vor. Der entstehende Aufwand kann durch Optimierungen verkleinert werden (siehe beispielsweise [MT00]), die allerdings häufig den SDL-Sprachumfang einschränken.

Ein weiterer Vorteil von SDL ist durch die Speicherbereiche der Agenten gegeben, die bis auf *input ports* bzw. *gates* völlig disjunkt voneinander sind[9]. Dass der Datenaustausch vollständig über Signale erfolgt, macht kritische Abschnitte durch Semaphore oder Locks für den Zugriff auf Variablen überflüssig. Daher kann bei dem Zugriff auf Variablen weder ein Deadlock entstehen, noch besteht das Problem der Prioritätsinvertierung (siehe Abschnitt 2.3). An dieser Stelle sei allerdings angemerkt, dass auf abstrakterer Ebene sehr wohl Deadlocks auftreten können. Der Standard geht auf dieses Problem im Zusammenhang mit *remote procedures* und *remote variables* ein [Int99].

Im Hinblick auf die Entwicklung von Hardware und den Trend zu Mehrkernsystemen ist das parallele Ausführungsmodell von SDL als weiterer Vorteil zu nennen. Durch den vollständig nebenläufigen Charakter der ASM-Agenten und den hohen Grad an vorhandener Parallelität scheint SDL für den Einsatz auf Mehrkernsystemen geradezu prädestiniert .

3.3 Stand der Forschung

Die aufwendige dynamische Semantik und die eingeschränkte Vorhersagbarkeit von SDL haben viele Forscher zum Anlass genommen, Erweiterungen von SDL und Methoden zur effizienten Implementierung zu entwickeln. Dieser Abschnitt fasst sowohl die Design-Entscheidungen für eine effiziente Implementierung von SDL als auch entwickelte Analysetechniken zur Vorhersagbarkeit einer SDL-Spezifikation zusammen.

3.3.1 Implementierungen des SDL-Standards

Das formale Ausführungsmodell von SDL legt den Grundstein für die Ausführbarkeit einer SDL-Spezifikation. Eine standardkonforme Implementierung erfordert aber keine Eins-zu-Eins-Umsetzung des formalen Ausführungsmodells, was aufgrund des hohen Grades an Parallelität und der Annahme über die Unbeschränktheit von Ressourcen ohnehin nicht möglich ist. Die im SDL-Standard fehlenden Implementierungsdetails und die oben diskutierten Freiheiten im Standard können bei der Implementierung von SDL-Systemen und der SDL-Laufzeitumgebung genutzt werden, um die Effizienz der Implementierung zu steigern.

Ein Buch, dass sich sehr ausführlich den Möglichkeiten und Design-Entscheidungen bei der Implementierung von SDL widmet, stammt von Mitschele-Thiel [MT00]. Darin steht die Effizienz und Performanz der Implementierung im Vordergrund und nicht die Vorhersagbarkeit einer SDL-Implementierung. Mitschele-Thiel behandelt in seinem Buch zahlreiche Aspekte von SDL und stellt alternative Implementierungswege vor. Dabei wird unter Anderem die Integration eines SDL-Systems in ein Hardware-System abgedeckt, bei welcher zwischen *tight* (intensives Nutzen von Diensten des Betriebssystems), *light* (kaum Interaktion zwischen SDL-System und zugrunde liegendem Betriebssystem) und *bare* (SDL-Laufzeitumgebung setzt ohne Betriebssystem direkt auf Hardware auf) unterschieden wird. Des Weiteren diskutiert Mitschele-Thiel unterschiedliche Möglichkeiten zum Warten auf Ereignisse (*active waiting*, *semi-active waiting* und *passive waiting*) sowie verschiedene Strategien zum effizienten Testen von Aktivierungsbedingungen einer Transition (*consumer-based testing* vs. *producer-based testing*). Ein wichtiges und sorgfältig behandeltes Thema ist außerdem der Umgang mit beschränkten Ressourcen, der sich zum

[9] Seit SDL-2000 ist es mit Zustandsaggregationen möglich, dass mehrere Agenten gemeinsame Variablen besitzen, die in dem sie umgebenden Agenten definiert sind [Int99]. Der Standard sieht in diesen Fällen aber eine sequentielle Abarbeitung der Transitionen dieser Agenten vor, sodass die Konsistenz von Variablen nie gefährdet ist.

Beispiel bei *input queues* bemerkbar macht. Diesbezüglich werden in dem Buch verschiedene Alternativen verglichen, wie zu reagieren ist, falls die Kapazität eines Puffers nicht mehr ausreicht (*prevention*, *discard signals*, *blocking*, *dynamic control*). Durch Aufweichung der *copy-by-value* Semantik von SDL, zum Beispiel bei Signalparametern, geht Mitschele-Thiel ebenfalls auf Effizienzsteigerungen durch die Vermeidung von (unnötigen) Kopieroperationen ein. Mitschele-Thiel stellt in dem Buch ebenfalls klar, dass durch Beschränkungen des SDL-Sprachumfangs und strikte Entwurfsregeln bei der Spezifikation eines SDL-Systems eine deutlich effizientere Implementierung möglich wird. Beispielsweise lassen sich *input ports* deutlich effizienter realisieren, wenn sie immer einer FIFO-Strategie folgen, was mitunter das Verbieten von priority inputs und *save*s erfordert.

Ein weiteres Standardwerk bezüglich der effizienten Implementierung von SDL stammt von Bræk and Haugen [BH93]. Die Autoren behandeln in ihrem Buch sowohl Aspekte des Hardware-/Implementierungs-Designs als auch des Software-Designs und definieren Regeln zum Finden eines Kompromisses zwischen Hard- und Software. Bei dem Implementierungs-Design werden Alternativen und Richtlinien vorgestellt, wie ein System auf physikalische Objekte verteilt werden kann und die Hardware-Plattformen bezüglich Ressourcen zu dimensionieren sind. Beispielsweise wird geraten, dass ein Hardware-System im Schnitt nicht mehr als 30% ausgelastet sein sollte, um genügend Restkapazitäten für kurzzeitige Peaks zu haben. In dem Software-Design gehen Bræk and Haugen auf Möglichkeiten zur Beachtung von Prioritäten in einem SDL-System ein, um zeitkritische Signale zu bevorzugen, und empfehlen eine dynamische Anpassung der Prioritäten und eine Unterbindung neuer externer Anfragen in Überlastsituation. Sie diskutieren ebenfalls Alternativen in der Kommunikation zwischen unterschiedlichen Software-Modulen, die der Terminologie von SDL folgend den Agenten entsprechen. Insbesondere gehen sie auch auf die Möglichkeit der Realisierung von SDL-Signalen in Form von Funktionsaufrufen ein, die zwar nicht mehr der asynchronen Kommunikation im SDL-Standard entspricht, aber aufgrund der Möglichkeit der Präemption von Agenten keineswegs gegen den Standard verstößt. Bei der Implementierung von SDL-Prozessen unterscheiden die Autoren zwischen drei verschiedenen Alternativen: *state-oriented*, *action-oriented* und *table-driven*. Bezüglich des Sprachumfangs und der Mächtigkeit von SDL, die hinsichtlich der Realisierung von zeitkritischen Systemen einige Nachteile mit sich führen, raten die Bræk and Haugen ebenfalls zu dem Verzicht auf verschiedene Sprachkonstrukte. Hierzu zählt beispielsweise ebenso *save*. In einem eigenen Kapitel demonstriert das Buch die Implementierung in der Programmiersprache C++.

Neben den beiden Standardwerken gibt es noch weitere Veröffentlichungen, die sich mit der effizienten Implementierung von SDL beschäftigen. Ein Beispiel ist [San00], wo unter Anderem ebenfalls die Möglichkeit der Gruppierung von SDL-Prozessen und deren Abbildung auf mehrere physikalische Objekte diskutiert wird. Ebenso wird die Realisierung von SDL-Signalen als Methodenaufrufe thematisiert. Diese Realisierung ist nur möglich, wenn das Verhalten des Systems dem eines *Aufruf-Baumes* entspricht, d.h. es darf insbesondere keine zyklischen Kommunikationsmuster geben. Ebenfalls behandelt [San00] das Optimierungspotential durch Aufweichung der *copy-by-value* Semantik bei Signalparameter, die oftmals nicht nur ineffizient ist, sondern auch überflüssig, da ein Prozess nicht zwingend schreibend auf ein empfangenes Datum zugreift. Bezüglich der Implementierung von abstrakten Datentypen wird geraten, die Signatur auf SDL-Ebene zu spezifizieren und die eigentliche Implementierung in der Ziel-Programmiersprache vorzunehmen.

Sowohl [BH93] als auch [San00] raten zu der Untergliederung eines SDL-Systems in mehrere Teile, die anschließend auf mehrere physikalische Hardwarekomponenten verteilt werden. Dieser

Prozess wird allerdings nicht von SDL unterstützt. Jedoch existieren Werkzeuge, die den Entwickler bei dieser Arbeit unterstützen. Ein Beispiel ist der *Deployment Editor* von Telelogic/IBM [IBMar], der anhand von UML-Komponentendiagrammen die physikalische Struktur eines Systems beschreibt [LEH00].

Selbst bei der Verwendung von SDL zur Entwicklung von zeitkritischen Systemen, die auf einer Hardwarekomponente zum Einsatz kommen und bei denen keine physikalische Verteilung vorgenommen wird, können SDL-Systeme auf mehrere Betriebssystem-Prozesse aufgeteilt werden. Durch eine adäquate Untergliederung des Systems und die Abbildung der entstehenden Betriebssystem-Prozesse auf Prioritäten eines Echtzeitbetriebssystems finden auf diese Weise ebenfalls Prioritäten in der Ausführung eines SDL-Systems Beachtung. Diese Vorgehensweise hat aber den Nachteil, dass eine zusätzliche *Deployment-* bzw. Implementierungsphase nötig wird, in welcher die Verwendung von SDL ausscheidet. Im Bereich von eingebetteten Systemen, bei denen Applikationen häufig direkt auf Hardware aufsetzen, führen Betriebssysteme außerdem einen unnötigen Overhead ein, wodurch die Ausführung des SDL-Systems ineffizienter wird. Bezogen auf das verwendete Entwicklungs-Framework, das in der AG Vernetzte Systeme aus *SdlRE*, *SEnF* und *ConTRaST* besteht (siehe Anhang A), entsteht durch die Integration eines Echtzeitbetriebssystems eine zusätzliche Abhängigkeit, da die Interprozess-Kommunikation die Unterstützung des Betriebssystems erfordert.

3.3.2 Analysierbarkeit und Vorhersagbarkeit von SDL

Neben der Problematik, SDL effizient zu implementieren, gab es einige Bestrebungen, die Analysierbarkeit von SDL-Spezifikation hinsichtlich temporaler Eigenschaften zu verbessern. Einerseits entstanden Analysetechniken, die mit zusätzlichen Annahmen das Zeitverhalten eines ausgeführten Systems quantitativ auswerten, andererseits entstanden auch Methoden, die die Beeinflussung des Zeitverhaltens erlauben. Im Folgenden werden die entsprechenden Arbeiten zusammengefasst.

3.3.2.1 Ein Ausführungsmodell für SDL mit Prioritäten

In [ÁDL+03] präsentieren die Autoren Maßnahmen, um die Vorhersagbarkeit und Echtzeitfähigkeit von SDL-Systemen speziell bei dem Einsatz auf Einkernprozessoren zu verbessern. Hierzu definieren sie ein Ausführungsmodell für SDL, das den Indeterminismus von SDL an vielen Stellen zu Gunsten der Vorhersagbarkeit einschränkt. In der Veröffentlichung sind nicht nur Analysetechniken beschrieben, die beispielsweise das SDL-System unter der Annahme eines speziellen Planungsverfahrens auf verschiedene Eigenschaften hin überprüfen, sondern es werden Erweiterungen von SDL in Form von Annotationen vorgestellt, die das Verhalten des Systems beeinflussen.

Die weitreichendste Erweiterung der SDL-Spezifikation betrifft die Ausführungsreihenfolge der Agenten. Die Autoren präsentieren ein präemptives Planungsverfahren, welches Agenten basierend auf transitionsabhängigen Prioritäten sortiert. Bei dem vorgestellten Ausführungsmodell distanzieren sich die *input ports* von Agenten vollständig von der FIFO-Eigenschaft. Es wird immer die höchstpriore Transition ausgeführt; die Reihenfolge, in der Signale in den *input port* eines SDL-Agenten eingefügt werden, spielt keine Rolle bei der Transitionsauswahl. Da sich allerdings an der *run-to-completion*-Semantik der Transitionen nichts ändert, gibt es eine Ausnahme dieser Regel: Ist der Agent bereits mitten in der Ausführung einer Transition und empfängt ein

Signal, welches eine höherpriore Transition aktiviert, muss zuerst die Ausführung der begonnen Transition beendet werden. In diesem Fall hebt die höherpriore Transition aber die Priorität des Agenten an, sodass die Ausführung anderer Agenten nicht zusätzlich zu einer Verzögerung beiträgt. Aufgrund des präemptiven Planungsverfahrens kann nur eine Transition des gleichen Agenten die Ausführung der höchstprioren Transition verzögern.
Des Weiteren werden in [ÁDL+03] SDL-Signale um Zeitstempel erweitert, welche Informationen über den Versand- und Empfangszeitpunkt enthalten. Anhand von zwei Funktionen kann ein empfangender Agent dadurch herausfinden, wie lange der Versand des Signals bereits in der Vergangenheit liegt, und seine nächsten Aktionen eventuell anpassen. Außerdem zeigen die Autoren eine Möglichkeit, wie Hardware-Treiber direkt in der SDL-Spezifikation realisiert werden können. Hintergrund dieser Maßnahmen ist die Tatsache, dass viele eingebettete Systeme direkt auf der Hardware aufsetzen, und die Realisierung der Treiber innerhalb des SDL-Environments dieser Tatsache nicht gerecht wird.
Basierend auf dem präemptiven Planungsverfahren wird in der Veröffentlichung eine Schedulability-Analyse vorgestellt, welche die *worst case response time* des Systems auf externe Ereignisse untersucht. Hierzu ist es erforderlich, dass die maximale Ausführungszeit aller Transitionen angegeben wird. Außerdem müssen Annahmen über das Intervall, in dem externe Ereignisse auftreten, getroffen werden. Bei der Analyse werden sowohl die Ausführungszeiten der Transitionen als auch die Blockierungszeiten durch höherpriore Transitionen beachtet. Der Overhead durch das Laufzeitsystem und die Präemption fließt nicht in die Berechnungen ein. Die Ergebnisse der Analyse können verwendet werden, um zu überprüfen, ob die Fristen bezüglich der Antwortzeiten auf externe Ereignisse eingehalten werden.
Durch die Änderung der Arbeitsweise der *input ports* sind die vorgestellten Maßnahmen nur bedingt konform mit dem SDL-Standard. Zusätzlich lassen sich *input ports* im Regelfall deutlich effizienter implementieren, wenn der Zugriff einer FIFO-Strategie entspricht. Ein weiterer Nachteil des Planungsverfahrens ist die erforderliche Neuberechnung der Priorität eines Agenten, die nach jedem Signalempfang und nach jeder Ausführung einer Transition neu zu bestimmen ist. Allgemein ist die Art der Bestimmung der Priorität eines Agenten für den Entwickler sehr intransparent. Zum Beispiel hat die Reihenfolge, in der mehrere Signale innerhalb einer Transition versendet werden, aufgrund der Präemption Auswirkungen auf die Ausführung des nächsten Agenten. Der Entwickler muss demnach nicht nur Prioritäten entsprechend den Anforderungen an Transitionen zuweisen, sondern ebenfalls bei der Aktivierung einzelner Agenten Vorsicht walten lassen.

3.3.2.2 Timed SDL (TSDL)

Timed SDL (TSDL) erweitert SDL-Spezifikationen um Zeitaspekte, die als Grundlage für eine auf Markov-Ketten basierende Performanzevaluation dienen [BB93, BB90]. Das Ziel von TSDL ist die automatisierte Auswertung von qualitativen und quantitativen Systemaspekten. Zu den quantitativen Aspekten zählt zum Beispiel die Bestimmung des durchschnittlichen Signaldurchsatzes, wodurch ebenfalls ein *Tuning* von Protokollen möglich wird. Quantitative Systemeigenschaften beinhalten beispielsweise das Finden von nicht-ausgeführten Transitionen, Deadlocks oder Livelocks. Hierzu wird in allen SDL-Prozessdefinitionen an jeder Transition die Dauer der Transition und die Wahrscheinlichkeit der Ausführung der Transition vermerkt. Um für die Performanzevaluation die relevanten Aspekte eines SDL-Systems zu erfassen, führt TSDL globale Variablen, die auf jeder Ebene der System-Hierarchie verwendet werden können, und Funktionen zur Betrachtung des Zustandes von Signal-Warteschlangen (*input ports*, *gates*) ein. Die Grö-

ße des Zustandsraums eines SDL-Systems erlaubt in der Regel keine vollständige Modellierung des Systems anhand von Markov-Ketten. Aus diesem Grund beinhalten die um TSDL entwickelten Werkzeuge verschiedene Analysetechniken, deren Auswertung sich auf Ausschnitte des Zustandsraumes beschränkt. Des Weiteren ist die Beschränkung der Kapazität von *input ports* eine notwendige Bedingung, damit der Zustandsraum überhaupt endlich bleibt.

Bei TSDL steht die Performanzevaluation, d.h. die Analyse eines bestehenden Systems, im Fokus und nicht die Beeinflussung der Ausführung. Dem Problem der Performanzevaluation wird dabei mit stochastischen Methoden begegnet, sodass Aussagen bezüglich des *worst cases* nicht unterstützt werden. Die gelieferten Ergebnisse hängen stark von der spezifizierten Wahrscheinlichkeit und Dauer einzelner Zustandsübergänge ab, was die Frage aufwirft, wie diese Parameter realistisch zu bestimmen sind.

3.3.2.3 Queueing SDL (QSDL)

Die Entwickler von *Queueing SDL* (QSDL), einer Erweiterung von SDL-92 mit nicht-funktionalen Systemeigenschaften, haben sich ebenfalls die Unterstützung der Performanzevaluation in SDL zur Aufgabe gemacht [DHMC95, DHMC97]. QSDL führt hierzu den Begriff von *Maschinen* in SDL ein, die dem aus der *queueing theory* bekannten Paradigma von Warteschlangenstationen folgen [Jai91]. In einer QSDL-Maschine ist unter Anderem die Menge an Diensten, die von der Maschine angeboten wird, die Menge an dienstabhängigen Bearbeitungszeiten der Maschine und das verwendete Planungsverfahren (*first come, first serve* (FCFS), zufällig oder statische Prioritäten) spezifiziert. QSDL-Maschinen können innerhalb eines SDL-Systems überall dort stehen, wo auch SDL-Prozesse erlaubt sind. Mit speziellen Bindungsmechanismen zwischen QSDL-Maschinen und SDL-Prozessen (*pipes* und *links*) stellt die Sprache außerdem Möglichkeiten zur Verfügung, Last, die in SDL-*tasks* von SDL-Prozessen erzeugt wird, an eine QSDL-Maschine zu binden. Durch Erweiterung der *tasks* kann somit jedem *task* ein von einer QSDL-Maschine bereitgestellter Dienst zugeordnet werden. Dabei kann auch zwischen der Priorität des *tasks*, die sich bei Verwendung des prioritätsbasierten Planungsverfahrens innerhalb der Maschine auswirkt, und der Menge an Last, die durch den *task* erzeugt wird, unterschieden werden. Zur Erzeugung von Systemstimuli stehen mehrere Verteilungsfunktionen zur Verfügung.

Ziel der Erweiterungen des SDL-Systems ist die automatisierte Erstellung eines ausführbaren Modells, mit welchem anschließend das System simulativ auf verschiedene Eigenschaften hin analysiert werden kann und Entscheidungen bezüglich der Optimierung von Datenstrukturen und Algorithmen getroffen werden können. Entsprechende durch die Simulation gewonnene Eigenschaften sind zum Beispiel die Nutzdatenrate oder der Zeit- und Ressourcenverbrauch. Zentraler Bestandteil der simulativen Analyse ist das Werkzeug QUEST [DHMC97]. Durch entsprechende Spezifikation der QSDL-Maschinen können durch die von QSDL eingeführte Methode implementierungsabhängige Details (zum Beispiel Grad an Nebenläufigkeit, Bearbeitungsverzögerungen oder das verwendete Planungsverfahren) in die Analyse einfließen.

QSDL nähert sich der Performanzanalyse ebenfalls mit stochastischen Methoden und Simulationen. Verglichen mit TSDL verfügt QSDL aber über deutlich mehr Konfigurationsparameter, um das System szenario- und plattformabhängig zu evaluieren. Allerdings ist auch bei QSDL fraglich, wie die Menge an Last, die ein *task* erzeugt, realistisch zu ermitteln ist. Des Weiteren ist die Tatsache, dass der Großteil an Last häufig nicht innerhalb eines *tasks* entsteht, sondern bei der Transitionsauswahl und der Zustellung von SDL-Signalen, eine weitere Einschränkung von QSDL.

3.3.2.4 Timed Extensions for SDL

Eine Fülle von Erweiterungen von SDL in Form von Annotationen mit zeitbasierten Eigenschaften wird in [BGM+01] vorgestellt. Die Annotationen lassen sich dabei in zwei Typen untergliedern: Annahmen (*assumptions*) und Zusicherungen (*assertions*). Annahmen entsprechen a priori-Wissen über die Umgebung oder das SDL-Laufzeitsystem, das von dem Entwickler spezifiziert werden kann und bei der Verifikation oder Codegenerierung seine Verwendung findet. Zusicherungen sind hingegen Erwartungen des Entwicklers an das Verhalten des Systems, welche nachzuweisen sind.

Bezüglich der Verzögerung in der Abarbeitung von Transitionen (vgl. Abschnitt 3.2.3) werden drei Annotationen eingeführt [BGK+00, BGM+01]: Ist eine Transition als *eager* gekennzeichnet, muss sie sofort ausgeführt werden, sobald sie aktiviert ist[10]. Dies bedeutet insbesondere, dass now nicht voranschreiten darf, bis die Transition abgearbeitet ist. Eine Transition, die als *lazy* markiert ist, kann hingegen beliebig verzögert werden, was dem *Default*-Verhalten im SDL-Standard entspricht. Der dritte Annotationstyp, *delayable*, spezifiziert ein Zeitintervall, für welches eine Transition verzögert werden darf. Spätestens am Ende dieses Intervalls muss die Transition jedoch abgearbeitet werden, d.h. eine mit *delayable* gekennzeichnete Transition ist zunächst *lazy*, wird aber *eager*, wenn die maximale Verzögerung erreicht ist.

Ebenfalls werden in [BGM+01] Annotationen zur Spezifikation der Verzögerung von Aktionen und Kanälen eingeführt. Diese Verzögerungen können in Form von Intervallen angegeben werden. Diesbezüglich weisen die Autoren außerdem darauf hin, dass die Ausdrucksstärke der Annotationen mit Wahrscheinlichkeitsfunktionen zusätzlich erhöht werden könnte. Zur Eingrenzung der Häufigkeit von periodischem Stimulus schlagen die Autoren des Weiteren die Angabe von Intervallen an Transitionen vor, die angeben, mit welcher Rate entsprechende Signale eintreffen können. Zuletzt werden zwei weitere Annotationen für SDL-Kanäle vorgestellt, mit welchen Eigenschaften des Kanals bezüglich Zuverlässigkeit und FIFO-Charakteristik spezifiziert werden können.

Neben den Erweiterungen von SDL in Form von Annotationen werden zusätzliche Ergänzungen vorgeschlagen, um Konzepte aus der Programmierung von Echtzeitsystemen nach SDL zu übernehmen. Eines dieser Konzepte betrifft SDL-Timer, für welche mehrere Erweiterungen diskutiert werden. Die weitreichendste Erweiterung ist die Einführung von unterbrechenden SDL-Timern, welche die Mächtigkeit haben, die Ausführung von Transitionen zu stoppen, und zur Realisierung von echten Notfall-Timeouts angedacht sind.

Die vorgestellten Erweiterungen versuchen einige fehlende Sprachmöglichkeiten von SDL zu ergänzen und tragen durch die Angabe von Häufigkeiten und Verzögerungen zur Verbesserung der Vorhersagbarkeit bei. Die eingeführten Annotationen stellen aber nur eine implizite Beeinflussung des Laufzeitverhaltens dar. Es bleibt zum Beispiel offen, ob Angaben über die Periodizität von Transitionen Auswirkungen auf das Planungsverfahren haben und wenn ja, wie deren konkreter Einfluss aussieht.

3.3.2.5 Abbildung von SDL auf *timed automata*

[OK01] beschreibt einen Ansatz zur Verifikation von temporalen Eigenschaften einer SDL-Spezifikation, der auf der Abbildung von SDL auf gezeitete Automaten (*timed automata*) basiert. Die

[10] Wird eine Transition des gleichen SDL-Agenten gerade ausgeführt, muss diese natürlich zuerst abgeschlossen werden. An der *run-to-completion*-Semantik von Transitionen ändert sich nichts.

Autoren werden von der Tatsache motiviert, dass viele Verifikationstechniken für SDL vollständig von der Zeit abstrahieren und meist die SDL-Zeit überhaupt nicht voranschreiten lassen, solange noch aktive Transitionen in dem System sind. Zur Erhöhung der Ausdrucksfähigkeit von SDL-Timern und zur Beschreibung des Determinismus ihrer Abarbeitung führen die Autoren analog zu [BGM+01] Annotationen zur Unterscheidung zwischen *eager*, *lazy* und *delayable* Timern ein. Ebenso werden dem Sprachumfang Annotationen zur Spezifikation des Zeitverbrauchs von SDL-*tasks* hinzugefügt. Durch die Abbildung von SDL auf gezeitete Automaten entsteht die Möglichkeit, Techniken und Werkzeuge aus dem Bereich der gezeiteten Automaten zu verwenden, um Eigenschaften eines SDL-Systems zu prüfen. Im Konkreten wird hierzu in der Veröffentlichung eine erweiterte Version des *ObjectGEODE*-Verifikationswerkzeugs verwendet[11]. Das Werkzeug besitzt mit *GOAL* eine Sprache zur Spezifikation von temporalen Eigenschaften, wie zum Beispiel Deadlock-Freiheit, *non-zenoness of runs* (unendliche *runs* ohne Voranschreiten der Zeit), oder allgemeinere Invarianten. Mithilfe von *GOAL* können solche Eigenschaften eines SDL-Systems angegeben werden, die anschließend mit der Abbildung der SDL-Spezifikation auf gezeitete Automaten und dem Verifikationswerkzeug überprüft werden.

Die durch *ObjectGEODE* und *GOAL* realisierten Verifikationsmöglichkeiten umfassen nicht den vollständigen SDL-Standard. Zum Beispiel ist die Verwendung von SDL-Zeit und insbesondere von now nur eingeschränkt möglich. Im Mittelpunkt der Veröffentlichung steht die Analyse temporaler Eigenschaften einer SDL-Spezifikation basierend auf der formalen Grundlage der gezeiteten Automaten. Möglichkeiten zur Beeinflussung des zeitlichen Verhaltens sind kein Gegenstand der Veröffentlichung.

3.3.2.6 Abbildung von SDL auf *task networks*

Die Autoren in [KF98] identifizieren den Indeterminismus durch die nicht festgelegte Ausführungsreihenfolge von Agenten als eine der Hauptursachen für die fehlende Echtzeitfähigkeit von SDL. Zur Steigerung der Echtzeitfähigkeit schlagen sie die Integration eines *Earliest-Deadline-First*-Planungsverfahrens in SDL vor. Es wird außerdem eine Methode vorgestellt, mit welcher SDL-Spezifikationen auf aus *task networks* bestehende Echtzeit-Analysemodelle abgebildet werden können, um die Planbarkeit von Agenten eines SDL-Systems mithilfe von Schedulability-Analysen zu überprüfen. Bei dieser Abbildung wird allerdings nur ein Bruchteil des SDL-Sprachumfangs unterstützt. Im Konkreten werden keine dynamischen Instanziierungen von SDL-Prozessen, keine priority inputs, keine continuous signals und kaum Element der Objekt-Orientierung von SDL unterstützt. Die Schedulability-Analyse erfordert die Angabe von Deadlines und die Bestimmung der maximalen Ausführungszeit (*worst case execution time* (WCET)) aller SDL-Agenten. Hierbei wird pro SDL-Agent und Signal, welches der Agent empfangen kann, genau eine Ausführungszeit berechnet. Kann das Signal in mehreren Zuständen empfangen werden, wird das Maximum der Ausführungszeiten aller entsprechenden Transitionen ausgewählt. Das durch die Umformungen des SDL-Systems entstehende *task network* kann anschließend hinsichtlich der Einhaltung der Fristen analysiert werden.

Die präsentierte Vorgehensweise behandelt die Analyse einer SDL-Spezifikation hinsichtlich der Einhaltung von Fristen. Die konkrete Umsetzung des EDF-Planungsverfahrens in die SDL-Laufzeitumgebung ist nicht behandelt. Ein Nachteil ist, dass nur ein kleiner Teil der Möglichkeiten von SDL unterstützt wird. Viele Sprachelemente, die gerade die Mächtigkeit des Sprachumfangs

[11] Das von der französischen Firma VERILOG entwickelte Werkzeug wird schon lange nicht mehr weiterentwickelt. Daher existieren keine aktuellen Referenzen zu *ObjectGEODE*.

von SDL ausmachen, wurden in der vorgestellten Analysetechnik nicht beachtet und es ist fraglich, ob sie ohne explizites Wissen über deren Implementierung erfasst werden können.

3.4 Bewertung

SDL als Spezifikations- und Beschreibungssprache hat sich in ihrer langjährigen Geschichte etabliert und in verschiedenen Szenarien ausgezeichnet. Die breite Werkzeugunterstützung und die Möglichkeiten zur hierarchischen Strukturierung, Modularisierung und Wiederverwendung machen die Sprache auch heute noch zu einem geeigneten Kandidaten für die Spezifikation von verteilten Systemen.

Die zahlreichen Veröffentlichungen, die Verbesserungen der Analysierbarkeit von SDL und der Effizienz von Implementierungen behandeln, zeigen jedoch, dass SDL primär nicht als Implementierungssprache entwickelt wurde und Nachteile bezüglich der Realisierung von zeitkritischen Systemen aufweist. Motiviert durch diese Nachteile entstanden Erweiterungen und Analysetechniken für SDL, die sich zusammenfassend in drei Gruppen untergliedern lassen:

- Möglichkeiten zur effizienten Implementierung des SDL-Standards.
- Analysetechniken (vorwiegend mit stochastischer Herangehensweise) zur Überprüfung temporaler Eigenschaften.
- Maßnahmen zur aktiven Beeinflussung des Laufzeitverhaltens.

Bei den vorgestellten Arbeiten besteht der Konsens, dass sowohl Effizienz als auch Vorhersagbarkeit von SDL nur zu erreichen sind, wenn im Standard einschränkende Annahmen getroffen werden. SDL muss es außerdem schaffen, einerseits eine high-level Spezifikationssprache, die generisch und plattformunabhängig ist, zu bleiben, andererseits aber an Ausdrucksstärke bezüglich Dienstgüteeigenschaften und dem zu verwendenden Planungsverfahren zu gewinnen [BGM+01]. Erst wenn dies erreicht ist und realistische Annahmen über den Stimulus des Szenarios getroffen wurden, lassen sich basierend auf einer SDL-Spezifikation Aussagen über das Zeitverhalten des ausgeführten Systems treffen.

4. KAPITEL

Maßnahmen zur Verbesserung des Laufzeitverhaltens

Die Diskussion der Probleme von SDL hinsichtlich der Vorhersagbarkeit und der Erstellung von Implementierungen in Abschnitt 3.2 zeigt, dass zwischen dem formalen Modell und dem auf realer Hardware ausgeführten SDL-System ein enormer Unterschied bestehen kann. Gerade die serialisierende Abbildung der nebenläufigen ASM-Agenten auf einen Einkernprozessor stellt einen der wichtigsten Gründe für diesen Unterschied dar. Zur Minimierung dieser Differenzen sind zusätzliche Maßnahmen erforderlich, die dem Entwickler auf Entwurfsebene die Möglichkeit geben, die Serialisierung zu beeinflussen, um den Prioritäten unterschiedlicher Agenten des SDL-Systems gerecht zu werden, und die Auslastung der Hardware-Plattform zu kontrollieren.

In diesem Kapitel wird durch das Einführen eines prioritätsbasierten Planungsverfahrens eine solche Maßnahme vorgestellt. Abschnitt 4.1 stellt zunächst allgemeine Entwurfsrichtlinien vor, die ohne die Notwendigkeit einer Erweiterung von SDL zu einer verbesserten Effizienz der Implementierung beitragen. Mit *Time-critical SDL* (TC-SDL) wird anschließend in Abschnitt 4.2 eine Erweiterung von SDL, die durch Annotationen die Ausführungsreihenfolge von SDL-Agenten beeinflusst, und deren Umsetzung in bestehende Werkzeuge präsentiert.

4.1 Laufzeiteffizienz auf Entwurfsebene

In [BCG09] wurden vier Richtlinien für zeitkritische Spezifikationen auf Entwurfsebene und Maßnahmen zur Erhöhung der Effizienz auf Implementierungsebene vorgestellt. Diese raten den Entwicklern zu kurzen Signalrouten und priority inputs für zeitkritische Signale, dem Aufsplitten langer Transitionen in mehrere kurze Transitionen sowie der Vermeidung von Überlastsituationen. Durch die Verwendung von *copy-by-reference*-Datentypen kann die Effizienz der Implementierung erhöht werden. Anhand eines frühen Zeitstempelns von äußeren Ereignissen, beispielsweise eines *Clear Channel Assessment*-Interrupts (CCA), kann die Genauigkeit, mit der das Ereignis im SDL-System wahrgenommen wird, verbessert werden.

Durch weitere Erfahrungen im Bereich der Spezifikation zeitkritischer Systeme mit SDL wird die Liste der Entwurfsrichtlinien um folgende Punkte ergänzt:

- **Timer statt Taktsignale:** Der Aufwand durch SDL-Signale ist deutlich größer als der Aufwand durch Timer, da hierbei die Ermittlung des Signalweges als zusätzlicher Overhead

hinzukommt. Aus diesem Grund sollte von einem systemweiten Taktgeber abgesehen und die Funktionalität in den einzelnen Agenten in Form von Timern realisiert werden.

- **Timer nur verwenden, wenn notwendig:** Bei Verwendung eines zeitbasierten Mediumslottings, wie es zum Beispiel für *Time Division Multiple Access* (TDMA) vorgenommen wird, ist es sinnvoll, keine Timer für das Zählen einzelner Slots zu verwenden. Stattdessen sollten Timer nur bei Slots ablaufen, bei denen der Knoten aktiv etwas sendet bzw. auf das Ausbleiben eines Ereignisses reagieren muss (zum Beispiel nach einem Timeout). Das Konzept des regelmäßigen Slottings sollte im Hintergrund stehen und nicht eins zu eins in die Spezifikation übernommen werden.

- **Vermeidung von Prozeduren und Konnektoren:** Prozeduren und Konnektoren bieten Möglichkeiten zur Wiederverwendung und zum Kapseln von Funktionalität. Insbesondere die Mächtigkeit von Prozeduren, die unter Anderem das Definieren eigener Zustände erlaubt, wird an vielen Stellen allerdings nicht benötigt und kann durch eine Vergrößerung des Zustandsraumes einen nicht zu vernachlässigenden Aufwand zur Folge haben. Aus diesem Grund sollten unnötige Prozeduren und Konnektoren aus Effizienzgründen vollständig vermieden oder durch SDL-Makros ersetzt werden. SDL-Makros, deren Funktionsweise mit *inline*-Funktionen bzw. C-Makros vergleichbar ist, können zwar zu einer Vergrößerung des Zielcodes führen, erlauben dafür aber die effizientere Kapselung von Funktionalität.

- **Entfernen von künstlichen Treiberkomponenten:** Treiberkomponenten auf Entwurfsebene, die ausschließlich eine Umbenennung von Signalen durchführen, um das restliche System hardwareunabhängig zu entwerfen, sollten vermieden werden, da sie zusätzlichen Aufwand zur Laufzeit nach sich ziehen. Stattdessen sollten Signale, wenn nötig, zur Compilezeit umbenannt werden. Die beste Möglichkeit hierzu ist nach der Erstellung der SDL/PR-Datei und vor der Generierung des Codes in der Zielsprache.

- **Keine nicht-verwendeten Signale deklarieren:** Signale, die in keiner Transition innerhalb des Systems explizit empfangen werden und mit keinem SDL-Kanal verknüpft sind, sollten auch nicht deklariert werden. Beispielsweise werden häufig Signale, die von einem Treiber in dem SDL-Environment erzeugt werden können, in einem speziellen treiberspezifischen SDL-Package deklariert, welches unabhängig von der Verwendung der einzelnen Signale in das SDL-System eingebunden wird. Als Beispiel sei hier das SDL-Package des CC2420-Transceivers zu nennen, dessen Fülle an Events (CCA, SFD. ...) häufig nicht innerhalb des SDL-Systems interessieren. Abhängig von der genauen Implementierung könnten solche Events aufgrund ihrer Deklaration zur Erzeugung von SDL-Signalen während der Laufzeit führen, um erst bei Ausführung des Environment-Agenten aufgrund des fehlenden Signalpfads implizit konsumiert zu werden. Dieser Aufwand sollte zur Laufzeit durch Auskommentieren der entsprechenden Deklarationen vermieden werden.

Diese Richtlinien zielen zwar auf Maßnahmen auf Entwurfsebene ab, sind aber durch Eigenschaften der verwendeten Werkzeuge und Laufzeitumgebung motiviert. Im Konkreten umfassen diese in unserem Fall *IBM Tau* [IBMar], *ConTraST* [FGW06], *SDL Runtime Environment* (SdlRE) und *SDL Environment Framework* (SEnF) [FGJ+05]. Die Aufgaben der einzelnen Werkzeuge sind in Anhang A beschrieben.

4.2 Time-critical SDL (TC-SDL)

Time-critical SDL, kurz TC-SDL, bezeichnet eine durch Annotationen ergänzte Erweiterung von SDL. Sie ist für Szenarien entwickelt, in denen die Ausführungsreihenfolge einzelner ASM-Agenten nicht dem „Zufall“ überlassen werden darf, d.h. der Interpretation des SDL-Standards bezüglich der Serialisierung der nebenläufigen ASM-Agenten. Diese Serialisierung wird immer dann notwendig, wenn ein SDL-System auf einem Prozessor ausgeführt wird, dessen Anzahl an Kernen kleiner ist als die Anzahl an ASM-Agenten. In praxisnahen Anwendungen gilt dies de facto immer. Ohne Erweiterungen, wie sie im Folgenden vorgestellt werden, hängt die Reihenfolge der Serialisierung stark von den verwendeten Werkzeugen ab und ist damit für den Entwickler intransparent und schwer zu verändern. Mit der Einführung von TC-SDL wird die Serialisierung transparenter, indem der Entwickler das verwendete Planungsverfahren, welches die Serialisierung von Agenten steuert, explizit innerhalb der SDL-Spezifikation festlegen kann.

4.2.1 Ziele

Im Folgenden werden die Ziele zusammengefasst, die bei der Umsetzung von TC-SDL verfolgt wurden:

- **Flexible Wahl eines Planungsverfahrens:** Dem Entwickler soll es möglich sein, ein für das Szenario passendes Planungsverfahren innerhalb der Spezifikation auszuwählen. Hierfür soll in der SDL-Laufzeitumgebung *SdlRE* ein Framework geschaffen werden, das unterschiedliche Planungsverfahren zur Verfügung stellt und eine einfache Auswahl und Konfiguration ermöglicht. Die Wahl des Verfahrens und die etwaige Konfiguration sollen außerdem dem modellgetriebenen Entwicklungsgedanken folgend innerhalb der SDL-Spezifikation vorgenommen werden können.
- **Einführung eines prioritätsbasierten Planungsverfahrens:** Ein Planungsverfahren, das Aktivitäten nach Prioritäten ausführt, ist in jedem Echtzeitbetriebssystem Voraussetzung zur Erfüllung von Zeitanforderungen [Sta92]. Durch die Eingliederung eines Planungsverfahrens mit festen Prioritäten in die SDL-Laufzeitumgebung *SdlRE* sollen die Wartezeiten von wichtigen Agenten minimiert werden, wobei die Auswahl dieses Verfahrens und die Zuordnung von Prioritäten an Agenten von dem Entwickler innerhalb der SDL-Spezifikation vorgenommen werden soll.
- **Integration in den bestehenden Entwicklungsprozess:** Der durch SDL ermöglichte modellgetriebene Ansatz soll durch die vorgestellten Maßnahmen nicht beeinträchtigt werden. Vielmehr sollen sich die Maßnahmen in den Entwicklungsprozess eingliedern und weiterhin automatisierte Implementierungen ermöglichen. Dies bedeutet auch, dass der Entwickler weder Anpassungen an dem vorhandenen Quellcode der Laufzeitumgebung noch an dem aus dem SDL-Modell generierten Quellcode vornehmen muss.
- **Keine Verletzung der SDL-Semantik:** Die vorgestellten Maßnahmen sollen den SDL-Standard einschränkend interpretieren, ohne dabei der Semantik zu widersprechen. Zum Beispiel soll die Reihenfolge von Signalen in den Warteschlangen einzelner Agenten nicht verändert werden.
- **Kompatibilität mit vorhandenen Werkzeugen:** Die zur Analyse und Codegenerierung verwendeten Werkzeuge sollen auch für erweiterte SDL-Systeme nutzbar bleiben. Im Speziellen heißt dies, dass vorhandene Werkzeuge (im Konkreten *IBM Tau* [IBMar]) verwendet werden sollen, um ein SDL-System mit den von TC-SDL eingeführten Annotationen

zu ergänzen. Dabei soll die Syntax der erweiterten SDL-Systeme verträglich mit der im SDL-Standard definierten Syntax sein und die Syntaxanalyse mit existierenden Werkzeugen keine zusätzlichen Syntaxfehler melden.

- **Abwärtskompatibilität der Laufzeitumgebung:** Erweiterte SDL-Modelle sollen kompatibel mit bestehenden Versionen der Laufzeitumgebung *SdlRE* bleiben. Dies besagt einerseits, dass ein in TC-SDL spezifiziertes System ebenso ein gültiges SDL-System darstellt, welches durch ein Laufzeitsystem, das keine Kenntnisse von TC-SDL besitzt, in der bisherigen Art und Weise ausgeführt werden kann. Andererseits bedeutet dies, dass SDL-Systeme ohne Erweiterung durch TC-SDL innerhalb der für TC-SDL erweiterten Laufzeitumgebungen lauffähig bleiben sollen. In diesem Fall soll sich das Verhalten des ausgeführten Systems nicht von dem Verhalten bei einer Ausführung des Systems innerhalb einer Laufzeitumgebung, die keine Kenntnisse von TC-SDL besitzt, unterscheiden.

4.2.2 Aufgaben der Laufzeitumgebung

Bei Ausführung eines SDL-Systems werden die strukturellen Elemente der Spezifikation (SDL-Block, SDL-Prozess, ...) durch entsprechende ASM-Agenten (SDL-Agent, SDL-Agentenmenge oder Linkagent) repräsentiert (vgl. Abschnitt 3.2.2). Neben der Verwaltung dieser ASM-Agenten muss das SDL-Laufzeitsystem zusätzlich Interaktion zwischen den Agenten gewährleisten und mit Hilfe von Treibern eine Abstraktion für angeschlossene Geräte, wie zum Beispiel Sensoren oder Aktuatoren, bereitstellen. Die Aufgaben einer SDL-Laufzeitumgebung sind demnach vergleichbar mit den Aufgaben eines Betriebssystems.

Bei der Implementierung eines SDL-Systems und der Laufzeitumgebung stellt sich die Frage nach dem Grad an Nebenläufigkeit, den das System zur Laufzeit besitzen soll. Gemäß der formalen Semantik von SDL anhand asynchroner ASMs, werden alle ASM-Agenten vollständig nebenläufig ausgeführt, ohne Hardwarebeschränkungen in irgendeiner Form beachten zu müssen. In der Realität lässt sich die vollständig *physikalisch* parallele Ausführung in der Regel nicht realisieren [BH93]. Bei einem Hardwaresystem mit Einkernprozessor muss sie sogar vollständig aufgegeben werden. Man kann dennoch auch in diesem Fall zwischen dem möglichen Grad an *logischer* Nebenläufigkeit in Form von Betriebssystem-Prozessen oder Threads unterscheiden, die sich den Prozessor als gemeinsame Ressource teilen. Zum Beispiel weist [San00] explizit darauf hin, dass keine Abhängigkeit zwischen einem Betriebssystem-Prozess und einem Agenten besteht und eine n:1-Abbildung von Agenten auf Betriebssystem-Prozesse möglich ist. Es ist demzufolge möglich, die Menge an Agenten eines SDL-Systems auf mehrere Betriebssystem-Prozesse zu verteilen. Das Planungsverfahren, mit dem Agenten innerhalb eines Betriebssystem-Prozesses ausgeführt werden, ist dabei völlig unabhängig von dem Planungsverfahren, mit welchem den Betriebssystem-Prozessen bzw. Threads der Prozessor zugeteilt wird.

Abbildung 4.1 stellt die Möglichkeiten einer Abbildung der asynchronen ASM-Agenten grafisch dar. Auf formaler Ebene des ASM-Modells ist die Ausführung von Linkagenten, SDL-Agenten und SDL-Agentenmengen völlig nebenläufig. Das Verhalten eines einzelnen SDL-Agenten ist sequentieller Natur und durch die Prozessdefinition innerhalb der SDL-Spezifikation festgelegt. Die Abbildung auf reale Hardware zeigt die unterschiedlichen Grade an Nebenläufigkeit, die das ausgeführte SDL-System besitzen kann. Abbildung 4.1a zeigt eine Verteilung von ASM-Agenten auf mehrere physikalisch verschiedene Objekte (Prozessoren). In Abbildung 4.1b findet die Ausführung des SDL-Systems auf einem physikalischen Objekt statt. Es gibt aber eine logische Nebenläufigkeit anhand von Betriebssystem-Prozessen und Threads. Abbildung 4.1c zeigt

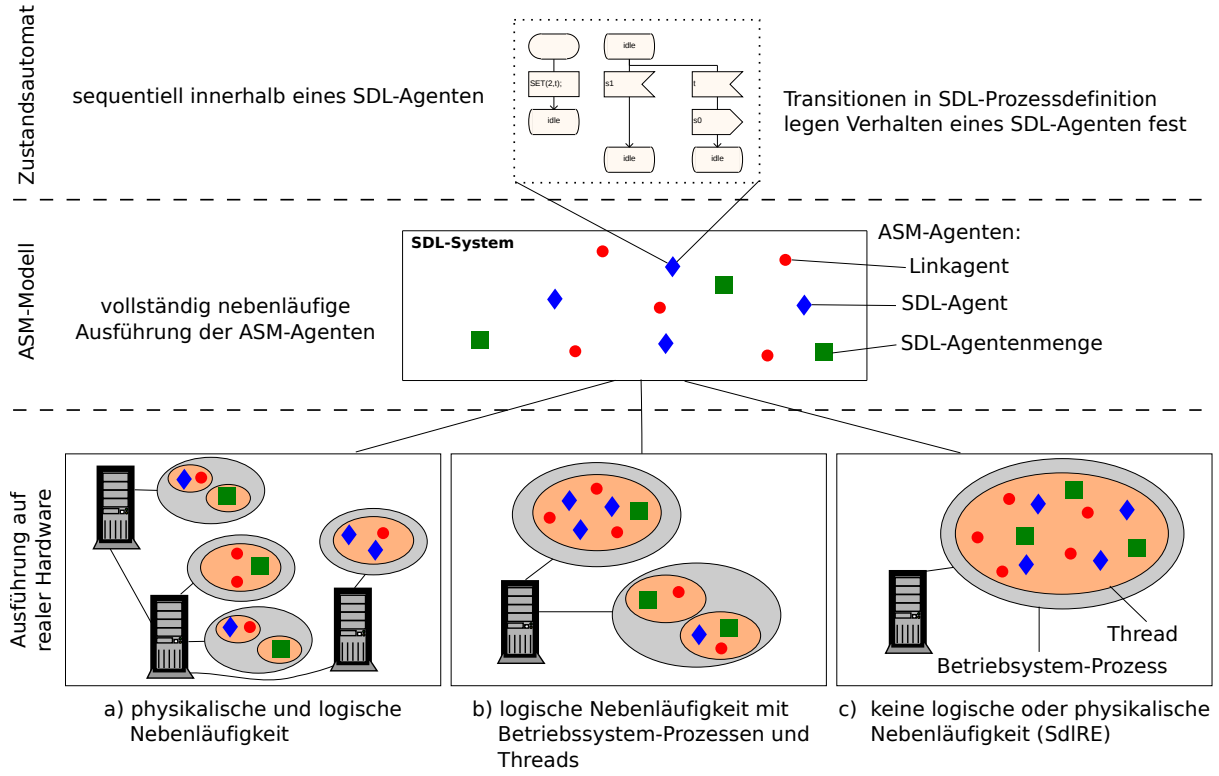
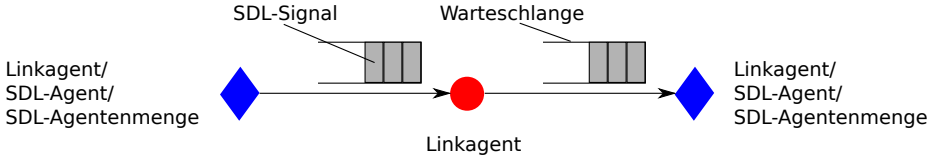


Abbildung 4.1.: Unterschiedliche Grade an physikalischer und logischer Nebenläufigkeit eines SDL-Systems bei Ausführung auf realer Hardware.

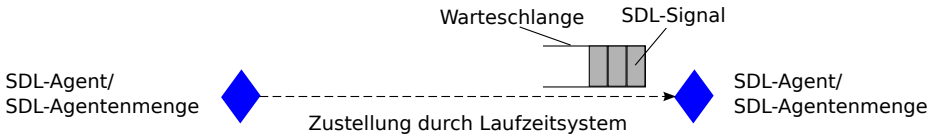
die Umsetzung in der Laufzeitumgebung *SdlRE*, in welcher die in diesem Kapitel beschriebenen Maßnahmen implementiert wurden. Dort werden alle ASM-Agenten in einem Thread ausgeführt und die Ausführungsreihenfolge der Agenten ist vollständig durch die Laufzeitumgebung bestimmt.

Prinzipiell sind hinsichtlich der Kommunikation innerhalb eines SDL-Systems zwei Modelle denkbar, die beide standardkonform unter Berücksichtigung der notwendigen Serialisierung sind, aber eine unterschiedliche Sicht auf die Agenten eines SDL-Systems bieten:

1. **Linkagenten als eigene Aktivitäten:** Im SDL-Standard ist jede Entität, welche den Zustand des Systems verändern kann, durch einen ASM-Agenten beschrieben. Linkagenten sind eine Art dieser Agenten und haben die Aufgabe, Signale aus ihrer Warteschlange in die Warteschlange des nächsten Linkagenten, SDL-Agenten oder der nächsten SDL-Agentenmenge entlang des Signalpfads zuzustellen [Int00]. Dieses Modell ist in Abbildung 4.2a skizziert. Demzufolge sind Linkagenten für die systeminterne Kommunikation verantwortlich und operieren auf den Warteschlangen als Teil eines gemeinsamen Speichers. Eine naheliegende, aber nicht zwingende, Umsetzung wäre die Abbildung aller Agenten auf Threads, die als nebenläufige Komponenten auf gemeinsamem Speicher arbeiten können. Der Signaltransfer entspricht in diesem Fall einer programminternen Kopieroperation.
2. **Kommunikation als Basisdienst der Laufzeitumgebung:** Aus Sicht von modernen Betriebssystemen ist es eher unüblich, die Kommunikation zwischen Aktivitäten selbst als aktive Komponente mit gemeinsamem Speicher zu realisieren. Stattdessen haben sich Verfahren wie *(POSIX-)Signale*, *Pipes* oder *Message-Queues* etabliert, die als Basisdienste durch das Betriebssystem zur Verfügung gestellt werden und eine Kommunikation zwischen Prozessen mit disjunktem Speicher ermöglichen [NS01]. Das Konzept hinter diesen Verfah-



(a) Linkagenten leiten Signale entlang des Signalpfads weiter.



(b) Keine explizite Planung von Linkagenten.

Abbildung 4.2.: Modelle für die systeminterne Interaktion zwischen Agenten eines SDL-Systems.

ren kann ebenso in die Laufzeitumgebung von SDL einfließen, indem Linkagenten nicht als eigenständige Aktivitäten ausgeführt werden, sondern die Zustellung von SDL-Signalen durch einen Basisdienst der Laufzeitumgebung erfolgt [12]. Abbildung 4.2b stellt diese Sicht auf die Interaktion zwischen Agenten schematisch dar. Dieses Modell verlangt keineswegs einen vollständig neuen Mechanismus bezüglich dem Finden von Signalpfaden zwischen SDL-Agenten und SDL-Agentenmengen. Es ist ebenso möglich, dass die bestehende Funktionalität der Linkagenten für die Zustellung der Signale verwendet wird, allerdings ohne die SDL-Signale in eigenen Warteschlangen der Linkagenten zwischenzuspeichern oder die Linkagenten als eigene Entitäten in dem Planungsverfahren explizit zu berücksichtigen.

In *SdlRE* existieren Implementierungen für beide oben aufgeführte Modelle in Form von Optimierungen [Fli09]. Durch Definition des Flags `OPTIMIZE_SIGNALTRANSFER` zur Übersetzungszeit erfolgt der Transfer von Signalen direkt ohne die Zustellung durch explizit ausgeführte Linkagenten. Des Weiteren existiert auch eine Optimierung zum Überspringen von SDL-Agentenmengen, falls diese nur einen SDL-Agenten enthalten (Flag: `OPTIMIZE_DeliverSignal`). Beide Flags sind standardmäßig gesetzt.

Neben den Optimierungen bezüglich der Zustellung von Signalen gibt es in *SdlRE* eine weitere Optimierung (Flag: `EXECUTE_CONTINUOUS`), die einen entscheidenden Einfluss auf die Dauer der Ausführung von Agenten hat [Fli09]. Ist das Flag gesetzt (*default*), endet die Ausführung eines SDL-Agenten erst, wenn keine feuerbereite Transition mehr vorliegt. Dadurch kann der Agent bei einer Ausführung mehrere Signale aus seiner Warteschlange konsumieren. Ähnlich wie bei einem Multitasking-Betriebssystem hat diese Optimierung das Ziel, die Zahl der Kontextwechsel im Sinne von Wechseln zwischen SDL-Agenten klein zu halten, um den Overhead durch „Umschaltvorgänge“ zu minimieren.

Eine weitere bedeutende Rolle für die Optimierung der Ausführung spielt das Flag `OPTIMIZE_EXECUTION`, welches, wenn gesetzt, dafür sorgt, dass nur Agenten, die auch tatsächlich „etwas zu tun“ haben, gemäß einer *First-Come-First-Serve*-Reihenfolge (FCFS) in den Ablaufplan aufgenommen werden. Dadurch werden zum Beispiel SDL-Agenten ohne feuerbereite Transition nicht ausgeführt. Entsprechend der ASM-Semantik würden auch solche Agenten immer ausgeführt werden, unabhängig davon, ob Transitionen aktiv sind und dadurch einen Zustandswech-

[12] Die strenge Auslegung dieses Ausführungsmodells kann zu Problemen bei der Initialisierung des Systems oder der dynamischen Erzeugung von SDL-Blöcken/-Prozessen führen, da der komplette Signalpfad eventuell noch nicht vollständig erstellt wurde. In diesem Fall muss die Zustellung des Signals gegebenenfalls verzögert werden, bis alle notwendigen Strukturen aufgebaut sind.

sel herbeiführen können. `OPTIMIZE_EXECUTION` ist standardmäßig gesetzt und Voraussetzung des prioritätsbasierten Planungsverfahrens.
SdlRE bietet zusammen mit *SEnF* Unterstützung für die PC-Plattformen Linux/Windows, für die AVR- und ARM-Plattformen aus dem Bereich der eingebetteten Systeme und für den Netzwerksimulator *PartsSim* [BGK08]. Eine Besonderheit der AVR- und ARM-Plattform ist die direkte Ausführung des SDL-Systems auf Hardware ohne Unterstützung eines zusätzlichen Betriebssystems, wodurch Overhead eingespart wird und eine vollständige Kontrolle über Hardware und Interrupts gegeben ist.

4.2.3 Auswahl an Planungsverfahren

SdlRE sah bisher keine Wahl zwischen verschiedenen Planungsverfahren vor. Stattdessen wurden alle Agenten gemäß einer *First-Come-First-Serve*-Strategie ausgeführt, wobei sich die Planung bei Aktivieren des `OPTIMIZE_EXECUTION`-Flags auf ausführbare ASM-Agenten beschränkte. Das Planungsverfahren war nicht-präemptiv, sah allerdings eine Konfiguration der Ausführungsdauer mittels des oben genannten `EXECUTE_CONTINUOUS`-Flags vor.
Die Laufzeitumgebung *SdlRE* wurde als Teil dieser Arbeit umstrukturiert, sodass sie nun ein Framework bietet, welches eine Auswahl zwischen verschiedenen Planungsverfahren und die einfache Ergänzung mit weiteren Verfahren erlaubt. Die Wahl und Konfiguration des gewünschten Verfahrens erfolgt zur Compilezeit mit C-Makrodefinitionen, die in einer Header-Datei oder dem (generierten) Makefile definiert werden. Zur Wahl und Konfiguration des zur Laufzeit verwendeten Planungsverfahrens ist der Entwickler eines SDL-Systems nicht gezwungen, den Quellcode oder das Makefile anzupassen. Die entsprechenden Einstellungen lassen sich ebenso anhand von Annotationen im SDL-System vornehmen (siehe Abschnitt 4.2.4). Das benötigte Makefile wird anschließend von *ConTraST* automatisiert generiert. Diese Umsetzung erlaubt sowohl die Konfiguration des Planungsverfahrens basierend auf dem SDL-Modell als auch eine Modifikation durch Ändern der entsprechenden Makros, ohne dass *ConTraST* erneut zur Codegenerierung ausgeführt werden muss.
Bisher werden drei Planungsverfahren zur Verfügung gestellt. Alle drei Verfahren sind nicht-präemptiv, wobei die Optimierung `EXECUTE_CONTINUOUS` weiterhin Möglichkeiten zur Beeinflussung der Ausführungsdauer gibt. Nicht-präemptive Verfahren, die die Unterbrechung eines Agenten innerhalb einer Transition verbieten, stellen einen klaren Nachteil dar, weil sie verhindern, dass hochpriore Agenten unmittelbar nach ihrer Aktivierung ausgeführt werden. Allerdings reduzieren sie die Komplexität und den Overhead der Laufzeitumgebung enorm, da ein präemptives Verfahren mit Aktivitäten auf gemeinsamem Speicher vergleichbar mit Multi-Threading ist und dadurch verlangt, dass der Zugriff auf alle gemeinsam genutzten Datenstrukturen durch Synchronisation geschützt wird. Die aktuelle Version von *SdlRE* sieht solche Maßnahmen nicht vor. Weiterhin haben nicht-präemptive Verfahren den Vorteil, dass die Ausführungszeiten von ähnlichen Transitionen vergleichbar sind und die Ausführung konzeptionell der *run-to-completion*-Semantik von SDL-Transitionen entspricht[13].
Folgende drei Planungsverfahren werden momentan von *SdlRE* unterstützt:

- *Nicht-optimierte Planung ohne Prioritäten*: Dieses Verfahren entspricht der Ausführung ohne `OPTIMIZE_EXECUTION`-Optimierung. Jeder Agent wird ausgeführt und entscheidet selbst, ob eine Transition feuerbereit ist.

  Folgendes Listing zeigt beispielhaft einen Schedule mit diesem Planungsverfahren:

[13] Die Ergebnisse mit Präemption wären ebenfalls standardkonform, aber nicht zwangsweise identisch.

```
|--- Active ---------------
>agentset : p2(1 agents)
|agent    : p2_0 (State=idle)(Queue=|,t) [at 4s:000000us in 0s:999477us]
|agentset : p0(1 agents)
|agent    : p0_0 (State=idle)(Queue=t|) [at 3s:000000us in -0s:00523us]
|agentset : p1(1 agents)
|agent    : p1_0 (State=idle)(Queue=t|) [at 3s:000000us in -0s:00523us]
|link     : p0(Queue=|)
|link     : p1(Queue=|)
|link     : INTERNAL_CHANNEL(Queue=|)
|link     : INTERNAL_CHANNEL(Queue=|)
|agent    : env_0 (State=<null>)(Queue=|)
|agentset : system(1 agents)
|agentset : Environment(1 agents)
|agent    : System_0 (State=<null>)(Queue=|)
|agentset : B(1 agents)
|agent    : B_0 (State=<null>)(Queue=|)
```

Listing 4.1: Beispielhafter Schedule bei nicht-optimierter Planung.

Alle Elemente der Liste werden entsprechend der Reihenfolge von oben nach unten ausgeführt. Die Liste enthält sowohl die Namen (beispielsweise p2_0) als auch die Typen (`agent`, `agentset` oder `link`) der ASM-Agenten. Bei SDL-Agenten ist außerdem der aktuelle Zustand des Zustandsautomaten (beispielsweise idle) und der Inhalt der *input ports* angegeben. Wenn ein *input port* einen SDL-Timer enthält, ist zusätzlich vermerkt, wann der Timer abläuft (beispielsweise in p2_0 zum Zeitpunkt now = 4.0s) . Eine negative Zeitdauer besagt hierbei, dass der Timer bereits abgelaufen ist. Des Weiteren ist in jedem *input port* ein „Pipe-Zeichen“ (|) eingetragen. Signale, die auf der linken Seite des Zeichens stehen, können bereits von dem Agenten konsumiert werden; Signale auf der rechten Seite erreichen den Agenten erst in Zukunft. Diese Fälle entstehen durch SDL-Timer, die noch nicht abgelaufen sind, oder Signale, die über einen verzögernden Kanal übertragen werden. In dem Listing ist in den SDL-Agenten p0_0 und p1_0 jeweils ein Timer-Signal t abgelaufen, welches konsumiert werden kann. SDL-Agent p2_0 kann das Timer-Signal t hingegen erst konsumieren, nachdem es zum Zeitpunkt now = 4.0s abläuft.

Das Listing zeigt, dass auch SDL-Agentenmengen und Linkagenten in den Plan aufgenommen wurden, die aufgrund der in Abschnitt 4.2.2 beschriebenen Optimierungen gar nicht in der Planung erfasst werden müssten. Auch SDL-Agenten ohne konsumierbare Signale (vgl. p2_0) sind in der Planung enthalten und werden „unnötigerweise“ ausgeführt.

- *Optimierte Planung ohne Prioritäten*: Dieses Verfahren stellt das frühere *Default*-Verhalten dar. Es entspricht der Ausführung mit `OPTIMIZE_EXECUTION`-Optimierung, d.h. es werden nur ausführbare ASM-Agenten in den Schedule eingefügt. Aufgrund der in Abschnitt 4.2.2 beschriebenen Optimierungen werden in der Regel nur SDL-Agenten in den Schedule aufgenommen. SDL-Agenten, die zur Zeit nicht ausführbar sind, bei denen aber ein Timer in die Zukunft gesetzt ist, werden in einer eigenen Warteliste verwaltet. Erst bei Ablauf des Timers werden diese Agenten in den eigentlichen Schedule verschoben. Folgendes Listing verdeutlicht das Verfahren anhand von obigem Szenario:

```
|--- Active --------------
>agent    : p1_0 (State=idle)(Queue=t|)
|agent    : p0_0 (State=idle)(Queue=t|)
|
```

```
|--- Signal pending: ---
|agent    : p2_0 (State=idle)(Queue=|,t) [at 4s:000000us in 0s:999198us]
```

Listing 4.2: Schedule bei optimierter Ausführung ohne Prioritäten.

Nur zwei SDL-Agenten, p1_0 und p0_0, sind in dem Beispiel ausführbar. SDL-Agent p2_0 wartet auf das Ablaufen des Timers. Alle anderen Agenten werden erst in die Liste aufgenommen, wenn sie durch Erhalt eines Signals wieder ausführbar werden.

- *Optimierte prioritätsbasierte Planung*: Dieses Planungsverfahren entstand aus der Notwendigkeit zur Unterscheidung der Priorität einzelner Agenten. Wie bei dem vorherigen Planungsverfahren werden bei diesem Verfahren nur ausführbare SDL-Agenten in den Schedule eingefügt. SDL-Agenten mit einem Timer in der Zukunft werden in einer eigenen Warteliste verwaltet und nach Ablauf des Timers in den Schedule verschoben. Der Schedule unterteilt sich in eine konfigurierbare Anzahl an Prioritätsklassen. Bei n Prioritätsklassen wird jedem Agenten eine Priorität aus der Menge {0,1,2,3,...,n-1} zugewiesen, wobei 0 der höchsten und n-1 der niedrigsten Priorität entspricht. Agenten, die durch Zustellung eines Signals oder durch Ablauf eines Timers ausführbar werden, werden entsprechend ihrer Priorität in den Schedule eingefügt. Die Zuweisung von Prioritäten an Agenten ist in Abschnitt 4.2.5 thematisiert. Einfügungen innerhalb einer Prioritätsklasse folgen einer FCFS-Strategie. Für die nächste Ausführung wird der erste Agent in der höchsten nicht-leeren Prioritätsklasse gewählt.

  Folgendes Listing zeigt beispielhaft eine Ausführung mit dem bekannten Szenario und der prioritätsbasierten Planung. Hierzu wurde die Anzahl der Prioritätsklassen auf fünf gesetzt.

```
|--- Active --------------
Priority: 0
>agent    : p0_0 (State=idle)(Queue=t|)
Priority: 1
>agent    : p1_0 (State=idle)(Queue=t|)
Priority: 2
Priority: 3
Priority: 4
|
|--- Signal pending: ---
|agent    : p2_0 (State=idle)(Queue=|,t) [at 4s:000000us in 0s:999660us]
```

Listing 4.3: Schedule bei optimierter prioritätsbasierter Ausführung.

In dem Beispiel wurde die Priorität von SDL-Agent p0_0 (willkürlich) auf 0 gesetzt, also auf die höchste Prioritätsstufe. Der Agent p1_0 erhielt die Priorität 1 und wird erst ausgeführt, wenn die Ausführung von p0_0 abgeschlossen ist. Die Priorität des Agenten p2_0 spielt zu diesem Zeitpunkt noch keine Rolle, da sein Timer noch nicht abgelaufen ist.

Im Gegensatz zu den vorherigen Planungsverfahren ist die prioritätsbasierte Planung nicht fair, da nicht sichergestellt ist, dass niederpriore Agenten in jedem Fall ausgeführt werden. Allerdings sollte dies keinen Nachteil darstellen, denn wenn solche Fälle auftreten, sind entweder die Prioritäten ungünstig vergeben oder das System ist überlastet. Außerdem ist Fairness für die angestrebte Verwendung in eingebetteten Systemen eher zweitrangig.

Selbst mit der prioritätsbasierten Planung sollte die Serialisierungsverzögerung eines hochprioren Agenten durch einen niederprioren Agenten nicht unterschätzt werden. Um die Verzögerung zu verringern, wurde daher die Bedeutung von EXECUTE_CONTINUOUS leicht

eingeschränkt: Die Laufzeitumgebung prüft nun nach Ausführung einer Transition, ob in der Zwischenzeit ein höherpriorer Agent durch ein Signal oder einen Timer ausführbar wurde. Wenn dies der Fall ist, wird der gerade ausgeführte Agent nach Ausführung seiner Transition unterbrochen, unabhängig davon, ob dieser weiterhin ausführbar ist.

Neben diesen drei Planungsverfahren sind weitere Verfahren wie zum Beispiel Shortest-Job-First oder Rate Monotonic denkbar. Diese Verfahren könnten entweder durch das Sammeln statistischer Daten zur Laufzeit oder anhand zusätzlicher Informationen durch den Entwickler parametrisiert werden. Der folgende Abschnitt zeigt anhand bereits unterstützter Planungsverfahren, wie das Verfahren auf SDL-Ebene gewählt und konfiguriert wird, und wie neue Verfahren dem Framework hinzugefügt werden können.

4.2.4 Wahl des Verfahrens und Konfiguration

Die Wahl und Konfiguration der unterschiedlichen Planungsverfahren sollte bereits auf Ebene des SDL-Modells möglich sein, um dem Entwickler manuelle Anpassungen des Makefiles oder gar des Quelltextes zu ersparen. Dabei sollte jedoch das SDL-Modell weiterhin mit bestehenden Werkzeugen (in unserem Fall *IBM Tau* [IBMar]) erstellt, bearbeitet und analysiert werden können. Aus diesem Grund schieden neue graphische SDL-Objekte oder SDL-Bezeichner zur Angabe des Planungsverfahrens aus. Annotationen bieten eine Möglichkeit, um das gewünschte Planungsverfahren dennoch innerhalb des SDL-Systems konfigurieren zu können. Es handelt sich dabei um formale SDL-Kommentare, die bei der Syntaxanalyse ignoriert werden. Auf diese Weise kann der Entwickler durch spezielle Schlüsselworte innerhalb eines SDL-Kommentars das Laufzeitsystem konfigurieren. Die Kommentare werden bei der Codegenerierung durch *ConTraST* ausgewertet, der die entsprechenden Annotationen in C-Makros und C++-Quellcode einfließen lässt. Wie bereits in Abschnitt 4.2.3 erwähnt, setzt *ConTraST* die Annotationen in dem generierten Makefile um, sodass ein schnelles Umschalten ohne erneute Codegenerierung möglich ist.

Abbildung 4.3 zeigt, wie auf Systemebene das Planungsverfahren innerhalb des Kopfsymbols spezifiziert werden kann. Anhand des Schlüsselwortes **SCHEDULING** erkennt *ConTraST*, dass der SDL-Kommentar Annotationen zur Konfiguration des Planungsverfahrens enthält. Die genaue Syntax ist in Anhang B in Form einer kontextfreien Grammatik gegeben. In der Regel folgt dem Schlüsselwort **SCHEDULING** eine Menge von Schlüssel/Wert-Paaren. Eine Unterscheidung zwischen Groß- und Kleinschreibung findet weder bei den Schlüsseln noch bei den

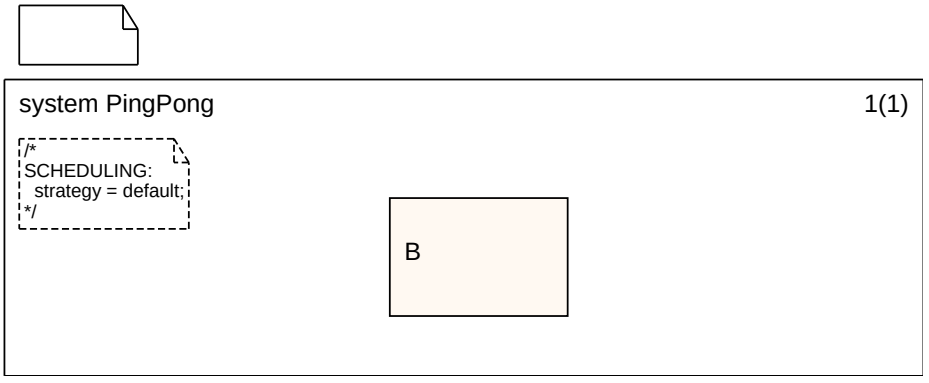


Abbildung 4.3.: Beispiel zur Wahl des Planungsverfahrens innerhalb der SDL-Spezifikation. Das Beispielsystem ist auf der beiliegenden CD unter `/SDL/scheduling_beispiel` zu finden.

Werten statt (*case insensitive*). In Abbildung 4.3 folgt ausschließlich der Schlüssel `strategy`, der spezifiziert, welches Planungsverfahren verwendet werden soll. Das spezifizierte Planungsverfahren gilt dabei für die systemweite Planung der Agenten, d.h. die Ausführung aller Agenten richtet sich nach dem gleichen Planungsverfahren. Der Schlüssel `strategy` kann folgende Werte annehmen:

- `non-optimized`: Die nicht-optimierte Planung wird verwendet.
- `default`: Die optimierte Planung ohne Prioritäten wird verwendet.
- `static-priorities`: Die prioritätsbasierte Planung wird verwendet.

Abhängig von dem Wert von `strategy`, im Beispiel `default`, können weitere Schlüssel/ Wert-Paare für eine erweiterte Konfiguration des Verfahrens angegeben werden. Anhang B geht auf die genauen Konfigurationsmöglichkeiten ein. Wird für `strategy` kein Wert angegeben, wird das optimierte Planungsverfahren ohne Prioritäten (`default`) verwendet.

4.2.5 Prioritätsbasiertes Planungsverfahren

Die vor Beginn dieser Arbeit vorhandene Version der Laufzeitumgebung *SdlRE* unterstützte nur das nicht-optimierte Planungsverfahren und die optimierte Planung ohne Prioritäten. Diese beiden Verfahren unterscheiden sich darin, *was* in den Schedule aufgenommen wird, führen die Agenten aber beide gemäß einer FCFS-Strategie aus. Mit dem prioritätsbasierten Planungsverfahren wurde die Laufzeitumgebung nun um eine Strategie erweitert, die aktiv die Schedulingreihenfolge beeinflusst. Das Verfahren bildet die Grundlage für die Einteilung von Agenten in unterschiedliche Prioritätsklassen. Dabei sollte allerdings beachtet werden, dass ein Planen nach festen Prioritäten nicht die Auslastung erreichen kann wie zum Beispiel ein Planen nach Fristen [Zöb08] (siehe auch Abschnitt 2.2.1). Das vorgestellte Planungsverfahren mit festen Prioritäten ist andererseits jedoch effizienter als ein Planen nach Fristen.

4.2.5.1 Fallbeispiel: Spezifikation des prioritätsbasierten Planungsverfahrens

Anhand des Beispiels in Abbildung 4.4 soll die Verwendung des prioritätsbasierten Planungsverfahrens vorgestellt werden. Die Syntax aller Parameter ist in Anhang B formal definiert. Neben der Wahl des Planungsverfahrens innerhalb des Kopfsymbols des SDL-Systems (`strategy = static-prorities`), werden in diesem Beispiel die Anzahl der Prioritätsklassen (`levels = 5`) und die *Default*-Priorität (`priority = 4`) festgelegt. Das Beispiel zeigt außerdem die Zuweisung der Priorität 3 an den Block-Typ SysBlockType. OneBlock, der diesen Block-Typ instanziiert, überschreibt diese Priorität aber mit 0, sodass die *Default*-Priorität von Agenten, die aus OneBlock zur Laufzeit hervorgehen, nicht 3 ist sondern 0. Dem Block BlockWithoutPriority wurde keine explizite Priorität zugewiesen, sodass für ihn die *Default*-Priorität 4 aus dem Kopfsymbol des Systems gilt. Es sei allerdings an dieser Stelle schon angemerkt, dass den SDL-Komponenten innerhalb eines Blocks ebenfalls Prioritäten zugewiesen werden können, die die *Default*-Prioritäten der Blöcke überschreiben können. Anhand von Abbildung 4.4 lässt sich demnach nicht schlussfolgern, dass alle Agenten, die aus OneBlock bzw. BlockWithoutPriority zur Laufzeit hervorgehen, die Priorität 0 bzw. 4 erhalten. Die Möglichkeiten der Zuweisung von Prioritäten werden im folgenden Abschnitt im Detail behandelt.

Das folgende Listing zeigt, wie die Annotationen aus dem Kopfsymbol des SDL-Systems Eins-zu-Eins durch *ConTraST* in dem generierten Makefile umgesetzt werden, um sie nachträglich

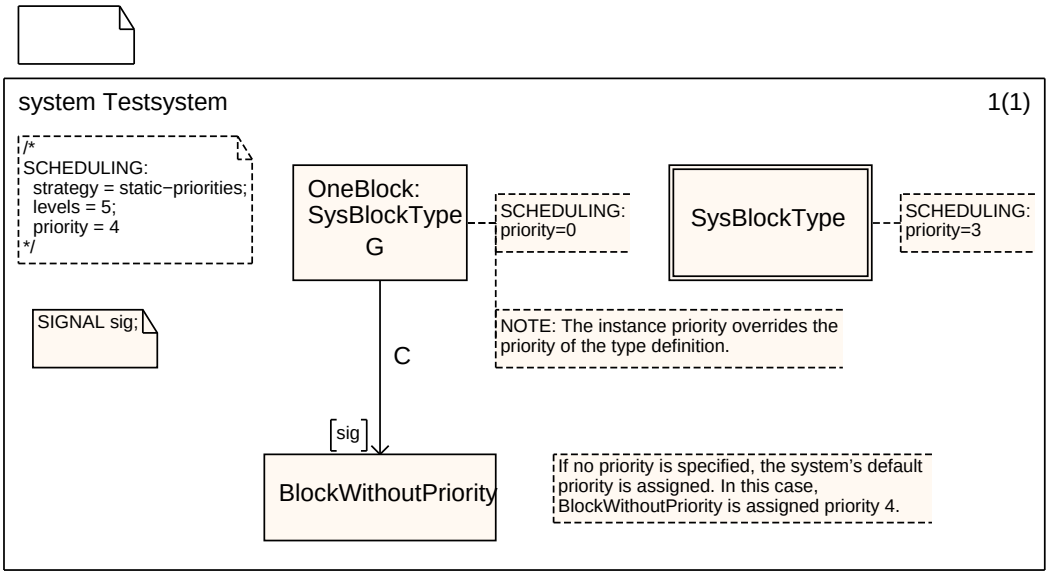


Abbildung 4.4.: Wahl des prioritätsbasierten Planungsverfahrens auf Systemebene. Das Beispiel ist auf der beiliegenden CD unter `/SDL/scheduling_beispiel` zu finden.

ohne erneute Codegenerierung verändern zu können. Neben der Wahl des Verfahrens (`SDL_SCHEDULER=SCHEDULING_POLICY_PRIO`) ist ebenfalls die Anzahl an Prioritätsstufen (`SDL_SCHEDULER_PRIORITIES=5`) und die Standardpriorität (`SDL_SCHEDULER_DEFAULT_PRIORITY=4`) in dem Makefile zu finden.

```
1 ...
2 # Scheduling information specified by SDL system (e.g., scheduling strategy)
3 SCHEDULING_OPTIONS =  -DSCHEDULING_POLICY_PRIO=1 -DSCHEDULING_POLICY_GENERIC=2
    -DSCHEDULING_POLICY_NON_OPTIMIZED=3   -DSDL_SCHEDULER=SCHEDULING_POLICY_PRIO
     -DSDL_SCHEDULER_DEFAULT_PRIORITY=4  -DSDL_SCHEDULER_PRIORITIES=5
4 ...
```

Listing 4.4: Ausschnitt aus dem von *ConTraST* generierten Makefile zeigt das verwendete Planungsverfahren sowie dessen Konfiguration.

Im Unterschied zu priority inputs hat die Priorität eines Agenten einen Einfluss auf die globale Ausführungsreihenfolge aller ausführbaren Agenten. Priority inputs hingegen beeinflussen nur die Transitionsauswahl innerhalb eines Agenten [Int99]. Ihre Semantik wird durch das prioritätsbasierte Planungsverfahren nicht verändert.

4.2.5.2 Zuordnung von Prioritäten

Es war die Zielsetzung bei der Zuordnung von Prioritäten, dem Entwickler möglichst viele Freiheiten zu geben, die Priorität eines Agenten zu spezifizieren. Die Zuweisung der Priorität sollte auf unterschiedlichen Ebenen innerhalb der Hierarchie eines SDL-Systems möglich sein, um zum Beispiel allen Agenten, die zur Laufzeit aus einem Block hervorgehen, eine *Default*-Priorität zuzuordnen. Die Syntax für die Zuweisung einer Priorität ist unabhängig von dem spezifischen SDL-Objekt immer identisch und wurde bereits in Abbildung 4.4 eingeführt. Die Vergabe einer Priorität geschieht in einer SDL-Kommentarbox, die mit dem entsprechenden Objekt verknüpft ist, und wird durch das Schlüsselwort **SCHEDULING** eingeleitet. Mit `priority = <prio>` folgt dem Schlüsselwort die Zuweisung der Priorität <prio>.

Folgende SDL-Objekte werden unterstützt:

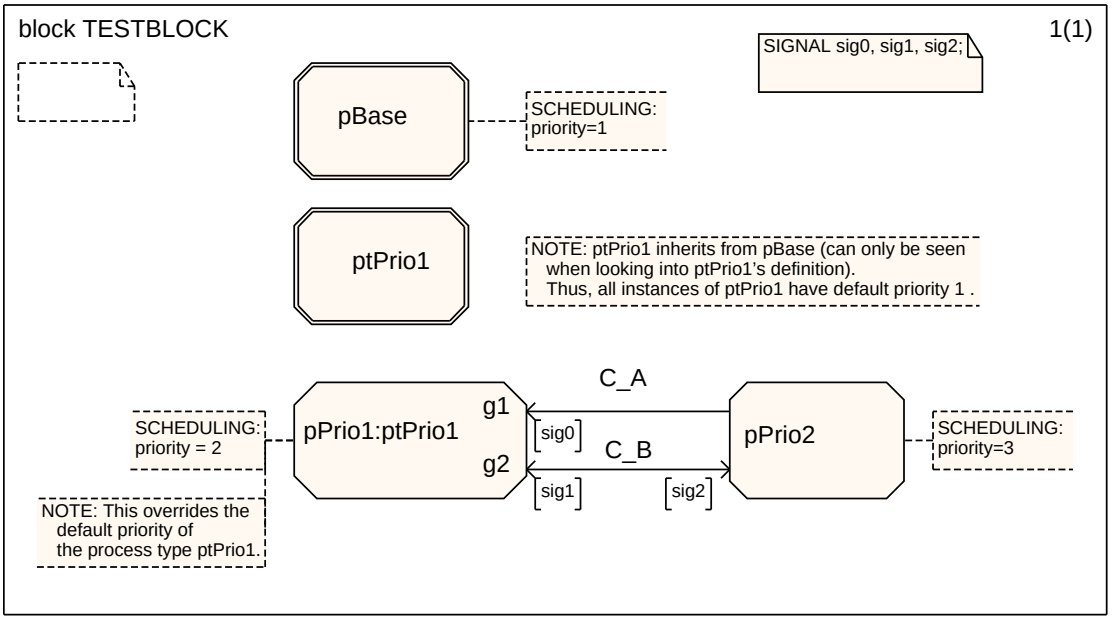


Abbildung 4.5.: Vergabe von Prioritäten an Prozesse innerhalb eines Blocks. Das Beispiel ist auf der beiliegenden CD unter `/SDL/scheduling_beispiel` zu finden.

1. SDL-Block-Ebene (siehe Abbildung 4.4): Um mehreren Prozessen innerhalb eines SDL-Blocks eine Priorität zuzuweisen, können Prioritäten mit SDL-Blöcken, SDL-Block-Typen und SDL-Block-Instanzen verknüpft werden.
2. SDL-Prozess-Ebene (siehe Abbildung 4.5): Die Zuweisung einer Priorität an einen Prozess scheint zunächst die naheliegendste Möglichkeit, da ein Prozess den SDL-Agenten definiert, dessen Verhalten explizit von dem Entwickler spezifiziert wurde. Es wird die Zuweisung zu SDL-Prozessen, SDL-Prozess-Typen und zu SDL-Prozess-Instanzen unterstützt. Ebenso wird Vererbung zwischen SDL-Prozessen unterstützt, bei der der Subtyp die Priorität des Supertyps erhält. Durch explizite Vergabe einer Priorität an den Subtyp kann die Priorität überschrieben werden (siehe pBase und ptPrio1 in Abbildung 4.5).
3. SDL-Service-Ebene (siehe Abbildung 4.6): Prioritäten können SDL-Services, SDL-Service-Typen und SDL-Service-Instanzen zugeordnet werden. SDL-Services wurden zwar durch die allgemeineren composite states ersetzt und sind seit SDL-2000 kein Teil des SDL-Standards mehr [Int99], wurden aber aufgrund der fehlenden Werkzeugunterstützung für composite states für die Vergabe von Prioritäten berücksichtigt. Aus einem SDL-Service entsteht zur Laufzeit kein Agent und somit auch keine Entität, die in dem Planungsverfahren berücksichtigt wird. Stattdessen gehen das Verhalten und die Priorität eines Services in das Verhalten des SDL-Agenten ein, der aus dem zugehörigen SDL-Prozess hervorgeht. Die resultierende Priorität des SDL-Agenten entspricht der höchsten Priorität der enthaltenen SDL-Services (siehe auch Abschnitt 4.2.5.3).

Die Zuweisung von Prioritäten an Block-, Prozess- oder Service-Typen hat zur Folge, dass jede Instanz des gleichen Typs die gleiche Priorität erhält, außer die Priorität wird explizit bei der Instanziierung gesetzt. Zeitkritische Systemteile, wie zum Beispiel eine Synchronisationskomponente, die in einem SDL-Package als Prozess-Typ spezifiziert wurde, können dadurch eine systemunabhängige Priorität erhalten, die bei allen Instanziierungen identisch ist.

Mit der Vergabe von Prioritäten an Block-, Prozess- oder Service-Instanzen kann der Entwickler gezielt die Priorität setzen oder überschreiben. Als Beispiel kann hier ein Multiplexer dienen,

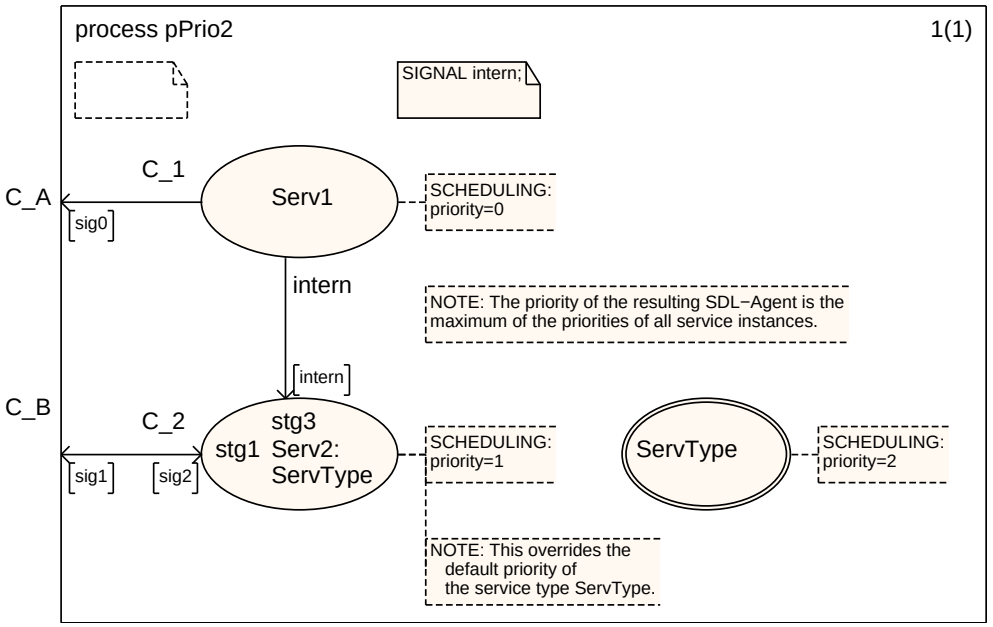


Abbildung 4.6.: Vergabe von Prioritäten an SDL-Services innerhalb eines Prozesses. Das Beispiel ist auf der beiliegenden CD unter `/SDL/scheduling_beispiel` zu finden.

der zweimal innerhalb eines SDL-Systems in zwei unterschiedlich zeitkritischen Systemteilen instanziiert wird. Durch explizite Zuweisung von unterschiedlichen Prioritäten kann diese Tatsache in die Priorisierung des zeitkritischen Multiplexers einfließen.

4.2.5.3 Bestimmung der Priorität eines Agenten

Die Priorität, mit der ein Agent in den Schedule eingefügt wird, wird einmalig zur Laufzeit bestimmt. Eine statische Bestimmung ist aufgrund von Vererbung und der möglichen dynamischen Erzeugung von SDL-Instanzen nicht möglich. Prioritäten werden ausschließlich SDL-Agenten zugewiesen. Andere ASM-Agenten, wie zum Beispiel Linkagenten, werden stets mit der höchsten Priorität ausgeführt. In der Regel werden aber nach der Systeminitialisierung ausschließlich SDL-Agenten ausgeführt (siehe Optimierungen in Abschnitt 4.2.2). Die Initialisierung eines Prozesses hat immer Vorrang und geschieht mit maximaler Priorität. Erst nachdem der SDL-Agent seine Starttransition durchlaufen hat, wird ihm die spezifizierte Priorität zugewiesen. Aus diesem Grund können während der Systeminitialisierung nur wenige Aussagen über die tatsächliche Ausführungsreihenfolge getroffen werden. Diese Tatsache sollte aber nicht stören, da die Systeminitialisierung unabhängig von dem Planungsverfahren immer als Ausnahmesituation gesehen werden muss, weil das Zeitverhalten durch die vielen Speicherallokationen und Initialisierungen von Hardwarekomponenten ohnehin sehr schwer vorhersagbar ist.

Die Priorität, die ein SDL-Agent letztendlich erhält, hängt von mehreren Faktoren ab. Allgemein gilt die Regel, dass die Priorität eines SDL-Agenten von der Spezifikation eines SDL-Prozesses startend von „innen nach außen“ gesucht wird. Die genaue Vorgehensweise zur Bestimmung einer Priorität wird mit folgendem Pseudocode verdeutlicht:

```
1 priority := undefined;  // resulting priority of SDL-Agent
2
3 // Check if there are included services with specified priorities
4 if (getServicesPriority() ≠ undefined):
5   // If there are several services, the maximal priority is returned
```

```
6   priority := getServicesPriority()
7
8 else:
9   // Check if instance of process or type definition
10  // of process has a specified priority
11  if (getNodePriority(self) ≠ undefined):
12    priority := getNodePriority(self)
13
14  // Check if one of the father nodes (blocks) or
15  // their type definitions obtain a priority
16  else:
17    node := self.father
18
19    while (node.type ≠ system):
20      if (getNodePriority(node) ≠ undefined):
21        priority := getNodePriority(node)
22        break
23      else:
24        node := node.father
25
26    if (priority = undefined):
27      // Check if there is (default) priority specified on system level
28      if (self.system.priority ≠ undefined):
29         priority := self.system.priority
30
31      // Use default priority of SdlRE
32      else:
33         priority := SdlRE.priority
```

Listing 4.5: Pseudocode zum Ermitteln der Priorität eines SDL-Agenten.

Der Pseudocode zeigt, wie zunächst in den Zeilen 4 bis 6 geprüft wird, ob ein SDL-Service in dem SDL-Prozess definiert wurde und ob dieser eine Priorität besitzt. Ist dies der Fall, erhält der SDL-Agent die ermittelte Priorität. Die Priorität eines SDL-Services hat demnach, wenn spezifiziert, immer Vorrang. Die Funktion getServicesPriority() ist in folgendem Listing näher beschrieben:

```
1 getServicesPriority():
2   prio := undefined;
3   // Determine maximum of priorities specified in all services
4   ∀ (service ∈ ServiceInstancesInProcess):
5     if (service.priority ≠ undefined):
6       prio := maxPrio(prio, service.priority);
7
8     else :
9       if (service.typeDefinition ≠ undefined):
10        if (service.typeDefinition.priority ≠ undefined):
11          prio := maxPrio(prio, service.typeDefinition.priority);
12
13  return prio;
```

Listing 4.6: Hilfsfunktion getServicesPriority() zum Bestimmen der Priorität einer Service-Menge.

In der Funktion werden alle SDL-Service-Instanzen (und SDL-Services) innerhalb des SDL-Prozesses durchsucht. Wurde einer Instanz keine Priorität zugewiesen, wird geprüft, ob eine entsprechende Typ-Definition existiert und ob dieser eine Priorität zugeordnet wurde. Unabhängig von der Definition der Priorität in der Instanz oder dem Typ, wird dem Prozess und damit dem

zur Laufzeit resultierenden SDL-Agenten das Maximum aller Instanz-Prioritäten zurückgegeben ($maxPrio(\ldots)$, Hinweis: 0 ist die systemweit höchste Priorität).

Sind keine Services mit Priorität gefunden, wird in den Zeilen 11 und 12 in Listing 4.5 geprüft, ob der SDL-Prozess eine Priorität besitzt. Die verwendete Hilfsfunktion getNodePriority(Node) hat dabei folgende Funktionalität:

```
1  getNodePriority(n):
2    // Determine priority of instance
3    if (n.priority ≠ undefined):
4        return n.priority;
5
6    else :
7        if (n.typeDefinition ≠ undefined):
8          // Determine priority of type if present
9          if (n.typeDefinition.priority ≠ undefined):
10           return n.typeDefinition.priority;
11
12         // Determine priority of supertype if present
13         else:
14           if (n.typeDefinition.superType ≠ undefined)
15             return n.typeDefinition.superType.priority;
16
17   return undefined;
```

Listing 4.7: Die Hilfsfunktion getNodePriority() ermittelt die Priorität eines SDL-Prozesses/Blocks.

Die Funktion prüft in Zeile 3, ob der Instanz eine Priorität zugeordnet ist und gibt diese ggf. zurück. Andernfalls wird, sofern vorhanden, die Priorität der Typ-Definition zurückgegeben. Falls diese ebenfalls nicht vorhanden ist, aber der Typ der Prozess-Instanz von einem anderen Typen erbt, wird dessen Priorität verwendet, sofern sie existiert.

Wurde auch dem SDL-Prozess selbst keine Priorität gegeben, wird entsprechend der Hierarchie des SDL-Systems der nächsthöhere Block gesucht, welcher eine Priorität besitzt (Zeilen 19 bis 24 in Listing 4.5). Ist auch diese Suche erfolglos, greifen die *Default*-Prioritäten, d.h. die Priorität aus dem Kopfsymbol des SDL-Systems oder, falls diese ebenfalls nicht vorhanden ist, die in *SdlRE* definierte *Default*-Priorität.

4.2.5.4 Blockieren niederpriorer Agenten

Das bis dato vorgestellte prioritätsbasierte Planungsverfahren leidet unter dem Problem der fehlenden Präemption. Ohne Präemption muss selbst der Agent mit der höchsten Priorität warten, bis die Ausführung eines niederprioren Agenten abgeschlossen ist. Abhängig von der Laufzeit der aktiven Transition kann daher die Serialisierungsverzögerung auch bei der Planung mit Prioritäten sehr hoch sein. Dieses Problem wurde bereits in [BCG09] erkannt und diskutiert. Als Lösung wurde vorgeschlagen, die Ausführung niederpriorer Agenten zeitweise zu unterdrücken. Auf diese Weise kann ein hochpriorer Agent die Laufzeitumgebung a priori konfigurieren, sodass vor einem kritischen Zeitabschnitt nur noch Agenten ab einer gegebenen Priorität ausgeführt werden. Bei einer sinnvollen Vergabe von Prioritäten ist es dadurch möglich, die Serialisierungsverzögerung von zeitkritischen Transitionen zu minimieren.

Um die Beeinflussung der Laufzeitumgebung aus SDL heraus zu ermöglichen, wurde das SDL-Package PriorityScheduling erstellt (siehe Abbildung 4.7). Das Package enthält zwei Prozeduren, disallowAgents() und allowAllAgents(), die einen *Wrapper* für die entsprechenden Funktionen der

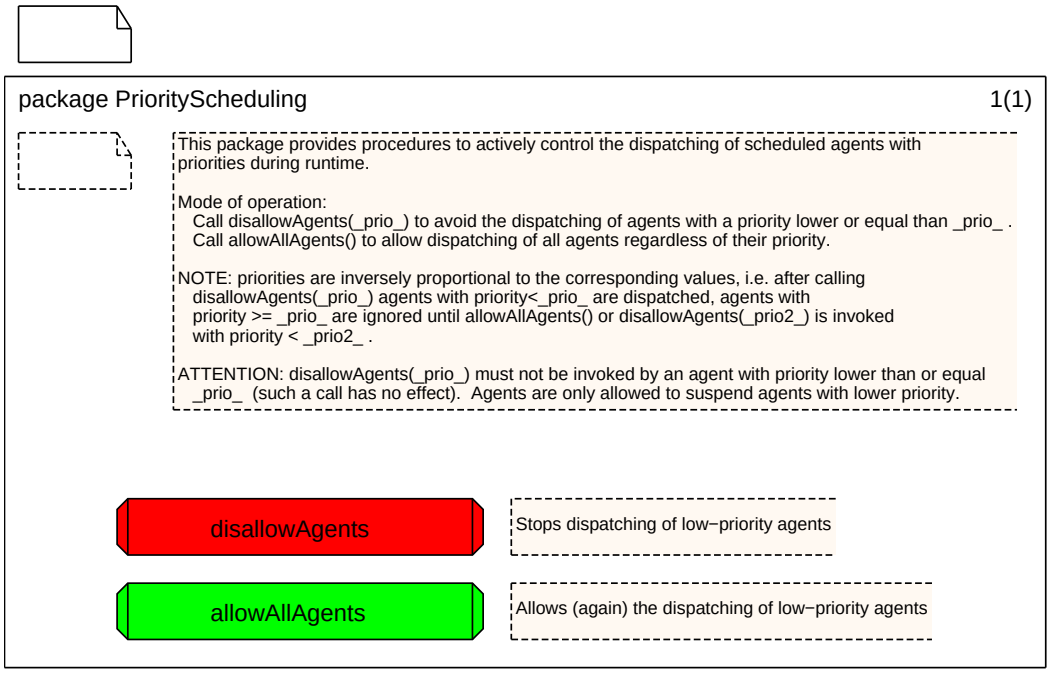


Abbildung 4.7.: Mit Hilfe der Prozeduren disallowAgents und allowAllAgents kann die Ausführung niederpriorer Prozesse zur Laufzeit unterbunden werden. Das SDL-Package ist auf der CD unter `/Werkzeuge/SEnF/sdl_packages/plugins/PriorityScheduling/` zu finden.

Laufzeitumgebung implementieren. Die Prozedur disallowAgents() besitzt einen Parameter, der angibt, ab welcher Prioritätsklasse keine Agenten mehr ausgeführt werden dürfen. Ein Agent, der disallowAgents() aufruft, muss eine Priorität besitzen, die höher ist als die Priorität, die er bei Aufruf der Prozedur als Argument mitgibt. Diese Restriktion wird einerseits der Tatsache gerecht, dass ein Agent nicht in der Lage sein sollte, höherpriore Agenten zu blockieren, andererseits wird auch verhindert, dass sich ein Agent selbst blockiert. Durch Aufruf der Prozedur allowAllAgents() werden alle Agenten wieder zugelassen.

Verglichen mit Präemption ist das Blockieren der Ausführung natürlich weniger komfortabel und flexibel, erreicht aber bis zu einer möglichen Einführung von Präemption in *SdlRE* eine spürbare Verbesserung. Eine große Gefahr sind allerdings die Risiken des „Verhungerns“ eines Agenten oder sogar eines Systemstillstands, wenn der Aufruf von allowAllAgents ausbleibt.

4.2.6 Umgang mit dem SDL-Environment

Im Vergleich zu anderen SDL-Agenten nimmt das SDL-Environment eine Sonderrolle ein, da es der einzige Agent ist, dessen Ausführung durch systemexterne Ereignisse angeregt werden kann. Es stellt sich daher die Frage, nach welchen Regeln die Ausführung des SDL-Environments zu planen ist. In wenig ausgelasteten Systemen ist diese Frage schnell beantwortet, da das Environment während der zahlreichen *idle*-Zeiten ausgeführt werden kann[14]. Hingegen bekommt die Frage in Systemen mit andauernder Last eine besondere Relevanz, da sich die Lastsituation mit der Anzahl der Ausführungen des Environments verbessern oder verschlechtern kann. Bei dem prioritätsbasierten Ansatz stellt sich außerdem die Frage, was die sinnvollste Priorität für das Environment ist. Prinzipiell käme nur die niedrigste Priorität in Frage, da an-

[14]Wenn man den Energieaspekt in die Betrachtung einfließen lässt, können diese Zeitintervalle auch zum Sparen von Energie benutzt werden, zum Beispiel durch Reduzieren der Taktrate.

dernfalls Agenten mit geringerer Priorität gar nicht mehr ausgeführt werden würden. Allerdings könnte dies dazu führen, dass das Environment gar nicht mehr oder zu selten ausgeführt wird.

Das oberste Ziel sollte es daher sein, das Environment *so häufig und so schnell wie erforderlich* auszuführen. Diese Kriterien sind jedoch sowohl treiber- als auch szenariospezifisch, sodass sich keine generische Strategie zum Erreichen dieses Ziels finden lässt. Beispielsweise sollte in einem System, das besonders schnell auf externe Stimuli reagieren muss, das Environment häufiger ausgeführt werden, um Ereignisse schnell in das System und mögliche Antworten ebenso schnell aus dem System heraus zu bekommen. Des Weiteren können zu dem jetzigen Zeitpunkt Treiber, die während der Ausführung des Environments angeschlossene Geräte pollen, nicht ausgeschlossen werden. Daher muss ebenfalls vorgesehen werden, dass das Environment regelmäßig ausgeführt wird, ohne dass ein externes Ereignis auftrat oder ein Signal an das Environment gesendet wurde. Es lassen sich demnach folgende Ursachen identifizieren, die eine Ausführung des Environments erfordern:

- **Interrupts**: Externe Ereignisse, die durch Hardware-Interrupts signalisiert werden (zum Beispiel der Empfang eines Rahmens)
- **Aufträge**: Signale von dem SDL-System an das Environment, um das Environment mit Aufgaben zu beauftragen (zum Beispiel Senden eines Rahmens)
- **Polling**: Ausführen des Environments, um Hardware zu pollen, wenn diese zum Beispiel keine Interrupts unterstützt

Diese drei Ursachen sollten abhängig von dem Szenario mit unterschiedlicher Priorität behandelt werden. In einem drahtlosen Sensornetzwerk sollten zum Beispiel Aufträge eine besonders hohe Priorität auf einem Messknoten erhalten, weil die Messdaten schnell versendet werden müssen. Da allerdings die Messdaten anderer Knoten für einen Messknoten nicht relevant sind, wäre es ausreichend, das Environment nach einem Interrupt verzögert auszuführen.

Anhand von Annotationen erhält der Entwickler die Möglichkeit, die Relevanz der drei Ursachen zu bewerten. Abbildung 4.8 zeigt deren Verwendung innerhalb des Kopfsymbols eines SDL-Systems. Die drei relevanten Parameter sind:

- `env-signal_in-threshold`: Spezifiziert die Anzahl an SDL-Signalen in der Eingangswarteschlange des SDL-Environments, die eine Ausführung des Environments erzwingen. Ist der Wert mit 1 spezifiziert, erhält das Environment die höchste Priorität, sobald ein Signal an dieses gesendet wurde.
- `env-signal_out-threshold`: Spezifiziert die Anzahl an SDL-Signalen, die von dem Environment *möglicherweise* erzeugt werden und zu einer Ausführung des Environments führen. Durch das *möglicherweise* soll angedeutet werden, dass die Anzahl an Signalen aus dem Environment erst bekannt ist, wenn dieses ausgeführt wird. Dies liegt unter Anderem an dem Design von *SEnF*. Beträgt der Wert 1, erhält das Environment die höchste Priorität, sobald ein Signal *möglicherweise* erzeugt werden kann.
- `env-agent-threshold`: Gibt die maximale Anzahl an Ausführungen systeminterner Agenten an, bevor das SDL-Environment ausgeführt werden muss. Dieser Parameter garantiert, dass das Environment unter allen Umständen ausgeführt wird. Sind Treiber vorhanden, die mit Polling bei Ausführung des Environments arbeiten, sollte dieser Wert nicht zu groß gewählt werden. Melden sich alle Ereignisse per Hardware-Interrupt, kann der Wert beliebig groß gewählt werden.

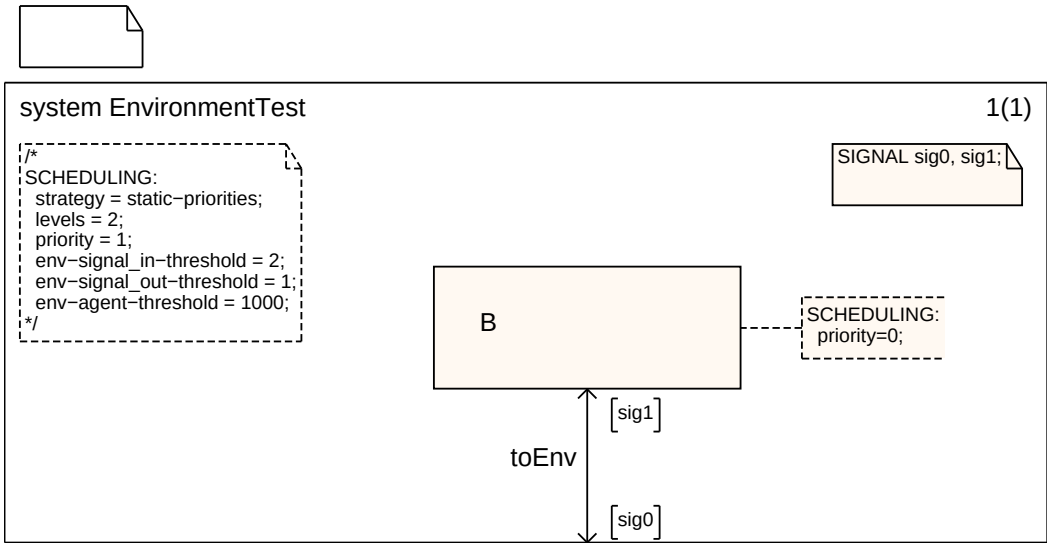


Abbildung 4.8.: Zusätzliche Richtlinien im *Header* des Systems erlauben die Konfiguration des Planungsverfahrens bezüglich des SDL-Environments. Das Beispiel ist auf der beiliegenden CD unter `/SDL/scheduling_beispiel` zu finden.

Für alle drei Parameter gilt, dass ihr Wert größer sein muss als 0. Andernfalls können andere Agenten nicht mehr ausgeführt werden. Unabhängig von den Parametern wird das SDL-Environment immer ausgeführt, wenn sonst keine Agenten ausführbar sind.

Die Konfigurationsparameter werden ebenfalls von *ConTraST* ausgewertet und zur Übersetzungszeit über Makros im Makefile an die Laufzeitumgebung übergeben. Dadurch können auch diese Parameter schnell geändert werden, ohne erneut den Schritt der Codegenerierung durchzuführen.

4.2.7 Umstrukturierung der Laufzeitumgebung

Die ursprüngliche Version der Laufzeitumgebung *SdlRE* sah bis auf die `OPTIMIZE_EXECUTION`-Optimierung keine Wahl zwischen verschiedenen Schedulingstrategien vor. Das verwendete FCFS-Planungsverfahren war eng mit dem *Dispatcher* verwoben, so dass zunächst die Grundlage zur Integration weiterer Planungsverfahren geschaffen werden musste. Hierzu wurde die starke Verbindung des *Dispatchers* mit dem Scheduler aufgetrennt und ein klares Interface definiert, über welches der *Dispatcher* den nächsten auszuführenden Agenten erfragt. Die Sortierung der ausführbaren Agenten ist nun vollständig in dem Scheduler gekapselt, was es dem *Dispatcher* erlaubt, von der Schedulingstrategie zu abstrahieren, und die schnelle Erweiterung mit neuen Strategien ermöglicht.

4.2.7.1 Architektur der Laufzeitumgebung

Zunächst soll das Resultat der Umstrukturierung mit Hilfe eines UML-Klassendiagramms präsentiert werden. Da *SdlRE* mittlerweile über 100 Klassen und knapp 20.000 Zeilen Code umfasst[15], würde ein vollständiges Klassendiagramm den Rahmen sprengen. Abbildung 4.9 zeigt daher nur die zusätzlich gekürzten Ausschnitte, die in Bezug auf die Planungsverfahren von Belang sind.

[15]Gemessen mit sloccount: `http://www.dwheeler.com/sloccount/`

Ein Design-Ziel bei der Umstrukturierung der Laufzeitumgebung war die einfache Erweiterung mit neuen Planungsverfahren. Als Resultat entstand die Klasse GlobalScheduler, welche alle Schedulingfunktionalitäten kapselt und über ein Singleton-Pattern implementiert [GHJV09]. Vorteil des Patterns ist es, dass maximal eine Instanz des Objektes existieren kann, auf welche durch die statische Methode getInstance() global zugegriffen wird. Diese „globale Variable“ folgt zwar nicht unbedingt der Idee von strenger Objektorientierung, ist aber in diesem Fall eine sinnvolle Lösung, da von mehreren Stellen aus auf den Scheduler zugegriffen werden muss. Zum Beispiel muss zum Sparen von Energie auch aus *SEnF* heraus auf den Scheduler zugegriffen werden, um zu erfragen, ob aktuell ausführbare Agenten in dem System sind.

Kern der Laufzeitumgebung ist die Klasse AsmRuntime. In einer Endlosschleife erfragt sie von GlobalScheduler den nächsten auszuführenden Agenten (getNextAgent()) und führt dessen ASM-Programm (program()) aus. AsmRuntime übernimmt daher die Rolle des *Dispatchers*.

Die Multiplizität (0..1) der Assoziation zwischen GlobalScheduler und den einzelnen Schedulern deutet bereits darauf hin, dass GlobalScheduler nicht immer Objekte aller Scheduler erstellt, sondern abhängig von dem verwendeten Planungsverfahren. In Falle des optimierten Planungsverfahrens ohne Prioritäten ist zum Beispiel die Kardinalität zwischen GlobalScheduler und GenericScheduler 1.

Bei Verwendung der optimierten Planung oder des prioritätsbasierten Planungsverfahrens wird nach der Ausführung eines Agenten entschieden, ob im dem ausgeführten Agenten Timer gesetzt sind und der Agent in eine Warteliste eingefügt werden muss. Die Warteschlange ist in der Klasse EDFScheduler implementiert, da ihre Funktionsweise, d.h. das Sortieren von Agenten anhand von Zeitpunkten, identisch zu einem EDF-Planungsverfahren ist. Im Falle der Verwendung einer Warteschlange ist die Kardinalität der Assoziation von GlobalScheduler zu EDFScheduler 1, andernfalls 0.

4.2.7.2 Interface für weitere Planungsverfahren

Um bei Hinzunahme eines weiteren Planungsverfahrens die Änderungen in der restlichen Laufzeitumgebung möglichst gering zu halten, wurde ein Interface definiert, das von allen Planungsverfahren verwendet werden muss. Die meisten denkbaren Verfahren kommen allerdings nicht ohne ergänzende Änderungen in der Laufzeitumgebung aus. Spätestens Verfahren, deren Metrik auf Statistiken aufbaut, müssen die Laufzeitumgebung für das Sammeln von statistischen Daten erweitern.

Listing 4.8 stellt das zu verwendende Interface analog zu Abbildung 4.9 vor, wobei die Funktionen durch Kommentare im Quellcode selbsterklärend sein sollten.

```
1  class BaseScheduler {
2       protected:
3            // protected constructors: Don't create pure instances,
4            //       this class defines the interface only
5            BaseScheduler();
6            // Destructor (requires declaration + definition)
7            virtual ~BaseScheduler();
8
9       public:
10           /* Interface to access the schedule */
11
12           // Access the first item in the list according to strategy
13           virtual Agent* getNextAgent() const = 0;
14
```

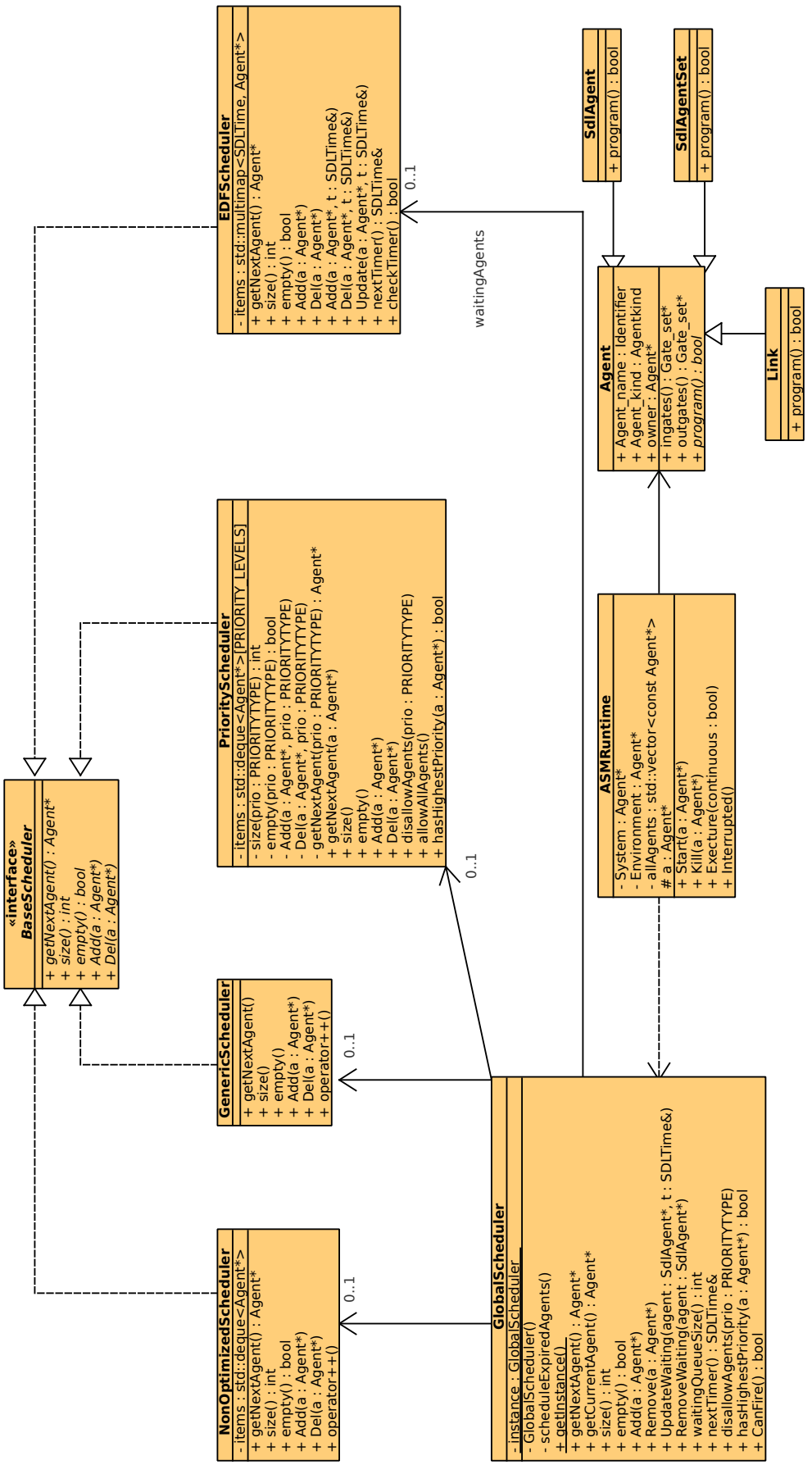


Abbildung 4.9.: Stark gekürztes Klassendiagramm der Laufzeitumgebung. Das Diagramm beschränkt sich auf die für die Planungsverfahren relevanten Teile.

```
15          // Status of schedule
16          virtual int size() const = 0;
17          virtual bool empty() const = 0;
18
19          // Adds/Removes an agent to/from the schedule
20          virtual void Add(Agent* a) = 0;
21          virtual void Del(Agent* a) = 0;
22 };
```

Listing 4.8: Das Interface BaseScheduler definiert Funktionen, die von allen Planungsverfahren zu implementieren sind.

4.2.8 Kompatibilität zwischen unterschiedlichen Versionen

Gerade in der Übergangszeit ist es wünschenswert, dass verschiedene Versionen von *ConTraST* und *SdlRE* miteinander funktionieren. Außerdem sollen SDL-Spezifikationen ohne TC-SDL-Annotationen weiterhin korrekt verarbeitet werden.

Unabhängig von den verwendeten Versionen kommt immer das frühere *Default*-Planungsverfahren (optimierte Planung ohne Prioritäten) zum Einsatz, wenn keine entsprechenden Annotationen im Kopfsymbol des Systems zu finden sind. Bei einer neuen *ConTraST*-Version werden in diesem Fall dennoch entsprechende Makros in dem Makefile definiert (siehe Listing 4.4), um dem Entwickler ein schnelles Ändern zu ermöglichen.

Wird eine ältere Version von *ConTraST*, die keine durch TC-SDL eingeführten Annotationen kennt, und eine neue Version von *SdlRE* verwendet, werden die Annotationen beim Generierungsschritt ignoriert, da sie als gewöhnliche Kommentare interpretiert werden. Das Verhalten entspricht in diesem Fall dem *Default*-Verhalten mit optimiertem Planungsverfahren ohne Prioritäten.

Vice versa – d.h. mit einer neuen *ConTraST*- und alten *SdlRE*-Version – sind hingegen Maßnahmen notwendig, um die Kompatibilität zu gewährleisten. In diesem Fall werden in dem Code Methodenaufrufe auf der Laufzeitumgebung generiert, die erst mit der Einführung von TC-SDL entstanden sind. Ohne weitere Maßnahmen würde die Übersetzung scheitern. Dieses Problem ist zusammen mit der Lösung in dem Codebeispiel in Listing 4.9 zu sehen.

```
1  void Testsys::SysBlockT::TBLOCK::pPrio1::StartTransition::fire(){
2          /* Scheduling: Determination of priority at runtime. */
3          #if defined(SDL_ENHANCED_SCHEDULING) && defined(SDL_SCHEDULER) &&
               SDL_SCHEDULER == SCHEDULING_POLICY_PRIO
4                  SDL_SET_PRIORITY(owner->getInstancePriority());
5          #endif
6          switch (offset) {
7          case 0:
8                  VAR->i = static_cast<Integer>(0);
9                  NextState(State_s0::ID());
10                 break;
11         default: break;
12         }
13 }
```

Listing 4.9: Für die Darstellung leicht gekürzte Starttransition. Falls die Laufzeitumgebung keine Prioritäten kennt, werden die entsprechenden Codezeilen ausgelassen.

Das (generierte) Codebeispiel zeigt das Setzen der Priorität eines Agenten in dessen Starttransition. Falls zur Übersetzungszeit eine alte *SdlRE*-Version verwendet wird, welche kein prioritätsbasiertes Scheduling unterstützt, ist SDL_ENHANCED_SCHEDULING nicht definiert, wodurch der für das Planungsverfahren spezifische Code nicht übersetzt werden würde. Es wird also erst zum Übersetzungszeitpunkt geprüft, ob die Laufzeitumgebung die erforderliche Unterstützung bietet. Dadurch ist es ebenfalls möglich, ohne erneute Codegenerierung zwischen verschiedenen *SdlRE*-Versionen zu wechseln.

4.2.9 Bewertung

Durch Einführung eines Frameworks zur flexiblen Wahl und Erweiterung mit neuen Planungsverfahren wurde eine Möglichkeit geschaffen, die Ausführungsreihenfolge von Agenten eines SDL-Systems entsprechend den Anforderungen des Szenarios anzupassen. Die Wahl und Steuerung innerhalb der SDL-Spezifikation ist dabei verträglich mit dem modellgetriebenen Entwicklungsprozess. Insbesondere kann sie als ergänzender Entwicklungsschritt angesehen werden, der notwendig wird, wenn das *platform-independent model* (PIM) einer SDL-Spezifikation zu einem *platform-specific model* (PSM) erweitert wird, um das SDL-Modell für die Ausführung auf einer bestimmten Hardware-Plattform zu parametrisieren [Got07]. Im Unterschied zum aktuellen Stand der Technik in der Entwicklung für eingebettete Systeme mit SDL, der eine Deployment-Phase und die Ausführung basierend auf einem zusätzlichen Betriebssystem vorsieht, verringert das modellgetriebene Vorgehen die Distanz zwischen Modell und ausgeführtem System. Mit dem prioritätsbasierten Planungsverfahren wurde eine Schedulingstrategie vorgestellt, welche die Priorität einzelner SDL-Agenten in den Ablaufplan einfließen lässt und bereits andeutet, wie zusätzliche Planungsverfahren die Stärke des Frameworks weiter ausspielen können.

Die Vergabe der Prioritäten an Agenten erfordert unter Umständen die Umstrukturierung bestehender Spezifikationen, da Funktionalitäten entsprechend ihrer Priorität zu kapseln sind. An dieser Stelle wäre eine Erweiterung denkbar, die Prioritäten auch abhängig von dem aktuellen Zustand der Agenten vergibt. Die Prioritäten der Agenten würden in diesem Fall nicht mehr statisch sein, sondern dynamisch zur Laufzeit wechseln. Eine weitergehende Erweiterung auf transitionsspezifische Prioritäten ist allerdings wenig praktikabel, da continuous signals, enabling conditions oder priority inputs eine ständige Neuberechnung der Priorität erfordern würden (siehe auch Abschnitt 3.2.2).

Bei der prioritätsbasierten Planung ist auch zu beachten, dass die globale zeitliche Anordnung verschiedener Ereignisse eines SDL-Systems nicht zwangsweise eingehalten wird. Wenn beispielsweise ein Timer zum Zeitpunkt $1{,}2\,s$ abläuft und ein anderer Timer in einem anderen SDL-Agenten mit höherer Priorität zum Zeitpunkt $1{,}3\,s$, kann der spätere Timer vor dem früheren Timer in einer Transition konsumiert werden, wenn das System zum Zeitpunkt $1{,}2\,s$ beschäftigt war. Dieses Ergebnis steht aber keineswegs im Widerspruch zur SDL-Semantik und kann auch bei den anderen Planungsverfahren in weniger offensichtlichen Fällen vorkommen.

Gewöhnlich gilt die Regel, dass je flexibler und „optimaler“ ein Verfahren ist, desto größer ist der Aufwand zur Laufzeit. In der Laufzeitumgebung *SdlRE* wurde der Laufzeitoverhead nicht nur bei dem prioritätsbasierten Verfahren erhöht, sondern es ist auch zu vermuten, dass sich der Overhead allgemein mit der Trennung zwischen *Dispatcher* und *Scheduler* und der daraus resultierenden Kapselung des Schedulers erhöht hat: Was vor der Einführung des Frameworks noch stark in den *Dispatcher* integriert war, auf das muss nun über eine Schnittstelle des Schedulers zugegriffen werden; die Kehrseite von Modularisierung und Austauschbarkeit.

Der bisherige Stand von TC-SDL ist erst der Anfang von Maßnahmen zur Verbesserung der Echtzeitfähigkeit von SDL und hat daher noch einige Nachteile. Ein bereits erwähnter Nachteil ist die Granularität der Prioritäten des prioritätsbasierten Planungsverfahrens. Insbesondere bei dem SDL-Environment, das nicht gemäß Prioritäten in mehrere Agenten gekapselt werden kann, wäre es sinnvoll, zwischen den Signaltypen zu unterscheiden. Dadurch könnte das Environment zum Beispiel nach dem Empfang eines Datenrahmens sofort ausgeführt werden (hohe Priorität), nach der Meldung über einen erfolgreichen Versand hingegen verzögert werden (niedrige Priorität).

Des Weiteren kann die fehlende Fairness des prioritätsbasierten Planungsverfahrens zu neuen Problemen führen und in besonders schweren Fällen sogar zu einem unbrauchbaren System. Nehmen wir beispielsweise einen hochprioren Agenten, der in einer Transition einem anderen Agenten und sich selbst ein Signal sendet, das wiederum die Ausführung der gleichen Transition veranlasst. In jedem Schritt würde dieser Agent seine eigene Queue leeren, aber den *input port* des anderen Agenten um ein Signal vergrößern. Der Speicherverbrauch würde ungebremst ansteigen und zu einem Fehler führen, der bei einem fairen Planungsverfahren nicht auftreten würde. Als Lösung dieses Problems wäre es denkbar, einen Agenten, dessen Input-Queue einen Grenzwert überschreitet, unabhängig seiner Priorität auszuführen. Allerdings wurde auf diese Lösung verzichtet, da sie mit zusätzlichem Aufwand zur Laufzeit verbunden und intransparent für den Entwickler wäre. Außerdem ist es sehr fragwürdig, ob es überhaupt sinnvolle Fälle gibt, in denen dieses Problem auftritt.

5. KAPITEL

Black Burst Synchronization – Realisierung und Integration

Der aktuelle Stand von TC-SDL und die bisher unterstützte Menge an Planungsverfahren ermöglicht die Verringerung von Wartezeiten hochpriorer Agenten, ohne dabei deterministische Garantien für deren *worst case*-Ausführung geben zu können. In vielen Anwendungsszenarien stellen diese Vorteile von TC-SDL bereits einen großen Fortschritt dar, sind allerdings in manchen Fällen, in denen statistische Garantien nicht ausreichen, keine zufriedenstellende Lösung. Ein solcher Fall ist *Black Burst Synchronization* (BBS), deren harte zeitliche Anforderungen sowohl Vorhersagbarkeit als auch eine effiziente Implementierung erfordern. Für die Realisierung von BBS musste daher ein anderer Weg beschritten werden. Da eine vollständige Realisierung in SDL ausschied, wurden Teile des Protokolls in Form eines Treibers realisiert.

Dieses Kapitel stellt am Beispiel von BBS vor, wie derartige zeitkritische Protokolle realisiert und in ein SDL-System integriert werden können. Die Lösung besteht dabei aus einem Kompromiss, der einerseits auf einer effizienten Implementierung von BBS in der Programmiersprache C basiert, aber andererseits eine Vielzahl von Konfigurationsmöglichkeiten aus SDL heraus bietet. Dadurch kann das Verhalten von BBS an die Anforderungen des konkreten Szenarios angepasst werden, ohne dass der Entwickler Modifikationen des C-Codes vornehmen muss.

Der Aufbau des Kapitels ist wie folgt: Abschnitt 5.1 gibt zunächst einen Überblick über das Konzept hinter „Black Bursts“ und stellt das Synchronisationsprotokoll BBS zusammenfassend vor. Die Zusammenfassung legt dabei einen Fokus auf Maßnahmen zur Erhöhung der Robustheit des Protokolls. Anschließend präsentiert Abschnitt 5.2 die Realisierung des BBS-Treibers im Detail und erklärt, wie Dienstnutzer ihn verwenden können.

5.1 BBS - Eine Zusammenfassung

Black Burst Synchronization (BBS) ist ein multi-hop Synchronisationsprotokoll für drahtlose Ad-Hoc-Netzwerke [GK08]. Die bedeutendsten Vorteile im Vergleich zu anderen Synchronisationsprotokollen sind sein dezentraler Charakter, seine Robustheit gegen Knotenausfälle und seine deterministischen Garantien bezüglich der Synchronisationsgenauigkeit $d_{maxOffset}$ und der Konvergenzzeit $d_{syncPhase}$. Die folgenden Unterabschnitte geben eine Zusammenfassung von BBS, wobei Implementierungsdetails zunächst ausgelassen werden. Ebenfalls wird auf die Herleitung einzelner Parameter verzichtet. Entsprechende Herleitungen sind in [GK08] zu finden.

5.1.1 Überblick

BBS ist ein Protokoll zur netzwerkweiten Ticksynchronisation. Verglichen mit einem Synchronisationsprotokoll, das Knoten nicht nur auf einen Referenztick synchronisiert, sondern zusätzlich Uhrenwerte mit den Ticks assoziiert, sind Protokolle zur Ticksynchronisation im Allgemeinen leichtgewichtiger, da keine Uhrenwerte ausgetauscht werden müssen. In vielen Fällen ist eine Ticksynchronisation bereits ausreichend, zum Beispiel zum Entwickeln von TDMA-basierten MAC-Protokollen oder zum synchronen Abgreifen von Sensorwerten.

BBS basiert auf dem rundenbasierten Senden von Tick-Rahmen, deren Inhalt durch sogenannte *Black Bursts* kollisionsresistent kodiert wird (siehe Abschnitt 5.1.2). Periodisch in einem Intervall d_{macro} führt BBS *Synchronisationsphasen* durch, um einerseits neue Knoten in das Netz aufzunehmen und um andererseits die aufgrund von unterschiedlich schnell laufenden Hardware-Quarzen sinkende Synchronisationsgenauigkeit zu kompensieren (*clock skew*). Anhand der periodischen Synchronisationsphasen wird das Medium in sogenannte *Macro-Slots* unterteilt (siehe Abbildung 5.1), die wiederum in sogenannte *Micro-Slots* untergliedert werden können. Micro-Slots können zu Slot-Regionen zusammengefasst werden, in denen anschließend unterschiedliche Medienzugriffsverfahren eingesetzt werden [BGK07]. Ebenso können sie bei einem reservierungsbasierten Medienzugriff verwendet werden, um die Granularität von Reservierungen festzulegen.

Die Anzahl der Runden pro Synchronisationsphase, in denen Tick-Rahmen gesendet werden, entspricht dem maximalen Durchmesser des Netzwerks $n_{maxHops}$. Dabei muss der maximale Durchmesser, der a priori festgelegt wird, nicht dem tatsächlichen Durchmesser entsprechen, sondern lediglich einer oberen Schranke. Die Dauer einer Synchronisationsrunde d_{round} hängt von dem Inhalt eines Tick-Rahmens und von Parametern des Transceivers ab, nicht aber von der Anzahl an Netzwerkknoten. Der Inhalt, d.h. die Nutzdaten, die in einem Tick-Rahmen während einer Synchronisationsrunde transportiert werden, richtet sich nach der verwendeten BBS-Version (siehe Abschnitt 5.1.3). Da der Umfang an Nutzdaten, der maximale Netzwerkdurchmesser $n_{maxHops}$ sowie die Parameter des Transceivers bekannt sind, ist die Dauer einer Runde d_{round} konstant und kann a priori berechnet werden. Somit ist auch die Dauer der Synchronisationsphase (Konvergenzzeit) bekannt und beträgt $d_{round} \cdot n_{maxHops}$. Das Synchronisationsverfahren terminiert somit in deterministischer Zeit.

Die oben erwähnte deterministische Garantie der Synchronisationsgenauigkeit $d_{maxOffset}$ lässt sich ebenfalls anhand der Eigenschaften des Transceivers und des maximalen Netzwerkdurchmessers herleiten. Insbesondere gilt auch für die Synchronisationsgenauigkeit, dass sie unabhängig von der Anzahl der Netzwerkknoten ist. Bei dem Transceiver sind im Konkreten die Umschaltzeiten zwischen Empfangs- und Sendemodus $d_{accessTx}$ sowie die Verzögerung bei der

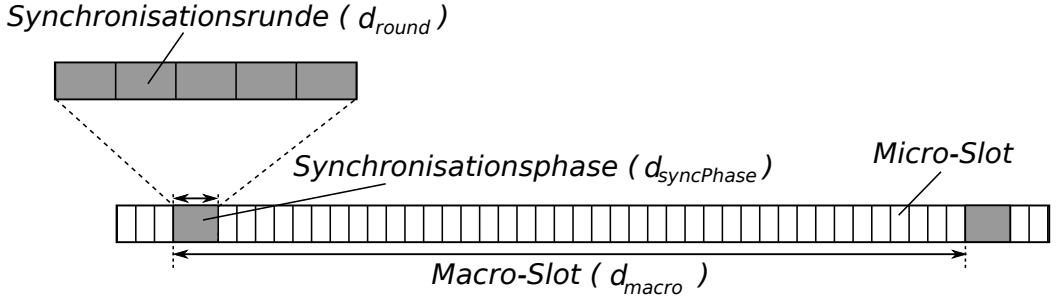


Abbildung 5.1.: Aufbau eines Macro-Slots mit BBS [GK08].

Erkennung einer Medienbelegung d_{maxCCA} ausschlaggebend. Bei einer optimierten Version von BBS, in welcher ein Master-Knoten die Synchronisation initiiert, hängt die Synchronisationsgenauigkeit nur von dem Durchmesser $n_{maxHops}$ und der Verzögerung bei der Erkennung einer Medienbelegung d_{maxCCA} ab (siehe Abschnitt 5.1.3).

Bisher existierte von BBS eine Implementierung für die MICAz-Plattform [Cro07], auf welcher die deterministischen Zeitschranken experimentell bestätigt wurde [GK08]. Bereits im Vorfeld dieser Arbeit entstand eine Implementierung für die Imote2-Plattform [Cro09]. Da MICAz und Imote2 beide mit einem CC2420-Transceiver ausgestattet sind, sind die Parameter (Synchronisationsgenauigkeit, Konvergenzzeit) von BBS bei beiden Sensorknoten identisch, obwohl sich die beiden Knoten in anderen Eigenschaften (CPU, Speicher, Hardware-Timer, ...) stark unterscheiden. Im Zuge der vorliegenden Arbeit wurde die Implementierung von BBS für den Imote2 erweitert und in Form eines Treibers in das *SDL Environment Framework* (*SEnF*) integriert. Die Aufgaben von *SEnF* und der Zusammenhang zu *SdlRE* fasst Anhang A zusammen.

5.1.2 Black Bursts: Konzept und Umsetzung

Die Informationen innerhalb eines Tick-Rahmens werden mit einer Folge von Black Bursts kodiert. Konzeptionell entspricht ein Black Burst einer Belegung des Mediums, die als Energie auf dem Äther wahrgenommen wird [KdI07, GK08]. Daten können in der Dauer und dem Startzeitpunkt der Medienbelegung kodiert werden. Diese beiden Informationen können mit dem *Clear Channel Assessment*-Mechanismus (CCA), den jeder IEEE 802.15.4-kompatible Transceiver zur Durchführung von CSMA/CA besitzen muss, erkannt werden [IEE03]. Des Weiteren können beide Informationen immer noch mit dem Energie-Detektor des CCA-Mechanismus erfolgreich dekodiert werden, wenn sich mehrere Black Bursts relativ zeitgleich überlagern. Black Bursts eröffnen damit die Möglichkeit zur kollisionsgeschützten Übertragung.

Generell kann ein Black Burst durch die Dauer der Medienbelegung mehrere Informationsbits pro Übertragung kodieren. In BBS kodiert ein Black Burst genau ein Bit. Die Übertragung eines Black Bursts der Länge $d_{BB} > 0$ entspricht einer binären 1, ein Black Burst der Länge $d_{BB} = 0$, d.h. keine Übertragung, hat die Bedeutung einer binären 0. Durch diese Art der Kodierung wird eine binäre 1 dominant und eine binäre 0 rezessiv übertragen. Werden zwei Bitsequenzen, die mit Black Bursts kodiert wurden, zeitgleich übertragen, erhält der Empfänger das *OR* beider Sequenzen. Diese Art der Kodierung hat außerdem den essentiellen Vorteil, dass ein Knoten, der eine binäre 0 „sendet", gleichzeitig das Medium beobachten und die mögliche Übertragung einer binären 1 erkennen kann. Diese Eigenschaften wurden in speziellen Transfer-Protokollen ausgenutzt, um Informationen in deterministischer Zeit in einem Netzwerk zu verbreiten (weitere Informationen in [Kuh09, CGK09]).

Es existieren Implementierungen von Black Bursts für die Sensorknoten MICAz und Imote2. Beide Implementierungen stellten ihre Vorteile bereits in experimentellen Evaluationen erfolgreich unter Beweis [KdI07, GK08, Chr09]. Sie nutzen beide den CC2420-Transceiver und zeigen damit, dass Black Bursts mit handelsüblichen Transceivern realisiert werden können. In beiden Implementierungen entspricht ein dominanter Black Burst der Übertragung eines regulären MAC-Rahmens mit einem Payload der Länge 0. *Header* und *Trailer* des MAC-Rahmens werden – soweit es die Hardware zulässt – gekürzt. Die endgültige Länge des MAC-Rahmens besteht nach Kürzung der möglichen Felder aus 5 Bytes[16]. Bei einer Übertragungsrate von 250 kbps des CC2420-

[16] 3 Bytes Präambel, 1 Byte SFD und 1 Byte Längenfeld. Ein Rahmen dieses Formates ist nicht 802.15.4-konform und wird durch den Transceiver nicht als regulärer Rahmen erkannt.

Transceivers benötigt die Übertragung eines Black Bursts somit 160 μs, wobei mindestens zusätzliche $2 \cdot 192\,\mu$s für das Umschalten zwischen Sende-/Empfangsmodus hinzukommen.

Die Implementierung von Black Bursts, die im Folgenden für die Realisierung von BBS verwendet wird, nutzt ebenfalls den CC2420-Transceiver des Imote2, verzichtet allerdings auf die Kürzung der Felder in *Header* und *Trailer*. Diese Maßnahme erscheint zunächst wenig plausibel, da die Implementierung eines Black Bursts dadurch deutlich ineffizienter ist. Im Vergleich zu den 5 Bytes pro Black Bursts werden nun nämlich 8 Bytes (4 Bytes Präambel, 1 Byte SFD, 1 Byte Längenfeld, 2 Bytes CRC) benötigt. Das Problem bei dem gekürzten Format ist die Inkompatibilität mit regulären Übertragungen. Da BBS in der Regel als Grundlage für andere Protokolle dient, die reguläre Rahmenübertragungen nutzen, muss ein effizientes Umschalten zwischen den verschiedenen Rahmenformaten möglich sein. Allerdings liegt genau hier das Problem, da bis dato kein Weg gefunden wurde, das Rahmenformat ohne Neustart des Transceiver-Oszillators zu wechseln. Dies ist jedoch so zeitintensiv (im Datenblatt ist die typische *start-up time* des Transceivers mit 1 ms angegeben [Chi07]), dass es keine Vorteile bringt. Durch zukünftige Arbeiten mit dem Transceiver werden andere Wege gesucht, um dieses Problem zu lösen.

5.1.3 Versionen von BBS

Insgesamt existieren drei Versionen von BBS [GK08]: Eine master-basierte Version, in welcher ein designierter Knoten die Synchronisation initiiert, eine dezentrale Version, welche eine geringere Synchronisationsgenauigkeit aufweist, dafür aber ohne *Single-Point-of-Failure* auskommt, und eine hybride Version, welche die Vorteile beider Versionen vereint. Die Implementierung für den Imote2-Knoten beschränkt sich zur Zeit auf die master-basierte Version, die Implementierungen der anderen Versionen sind in Planung.

Die master-basierte Synchronisation geht immer von einem einzelnen Knoten aus, d.h. in der ersten Runde der Synchronisationsphase sendet ausschließlich ein Knoten. Dieser sogenannte *Master-Knoten* muss netzwerkweit eindeutig sein. Bei einem Ausfall des Masters kann ein beliebiger Knoten dessen Rolle übernehmen, da hierfür keine spezielle Hardware-Unterstützung benötigt wird.

Die zur master-basierten Synchronisation verwendeten und mit Black Bursts kodierten Tick-Rahmen bestehen aus einem dominanten *Start-Of-Frame*-Bit (SOF), der aktuellen Rundennummer und einer Prüfsumme. Zu Beginn der Synchronisationsphase überträgt der Master-Knoten einen Tick-Rahmen mit der Rundennummer 0. Alle Knoten, die sich in Sendereichweite befinden, synchronisieren sich auf den Start des Tick-Rahmens und leiten den Rahmen mit der um eins erhöhten Rundennummer in der nächsten Runde weiter. Da die Knoten identische Tick-Rahmen senden und die Tick-Rahmen aufgrund der Kodierung durch Black Bursts kollisionsresistent übertragen werden, bleiben die Informationen eines Tick-Rahmens bei simultaner Übertragung durch mehrere Knoten erhalten. Auf diese Weise werden Tick-Rahmen durch das Netzwerk propagiert und jeder Empfänger kann anhand der Rundennummer und dem Empfangszeitpunkt den Beginn der Synchronisationsphase berechnen. Da der maximale Netzwerkdurchmesser $n_{maxHops}$ bekannt ist, terminiert eine Synchronisationsphase nach $n_{maxHops}$ Runden. Wenn der tatsächliche Durchmesser geringer ist, endet die Synchronisation ebenfalls nach $n_{maxHops}$ Runden, da den Knoten der tatsächliche Durchmesser nicht zwingend bekannt ist. Allerdings bleiben die letzten Synchronisationsrunden in diesem Fall ungenutzt.

Bei der dezentralen Version von BBS beginnen alle Knoten zu Beginn der Synchronisationsphase mit der Übertragung von Tick-Rahmen. Die verwendeten Tick-Rahmen bestehen ausschließlich

aus zwei dominanten Black Bursts, die in in jeder Runde von jedem synchronisierten Knoten gesendet werden. Erkennt ein Knoten den Beginn eines Black Bursts, bevor er mit der Übertragung seines eigenen Tick-Rahmens beginnt, was genau dann der Fall ist, wenn die Synchronisationsungenauigkeit zu groß ist, synchronisiert er sich zu dem empfangenen Black Burst und beginnt unmittelbar mit der Übertragung seines Tick-Rahmens. Verglichen mit der master-basierten Version trägt bei dieser Version von BBS daher neben der Erkennungsverzögerung der Medienbelegung die Umschaltzeit vom Empfangs- in den Sendemodus zusätzlich zur Synchronisationsungenauigkeit bei.

Das hybride Verfahren sendet sowohl dezentrale als auch master-basierte Tick-Rahmen entsprechend den oben beschriebenen Protokollbeschreibungen. Die Idee hinter der hybriden Version ist es, von der Genauigkeit der master-basierten BBS-Version zu profitieren und die dezentrale Version als Backup-Lösung bereit zu behalten, um bei Ausfall des Master-Knotens die Synchronisation weiterhin aufrecht zu halten.

Es sei außerdem darauf hingewiesen, dass zwischen der Dauer eines master-basierten Black Bursts $d_{masBurst}$ und der Dauer eines dezentralen Black Bursts $d_{decBurst}$ zu unterscheiden ist. Die reine physikalische Übertragungsdauer beträgt zwar in beiden Fällen 160 μs bzw. bei dem nicht-optimierten MAC-Rahmenformat 256 μs (siehe Abschnitt 5.1.2), allerdings müssen aufgrund der unterschiedlichen Synchronisationsgenauigkeit verschiedene *guard times* beachtet werden. Weitere Informationen hierzu sind in [GK08] zu finden.

5.1.4 Robustheit und Garantien

Die bisherige konzeptionelle Vorstellung von BBS geht von einem perfekten Medium ohne externe Störquellen oder transiente Links aus. Lediglich die bereits in Abschnitt 5.1.3 erwähnte Prüfsumme lässt vermuten, dass in einer Implementierung zusätzliche Maßnahmen notwendig werden. Dieser Abschnitt stellt nun einige Einflüsse vor, welche die korrekte Funktionalität von BBS in einer realistischen Umgebung infrage stellen, und wie die Implementierung diesen Einflüssen entgegenwirken kann. Viele der genannten Maßnahmen zur Erhöhung der Robustheit basieren auf [GK08], wobei im Folgenden zusätzliche Implementierungsdetails aufgeführt werden. Da bisher nur eine Implementierung für die master-basierte BBS-Version existiert, beziehen sich alle folgenden Aussagen auf die master-basierte Synchronisation.

5.1.4.1 Notwendigkeit zur Erhöhung der Robustheit

Dass zusätzliche Maßnahmen zur Erhöhung der Robustheit notwendig sind, wird anhand der Grafiken 5.2 und 5.3 sowie an Tabelle 5.1 deutlich, die das Resultat einer zweitägigen Messung zeigen. Die Messung wurde in einer realistischen Umgebung (Raum 36/422) durchgeführt, in welcher Rauschen und Interferenzen mit anderen Netzwerken auftreten. Das Experiment wurde mit Hilfe eines Imote2-Knotens durchgeführt, welcher zyklisch die 802.15.4-Kanäle 11 bis 26 mit einer Periode von einer Sekunde durchlief. Weitere technische Details zum Imote2 sind in Abschnitt 6.1 zusammengefasst.

In Abbildung 5.2 ist für jeden der 16 Kanäle dargestellt, wie lange im Durchschnitt der CCA-Mechanismus in einem Zeitfenster von 1 s eine Medienbelegung erkennt. Spitzenreiter sind hierbei die Kanäle 16 und 19, welche durchschnittlich für jeweils ungefähr 20 ms belegt sind. Dies entspricht einer prozentualen Belegung von 2%. Anhand von Tabelle 5.1 kann erkannt werden, dass die Kanäle keineswegs konstant belegt waren, sondern dass die Belegungsdauer stark variierte. Hierbei weist Kanal 26 die stärkste Varianz auf (größte Standardabweichung σ) mit einer

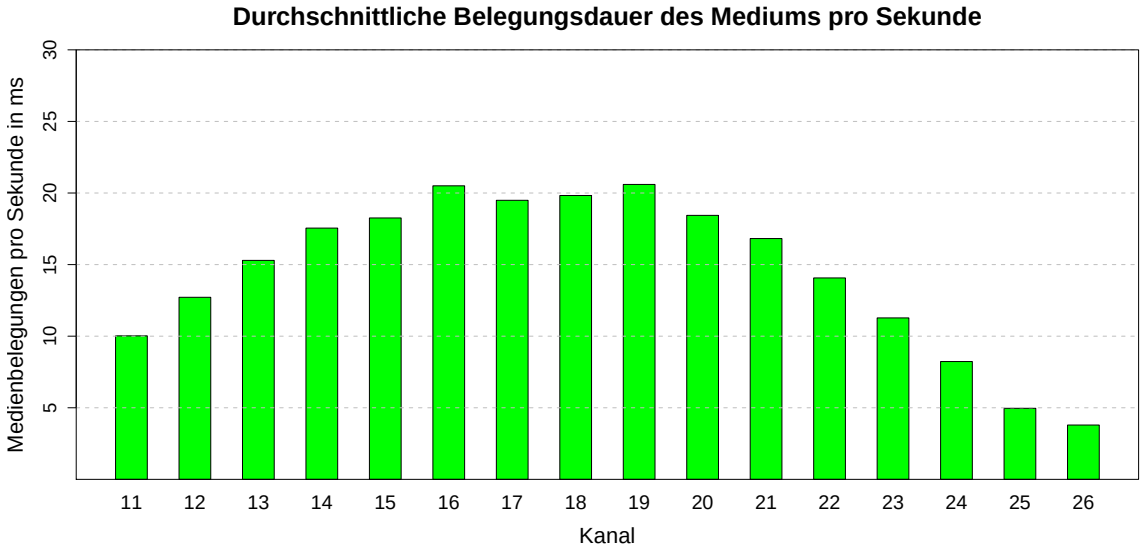


Abbildung 5.2.: Grafik zur Verdeutlichung der Belegung des Mediums. Durch Rauschen und interferierende Netzwerke meldet der CCA-Mechanismus des Transceivers eine durchschnittliche Medienbelegung von bis zu 20 ms.

maximalen Belegungsdauer von 907 *ms* pro Sekunde. Dies entspricht einer Auslastung von rund 91% und macht eine Anwendung von Black Bursts unmöglich.

In einem weiteren Schritt wurde die Dauer der einzelnen Belegungen des Mediums im Detail betrachtet. Es wurde untersucht, wie viele der Belegungen aufgrund der zeitlichen Dauer der Belegung als dominanter Black Burst dekodiert werden würden. Das Resultat dieser Analyse ist in Abbildung 5.3 dargestellt. Die Grafik verdeutlicht, dass auf den meisten Kanälen der Großteil der Medienbelegungen keinem regulären dominanten Black Burst entsprach. Problematisch sind die Fälle, in denen die Dauer der Belegung einen regulären dominanten Black Burst darstellt (*false positives*), was im Schnitt etwa 10 mal pro Sekunde zutraf. Wie auch bei der Gesamtdauer der Belegung, waren diese Werte stark variabel (siehe Tabelle 5.1). Das Maximum der Anzahl an Belegungen des Mediums lag auf Kanal 13 bei 611 Belegungen pro Sekunde. Die maximale Anzahl an *false positives* pro Sekunde lag auf Kanal 12 und entsprach 460 gültigen dominanten Black Bursts.

Diese Zahlen verdeutlichen, dass die starken Interferenzen ein Problem für die korrekte Anwendung von Black Bursts darstellen und Maßnahmen notwendig sind, um die Beeinträchtigung der korrekten Funktionsweise zu minimieren. Die genauen Auswirkungen der Interferenzen und entsprechende Gegenmaßnahmen werden in den Abschnitten 5.1.4.5 und 5.1.4.4 für die master-basierte Version von BBS diskutiert. Zunächst werden aber mit dem Ausfall des Master-Knotens und mit der Fusionierung von Netzwerken zwei Fälle betrachtet, die unabhängig von etwaigen Störquellen oder Interferenzen auftreten können.

5.1.4.2 Ausfall des Master-Knotens

Das Ausbleiben von Tick-Rahmen während einer Synchronisationsphase kann mehrere Ursachen haben. Eine Möglichkeit ist der Ausfall des Master-Knotens oder die Bewegung des Master-Knotens aus der Sendereichweite des Netzwerks. Ebenso ist es möglich, dass sich der Knoten, der vergebens auf Tick-Rahmen wartet, aus der Sendereichweite des Netzwerks entfernt hat.

	Kanal	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Dauer erkannter Belegung des Mediums in ms	Ø	10	12.7	15.3	17.5	18.3	20.5	19.5	19.8	20.6	18.4	16.8	14.1	11.3	8.2	5	3.8
	Max	647.7	298.9	447.5	349.2	37	55.6	46.2	58.9	48.7	42.3	379.6	444	167.5	191.4	13.5	**906.7**
	Min	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
	σ	16.2	12.1	12.8	9.6	3.3	3.8	3.5	4	3.8	3.5	5.7	6.9	4.3	2.8	1.4	26.8
Anzahl Belegungen des Mediums	Ø	22.5	27	31.3	36	35.6	39.8	36	37.1	40.8	36.7	32.8	27	22.5	18.2	13.7	8.8
	Max	569	493	**611**	481	61	82	83	90	140	131	340	306	134	279	29	151
	Min	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	σ	26.9	27.5	29.5	24.4	6	7	6.3	7.2	7.7	6.7	7.7	7.5	5.3	5.1	3.4	3.9
Anzahl erkannter Black Bursts	Ø	13.3	13.5	13.5	14	9.8	11.1	7.2	8.5	12.7	11.6	10.9	9.7	10	10.3	10.1	8.3
	Max	449	**460**	405	436	33	35	30	57	108	107	112	109	70	113	22	15
	Min	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
	σ	22.7	21.9	23.9	20.6	3.8	4.3	3.1	4	4.7	4	4.8	4.1	3.2	3.4	3	2.9

Tabelle 5.1.: Analyse des Drahtlosmediums. Die Tabelle gibt die Ergebnisse einer zweitägigen Messung in rauschbehaftetem Umfeld wieder. Die Ergebnisse beziehen sich jeweils auf ein Zeitfenster von 1 s.

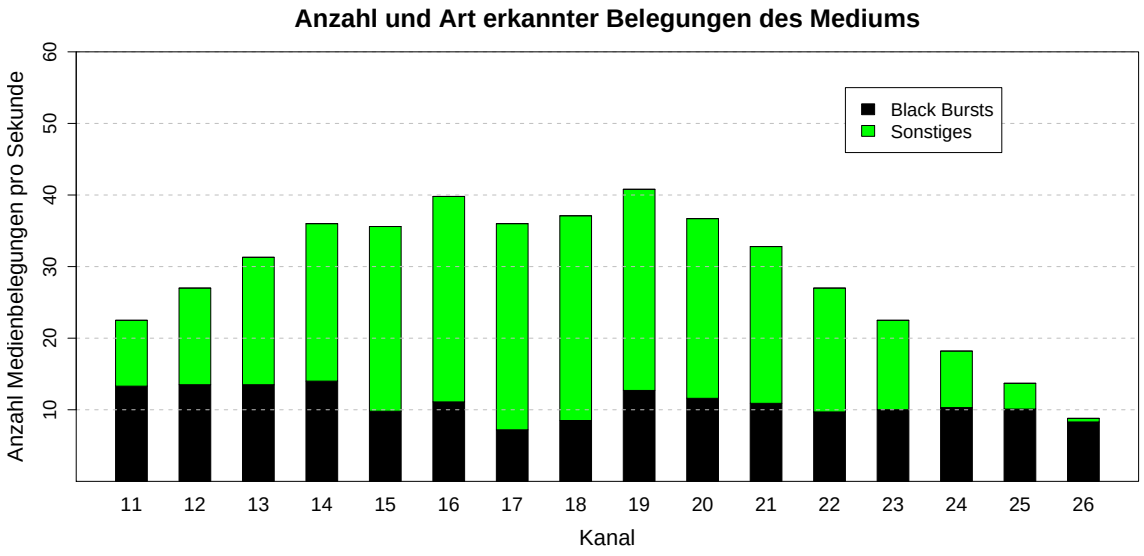


Abbildung 5.3.: Anhand der Dauer der Belegung des Mediums durch Rauschen und Interferenzen würden im Durchschnitt ca. 10 Black Bursts pro Sekunde erkannt werden.

Unabhängig von der Ursache muss ein Knoten entscheiden, wie er auf diesen Vorfall reagieren soll, und gegebenenfalls Maßnahmen zur Bestimmung eines neuen Masters treffen. Dies sollte dabei einerseits schnell geschehen, um bei tatsächlichem Ausfall des Masters das Netzwerk möglichst schnell wieder in einen synchronisierten Zustand zu bekommen, andererseits muss die Eindeutigkeit des Master-Knotens weiterhin sichergestellt sein.

Daher warten Knoten, die keine Tick-Rahmen innerhalb einer Synchronisationsphase empfangen, zunächst weitere Synchronisationsphasen ab, bevor sie Maßnahmen ergreifen. Erst wenn gültige Tick-Rahmen in mehreren aufeinanderfolgenden Synchronisationsphasen ausbleiben, wird ein neuer Master bestimmt. Die genaue Anzahl der Synchronisationsphasen, die ohne Erhalt von Tick-Rahmen toleriert werden, ist ein konfigurierbarer Parameter. Die Bestimmung eines neuen Masters besteht darin, dass sich ein Knoten selbst nach einer zusätzlichen Wartezeit als neuer Master deklariert und einen Tick-Rahmen sendet. Die Wartezeit kann zufällig aus einem ausreichend großen Zahlenintervall gezogen oder in Abhängigkeit der eindeutigen Knoten-ID berechnet werden. Der zweite Ansatz hat den Vorteil, dass die Wartezeiten garantiert eindeutig sind und immer genau ein Master-Knoten hervorgeht.

Nicht in allen Szenarien macht es wirklich Sinn, auf den Ausfall des Masters zu reagieren. Zum Beispiel ist es in einem typischen WSN, welches Sensorwerte sammelt und entlang einer Baumtopolgie zur Wurzel (Senke) des Baumes transportiert, sinnvoll, die Wurzel des Baumes als Master festzulegen. Diese Konfiguration hat unter Anderem den Vorteil, dass $n_{maxHops}$ nicht dem Netzwerkdurchmesser entsprechen muss, sondern lediglich der Distanz zwischen Wurzel und entferntestem Blattknoten. Sollte die Wurzel des Baumes ausfallen, kann das WSN seine Aufgabe nicht mehr erfüllen. Eine Synchronisation ohne den Wurzel-Knoten wäre für die Anwendung nutzlos, insbesondere wenn die Distanz zwischen neuem Master und entferntestem Knoten größer wäre als $n_{maxHops}$. Vielmehr sollte in diesem Fall sogar verhindert werden, dass bei zeitweiser Partitionierung des Netzwerks weitere Master-Knoten entstehen, die bei Zurückkehren des Wurzel-Knotens zu aufwändigem Zusammenführen der Partitionen führen würden. In Ad-Hoc-Netzen könnte es hingegen sinnvoll sein, dass ein Knoten als Master agieren *kann*, falls kein anderer Master-Knoten vorhanden ist. In diesem Fall ist es für den Entwickler ausrei-

chend, dass eine Synchronisation gegeben ist, aber es spielt keine Rolle, wer als Master-Knoten die Übertragung von Tick-Rahmen initiiert. Die Realisierung von BBS bietet daher die Option, zwischen verschiedenen Knotenrollen zu unterscheiden (siehe Abschnitt 5.2.2 und Anhang C.2).

5.1.4.3 Fusionierung zweier Netzwerke

Bei dynamischen Topologien kann es passieren, dass sich aus einem Netzwerk mehrere Cluster bilden. Ebenso ist es möglich, dass zwei Netzwerke unabhängig voneinander entstehen, sich später aber aufgrund von Knotenmobilität zu einem Netzwerk vereinen. Diese Ereignisse müssen auch von BBS vorgesehen werden, um nach der Fusionierung zweier Netze einen gemeinsamen Referenztick zu erhalten.

Ein Knoten bemerkt das Vorhandensein eines anderen Netzwerks, indem Tick-Rahmen zu unerwarteten Zeitpunkten innerhalb eines Macro-Slots empfangen werden. Allerdings kann der Knoten nicht davon ausgehen, dass alle Knoten seines Netzwerks ebenfalls den „fremden“ Tick-Rahmen empfangen haben. Er muss also zunächst das restliche Netzwerk über seine Beobachtung informieren. In [GK08] wurde hierzu ein *Jamming-Rahmen*, d.h. eine vordefinierte Folge von Black Bursts, innerhalb eines reservierten Micro-Slots versendet und bei Empfang durch das Netzwerk weitergeleitet. Dieses Vorgehen hat den Nachteil, dass es gerade bei dem CC2420-Transceiver relativ viel Zeit kostet, da insbesondere die ineffizienteren dezentralen Black Bursts zur Anwendung kommen müssen, weil sich Jamming-Rahmen theoretisch von zwei Seiten eines Netzwerks ausbreiten können. Ein weiterer Nachteil ist, dass durch die Verwendung von dezentralen Black Bursts nicht sichergestellt ist, dass die reservierte Jamming-Sequenz nicht doch einem gültigen Tick-Rahmen entspricht. Die Unterscheidung zwischen Jamming-Sequenz und Tick-Rahmen kann zwar anhand des Empfangszeitpunkts getroffen werden, allerdings nur von Knoten, die bereits synchronisiert sind. Knoten, die gerade eingeschaltet wurden, könnten hingegen eine Jamming-Sequenz irrtümlich als regulären Tick-Rahmen interpretieren.

In der vorliegenden Implementierung wurde daher anstelle der mit Black Bursts kodierten Jamming-Rahmen ein regulärer Rahmen, welcher minimal größer ist als ein Black Burst, zur Signalisierung eines Konfliktes verwendet. Diese Konfliktsignalisierungsrahmen werden ebenso in einem vordefinierten Zeitfenster direkt vor der eigentlichen Synchronisationsphase gesendet. Ihre Erkennung geschieht ebenfalls anhand des CCA-Mechanismus und der Dauer der Medienbelegung, um parallele Übertragungen zu ermöglichen. Diese Lösung ist etwas effizienter als die Lösung mit Jamming-Rahmen und hält außerdem die Protokollkomplexität geringer, da nicht unterschieden werden muss, ob empfangene Black Bursts Teil eines Tick-Rahmens oder Teil eines Jamming-Rahmens sind.

Knoten, die entweder direkt durch den Empfang von fremden Tick-Rahmen oder indirekt durch den Empfang eines Konfliktsignalisierungsrahmens von einem Konflikt erfahren haben, unternehmen zunächst nichts zur Auflösung des Konfliktes. Erst wenn in mehreren aufeinanderfolgenden Macro-Slots ein Konflikt erkannt wurde, kommt es zur Fusionieren beider Netze[17]. Diese zusätzliche Absicherung soll verhindern, dass in einer Umgebung mit starkem Rauschen und vielen *false positives* bereits ein irrtümlich entdeckter Tick- oder Konfliktsignalisierungsrahmens zu einer Fusionierung führt. Falls es doch zu einer Fusionierung kommt, gibt der Master-Knoten zunächst seine Rolle auf und wartet eine von seiner Knoten-ID abhängige Zeitspanne. Wird innerhalb dieser Zeitspanne kein anderer Tick-Rahmen empfangen, deklariert sich der Knoten

[17] Die genaue Anzahl der tolerierten Konflikte ist konfigurierbar.

wieder als Master und beginnt mit einer neuen Synchronisationsphase. In der Regel geht also einer der ursprünglichen Master-Knoten auch als neuer Master hervor. Fallen beide Master während der Fusionierung aus, greifen die gleichen Maßnahmen von Abschnitt 5.1.4.2.

5.1.4.4 Detektion nicht gesendeter Black Bursts

Durch Rauschen und Interferenzen kann es vorkommen, dass eine Belegung des Mediums erkannt wird, die einem gültigen Black Burst entspricht, obwohl kein Black Burst gesendet wurde (*false positives*). Diese falsch erkannten Black Bursts können ohne Gegenmaßnahmen dazu führen, dass sich Knoten auf den Empfang von *false positives* synchronisieren. Ebenfalls problematisch sind *false positives* bei der Erkennung fremder Netzwerke, da bereits synchronisierte Knoten aufgrund von falsch erkannten Black Bursts irrtümlich auf die Existenz eines zweiten Netzwerks schließen könnten.

Um den Auswirkungen der Probleme entgegenzuwirken, existieren zahlreiche Überprüfungen, welche die Plausibilität detektierter Black Bursts einschätzen. Zum Beispiel findet sich in der Implementierung des BBS-Treibers eine Funktion namens isSofExpected(t_{start}, t_{end}), welche auf einem bereits synchronisierten Knoten bei Erhalt eines Black Bursts prüft, ob der Empfangszeitpunkt des erkannten Black Bursts zu dem SOF-Bit eines erwarteten Tick-Rahmens passt. Dies ist möglich, da die Synchronisationsungenauigkeit bekannt und deterministisch beschränkt ist, sodass ein bereits synchronisierter Knoten den Empfangszeitpunkt von neuen Tick-Rahmen zeitlich einschränken kann. Die Filterung von „falschen“ SOF-Bits ist demnach durch die Überprüfung des Empfangszeitpunkts sehr zuverlässig möglich.

Dekodiert ein Knoten hingegen einen dominanten Black Burst während des Empfangs eines Tick-Rahmens an einer Bitposition, an der kein dominanter Black Burst gesendet wurde, kann der Knoten den Fehler nicht anhand des Empfangszeitpunkts des Black Bursts identifizieren. Er kann den Fehler allerdings nach vollständigem Empfang des Tick-Rahmens mithilfe der Prüfsumme erkennen. Bei kurzen Tick-Rahmen hat sich das Hinzufügen eines Paritätsbits, welches den Tick-Rahmen auf ungerade Parität ergänzt, als zusätzliche Redundanz als ausreichend erwiesen. Mit einem Paritätsbit kann jede ungerade Anzahl an fehlerhaften Bits innerhalb eines Tick-Rahmens erkannt werden. Bereits synchronisierte Knoten haben zusätzlich die Möglichkeit, mittels der in dem Tick-Rahmen kodierten Rundennummer und dem Empfangszeitpunkt des Tick-Rahmens die Plausibilität des empfangenen Rahmens zu prüfen. Da die Synchronisationsungenauigkeit $d_{maxOffset}$ im Allgemeinen deutlich geringer ist als die Rundendauer d_{round}, erwartet ein synchronisierter Knoten in einem von der Synchronisationsungenauigkeit vorgegebenem Zeitfenster den Empfang einer bekannten Rundennummer und kann Tick-Rahmen, deren Rundennummer abweicht, verwerfen.

Ein unsynchronisierter Knoten, der nach dem Einschalten nach bestehenden Netzen sucht, hat kein Vorwissen, um aus dem Empfangszeitpunkt Rückschlüsse zu ziehen. Damit sich der Knoten nicht ausschließlich auf das Paritätsbit verlassen muss, existieren zwei Vorsichtsmaßnahmen. Zunächst werden Rundennummern innerhalb gültiger Tick-Rahmen in einer Weise kodiert, die die Wahrscheinlichkeit, dass Rauschen oder Interferenzen gültige Tick-Rahmen erzeugen, minimiert. Im Konkreten beginnt die Zählung der Rundennummer bei 1, was zusammen mit dem dominanten SOF-Bit und dem ungeraden Paritätsbit dafür sorgt, dass pro Tick-Rahmen mindestens drei dominante Black Bursts gesendet werden. Weniger dominante Black Bursts bedeuten automatisch einen verfälschten Tick-Rahmen. Bei durchschnittlich zehn „falschen“ Black Bursts pro Sekunde (siehe Abschnitt 5.1.4.1) ist die Wahrscheinlichkeit sehr klein, dass drei der falschen

Black Bursts in ein Zeitfenster der Größe d_{round} fallen[18]. Zusätzlich synchronisiert sich ein Knoten erst nach mehreren erfolgreichen aufeinanderfolgenden Synchronisationsphasen zu einem bestehenden Netzwerk[19], was die Wahrscheinlichkeit nochmals verringert, dass sich ein Knoten auf durch Rauschen erzeugte Black Bursts synchronisiert.

5.1.4.5 Verlust von Black Bursts

Der Verlust von dominanten Black Bursts kann mehrere Ursachen haben[20]. Zunächst kann eine zu große Distanz zwischen Sender und Empfänger dazu führen, dass die Medienbelegung nicht korrekt durch den CCA-Mechanismus des Transceivers erkannt wird. In diesem Fall bemerkt der Empfänger entweder gar keine Medienbelegung oder die Dauer der Medienbelegung stimmt nicht mit der gültigen Dauer eines Black Bursts überein. Des Weiteren kann die Medienbelegung, die durch einen regulär gesendet Black Burst verursacht wurde, auch durch Interferenzen oder starkes Rauschen so stark verlängert werden, dass die gemessene Medienbelegung nicht mehr der Dauer eines gültigen Black Bursts entspricht.

Beide Fälle resultieren darin, dass ein dominant gesendeter Black Burst als rezessiver Black Burst empfangen wird, da die Implementierung an dieser Stelle nicht zwischen der Detektion einer für einen Black Burst zu langen oder zu kurzen Medienbelegung und der Detektion gar keiner Medienbelegung unterscheidet. Abhängig von dem Zeitpunkt, zu dem dies passiert, kann der Verlust des Black Bursts unterschiedliche Folgen haben: Steht der verloren gegangene Black Burst für das SOF-Bit eines Tick-Rahmens, kann sich der Knoten in der aktuellen Runde nicht mehr auf den Referenztick synchronisieren, da der Referenztick aus dem Empfangszeitpunkt eines SOF-Bits und der empfangenen Rundennummer errechnet wird. Um zu verhindern, dass andere dominante Bits des Tick-Rahmens als SOF-Bit missgedeutet werden, findet einerseits das Paritätsbit und andererseits die bereits erwähnte Funktion isSofExpected(t_{start}, t_{end}) ihre Anwendung. Geht ein Black Burst nach erfolgreich empfangenen SOF-Bit innerhalb eines Tick-Rahmens verloren, greift ebenfalls das Paritätsbit. Zusätzlich wird, wie oben beschrieben, der Tick-Rahmen anhand der dekodierten Rundennummer auf seine Plausibilität hin geprüft.

Es existieren demnach viele Prüfungen, die vermeiden, dass sich Knoten auf einen falschen Referenzzeitpunkt synchronisieren. Das Ignorieren von fehlerhaften Tick-Rahmen hat allerdings zur Folge, dass Knoten während einer Synchronisationsphase möglicherweise gar nicht neu synchronisiert werden. In der ursprünglichen Version von BBS sendete jeder synchronisierte Knoten maximal einen Tick-Rahmen pro Synchronisationsphase [GK08]. Der Master-Knoten blieb zum Beispiel nach Abschluss der ersten Synchronisationsrunde für den Rest der Synchronisationsphase passiv. Dieses Verhalten wurde dahingehend geändert, dass der Master-Knoten und jeder Knoten, der einen Tick-Rahmen fehlerfrei empfängt, in allen folgenden Runden der aktuellen Synchronisationsphase Tick-Rahmen mit der entsprechenden Rundennummer senden. Diese Maßnahme verhindert, dass temporäre Störungen während einer der ersten Synchronisationsrunden einen Großteil des Netzwerks unsynchronisiert lassen. Die Idee lässt sich auch

[18] Für den Imote2 ergibt sich bei $n_{maxHops} = 3$ und $d_{macro} = 1\,s$ $d_{round} < 4ms$ (weitere Informationen siehe [GK08]), was deutlich geringer ist als das durchschnittliche Auftrittsintervall eines *false positives* von $1\,s/10 = 100\,ms$. Allerdings müsste durch weitere Messungen untersucht werden, ob als Black Burst erkannte Störungen statistisch unabhängig sind.

[19] Die Anzahl an notwendigen gültigen Synchronisationsphasen ist ein konfigurierbarer Parameter.

[20] In diesem Unterabschnitt wird nur der Verlust von Black Bursts betrachtet, die Teil eines regulär erwarteten Tick-Rahmens sind. Der Verlust von Black Bursts aus benachbarten Netzwerken hat in der Regel kaum Auswirkungen und wird daher nicht behandelt.

weiterentwickeln, indem $n_{maxHops}$ bewusst größer als der maximale Netzwerkdurchmesser gewählt wird. Dadurch existieren zusätzliche „Reserverunden“ und Störungen sind bis zu einem gewissen Grad ohne Einschränkung der Synchronisation tolerierbar.
Außerdem hat das Senden von Tick-Rahmen während der kompletten Synchronisationsphase den Vorteil, dass sich neue Knoten, die nahe am Master-Knoten positioniert sind, schneller zu einem bestehenden Netzwerk synchronisieren können. Wie oben beschrieben, warten neue Knoten mehrere gültige Tick-Rahmen ab, bevor sie sich zu einem Netzwerk synchronisieren. Diese Vorsichtsmaßnahme verlangt keineswegs das Abwarten mehrerer Macro-Slots. Ausreichend ist bereits das Abwarten mehrerer gültiger Tick-Rahmen in aufeinanderfolgenden Runden. Durch das anhaltende Senden von Tick-Rahmen durch bereits synchronisierte Knoten können diese Bedingungen früher erfüllt sein und neue Knoten können dem Netz schneller beitreten.

5.1.5 Flexibilität der rundenbasierten Weiterleitung

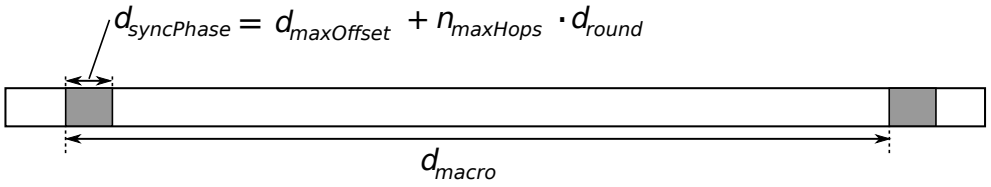
BBS sieht ursprünglich vor, die Synchronisation netzwerkweit in einem Block durchzuführen. Aus Effizienzgründen ist dies äußerst sinnvoll, da Knoten, die einen Tick-Rahmen für die Resynchronisation erwarten, bereits $d_{maxOffset}$ vor dem berechneten Empfang des SOF-Bits aufgrund der Synchronisationsungenauigkeit mit dem Start der Übertragung rechnen müssen. Wird die Synchronisation in einem Block durchgeführt, muss der Overhead dieser zusätzlichen *guard time* genau einmal vor der Synchronisationsphase entrichtet werden.
In manchen Anwendungsszenarien ist es allerdings nicht möglich, die Synchronisation in einem Block durchzuführen, da die Unterbrechung der regulären Kommunikation durch die Synchronisationsphase zu lang und nicht hinnehmbar ist. Beispiele solcher Systeme findet man in der Regelungstechnik bei *Networked Control Systems* (NCS), beispielsweise bei dem Anwendungsfall des inversen Pendels. Diese Systeme haben die Eigenschaft, dass Sensor- und Stellwerte in einem streng periodischen Muster übertragen werden müssen. Das Medium ist hierbei nicht unbedingt völlig ausgelastet, sodass noch genügend Kapazität für die Durchführung der Synchronisation vorhanden ist, allerdings nicht in einem Block.
Um den Anforderungen solcher Systeme gerecht zu werden, wurde ein Parameter $d_{roundInterval}$ eingeführt, mit welchem die Synchronisationsrunden in einem konfigurierbaren Intervall auf den Macro-Slot verteilt werden können. Abbildung 5.4 zeigt die Möglichkeiten der Verwendung dieses Parameters:

(a) $d_{roundInterval} = 0$, keine Verteilung: Die bisher vorgestellte Verteilung der Runden. Alle Knoten werden in einem Block synchronisiert.

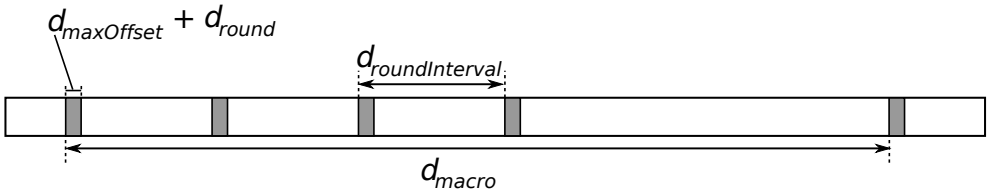
(b) $n_{maxHops} \cdot d_{roundInterval} < d_{macro}$, schwach periodische Verteilung: Synchronisationsrunden finden in festen Abständen statt, allerdings entspricht der Abstand zwischen der letzten Synchronisationsrunde und dem Start des folgenden Macro-Slots nicht dem Abstand zwischen den Runden innerhalb eines Macro-Slots.

(c) $n_{maxHops} \cdot d_{roundInterval} = d_{macro}$, streng periodische Verteilung: Synchronisationsrunden finden vollständig periodisch statt. Die Grenzen von Macro-Slots sind nicht mehr anhand der Aufteilung der Runden zu unterscheiden. Rundennummern werden aber weiterhin in Tick-Rahmen mitgesendet und zyklisch durchnummeriert. Ebenfalls beginnt der Master-Knoten zu Beginn jedes Macro-Slots als einziger Knoten mit der Übertragung von Tick-Rahmen in Runde 1.

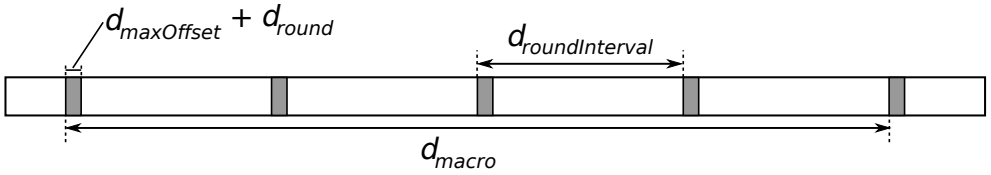
Es wäre weitergehend möglich, die BBS-Runden noch flexibler auf einen Macro-Slot zu verteilen und zum Beispiel Gruppierungen der Runden oder deren aperiodische Verteilung innerhalb des



(a) Netzwerkweite Synchronisation findet innerhalb eines Blocks pro Macro-Slot statt.



(b) Schwach periodische Verteilung der BBS-Runden.



(c) Streng periodische Verteilung der BBS-Runden.

Abbildung 5.4.: Möglichkeiten wie BBS-Runden auf einen Macro-Slot verteilt werden können.

Macro-Slots zu erlauben. Da hierfür allerdings bisher keine Notwendigkeit vorlag, wurde aus Komplexitätsgründen in der Implementierung darauf verzichtet.

Da bisher nur eine Implementierung der master-basierten Version von BBS vorliegt, beschränkt sich die vorgestellte Lösung auf das master-basierte Verfahren. Für das dezentrale und hybride Verfahren wäre eine flexiblere Aufteilung der Synchronisationsrunden auf einen Macro-Slot ebenfalls sinnvoll und erstrebenswert. Da dezentrale Tick-Rahmen jedoch keine Rundennummer beinhalten, kann die beschriebene Lösung nicht Eins-zu-Eins übernommen werden.

Die Verwendung von $d_{roundInterval}$ sollte überlegt auf das Szenario angepasst werden. Ist die Anwendung der Synchronisation innerhalb eines einzigen Blocks möglich, sollte auf die Verteilung der Runden aus Effizienzgründen verzichtet werden, denn bei Verteilung der Runden wächst der Overhead der Synchronisation mindestens um $(n_{maxHops} - 1) \cdot d_{maxOffset}$ (siehe Abbildung 5.4). Des Weiteren muss beachtet werden, dass bei einem zu großen Wert für $d_{roundInterval}$ die Länge von gültigen Black Bursts wachsen kann und daher $d_{masBurst}$ unter Umständen angepasst werden muss. Der Hintergrund dieses Seiteneffekts liegt in der Synchronisationsungenauigkeit zweier Knoten, die zeitgleich Tick-Rahmen senden bzw. weiterleiten. Wird die Synchronisation in einem Block durchgeführt, leiten die beiden Knoten unmittelbar nach dem Empfang des Tick-Rahmens den Tick-Rahmen der folgenden Runde weiter. In diesem Fall kann die Ungenauigkeit durch unterschiedlich schnell laufende Hardware-Quarze ignoriert werden. Findet die Weiterleitung allerdings erst nach $d_{roundInterval}$ statt, kann die initiale Ungenauigkeit so stark angestiegen sein, dass die Länge von „gleichzeitig“ gesendeten Black Bursts signifikant wächst. Der aktuelle Stand der Implementierung ignoriert dieses Problem, da dieser Effekt nur bei großen Werten für $d_{roundInterval}$ auftritt[21]; und auch dann nur unter äußerst ungünstigen Umständen spürbar ist.

[21] Bei $d_{roundInterval} = 200\,ms$ kann die Genauigkeit der Synchronisation zwischen zwei Synchronisationsrunden auf dem Imote2 beispielsweise um 16 μs sinken [GK08]

5.2 Realisierung des BBS-Treibers

Die zeitliche Ausführung der Protokollschritte von BBS stellt hohe Anforderungen an die Implementierung des Protokolls in Software. Die Anforderungen beziehen sich sowohl auf die exakte Startzeit der Ausführung, die im Bereich von μs liegt, als auch auf die Ausführungsdauer der Protokollimplementierung, die zur Erfüllung der Anforderungen gering gehalten werden muss. Da in einem SDL-System auch mit den Erweiterungen durch TC-SDL keine deterministische Garantie für die Startzeit einzelner Agenten gegeben werden kann und SDL-Transitionen außerdem einen zu hohen Overhead für eine effiziente Spezifikation von BBS in SDL mit sich führen, scheidet die vollständige Spezifikation des BBS-Protokolls auf SDL-Ebene aus. Stattdessen wird im Folgenden ein Kompromiss zwischen dem modellgetriebenen Entwicklungsgedanken und einer effizienten hardwarenahen Implementierung von BBS in der Programmiersprache C vorgestellt.

5.2.1 Übersicht

Der wohl größte Vorteil bei der praktischen Anwendung von SDL ist die Unterstützung für einen modellgetriebenen Entwicklungsprozess, der es ermöglicht, automatisiert Implementierungen für unterschiedliche Plattformen basierend auf einer einzigen SDL-Spezifikationen zu erstellen [Got07]. Aufgrund der von BBS geforderten deterministischen Garantien ist man bei der Realisierung von BBS gezwungen, diesen holistischen Ansatz in Teilen aufzugeben. Dennoch heißt das Aufgeben dieses Vorteils nicht, dass ein Entwickler, der BBS als Basisdienst verwenden möchte, zu manuellen Änderungen im Quellcode gezwungen wird, um das Synchronisationsprotokoll an das gewünschte Szenario anzupassen.

Ziel der im folgenden präsentierten Umsetzung war es vielmehr, BBS sowohl effizient als auch konfigurierbar zu implementieren, sodass szenariospezifische Einstellungen innerhalb der SDL-Spezifikation vorgenommen werden können und in der effizienten Implementierung Beachtung finden. Zu den szenariospezifischen Einstellungen gehören zum Beispiel die Wahl der BBS-Version (master-basiert, dezentralisiert, hybrid) oder des Netzwerkdurchmessers $n_{maxHops}$. Die Realisierung von BBS besteht infolgedessen aus zwei Teilen: Einer SDL-Komponente, die die komfortable Konfiguration ermöglicht, und einem *SEnF*-Treiber, der die Protokollfunktionalität effizient und hardwarenah in C implementiert.

Diese zweigeteilte Realisierung von BBS ist daher nur in Teilen plattformspezifisch. Vielmehr führt die vorgestellte Lösung eine Abstraktionsstufe ein, die es ermöglicht, dass SDL-Komponenten als Dienstnutzer die Direktimplementierung des BBS-Treibers verwenden, ohne sich dabei um plattformabhängige Hardware-Timer oder Registerwerte kümmern zu müssen. Die Entwicklung der SDL-Spezifikation selbst bleibt demnach hardwareunabhängig. Hardwarespezifische Teile kommen erst im Generierungsschritt hinzu, wenn sich der Entwickler für eine Zielplattform entscheidet. Voraussetzung hierbei ist, dass BBS auf der gewählten Zielplattform unterstützt wird, d.h. dass ein entsprechender *SEnF*-Treiber vorhanden ist. Dies ist zur Zeit nur für die Imote2-Plattform umgesetzt. Werden Implementierungen für andere Plattformen, wie zum Beispiel den Simulator *ns+SDL* oder der MICAz-Plattform, nachgereicht, können bereits spezifizierte SDL-Systeme ohne Anpassung auch auf diesen Plattformen verwendet werden.

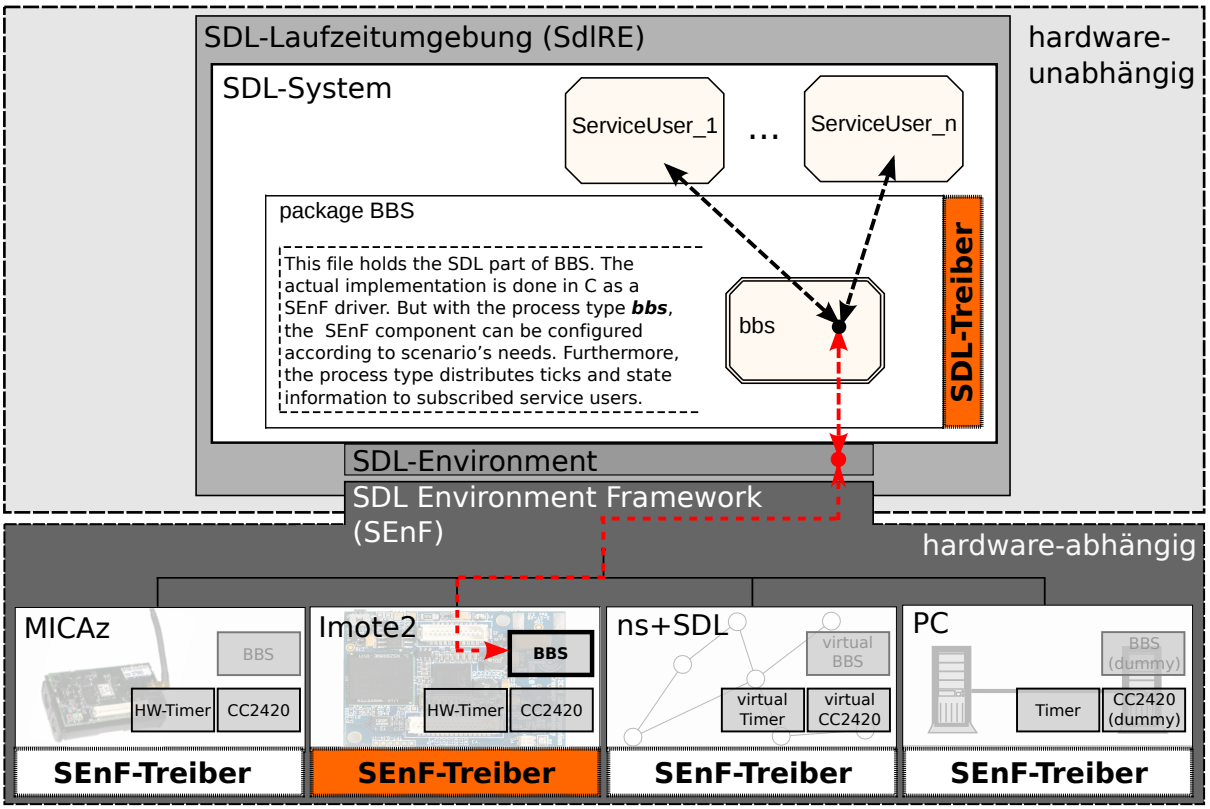


Abbildung 5.5.: Realisierung von BBS als SDL-Komponente und *SEnF*-Treiber.

5.2.2 Aufteilung: SDL-Treiber und SEnF-Treiber

Abbildung 5.5 stellt die zweiteilige Realisierung schematisch vor. Der erste Teil des BBS-Treibers, der BBS-SDL-Treiber, ist in der Abbildung auf Ebene des SDL-Systems zu erkennen und besteht aus einer Prozess-Typ-Definition. SDL-Komponenten, die BBS zur Synchronisation verwenden wollen, müssen diesen Prozess-Typ instanziieren. Sie haben hierüber außerdem die Möglichkeit, BBS für das Szenario zu konfigurieren. Des Weiteren sind Instanzen des Prozess-Typs nach erfolgreicher Synchronisation für die Verteilung von Referenzticks innerhalb des SDL-Systems verantwortlich. In Abschnitt 5.2.3 ist die Verwendung des SDL-Treibers im Detail beschrieben.

Über das SDL-Environment kommuniziert der BBS-SDL-Treiber mit dem BBS-SEnF-Treiber, dem zweiten Teil des BBS-Treibers. Die Kommunikation beinhaltet einerseits die szenariospezifische Konfiguration des BBS-Protokollverhaltens, andererseits dient sie auch zum Bekanntmachen von Referenzticks bei erfolgreicher Synchronisation. Im Unterschied zum SDL-Treiber muss der *SEnF*-Treiber spezifisch für die verwendete Plattform implementiert sein. Hinter diesem BBS-SEnF-Treiber steht allerdings keine reale Hardwarekomponente, wie es üblicherweise bei Treibern in *SEnF* der Fall ist. Stattdessen verwendet der BBS-SEnF-Treiber mit dem CC2420-Treiber einen anderen Treiber aus *SEnF*, um auf reale Hardware zuzugreifen. Der BBS-Treiber kann allerdings als Treiber einer virtuellen Hardware angesehen werden, die wie eine reale Hardware aus SDL heraus verwendet werden kann. Abschnitt 5.2.4 geht hierauf näher ein.

5.2.3 SDL-Treiber

Der SDL-Teil des BBS-Treibers besitzt drei Aufgaben. Einerseits muss er ein komfortables Interface zur Steuerung und Anpassung von BBS an das Szenario bereitstellen. Hierfür beinhaltet das Interface zwischen Dienstnutzer und SDL-Treiber vier Signale:

- SetParameter: Das Signal besitzt zwei Parameter. Der erste Parameter ist ein Schlüssel, der angibt welcher Konfigurationsparameter von BBS geändert werden soll. Der zweite Parameter gibt den Wert vor, welchen der Konfigurationsparameter erhalten soll.
- ResetParameter: Dieses Signal besitzt nur einen Parameter, den Schlüssel eines BBS-Konfigurationsparameters. Bei Erhalt dieses Signals setzt der SDL-Treiber den Parameter zurück auf den *Default*-Wert.
- Enable: Mit Enable kann der Dienstnutzer zur Aktivierung von BBS auffordern. Der BBS-SDL-Treiber leitet diese Aufforderung zusammen mit den aktuell gesetzten Konfigurationsparametern an den *SEnF*-Treiber weiter. Abhängig von den Werten der Parameter beginnt der *SEnF*-Treiber anschließend mit der Abarbeitung der BBS-Protokollfunktionalität.
- Disable: Dieses Signal führt zum Stopp der BBS-Funktionalität, bis ein Enable-Signal des Dienstnutzers das Protokoll erneut startet.

Insgesamt können mit SetParameter / ResetParameter neun Parameter von BBS verändert werden. Die wichtigsten Parameter hiervon sind die zu verwendende BBS-Version, der Netzwerkdurchmesser (entspricht $n_{maxHops}$), das Resynchronisationsintervall (entspricht d_{macro}) und die bereits oben erwähnte Knotenrolle, bei welcher BBS zwischen `SLAVE`, `MASTER` und `MASTER_IF_REQUIRED` unterscheidet. Anhang C.2 listet die möglichen Schlüssel zusammen mit ihren *Default*-Werten auf.

Bei Systemstart ist BBS immer deaktiviert, d.h. ein Dienstnutzer, der die *Default*-Werte der Konfiguration beibehalten möchte, muss mindestens ein Enable-Signal an den SDL-Treiber senden. Es ist zu beachten, dass Konfigurationsparameter nur verändert werden dürfen, wenn BBS nicht aktiviert ist. Ist BBS aktiviert und soll neu konfiguriert werden, muss es zuerst mit dem Disable-Signal deaktiviert und nach Neusetzen der Parameter mit dem Enable-Signal wieder aktiviert werden. Im Allgemeinen sollte dieser Fall aber äußerst selten vorkommen, denn das Parametrisieren von BBS ist in den meisten Anwendungsfällen genau einmal während des Systemstarts notwendig.

Die zweite Aufgabe des BBS-SDL-Treibers ist das Verteilen von Referenzticks innerhalb des SDL-Systems. Ist BBS aktiviert, erhält der SDL-Treiber nach jeder erfolgreichen Synchronisation einen Zeitstempel aus dem *SEnF*-Treiber, der dem lokalen Referenzzeitpunkt zum Zeitpunkt des nächsten Ticks entspricht. In der Regel ist dieser Tick nicht nur für einen, sondern für mehrere Dienstnutzer innerhalb des SDL-Systems von Interesse. Beispielsweise existieren in vielen Systemen neben Komponenten für das reservierungsbasierte/zeitgesteuerte Versenden von Rahmen weitere Komponenten zum Einsparen von Energie, die abhängig vom letzten Referenztick zwischen den verschiedenen Energiemodi wechseln. Diese zwei Beispiele zeigen nur exemplarisch, dass das Vorhandensein mehrerer Dienstnutzer kein Ausnahmefall ist.

Aus diesem Grund ist in dem SDL-Treiber ein *Publish/Subscribe-Pattern* (manchmal auch als *Observer-Pattern* bekannt) umgesetzt [GHJV09]: Dienstnutzer, die den Referenztick benötigen, können sich mit einem Subscribe4Tick-Signal für den Empfang von Referenzticks registrieren. Der SDL-Treiber speichert daraufhin die PID des Dienstnutzers in der Liste der Empfänger von Referenzticks, wodurch der Dienstnutzer bei zukünftigem Auftreten eines Referenzticks von dem

SDL-Treiber informiert wird[22]. Hierdurch ist einerseits sichergestellt, dass Referenzticks effizient innerhalb des Systems verteilt werden, da keine zusätzlichen Komponenten benötigt werden, und andererseits ist der Entwickler von zusätzlichem Aufwand befreit. Ein Nachteil dieses Ansatzes ist, dass jeder Dienstnutzer über einen expliziten Signalpfad mit dem SDL-Treiber verbunden sein muss.

In manchen Anwendungen ist es nicht ausreichend, dass die Dienstnutzer nur über Referenzticks informiert werden. Es sind ebenso Fälle denkbar, in denen weitere Informationen über den Zustand des BBS-Protokolls innerhalb des SDL-Systems von Interesse sind. Beispielsweise weiß ein Knoten, dessen Rolle mit `MASTER_IF_REQUIRED` konfiguriert ist, nicht, ob er die Master-Rolle tatsächlich ausführt oder ob ein anderer Master vor ihm vorhanden war und er die Rolle des Slaves übernimmt. Diese Information könnte aber auf höherer Netzwerkschicht von Interesse sein, zum Beispiel um zu entscheiden, ob ein Knoten, der aktuell außer der eventuellen Synchronisation keine Funktion mehr besitzt, sich ohne Auswirkungen aus dem Netz entfernen kann.

Aus diesem Grund werden verschiedene Zustandsinformationen aus dem *SEnF*-Treiber an den SDL-Treiber gemeldet, dessen dritte Aufgabe die Verteilung dieser Informationen innerhalb des Systems ist. Hierzu kommt ebenfalls ein *Publish/Subscribe-Pattern* zum Einsatz und Dienstnutzer können sich mit dem Signal Subscribe4Event für den Empfang von Zustandsänderungen anmelden. Neben der erwähnten tatsächlichen Rolle beinhalten die gemeldeten Zustandsinformationen ein *Flag*, das aussagt, ob der Knoten momentan synchronisiert ist, und die aktuelle Synchronisationsungenauigkeit, welche zum Beispiel ansteigen kann, falls ein Knoten aufgrund von transienten Links den Empfang eines Tick-Rahmens verpasst. Um unnötige Last zu vermeiden, werden diese Informationen nur im System verbreitet, wenn sich tatsächlich etwas am Zustand geändert hat. Bei Bedarf lassen sich die Zustandsinformationen um weitere Informationen erweitern. Ein guter Kandidat wäre die Hop-Distanz des Knotens zum Master-Knoten, falls der betrachtete Knoten nicht selbst die Master-Rolle einnimmt. Diese Information ist im BBS-SEnF-Treiber bei Verwendung der master-basierten oder hybriden BBS-Version aufgrund der Rundennummer innerhalb eines Tick-Rahmens verfügbar und könnte zum Beispiel in einem Szenario mit hoher Mobilität verwendet werden, um vor der Überschreitung des maximalen Netzdurchmessers $n_{maxHops}$ zu warnen.

Zum Abschluss dieses Abschnitts soll noch ein Beispiel die Verwendung des Treibers demonstrieren. Hierzu zeigt Abbildung 5.6 ein *Message Sequence Chart* (MSC, [Int04]), in welchem zwei Dienstnutzer BBS verwenden. Beide Dienstnutzer registrieren sich für Referenzticks, aber nur ServiceUser_1 registriert sich für die Weiterleitung von Zustandsänderungen. Des Weiteren ändert ServiceUser_2 zwei Parameter von BBS: Zuerst legt er die Rolle des Knotens mit SLAVE fest, was zur Folge hat, dass der Knoten zukünftig nie die Synchronisation anstößt, sondern sich nur auf empfangene Tick-Rahmen synchronisiert. Anschließend setzt er den Netzwerkdurchmesser auf 5 Hops und aktiviert BBS mit dem Enable-Signal. Der SDL-Treiber sendet daraufhin den „Startschuss“ zusammen mit der aktuell gesetzten Konfiguration an den *SEnF*-Treiber (BBS_ENABLE). Da die zu verwendende BBS-Version nicht explizit gesetzt wird, bleibt sie auf ihrem *Default*-Wert, was dem master-basierten Verfahren entspricht (siehe auch Anhang C.2). Nachdem die Synchronisation erfolgt ist, d.h. der Knoten erfolgreich Tick-Rahmen empfangen hat, meldet der *SEnF*-Treiber Referenzticks an das SDL-System, die daraufhin von dem SDL-Treiber an die beiden Dienstnutzer weitergeleitet werden.

[22] Die Spezifikation sieht aufgrund der fehlenden Notwendigkeit bisher noch kein UnsubscribeFromTick vor. Ein solches Signal könnte aber nachträglich analog hinzugefügt werden.

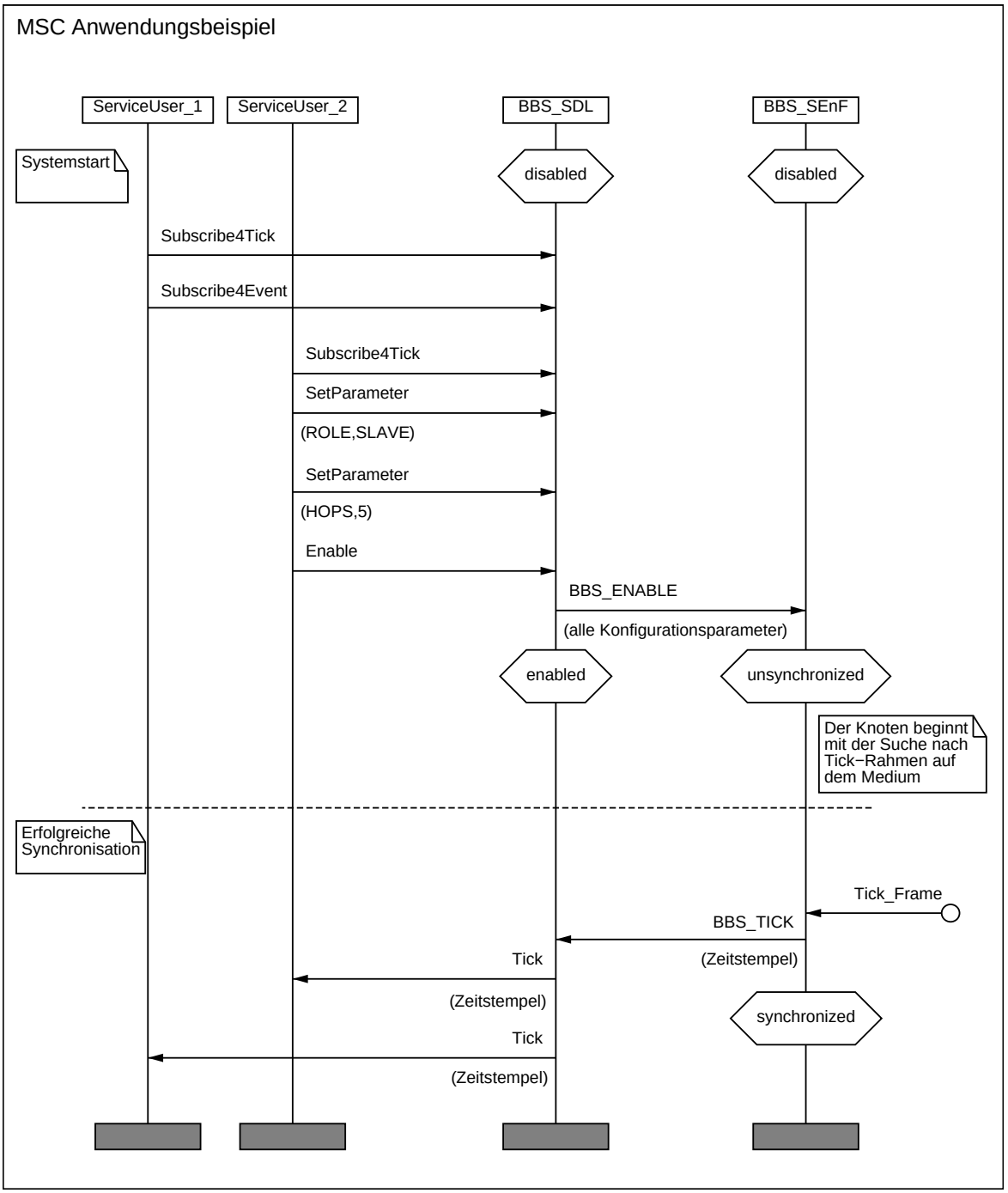


Abbildung 5.6.: Ein Beispiel, das einen Anwendungsfall von BBS mit zwei Dienstnutzern zeigt.

5.2.4 SEnF-Treiber

In dem BBS-SEnF-Treiber ist die eigentliche Protokollfunktionalität von BBS implementiert. Solange der Treiber nicht explizit aus dem SDL-System heraus aktiviert wird, werden empfangene Black Bursts ignoriert. Sobald der Treiber ein BBS_ENABLE-Signal erhält, startet der Knoten mit dem BBS-Protokoll und verhält sich entsprechend der in dem BBS_ENABLE-Signal mitgegebenen Konfiguration. Abhängig von der Konfiguration sendet er also Tick-Rahmen oder durchsucht das Medium nach einem bereits synchronisierten Netzwerk.

Nach Aufbau der Synchronisation und nach jeder erfolgreichen Resynchronisation meldet der *SEnF*-Treiber die neuen Referenzticks an das SDL-System. Unabhängig der Konfiguration von $d_{roundInterval}$ und von der Runde, in welcher ein Tick-Rahmen empfangen wurde, ist der lokale Referenzzeitpunkt immer mit dem Ende der Synchronisationsphase assoziiert. Abbildung 5.7 verdeutlicht diese Assoziation anhand eines Beispiels mit $n_{maxHops} = 4$ in Abhängigkeit von $d_{roundInterval}$ und zeigt zusätzlich mögliche Zählungen von Micro-Slots, die zyklisch nach jedem Referenztick bei 1 beginnen. Diese Zählungen stellen nur Beispiele dar, wie Micro-Slots durchnummeriert werden können. Insbesondere im Falle von $d_{roundInterval} > 0$ gibt es auch andere sinnvolle Alternativen einer Unterteilung des Macro-Slots. Entscheidend ist, dass alle Knoten die Micro-Slots in gleicher Art und Weise nummerieren.

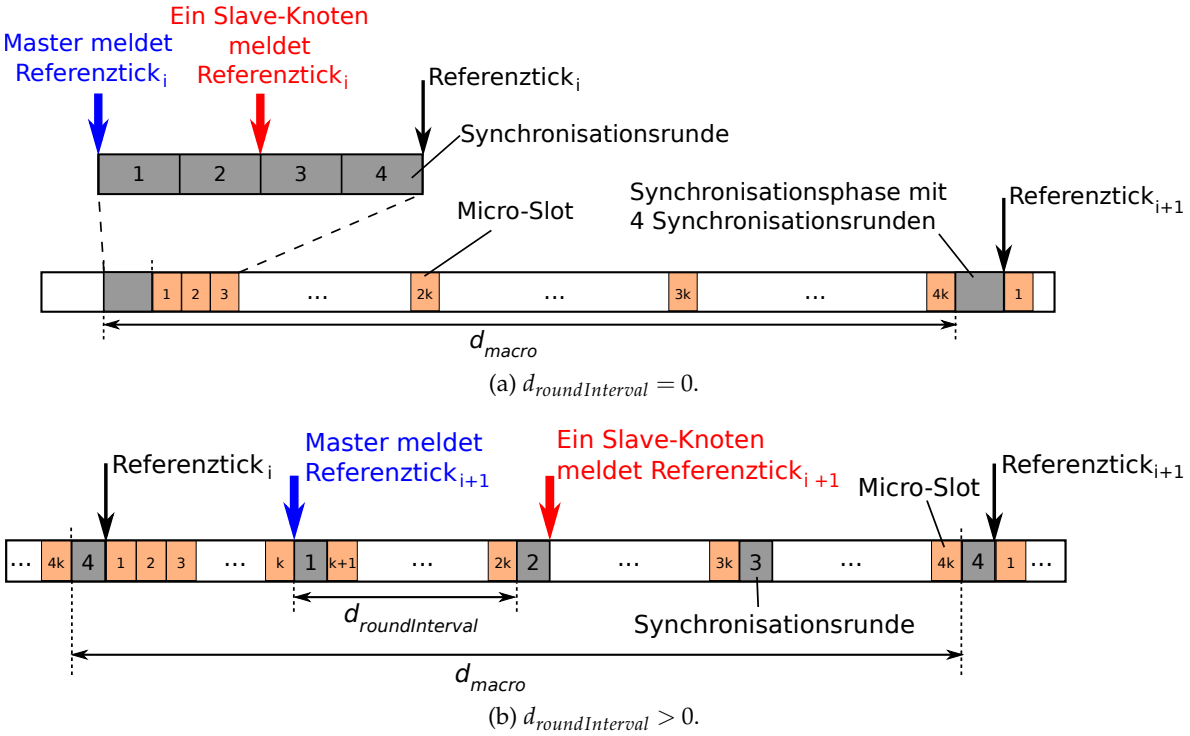


Abbildung 5.7.: Assoziationen von Referenzticks innerhalb eines Macro-Slots an einem Beispiel mit $n_{maxHops} = 4$.

Abbildung 5.7 beinhaltet ebenfalls beispielhaft für den Master-Knoten und einen (beliebig gewählten) Slave-Knoten die Zeitpunkte, zu denen der BBS-SEnF-Treiber die Referenzticks an das SDL-System meldet. Referenzticks, in der Abbildung $Referenztick_i$ und $Referenztick_{i+1}$, sind zwar mit den lokalen Zeitpunkten am Ende der letzten Synchronisationsrunde assoziiert, werden aber möglichst früh von dem *SEnF*-Treiber nach oben gemeldet. Bei dem Master-Knoten bedeutet dies, dass der Referenztick direkt zu Beginn der Synchronisationsphase, d.h. vor der ersten Synchronisationsrunde, an das SDL-System gemeldet wird. Auf Slave-Knoten kann der BBS-Treiber den Tick melden, sobald ein gültiger Tick-Rahmen empfangen wurde. In Abbildung 5.7 ist dieser Fall nach der zweiten Runde eingezeichnet. Im Regelfall wird ein Referenztick immer mit einem lokalen Zeitpunkt in der Zukunft assoziiert. Dem SDL-System bleibt dadurch Zeit, eine Aktion (zum Beispiel das Senden eines Datenrahmens) direkt zu Beginn des nutzbaren Anteils des Macro-Slots zu starten oder zu planen.

Statt der Assoziation des Referenzticks mit dem Ende der Synchronisationsphase, könnte der Referenztick auch mit den lokalen Zeitpunkten zu Beginn der Synchronisationsphase assoziiert

werden. Zum Beispiel ist in [GK08] der Referenztick bei dem Start der Synchronisationsphase gesetzt. Entscheidend ist, dass alle Knoten den Referenztick gleich assoziieren. Der hier gegangene Weg hat den Vorteil, dass der gemeldete Referenzzeitpunkt dem Zeitpunkt entspricht, bei dem höhere Protokolle das Medium wieder nutzen können. Bei der Assoziation des Ticks mit dem Beginn der Synchronisationsphase müsste die Dauer der Synchronisation auf den gemeldeten Zeitpunkt addiert werden, um den frühestmöglichen Zeitpunkt zum Versenden regulärer Datenrahmen zu erhalten. Bei der Planung und Einteilung eines Macro-Slots macht dies allerdings keinen Unterschied, da unabhängig von der Assoziation der Ticks die Dauer der Synchronisationsphase in höheren Protokollschichten – beispielsweise beim Definieren von Zeitintervallen zum Sparen von Energie oder beim periodischen Reservieren von Sende-Slots – Beachtung finden muss (siehe Anhang C.4).

Hinter dem BBS-SEnF-Treiber steht keine eigene reale Hardware. Der Treiber benötigt aber Hardware-Timer und muss eng mit dem CC2420-Treiber, hinter dem eine reale Hardware steht, kooperieren. Da die Anzahl an Hardware-Timern beschränkt ist und bereits viele der vorhandenen Timer der Imote2-Plattform anderweitig eingesetzt werden, verwendet die Implementierung des Treibers einen einzigen Hardware-Timer. Dieser kommt sowohl für die Bestimmung der Belegungsdauer des Mediums zur Erkennung von Black Bursts als auch für das Abwarten zwischen den Synchronisationsphasen und Synchronisationsrunden zum Einsatz.

Um die CCA-Interrupts des Transceivers zu erhalten, überschreibt der BBS-SEnF-Treiber den entsprechenden CC2420-Interrupt-Handler. Bei Eintreten eines CCA-Events merkt sich der BBS-Interrupt-Handler abhängig vom neuen Zustand des Mediums die Startzeit der Belegung oder prüft, ob die Belegungsdauer der Dauer eines gültigen Black Bursts entspricht. Außerdem wird das Ereignis dem regulären CC2420-Interrupt-Handler gemeldet, sodass dieser gegebenenfalls das Ereignis in Form eines SDL-Signals an das SDL-System melden kann.

Da durch die Anwendung von BBS viele Interrupts als „Nebenprodukte“ erzeugt werden, die für das SDL-System nicht von Interesse sind – etwa die Sendebestätigung eines Black Bursts oder der Empfang eines Black Bursts als regulärer Rahmen[23] – wurden die entsprechenden Interrupt-Handler im CC2420-Treiber um zusätzliche Überprüfungen ergänzt. Diese Überprüfungen haben das Ziel, die entsprechenden Ereignisse innerhalb einer Synchronisationsphase zu filtern, um zu verhindern, dass die Ereignisse den Weg in das SDL-System finden. Es wäre beispielsweise äußerst unsauber, wenn das SDL-System über einen erfolgreichen Rahmenversand informiert wird, obwohl aus Sicht des SDL-Systems gar kein Rahmen versendet wurde.

Aufgrund des unkontrollierbaren Drahtlosmediums hat die Implementierung des BBS-SEnF-Treibers noch mit zusätzlichen Problemen zu kämpfen, welche die deterministische Genauigkeit des Verfahrens bedrohen. So kann der Empfang eines Datenrahmens kurz vor oder während der Synchronisationsphase dazu führen, dass der SPI-Bus, der für die Kommunikation zwischen Mikrocontroller und CC2420-Transceiver verantwortlich ist, belegt ist und Black Bursts nicht rechtzeitig versendet werden können. Dieses Problem kann gelöst werden, indem die Übertragung eines empfangenen Rahmens vom CC2420-Transceiver in den RAM verzögert stattfindet, nachdem der zu sendende Black Burst zu dem Transceiver übertragen wurde. Auf zusätzliche Probleme, wie zum Beispiel das Herausrechnen der DMA/SPI-Transferzeiten, soll an dieser Stelle nicht weiter eingegangen werden.

[23] Da Black Bursts als reguläre MAC-Rahmen mit gültigem *Header/Trailer* implementiert sind, wird jeder Black Burst, der sich nicht mit anderen Black Bursts überlappt, in der Regel auch als regulärer MAC-Rahmen empfangen.

5.2.5 Probleme, Einschränkungen und Verbesserungspotential

Die aufgrund der langen Umschaltzeiten ohnehin sehr ineffiziente Implementierung von Black Bursts mit dem CC2420-Transceiver leidet in der aktuellen Implementierung zusätzlich unter der Verwendung des nicht-optimierten Rahmenformats. Es bleibt ein offenes Problem, effizient zwischen dem regulären Rahmenformat und dem optimierten Rahmenformat mit gekürzter Präambel und weggelassener CRC zu wechseln, das es in Zukunft zu lösen gilt.

Des Weiteren finden sich in der aktuellen Implementierung des CC2420-Treibers einige Quellen von Indeterminismus, welche die garantierte Genauigkeit des Synchronisationsverfahrens bedrohen. Eine dieser Quellen ist der DMA-Transfer von Black Bursts über den SPI-Bus, dessen Dauer von der Anzahl anderer offener DMA-Aufträge abhängt[24]. Eine weitere Quelle von indeterministischer Verzögerung ist die dynamische Allokation von Speicher, die zum Beispiel zum Zwischenspeichern von Rahmen vor dem Transfer über den SPI-Bus vorgenommen wird. In Teilen könnte hier das Vorziehen der Speicherreservierung in weniger zeitkritische Abschnitte und das Halten von „Reserve-Speicher“ (ähnlich dem Halten von Reserve-Threads im Worker-Modell des Apache-Webservers [Apaar]) Verbesserungen bringen.

Zusätzlich erfordert die Bestimmung der Ausführungszeiten der BBS-Implementierung weitere Betrachtungen. Hier ist es allerdings nicht ganz trivial, die Verzögerungen durch die Implementierung genau zu bestimmen, da die Zeiten nicht bei jeder Ausführung konstant sind. BBS führt zwar die meisten Aktionen innerhalb eines Interrupts (Timer- oder CCA-Interrupt) aus, sodass die Verzögerung der Ausführung verglichen mit der Serialisierungsverzögerung von SDL-Transitionen um etliche Größenordnungen geringer ist, allerdings darf sie auch nicht vollständig vernachlässigt werden. Gerade der Einsatz von BBS mit einem SDL-System, das evtl. noch andere Hardware verwendet, lässt befürchten, dass Interrupts durch andere Hardwaregeräte zu einer zusätzlichen Verzögerung in der Ausführung des Protokolls führen. Hinzu kommt, dass die Ausführungszeit der Protokollimplementierung unter Anderem auch von äußerst schwer zu quantifizierenden Faktoren abhängt, wie zum Beispiel der Anzahl an Cache-Misses.

5.3 Bewertung

Die vorgestellte Realisierung von BBS stellt einen Kompromiss dar, der den harten Zeitanforderungen des Protokolls und dem modellgetriebenen Entwicklungsgedanken gerecht werden will. Die Umsetzung profitiert von den Vorteilen, dass die Einhaltung von Zeitschranken aufgrund der effizienten Implementierung der BBS-Funktionalität in C mit deutlich weniger Problemen zu erreichen ist als es mit einer entsprechenden Realisierung in SDL möglich wäre. Außerdem wird der Knoten aufgrund der effizienten Implementierung nur wenig durch die Abarbeitung des BBS-Protokolls ausgelastet. Des Weiteren hat die vorgestellte Lösung den positiven Effekt, dass Hardwaredetails, wie zum Beispiel die Umschaltzeiten des Transceivers, die die Länge eines Black Bursts maßgeblich beeinflussen, innerhalb des SDL-Systems nicht im Detail bekannt sein müssen.

Die Kehrseite der geschilderten Lösung ist der Bruch mit dem holistischen modellgetriebenen Ansatz. Durch die Auslagerung von Protokollfunktionalität in plattformspezifischen C-Code werden manuelle Implementierungen des Protokolls für unterschiedliche Plattformen notwendig. Es bleibt eine Aufgabe für die Zukunft, durch Steigerung der Effizienz und Vorhersagbarkeit

[24] Der Imote2 besitzt 32 DMA-Kanäle, die in vier Prioritätsklassen eingeteilt sind. Der DMA-Controller wechselt zwischen aktiven Kanälen in Abhängigkeit ihrer Priorität und einer konfigurierbaren Burst-Größe [Cro09].

von SDL den Anteil der manuellen Implementierung immer weiter zu reduzieren und immer mehr Funktionalität nach SDL zu verlagern. Ein notwendiger Schritt hierzu ist die Reduzierung der Reaktionszeit eines SDL-Systems auf ein externes Ereignis und die Ermittlung einer oberen Schranke. Mit dem in Abschnitt 4.2.5.4 vorgestellten Suspendieren von Agenten ist bereits ein Grundstein gelegt, der zur Lösung dieser Aufgabe beitragen kann.

6. KAPITEL

Experimentelle Evaluation

Sowohl die Erweiterung der SDL-Laufzeitumgebung mit dem prioritätsbasierten Planungsverfahren als auch die Umsetzung des *Black Burst Synchronization*-Treibers (BBS) bedürfen einer sorgfältigen Evaluation ihrer korrekten Funktionsweise. Allerdings ist der Evaluation mit der Überprüfung der Funktionalität der Implementierungen noch nicht genüge getan. Im Falle des Planungsverfahrens müssen die Verbesserungen quantitativ erfasst werden, um das Verbesserungspotential in Zahlen zu verdeutlichen und den zusätzlichen Overhead durch die aufwändigere Implementierung abschätzen zu können. Ebenso muss überprüft werden, ob die Implementierung des BBS-Protokolls ausreichend zuverlässig und ausgereift ist, um als Grundlage für Protokolle auf höherer Netzwerkschicht zu dienen.

Dieser Abschnitt stellt die Ergebnisse experimenteller Evaluationen der um TC-SDL erweiterten SDL-Laufzeitumgebung und der Implementierung von BBS vor. Er hilft außerdem zu verstehen, wie sich einzelne Optimierungen der Laufzeitumgebung auf das zeitliche Verhalten eines Systems auswirken. Die Evaluationen wurden mit Hilfe von Imote2-Sensorknoten durchgeführt, die stellvertretend für andere eingebettete Systeme stehen sollen.

Zunächst wird in Abschnitt 6.1 die Imote2-Plattform vorgestellt. Anschließend evaluiert Abschnitt 6.2 das prioritätsbasierte Planungsverfahren und die korrekte Zuweisung von Prioritäten an Agenten. Außerdem gibt der Abschnitt eine quantitative Einschätzung über das Ausmaß an Verbesserungen, die das prioritätsbasierte Planungsverfahren zusammen mit einer günstigen Vergabe von Prioritäten erreichen kann. In Abschnitt 6.3 wird die Realisierung des masterbasierten BBS an zwei Szenarien überprüft. Das erste Szenario testet die Funktionsweise von BBS anhand eines TDMA-Medium-Slottings in einer Single-Hop-Topologie. Das zweite Szenario analysiert die Kollisionsresistenz und die Multi-Hop-Fähigkeit von BBS.

6.1 Die Imote2-Plattform

Alle Messungen, die in den folgenden Abschnitten präsentiert werden, wurden direkt auf der Imote2-Plattform vorgenommen. Abgesehen von dem Fehlen eines CC2420-Transceivers für die PC-Plattform, haben Messungen auf dem Imote2 im Vergleich zu einem PC den Vorteil, dass die Ergebnisse besser reproduzierbar sind. Grund ist die direkte Ausführung auf der Hardware ohne zusätzliches Betriebssystem, welches durch Unterbrechungen und Kontextwechsel die Messergebnisse stark beeinflussen kann. Da das Drahtlosmedium und die von ihm durch den CC2420-Transceiver erzeugten Interrupts nicht in vollem Maße kontrolliert werden können, sind

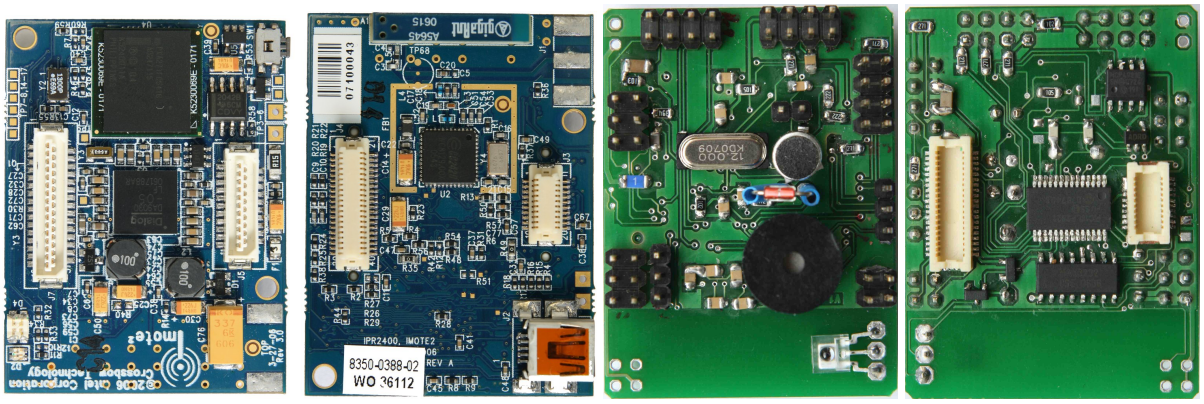


Abbildung 6.1.: Imote2 und Erweiterungsplatine mit zusätzlichen Sensoren, Maßstab 1:1, `http://vs.cs.uni-kl.de/en/people/kraemer/embed/`.

die Ergebnisse dennoch nicht zwangsweise identisch. Allerdings sollten unter ähnlichen externen Bedingungen die Ergebnisse vergleichbar sein.

Wie es bei Messungen häufig der Fall ist, besteht auch bei folgenden Experimenten das Problem, dass die Messung selbst das Verhalten und die Ausführungsdauer beeinflusst. Soweit möglich wurden die Einflüsse der Messung minimiert. Da hauptsächlich Vergleiche zwischen unterschiedlichen Verfahren gezogen werden, die von den Messungen in ähnlicher Weise beeinträchtigt werden, sollten die Auswirkungen im Ergebnis wenig spürbar sein.

Der Imote2 ist ein Knoten für drahtlose Sensornetzwerke [Cro09] (siehe Abbildung 6.1). Er ist ausgestattet mit einem auf der ARM-Architektur basierenden XScale-Prozessor, der Taktraten von 13 bis 416 MHz unterstützt. Der Knoten besitzt 32 kB SRAM, 32 MB Flash-Speicher und zusätzliche 32 MB SDRAM. Für die drahtgebundene Kommunikation verfügt der Imote2 unter Anderem über drei UARTs (*Universal Asynchronous Receiver Transmitter*s), von welchen einer zur Übertragung der Messergebnisse zwischen Imote2 und PC verwendet wurde. Des Weiteren ist der Knoten für die drahtlose Kommunikation mit einem CC2420-Transceiver von *Texas Instruments* ausgestattet [Chi07].

Im Hinblick auf die Realisierung von Black Bursts definiert der CC2420-Transceiver die grundlegenden Eigenschaften wie Übertragungsdauer und Erkennungsverzögerung. Der dem IEEE 802.15.4-Standard folgende Transceiver arbeitet im ISM-Band (*Industrial, Scientific, Medical Band*) im Frequenzbereich um 2,4 Ghz und bietet eine Transferrate von 250 kbps. Die Umschaltzeiten, welche für die Realisierung von Black Bursts entscheidend sind, betragen 192 μs vom Empfangs- in den Sendemodus und 320 μs vom Sende- in den Empfangsmodus. Die 320 μs enthalten bereits die notwendige Zeit, die der Transceiver zusätzlich benötigt, um nach Umschalten des Modus den tatsächlichen Belegungszustand des Mediums zu erkennen. Wenn sich der Transceiver bereits im Empfangsmodus befindet, kann sich die Erkennung einer Medienbelegung um bis zu 8 Symbole verzögern, was bei einer Symboldauer von 16 μs eine maximale Verzögerung von 128 μs ergibt.

6.2 Prioritätsbasiertes Scheduling

In diesem Abschnitt sollen die Auswirkungen der unterschiedlichen Planungsverfahren auf ein SDL-System untersucht werden. Im Fokus steht hierbei das neu eingeführte Verfahren mit statischen Prioritäten.

Die Planungsverfahren sind zunächst knoteninterne Maßnahmen zur Verbesserung der Laufzeit zeitkritischer Aktivitäten. Die beiden Szenarien folgender Experimente bestehen daher ausschließlich aus einem Knoten. Der CC2420-Transceiver des Knotens ist deaktiviert, sodass der Knoten von externen Einflüssen aus dem Drahtlosmedium abgeschottet ist. Die Ergebnisse in diesem Abschnitt werden dadurch kaum von der Umgebung beeinflusst und sollten sehr exakt reproduzierbar sein.

6.2.1 Verzögerungen entlang eines Signalpfades

Das erste Szenario dient primär einer funktionalen Evaluation, in welcher die korrekte Zuweisung von Prioritäten an einzelne SDL-Komponenten überprüft wird. Wie in Kapitel 4 vorgestellt, lassen sich Prioritäten an SDL-Services, SDL-Prozesse und SDL-Blöcke sowie deren Typen und Instanzen zuweisen. Des Weiteren werden Prioritäten auch bei der Vererbung zwischen SDL-Prozessen unterstützt. Die korrekte Erkennung dieser Menge an Möglichkeiten ist Thema des folgenden Szenarios.

6.2.1.1 Beschreibung des Szenarios

Das evaluierte SDL-System ist schematisch in Abbildung 6.2 skizziert. Das System besteht aus mehreren sogenannten *Traces*. Ein *Trace* besteht aus jeweils vier SDL-Blöcken, in denen die Zuweisung von Prioritäten an unterschiedliche SDL-Komponenten vorgenommen wird. Der oberste Block testet die Zuweisung von Prioritäten an SDL-Services. Ebenso wird überprüft, ob Service-Instanzen die Priorität der Service-Typen überschreiben. Die Anmerkungen in der Abbildung zählen die SDL-Komponenten auf, die in den jeweiligen Blöcken getestet werden.

Signale werden in einem *Trace* von „oben nach unten“ gesendet. Innerhalb eines *Traces* wird ein Signal in insgesamt 14 Transitionen konsumiert und weitergeleitet. Dabei bestehen die einzelnen Stationen aus drei SDL-Prozessen mit insgesamt fünf SDL-Services, und neun SDL-Prozessen ohne SDL-Services. Durch entsprechende Log-Meldungen innerhalb der Transitionen, können die Zeitpunkte, zu denen die Signale in den entsprechenden Transitionen konsumiert werden, ermittelt werden.

Alle Traces sind bis auf die Zuweisung der Prioritäten identisch. *Trace 1* erhält unter den drei *Traces* die höchste Priorität (`priority=1`), d.h. den SDL-Komponenten in den entsprechenden Blöcken wird die höchste Priorität zugewiesen. Analog erhält *Trace 2* die mittlere (`priority=2`) und *Trace 3* die niedrigste Priorität (`priority=3`). Den beiden Blöcken START und TARGET wird die systemweit höchste Priorität (`priority=0`) zugeteilt.

Die Funktionsweise des Systems ist mit einem „Wettrennen“ vergleichbar. Alle *Traces* sind zunächst inaktiv, werden aber gleichzeitig von START durch jeweils ein Signal an den obersten SDL-Block aktiviert[25]. Insgesamt werden demnach drei Signale von START versendet, die den virtuellen „Startschuss“ darstellen. Aufgrund der nicht-präemptiven Ausführungen innerhalb einer Transition ist sichergestellt, dass die drei Signale zum gleichen SDL-Zeitpunkt bei den drei *Traces* ankommen.

[25]Genaugenommen sendet nicht der SDL-Block START, sondern der Agent des (einzigen) SDL-Prozesses innerhalb des Blocks. Seit SDL-2000 ist es zwar möglich, dass SDL-Blöcke in einem expliziten Zustandsautomaten SDL-Signale erzeugen, dies wird aber von IBM Tau nicht unterstützt [Int99, IBMar]. Da es an dieser Stelle keinen Unterschied macht, wird die Terminologie etwas salopp gehandhabt.

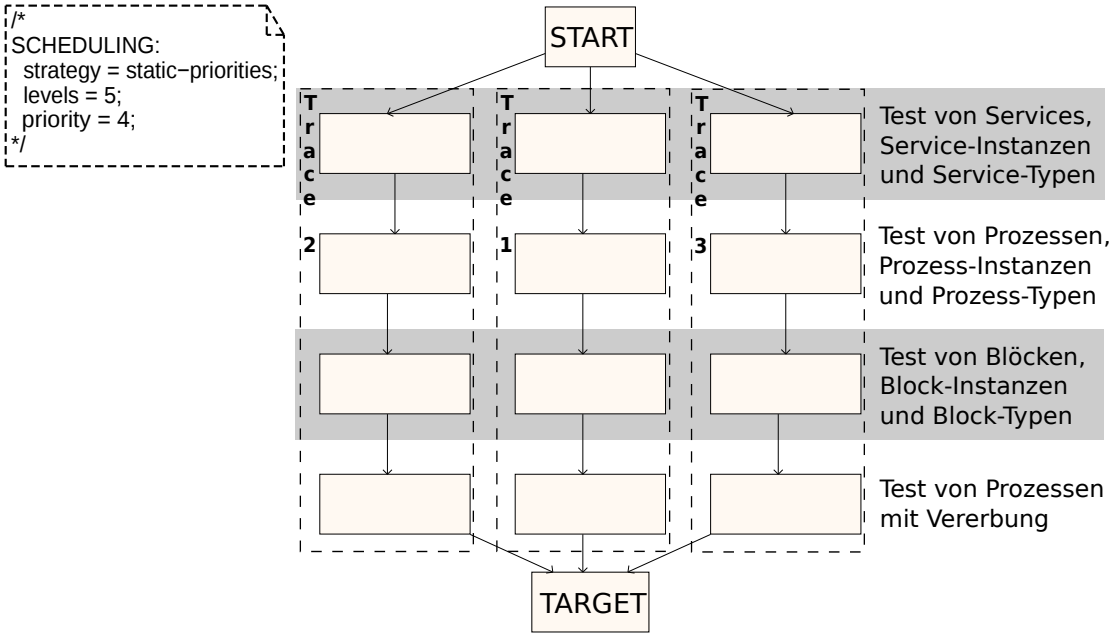


Abbildung 6.2.: Evaluiertes SDL-System. SDL-Komponenten von *Trace 1* erhalten die Priorität 1, von *Trace 2* die Priorität 2 und von *Trace 3* die Priorität 3.

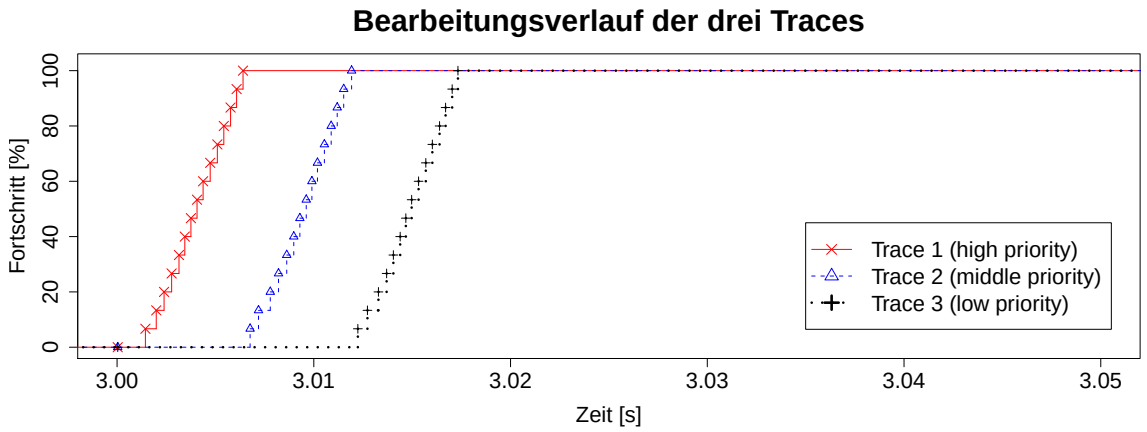
6.2.1.2 Ergebnisse

Im diesem Abschnitt wird untersucht, wie sich die unterschiedlichen Planungsverfahren auf die Weiterleitung der Signale auf den einzelnen *Traces* auswirken.

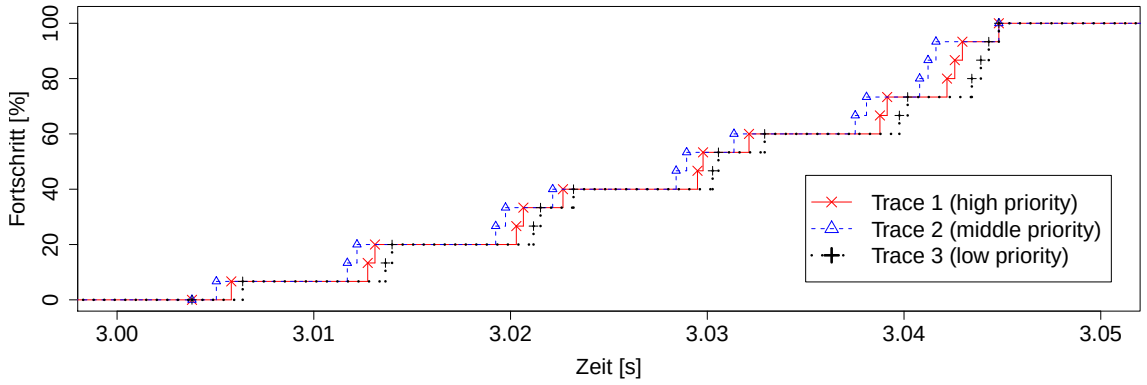
Zunächst soll die funktionale Korrektheit des neu eingeführten prioritätsbasierten Planungsverfahrens untersucht werden. Hierzu ist in Abbildung 6.3 der prozentuale Fortschritt eines „Wettrennens“ veranschaulicht, das zum Zeitpunkt $t = 3.0$ initiiert wird. Die Abbildung stellt die Zeitpunkte dar, zu denen Signale auf den drei *Traces* konsumiert werden. Abbildung 6.3a zeigt dabei den Fortschritt bei dem optimierten prioritätsbasierten Planungsverfahren, Abbildung 6.3b bei dem nicht-optimierten Planungsverfahren und Abbildung 6.3c bei dem optimierten Planungsverfahren ohne Prioritäten. Durch die Gegenüberstellung erlaubt die Abbildung außerdem einen Vergleich zwischen den drei Planungsverfahren.

Entsprechend der Vergabe der Prioritäten an SDL-Komponenten und den Zielsetzungen des prioritätsbasierten Planungsverfahrens zeigt Abbildung 6.3a das geforderte Verhalten. Signale auf den *Traces 2* und *3* werden erst bearbeitet, nachdem alle höherprioren SDL-Komponenten innerhalb von *Trace 1* das Signal bis zu dem Bestimmungsort weitergeleitet haben. Im Konkreten beendet *Trace 1* bei dem prioritätsbasierten Planungsverfahren 6.5 ms nach dem Startschuss seine Ausführung, *Trace 2* nach 11.9 ms und *Trace 3* nach 17.3 ms.

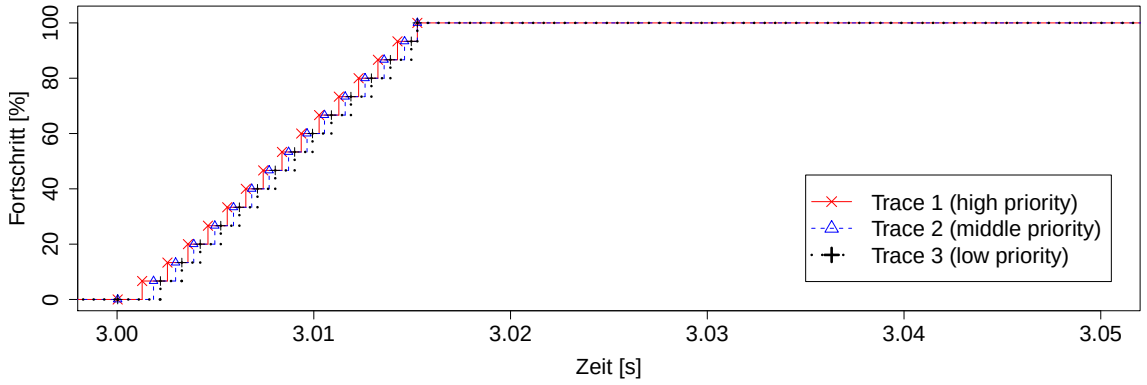
Obwohl bei dem nicht-optimierten Planungsverfahren und dem optimierten Verfahren ohne Prioritäten keine explizite Reihenfolge festgelegt wurde, ist bei beiden Verfahren ein Muster erkennbar. Beim nicht-optimierten Verfahren ist das Muster durch die Reihenfolge, in der die SDL-Agenten initialisiert werden, definiert, welche während der Ausführung zyklisch durchlaufen wird. Die Reihenfolge der Initialisierung ist keineswegs zufällig, sondern lässt sich auf die Reihenfolge, in welcher die Agenten in der SDL/PR-Datei stehen, zurückführen; ein Faktor, der von dem Entwickler nur bedingt zu erkennen und zu beeinflussen ist. Mit dem nicht-optimierten Planungsverfahren dauert die Zustellung der Signale verglichen mit den anderen Verfahren am längsten. Erst nach knapp 42 ms erreichen alle Signale „zeitgleich“ (d.h. während einer Ausfüh-



(a) Fortschritt der einzelnen Traces unter Verwendung des optimierten prioritätsbasierten Planungsverfahrens.



(b) Fortschritt der einzelnen Traces unter Verwendung des nicht-optimierten Planungsverfahrens.



(c) Fortschritt der einzelnen Traces unter Verwendung des optimierten Planungsverfahrens ohne Prioritäten.

Abbildung 6.3.: Messergebnisse zum Fortschritt der Bearbeitung auf den drei *Traces*.

rung des Zielagenten) ihren Bestimmungsort. Dass die *Traces* alle „zeitgleich“ beendet werden, liegt wiederum an der auf die Initialisierungsreihenfolge zurückzuführenden Ordnung.

Das erkennbare Muster bei dem optimierten Planungsverfahren ohne Prioritäten ist keine Folge der Reihenfolge der Initialisierung. Entscheidend ist stattdessen die Reihenfolge, in der die Agenten aktiviert werden. Dass *Trace 1* als erster *Trace* einen Fortschritt zu verzeichnen hat, liegt an der Reihenfolge, in der START die einzelnen *Traces* aktiviert. Obwohl die *Traces* zwar alle in derselben Transition aktiviert werden, geht das erste Signal an *Trace 1*, dessen erster Agent in Folge dessen als nächstes ausgeführt wird. Aufgrund des Aktivierungssignals macht *Trace 1*

nun immer zuerst den nächsten Schritt in Richtung Ziel, wobei die anderen *Traces* unmittelbar nachziehen. Wie auch bei dem nicht-optimierten Planungsverfahren erreichen alle Signale in einer Ausführung des Zielagenten ihr Ziel. Hierzu werden 15,3 ms benötigt. Das nicht-optimierte Verfahren ohne Prioritäten ist damit das effizienteste Verfahren bezüglich der Fertigstellung aller *Traces*. Allerdings werden zwei der drei *Traces* später beendet als bei dem prioritätsbasierten Planungsverfahren.

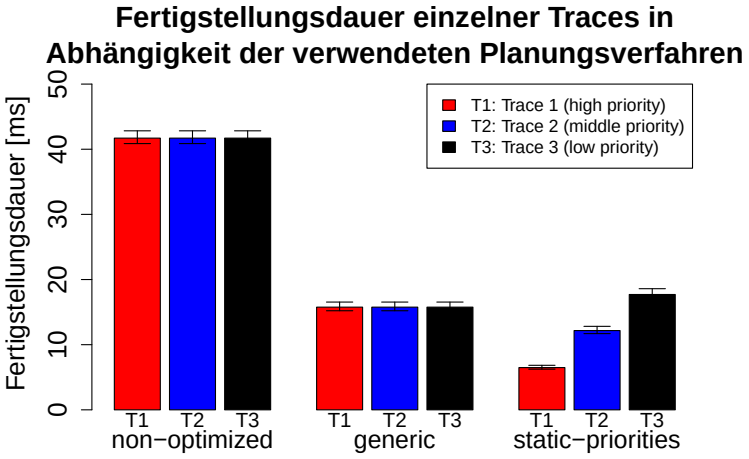


Abbildung 6.4.: Vergleich der durchschnittlichen, minimalen und maximalen Fertigstellungsdauer bei Verwendung unterschiedlicher Planungsverfahren.

Abbildung 6.3 zeigt beispielhaft nur einen möglichen zeitlichen Verlauf. Um eine umfangreichere quantitative Analyse durchzuführen, wurden die „Wettrennen“ 4000 mal wiederholt. Die Ergebnisse dieser Wiederholungen sind in Abbildung 6.4 dargestellt. Die Abbildung zeigt das arithmetische Mittel der Fertigstellungsdauer sowie deren Maximum und Minimum für alle drei Planungsverfahren. Die hohe Übereinstimmung zwischen den durchschnittlichen Fertigstellungsdauern und den Zeiten aus Abbildung 6.3 bestätigen, dass der betrachtete Ausschnitt repräsentativ den zeitlichen Verlauf wiedergibt. Der geringe Jitter (Differenz zwischen Maximum und Minimum) belegt, dass alle „Wettläufe“ nahezu identisch ablaufen. Das hinreichende, aber nicht notwendige Kriterium der disjunkten Min/Max-Intervalle bei dem prioritätsbasierten Planungsverfahren ist ein Beleg dafür, dass *Trace 1* in jedem Lauf vor *Trace 2* und *Trace 2* vor *Trace 3* fertiggestellt wurde. Dies zeigt außerdem, dass die Ergebnisse aus Abbildung 6.3 reproduzierbar und repräsentativ sind.

6.2.2 Genauigkeit von SDL-Timern

In diesem Szenario soll die Genauigkeit der Bearbeitung von SDL-Timern quantitativ untersucht werden. Anhand mehrerer Experimente wird hierbei die Verzögerung der Bearbeitung in Abhängigkeit von zusätzlicher Last innerhalb des Systems betrachtet.

6.2.2.1 Beschreibung des Szenarios

Das zur Evaluation verwendete System besteht aus zwei SDL-Blöcken: Ein Block, welcher zur Messung der Timer-Genauigkeit dient, und ein Block, welcher abhängig von einem Konfigurationsparameter Last zur Laufzeit erzeugt.

Der „Messblock“ besteht aus drei SDL-Prozessen. Die zur Laufzeit daraus entstehenden SDL-Agenten setzen periodisch einen Timer im Intervall von 100 ms. Die drei Agenten setzen dabei die Timer immer auf absolute und identische Zeitpunkte. Die Agenten der Prozesse erhalten unterschiedliche Prioritäten, um die Ausführungsreihenfolge bei dem prioritätsbasierten Planungsverfahren zu beeinflussen. Bei Konsumieren eines Timer-Signals geben die Agenten die Differenz zwischen aktueller Zeit und Stellzeitpunkt des Timers aus. Für die aktuelle Zeit wird sowohl der Wert von now als auch der tatsächlich durch die Hardware gemessene Zeitpunkt verwendet. Beide Zeiten sind nicht identisch, da now während einer Transition nicht aktualisiert wird.

Der „Lastgenerator“ besteht aus vier SDL-Prozessen und lässt sich mit einem Parameter konfigurieren, um verschiedene Lastsituationen bei der Analyse der Timer-Ungenauigkeit zu betrachten. Einer der vier Agenten, der aus den Prozessen entstanden ist, erzeugt periodisch in einem Intervall von 51 ms eine von dem Konfigurationsparameter abhängige Anzahl an Signalen, welche anschließend jeweils zweimal durch die anderen Agenten weitergeleitet werden. Da 51 und 100 relativ prim zueinander sind, wiederholt sich das Muster, d.h. der relative Abstand zwischen Zeitpunkt der Lasterzeugung und Stellwert der zu messenden Timer, erst nach 5.1 s.

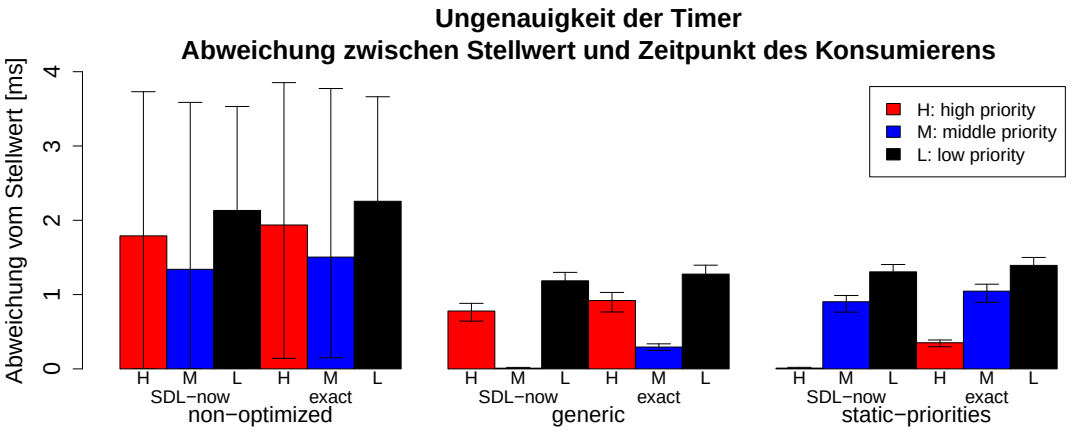
6.2.2.2 Ergebnisse

Abbildung 6.5 zeigt die Ergebnisse dieses Szenarios für drei Lastsituationen, die in jeweils einer Unterabbildung dargestellt sind. Abbildung 6.5a entstand ohne Last, d.h. die drei Agenten mit den zu messenden Timern waren die einzigen aktiven Komponenten. In Abbildung 6.5b wurden 50 Signale periodisch erzeugt, in Abbildung 6.5c waren es 100 Signale. Die Balken zeigen jeweils das arithmetische Mittel der Messergebnisse, die Linien den minimal/maximal gemessenen Wert. In allen Unterabbildungen sind die Ergebnisse für die drei Planungsverfahren dargestellt. Bei jedem Planungsverfahren ist die Abweichung vom Stellwert für alle drei Timer dargestellt. Zusätzlich zeigt die Abbildung den Unterschied bezüglich der Zeitgenauigkeit zwischen now und exakter (Hardware-)Zeit.

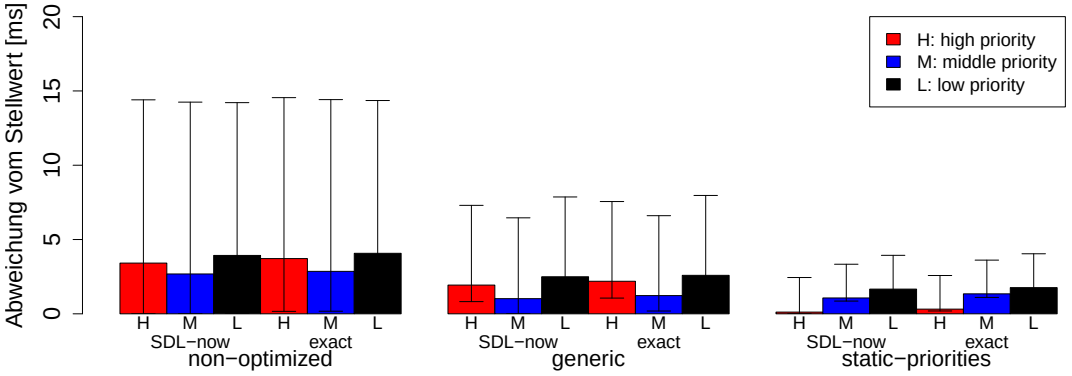
Pro Planungsverfahren/Lastsituation-Kombination liefen die Messungen über 10 Minuten. Bei einem Intervall der gemessenen Timer von 100 ms lief demnach je 6000 mal pro Agent und Messreihe ein Timer ab. Die Messergebnisse, die zu Beginn jeder Messreihe entstanden sind, wurden bei der Auswertung ignoriert, da sie aufgrund von kalten Caches und fehlenden Initialisierungen in allen Fällen deutlich höher waren als später im „eingeschwungenen“ Zustand. Diese Tatsache zeigt, dass die im Allgemeinen getätigte Unterscheidung zwischen Initialisierung und *steady state* auch bei SDL-Systemen notwendig ist.

Wenn man in Abbildung 6.5 den Blick zunächst auf einzelne Planungsverfahren richtet, fällt auf, dass bei allen Verfahren eine Rangordnung zu erkennen ist. Das prioritätsbasierte Verfahren zeigt das gewünschte Ergebnis, da der Agent des SDL-Prozesses mit der höchsten Priorität die geringste Timer-Ungenauigkeit aufweist. Bei dem optimierten Verfahren ohne Prioritäten fällt hingegen auf, dass der mittelpriore Agent unter der schlechtesten Genauigkeit leidet[26]. Dies ist ebenfalls kein Zufall, sondern lässt sich wie in Abschnitt 6.2.1 auf die Reihenfolge der Prozessdefinitionen innerhalb der SDL-Spezifikation zurückführen: Der mittelpriore Prozess steht in der SDL/PR-Datei vor den anderen beiden Prozessen, was zur Folge hat, dass er zuerst initialisiert wird und in allen folgenden Timer-Intervallen immer als erster den Timer setzt. Die SDL-Agenten der anderen beiden Prozesse setzen zwar ihre Timer auf die gleichen absoluten Zeitpunkte,

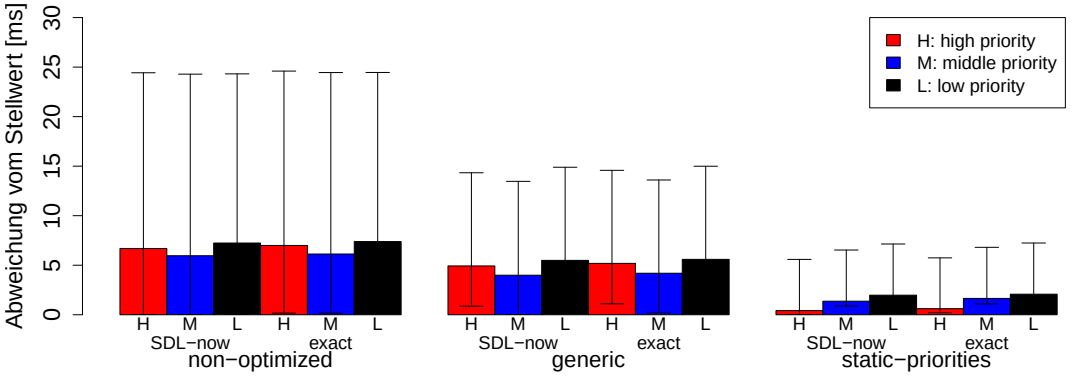
[26] Außer bei dem prioritätsbasierten Verfahren werden die zugewiesenen Prioritäten der Agenten ignoriert. Dennoch wird die Priorität bei allen Planungsverfahren zur Unterscheidung der Agenten verwendet.



(a) System ohne Last.



(b) System mit mittlerer Last.



(c) System mit hoher Last.

Abbildung 6.5.: Messergebnisse zum verzögerten Empfang von Timer-Signalen. Die Ergebnisse zeigen die Differenz zwischen Stellwert eines Timers und Bearbeitungszeitpunkt. Hinweis: Die Skalierung der x-Achse unterscheidet sich zwischen den einzelnen Abbildungen.

allerdings werden diese erst später gesetzt, sodass sie in der Warteschlange hinter dem Agent des mittelprioren Prozesses eingereiht werden.

In Abbildung 6.5a, in der die Timer nicht durch andere Last behindert werden, wird der Vorteil der beiden optimierten Planungsverfahren deutlich, da diese Verfahren nur die Agenten berücksichtigen, die ausführbar sind. Dadurch ist sowohl die durchschnittliche Ungenauigkeit als auch der Jitter deutlich geringer als bei dem nicht-optimierten Verfahren. Ohne auf die Prioritäten der

Agenten zu achten, schneidet das optimierte Verfahren ohne Prioritäten besser ab als das prioritätsbasierte Verfahren, da die Wartezeiten im Schnitt minimal geringer sind. Dieser Umstand ist vermutlich auf den etwas höheren Overhead zurückzuführen, der durch die Verwaltung mehrerer Prioritäts-Queues hinzukommt.

Bei mittlerem Lastaufkommen in Abbildung 6.5b werden die Ungenauigkeiten im Allgemeinen größer (Achtung: Andere Skalierung der y-Achse). Allerdings verschlechtern sich die Ungenauigkeiten bei dem prioritätsbasierten Planungsverfahren deutlich weniger als bei den anderen beiden Verfahren. Der Grund hierfür ist in der Optimierung `EXECUTE_CONTINUOUS` zu finden. Bei dem nicht-optimierten Verfahren und dem optimierten Verfahren ohne Prioritäten sorgt diese Optimierung dafür, dass ein Agent, der Last generiert, in einer Ausführung immer in diesem Fall 50 Signale in ebenfalls 50 Transitionen ohne Unterbrechung verarbeitet. Das prioritätsbasierte Verfahren unterbricht hingegen die Ausführung, sobald ein höherpriorer Agent zwischenzeitlich bereit wurde, indem zum Beispiel ein zu messender Timer abläuft. Dass die Ungenauigkeit bei dem prioritätsbasierten Verfahren dennoch steigt, liegt daran, dass Unterbrechungen nach nur einer Transition möglich sind. Ebenfalls auffallend in der Abbildung ist der stark gestiegene Jitter bei allen Planungsverfahren. Ursache hierfür ist der gestiegene Unterschied zwischen *best* und *worst case*: Der *best case* liegt immer noch dann vor, wenn in dem Moment, in dem die zu messenden Timer ablaufen, keine Last vorhanden ist. Bei dem *worst case* hat die Erzeugung von Last gerade begonnen und die Abarbeitung der Timer verzögert sich, bis alle ausführbaren Agenten die Last abgearbeitet haben. Da die Bearbeitungsdauer der Last stieg, ist demnach auch die *worst case*-Ungenauigkeit angestiegen.

Die Ergebnisse bei höchstem Lastaufkommen in Abbildung 6.5c bestätigen den Trend (Achtung: Andere Skalierung der y-Achse): Sowohl die Ungenauigkeiten als auch der Jitter steigen im Allgemeinen, aber bei dem prioritätsbasierten Verfahren ist die Verschlechterung wieder am geringsten.

Zusammenfassend stellt Abbildung 6.6 die drei gemessenen Timer mit der für Echtzeitsysteme interessanten maximalen Ungenauigkeit nebeneinander. Pro Abbildung werden jeweils alle Planungsverfahren verglichen. Bei allen drei Timern ist mit steigender Last ein linearer Anstieg der Ungenauigkeit zu erkennen, wobei die geringere Steigung bei dem prioritätsbasierten Planungsverfahren die Erkenntnisse aus den vorherigen Betrachtungen bestätigt. In absoluten Zahlen ausgedrückt, beträgt die maximale Ungenauigkeit des höchstprioren Agenten bei maximaler Last 5,7 ms. Damit ist sie weniger als halb so groß wie bei Verwendung des optimierten Planungsverfahrens ohne Prioritäten (14,6 ms) und weniger als ein Viertel so groß wie bei dem nicht-optimierten Verfahren (24.6 ms). An dieser Stelle sei angemerkt, dass die Ungenauigkeit bei dem prioritätsbasierten Verfahren nur steigt, weil bei steigender Last die Ausführungsdauer einer Transition steigt und die Ausführung einer Transition nicht unterbrochen werden kann. Würde hingegen nur die Anzahl an ausgeführten Transitionen steigen und nicht deren individuelle Ausführungszeit, wäre der Anstieg der Ungenauigkeit bei dem prioritätsbasierten Verfahren deutlich geringer. Diese Tatsache bestätigt erstens die Entwurfsrichtlinie, zeitaufwendige Transitionen in kleinere Transitionen aufzusplitten (siehe [BCG09]), und zeigt zweitens einen Anwendungsfall für das temporäre Blockieren der Ausführung von niederprioren Agenten, welches in Abschnitt 4.2.5.4 vorgestellt wurde.

6.2.2.3 Bessere Ergebnisse ohne Optimierung?

Zum Abschluss dieses Szenarios soll verdeutlicht werden, dass sich die Ergebnisse nicht nur bei unterschiedlichem Lastaufkommen unterscheiden, sondern ebenfalls von der Verteilung der

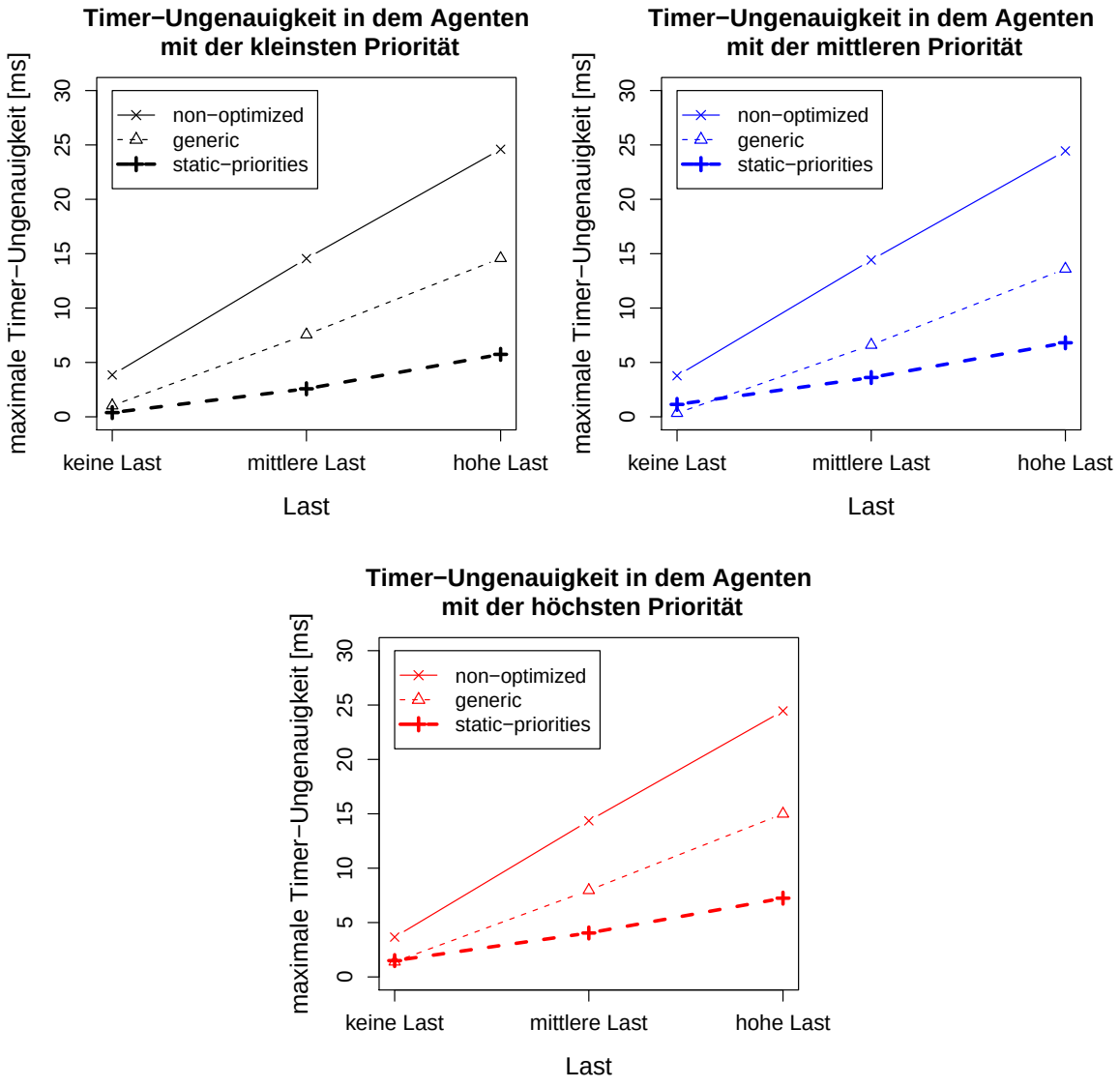
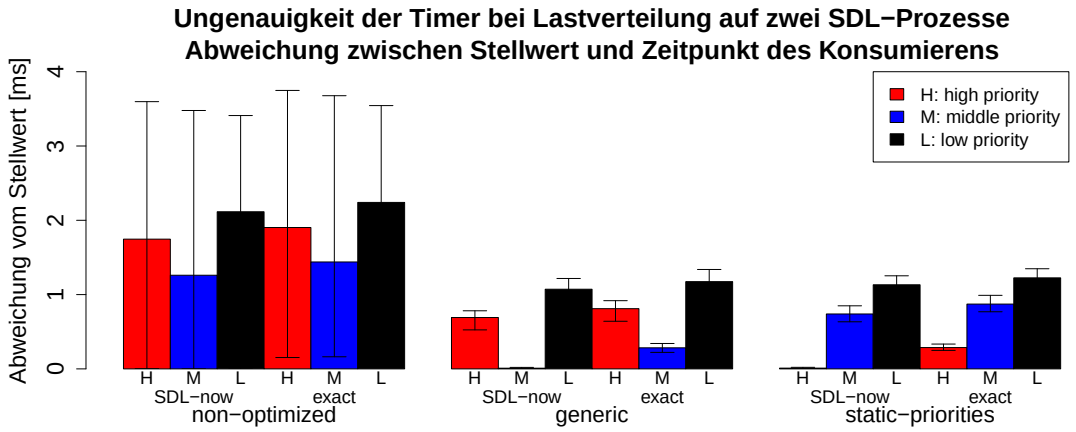


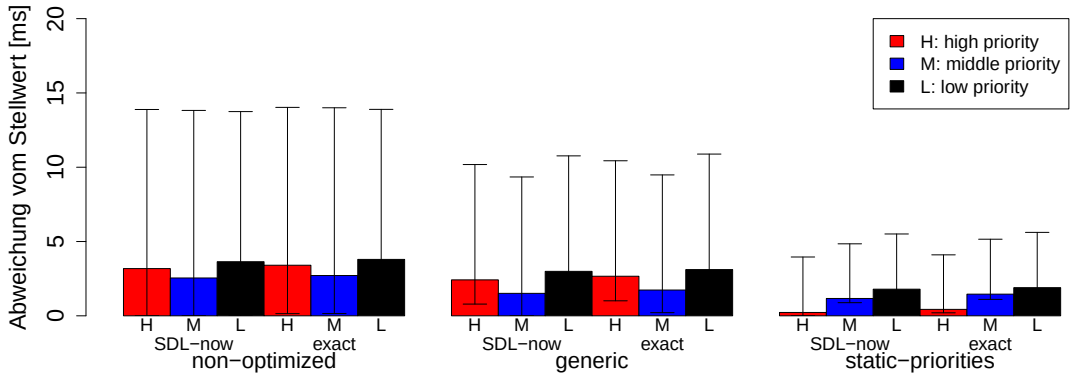
Abbildung 6.6.: Messergebnisse zur verzögerten Timer-Bearbeitung. Die Abbildungen zeigen die maximale Ungenauigkeit in Abhängigkeit der Planungsverfahren.

Last auf verschiedene Agenten abhängen. Dies betrifft zwar nur in sehr geringem Maße das optimierte Planungsverfahren mit Prioritäten, kann aber bei dem nicht-optimierten Verfahren und dem optimierten Verfahren ohne Prioritäten zu unerwarteten Ergebnissen führen: Verteilt sich die Last auf viele Agenten, verschlechtern sich in der Regel die Genauigkeiten der Timer bei dem nicht-optimierten Planungsverfahren. Wird allerdings die gleiche Menge an Last von beispielsweise nur zwei Agenten erzeugt, kann das nicht-optimierte Verfahren sogar bessere Genauigkeiten bezüglich des Empfangs eines Timer-Signals liefern als das optimierte Verfahren ohne Prioritäten.

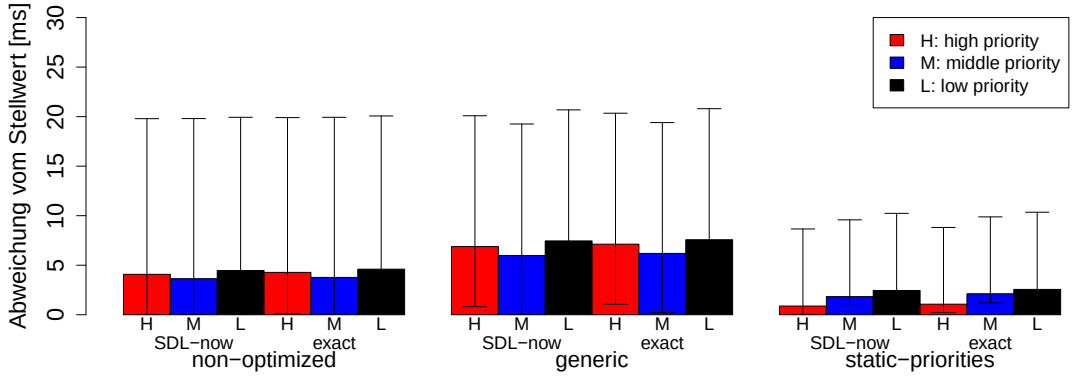
Entsprechende Ergebnisse, die dieses Phänomen zeigen, sind in Abbildung 6.7 dargestellt. Die Ergebnisse entstanden aus einem ähnlichen SDL-System mit der gleichen Menge an Last wie in Abbildung 6.5. Einzige Unterschiede sind, dass die Signale zur Erzeugung von Last nun nicht zwischen vier Agenten, sondern ausschließlich zwischen zwei Agenten gesendet werden, und dass bei mittlerem Lastaufkommen 75 Signale anstelle von 50 Signalen und bei maximaler Last 150 Signale anstelle von 100 Signalen erzeugt werden. Die Gesamtsumme an Signalen pro 51 ms-



(a) System ohne Last.



(b) System mit mittlerer Last.



(c) System mit hoher Last.

Abbildung 6.7.: Messergebnisse zum verzögerten Empfang von Timer-Signalen. In diesem Fall verteilt sich die Last auf weniger Agenten. Hinweis: Die Skalierung der x-Achse unterscheidet sich zwischen den einzelnen Abbildungen.

Intervall bleibt demnach bei jedem Lastaufkommen unverändert (150 bzw. 300 Signale), nur die Aufteilung auf Agenten wechselt.

Abbildung 6.7a zeigt zunächst, dass sich im Fall ohne Last wenig verändert. Einzig bei dem nicht-optimierten Verfahren reduzieren sich die Maximalwerte etwas, da nun weniger Agenten ausgeführt werden. Interessant wird es bei dem mittleren und hohen Lastaufkommen in Abbildung 6.7b bzw. Abbildung 6.7c, denn im Vergleich zu Abbildung 6.5 steigen die Ungenau-

igkeiten bei den beiden optimierten Verfahren leicht, fallen bei dem nicht-optimierten Verfahren allerdings etwas. Dass die Ungenauigkeit bei beiden optimierten Verfahren steigt, ist nicht weiter verwunderlich, da die Ausführungsdauer eines lasterzeugenden Agenten nun aufgrund der gestiegenen Signalanzahl ebenfalls ansteigt. Ebenfalls verwundert nicht die Tatsache, dass sich die Timer-Ungenauigkeit mit dem prioritätsbasierten Verfahren weniger durch die Art der Lastverteilung beeinflussen lässt als das optimierte Verfahren ohne Prioritäten, da bei dem prioritätsbasierten Verfahre ein niederpriorer Agent nach Abarbeitung einer Transition unterbrochen werden kann, was auf die veränderte Bedeutung von `EXECUTE_CONTINUOUS` zurückzuführen ist (siehe Abschnitt 4.2.3). Dass die Ungenauigkeit dennoch leicht ansteigt, liegt an der fehlenden Präemptionsmöglichkeit innerhalb einer Transition.

Seltsam in Abbildung 6.7c erscheint allerdings, dass das nicht-optimierte Verfahren geringere Ungenauigkeiten aufweist als das optimierte Verfahren ohne Prioritäten. Der Grund für dieses Phänomen ist etwas schwieriger zu finden, lässt sich aber dadurch erklären, dass bei dem nicht-optimierten Planungsverfahren ein Agent nie zweimal vor einem anderen Agenten ausgeführt wird, da die Menge der Agenten immer zyklisch durchlaufen wird. Bei den optimierten Verfahren kann dies jedoch passieren und sowohl die Durchschnitts- als auch die *worst case*-Verzögerung verschlechtern.

Allerdings verschweigt die Grafik, dass die optimierten Verfahren dennoch effizienter sind als das nicht-optimierte, d.h. die Gesamtdauer, die für die Bearbeitung der Last und der gemessenen Timer benötigt wird, ist bei dem nicht-optimierten Verfahren im Regelfall höher. Außerdem taucht das gezeigte Phänomen normalerweise nur in kleinen (Test-)Systemen auf. In größeren Systemen ist das nicht-optimierte Planungsverfahren üblicherweise in allen Belangen schlechter, da die Anzahl an „unnötig“ berücksichtigten Agenten steigt. Dennoch verdeutlicht dieser Umstand, dass der zeitliche Ablauf in komplexen Systemen mit chaotischen Lastsituationen unvorhersehbar werden kann. Mit den statischen Prioritäten wurde ein erster Schritt getan, die zeitliche Ausführungsreihenfolge von der Initialisierung oder Aktivierung von Agenten zu entkoppeln und die Transparenz des Laufzeitverhaltens zu erhöhen.

6.2.3 Fazit

Die Evaluation des prioritätsbasierten Planungsverfahrens zeigt bereits das große Potential, das durch eine sinnvolle Vergabe der Prioritäten Vorteile hinsichtlich des Laufzeitverhaltens erwarten lässt. Der bisherige Stand der Laufzeitumgebung leidet aber insbesondere unter der fehlenden Präemption, die einer sofortigen Ausführung höchstpriorer Agenten im Wege steht. Mit den Möglichkeiten zur Blockierung von niederprioren Agenten (siehe Abschnitt 4.2.5.4) existiert jedoch eine Lösung, um in zeitkritischen Abschnitten die fehlende Präemption zu kompensieren und die Ausführungsverzögerung weiter zu verringern.

6.3 Black Burst Synchronization

Dieser Abschnitt beschreibt die funktionale und quantitative Evaluation der Implementierung von BBS mit dem Ziel, die Korrektheit der Schnittstelle zwischen SDL-System und BBS-Treiber zu validieren und die Güte der Synchronisationsgenauigkeit zu analysieren. Im Gegensatz zu den vorherigen Evaluationen basieren die folgenden Experimente auf verteilten Systemen, in denen die Knoten drahtlos über den CC2420-Transceiver miteinander kommunizieren.

6.3.1 BBS als Grundlage für TDMA

In einem ersten Szenario wird eine Single-Hop-Topologie aufgebaut, in welcher die Synchronisation von BBS als Grundlage für TDMA dienen soll. Um die Ergebnisse qualitativ einschätzen zu können, werden die Experimente mit einer beacon-basierten Synchronisation in der gleichen Topologie verglichen. Für die beacon-basierte Synchronisation werden reguläre MAC-Rahmen verwendet, die ein designierter Master-Knoten zur Synchronisation aller Knoten sendet.

6.3.1.1 Beschreibung des Szenarios

Sowohl bei der Verwendung von BBS als auch bei der Verwendung der beacon-basierten Synchronisation dient die Synchronisation der Unterteilung des Mediums in Micro-Slots. Hierauf aufbauend bekommen die Sendeknoten in periodischen Abständen Sende-Slots zugewiesen. Die Sendeknoten senden in ihren zugewiesenen Sende-Slots reguläre Datenrahmen an einen Empfängerknoten. Für die Evaluation wird untersucht, wie konstant das Empfangsintervall bei dem Empfängerknoten ist. Hierdurch wird in indirekter Weise die Güte der Synchronisation analysiert: Ist die Synchronisation sehr genau, dann ist das Empfangsintervall konstant und entspricht dem Intervall der zugeteilten Sende-Slots. Ist die Synchronisation hingegen ungenau, dann „treffen“ die Sendeknoten aus Sicht des Empfängers ihre Sende-Slots nicht exakt und der Abstand zwischen dem Empfang von Rahmen unterschiedlicher Sender weicht von dem vorgesehenen Abstand der Sende-Slots ab. In dem ersten Szenario zur Evaluation von BBS wurden insgesamt acht Experimente durchgeführt, die sich in dem verwendeten Synchronisationsverfahren, den Übertragungsmodi für Datenrahmen und der Lastsituation auf den Sendeknoten unterschieden.

Topologie

Der Aufbau der Experimente erfolgte auf einem Waldparkplatz in etwa 1 km Entfernung zur nächsten Ortschaft, um den Einfluss von Störungen durch interferierende Netzwerke (IEEE 802.11 WLAN [IEE99], IEEE 802.15.1 Bluetooth[HE00], ...) zu minimieren und die Reproduzierbarkeit der Ergebnisse zu verbessern. Die Topologie ist in Abbildung 6.8 sowohl für die Synchronisation mit BBS als auch für die beacon-basierte Synchronisation skizziert. Sie besteht aus drei Knoten: Zwei Sendeknoten (Knoten 0 und Knoten 2), die in periodischen Abständen Datenrahmen senden, und einem Empfangsknoten (Knoten 1), der die Zeitpunkte der empfangenen Datenrahmen für eine spätere Auswertung aufzeichnet. Knoten 0 übernimmt für die Synchronisation die Rolle des Masters, d.h. er initiiert die Übertragung von Tick-Rahmen bzw. ist bei der beacon-basierten Synchronisation für die Übertragung von Beacons verantwortlich. Für die Experimente waren also zunächst alle Knoten in Sendereichweite voneinander, da aufgrund des Aufwands darauf verzichtet wurde, eine beacon-basierte Multi-Hop-Synchronisation umzusetzen, wie es zum Beispiel durch *Reference Broadcast Synchronization* (RBS) unterstützt wird [EGE02]. Zur besseren Vergleichbarkeit wurde daher auch BBS in der gleichen Topologie evaluiert. Die Multi-Hop-Fähigkeit von BBS ist Thema in Abschnitt 6.3.2.

Faktoren

Neben der bereits erwähnten Unterscheidung der Synchronisationsprotokolle wurden weitere Faktoren in der Evaluation betrachtet und analysiert. Einer dieser Faktoren ist der Übertragungsmodus von regulären Datenrahmen, bei dem zwischen *SEND* und *SEND_AT* unterschieden wird. *SEND* bezeichnet eine reguläre Übertragung, die direkt aus dem SDL-System her-

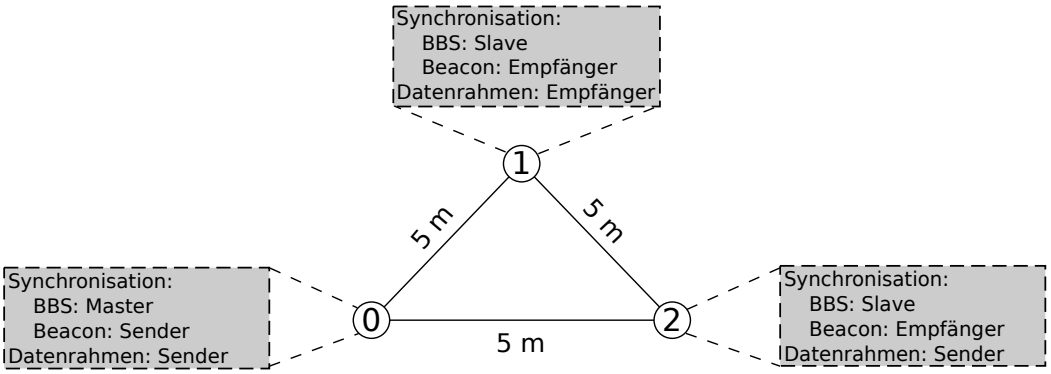


Abbildung 6.8.: Topologie zur Evaluation des BBS-Treibers anhand von TDMA.

aus angestoßen wird. *SEND_AT* ist eine Funktion, die es ermöglicht, einen Datenrahmen exakt zu einem vorgegeben Zeitpunkt ohne Verzögerung durch Laufzeiten der SDL-Signale mit dem CC2420-Transceiver zu versenden. Der Rahmen muss hierzu a priori an das SDL-Environment geschickt werden, welches anschließend einen Hardware-Timer setzt, den Rahmen bis zu dem spezifizierten Zeitpunkt zwischenspeichert und bei Ablauf des Hardware-Timers versendet. Der Begriff „Funktion“ ist etwas irreführend, da *SEND_AT* eigentlich ein SDL-Signal ist, welches in *SEnF* zu dem beschriebenen Resultat führt.

An dieser Stelle sei angemerkt, dass für die Beacons, die bei der beacon-basierten Synchronisation von dem Master-Knoten zur Synchronisation versendet wurden, kein *SEND_AT* verwendet wurde. Ebenso wurde auf das frühe Zeitstempeln bei dem Empfang von Beacons verzichtet (siehe Abschnitt 4.1). Mit diesen beiden Maßnahmen wäre eine deutlich genauere Synchronisation mit Beacons zu erreichen. Allerdings soll der Vergleich zwischen BBS und der beacon-basierten Synchronisation zeigen, wie sich die Realisierung von BBS im Vergleich zu einem Synchronisationsprotokoll schlägt, welches ausschließlich auf Mittel zurückgreift, die SDL „von Haus aus“ mitbringt. Ein frühes Zeitstempeln und die *SEND_AT*-Funktion zählen zu SDL erweiternden Implementierungsdetails, die den Eigenschaften realer Hardware geschuldet sind, um das Laufzeitverhalten zu verbessern.

Als weiterer Faktor wurde der Einfluss von Last untersucht. Hierzu wurde, ähnlich wie bei der Evaluation der Genauigkeit von SDL-Timern, eine unabhängige Last-Komponente zur Erzeugung einer zufälligen Last in das System aufgenommen (siehe Abbildung 6.9).

Zusammenfassend folgt eine Auflistung der drei Faktoren mit ihren möglichen Werten:

1. Synchronisation: BBS oder beacon-basiert.
2. Rahmenversand: Mit oder ohne *SEND_AT*.
3. Last: Mit oder ohne zusätzliche Last innerhalb des Systems.

Insgesamt führten alle Kombinationen zu acht unterschiedlichen Experimenten.

Protokollverhalten

Abgesehen von den unterschiedlichen Synchronisationsverfahren war das Protokollverhalten bei der Verwendung von BBS und der Verwendung von Beacons identisch. Abbildung 6.10 zeigt den Aufbau des Macro-Slots, in dem die Sende-Slots der beiden Sendeknoten gekennzeichnet sind. Ein Macro-Slot besaß eine Länge von $1\,s$ und war in Micro-Slots mit einer Länge von je 5 ms unterteilt. Knoten 0 begann 100 ms nach Ende jeder Synchronisationsphase für eine Dauer von 800 ms mit der Übertragung von Datenrahmen im Intervall von 10 ms. Knoten 2 folgte

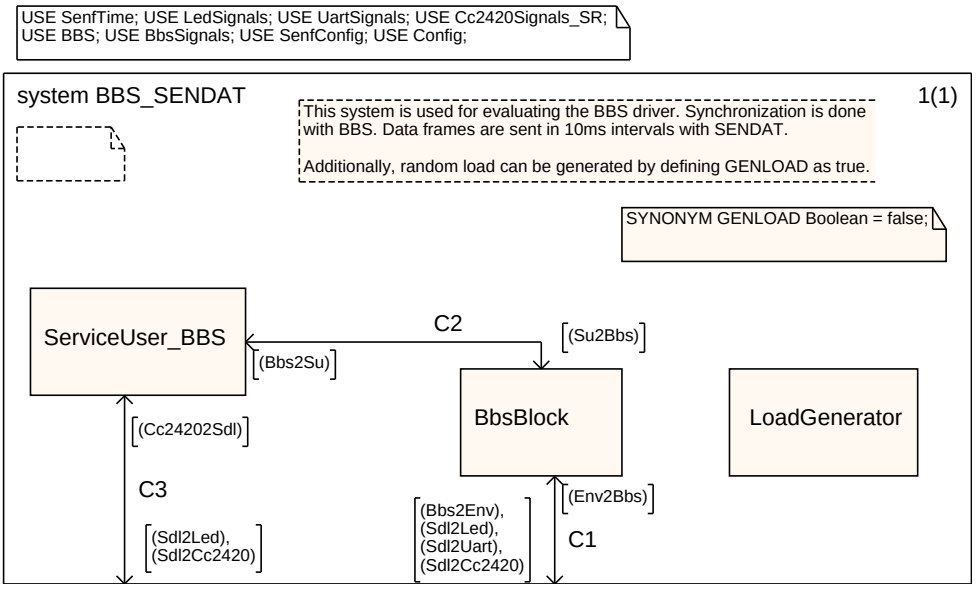


Abbildung 6.9.: Evaluiertes SDL-System mit BBS. BbsBlock beinhaltet den SDL-Treiber von BBS und propagiert die Ticks aus dem *SEnF*-Treiber an den Dienstnutzer. ServiceUser_BBS sendet Datenrahmen mit Hilfe der *SEND_AT*-Funktion. Last wird in LoadGenerator erzeugt, falls das Synonym GENLOAD gleich `true` ist.

dem Beispiel 105 ms nach Ende jeder Synchronisationsphase. Jeder gesendete Rahmen hatte eine Größe von 18 Bytes auf SDL-Ebene, hinzu kamen weitere 8 Bytes durch Header/Trailer. Bestandteil jedes Rahmens war eine Sequenznummer, deren Zählung in jedem Macro-Slot von 0 an begann und das Erkennen von Rahmenverlusten ermöglichte. Der Empfangsknoten, d.h. Knoten 1, maß den zeitlichen Abstand zwischen dem Empfang von Rahmen identischer Sequenznummer, die von den Knoten 0 und 2 gesendet wurden. Wenn $t_{0,seqId}$ der Empfangszeitpunkt eines von Knoten 0 gesendeten Rahmens mit der Sequenznummer $seqId$ ist und $t_{2,seqId}$ der ent-

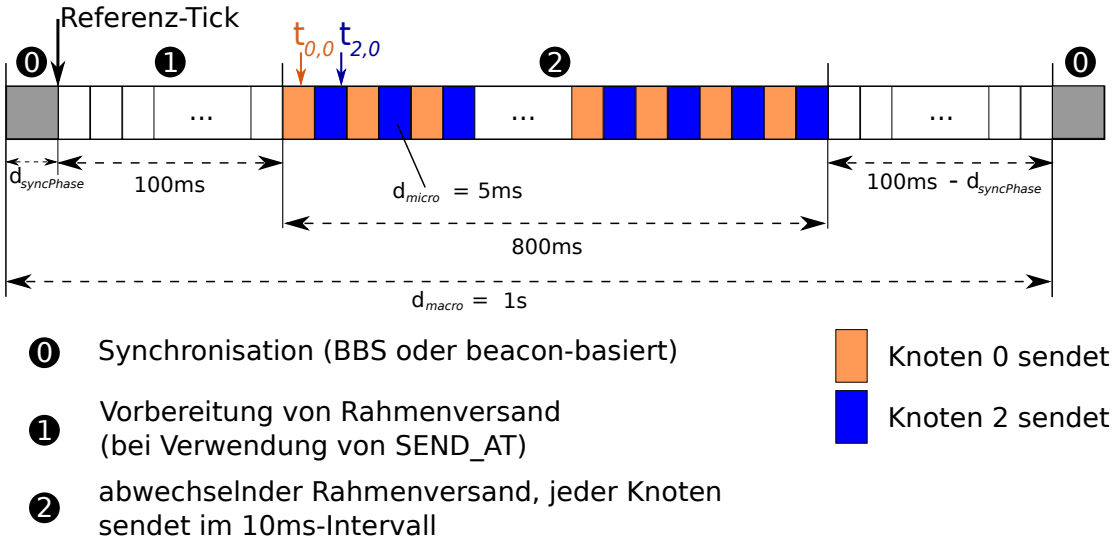


Abbildung 6.10.: Der Macro-Slot zeigt den zeitlichen Ablauf des Testprotokolls.

sprechende Empfangszeitpunkt des von Knoten 2 gesendeten Rahmens (siehe Abbildung 6.10), dann werden formal folgende Intervalle pro Macro-Slot berechnet:

$$d_{seqId} = t_{2,seqId} - t_{0,seqId} \quad , \quad 0 \leq seqId \leq 79 \tag{6.1}$$

Diese Intervalle berechnen sich im Idealfall zu 5 ms. Da jeder Knoten 80 Datenrahmen pro Macro-Slot sendete, empfing der Empfangsknoten im Idealfall 160 Rahmen pro Macro-Slot. Entsprechend konnten maximal 80 Empfangsintervalle pro Macro-Slot bestimmt werden. Jedes der acht Experimente lief für eine Dauer von 30 Minuten. Demnach gingen 1800 Macro-Slots und maximal 144000 Empfangsintervalle in die Ergebnisse eines Experiments ein.

Im Gegensatz zu Beacons wurden Datenrahmen auf dem Empfangsknoten direkt im Interrupt-Handler des CC2420-Treibers zeitgestempelt. Dies geschah bei beiden Synchronisationsverfahren. Die Messdaten, die auf diese Weise gewonnen wurden, sind dadurch unabhängig von dem Zeitpunkt, zu welchem der Empfangsknoten den Datenrahmen tatsächlich bearbeitete. Alle Abweichungen von dem konfigurierten Sendeintervall lassen sich somit nahezu vollständig auf Ungenauigkeiten bei dem Sender zurückführen. Es bleibt allerdings immer noch ein kleiner Unsicherheitsfaktor, da der Interrupt, der zum Zeitstempeln des Rahmenempfangs auf dem Empfängerknoten führt, durch andere Interrupts verzögert auftreten kann. Die verwendete Messmethodik eignet sich daher nur bedingt zum Nachweis der deterministischen Synchronisationsgenauigkeit von BBS.

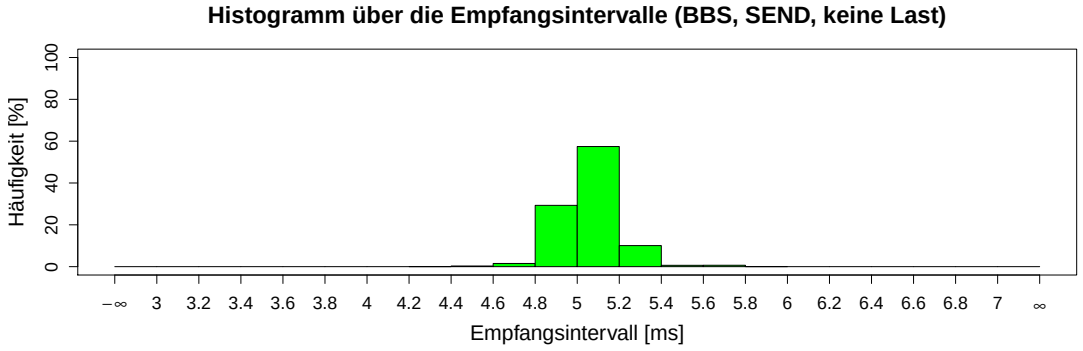
6.3.1.2 Ergebnisse

Dieser Abschnitt beschreibt die Auswertung der von Knoten 1 genommenen Messdaten.

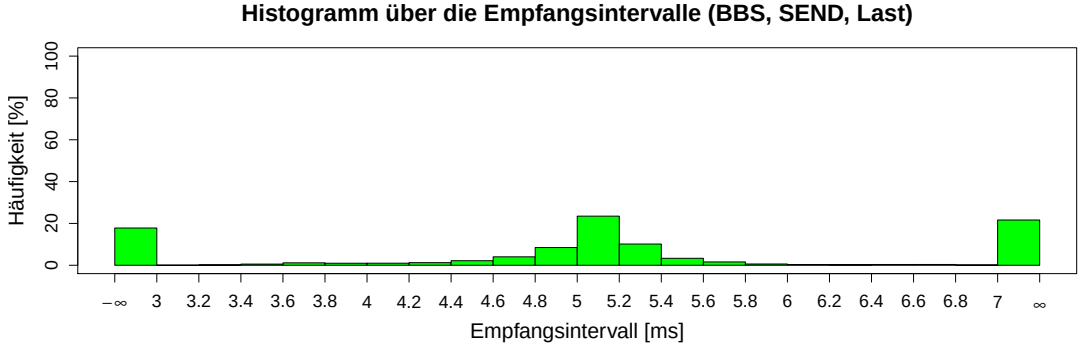
Empfangsintervall

Zunächst wird das oben beschriebene Empfangsintervall d_{seqId} näher analysiert. Hierzu zeigen die Abbildungen 6.11 und 6.12 Histogramme über die Häufigkeit, mit denen einzelne Empfangsintervalle von Knoten 0 gemessen wurden. Abbildung 6.11 zeigt die Ergebnisse, bei denen BBS zur Synchronisation verwendet wurde, Abbildung 6.12 zeigt entsprechende Ergebnisse bei beacon-basierter Synchronisation. Im Intervall [3 ms,7 ms] zeigen die Histogramme die Häufigkeit der Empfangsintervalle mit einer Intervallgröße von 200 μs. Unter 3 ms und oberhalb von 7 ms sind die restlichen Empfangsintervalle zusammengefasst. Dies ist damit begründet, dass in der Regel Abweichungen von mehr als 2 ms von einem vorgesehenen und reservierten Empfangszeitpunkt nicht hinnehmbar sind und damit auch nicht analysiert werden müssen.

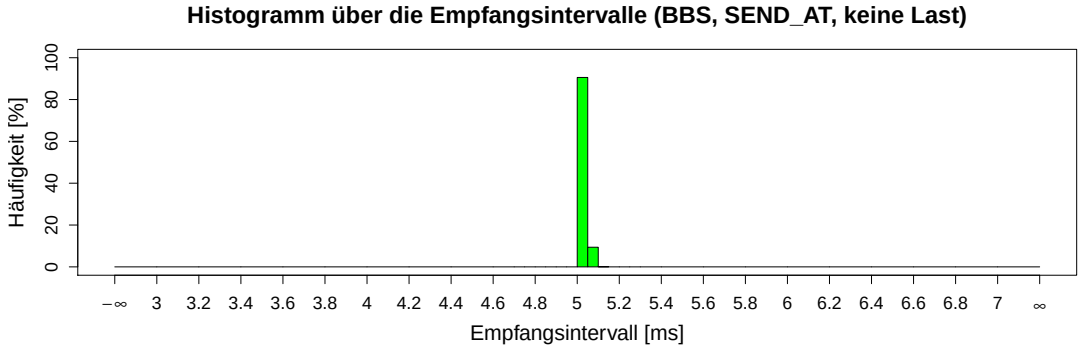
Ohne zunächst auf die unterschiedlichen Übertragungsmodi und Lastverhältnisse zu schauen, fällt bei dem Vergleich zwischen der beacon-basierten Synchronisation und BBS auf, dass die Empfangsintervalle mit BBS häufiger und näher an den nominellen 5 ms liegen. Besonders gut zu erkennen ist dies bei einem Vergleich der Abbildungen 6.11c und 6.12c. Dies ist dem Umstand geschuldet, dass die beacon-basierte Synchronisation vollständig in SDL umgesetzt wurde und Verzögerungen durch Bearbeitungszeiten und Schwankungen der Synchronisationsgenauigkeit den Sendeintervallen und damit auch den Empfangsintervallen stark schadeten. Durch ein frühes Zeitstempeln von Beacons direkt in dem Interrupt-Handler könnten diese Ungenauigkeiten deutlich reduziert werden. Bei BBS hingegen spielen Verzögerungen in SDL für die Synchronisationsgenauigkeit aufgrund der hardwarenahen Implementierung des zeitkritischen Protokolls keine Rolle.



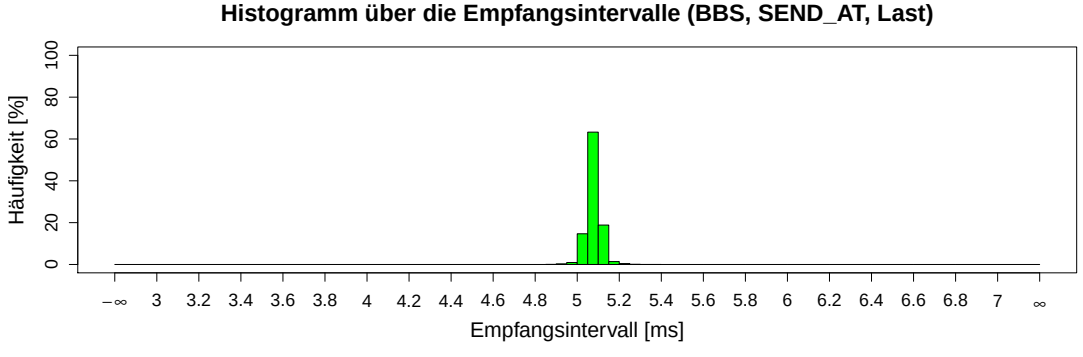
(a) Synchronisation mit BBS, Rahmenversand mit SEND, ohne Last.



(b) Synchronisation mit BBS, Rahmenversand mit SEND, mit Last.

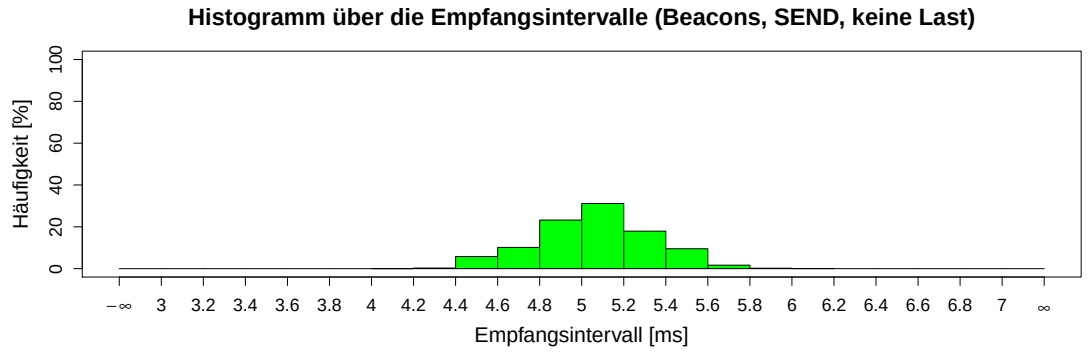


(c) Synchronisation mit BBS, Rahmenversand mit SEND_AT, ohne Last.

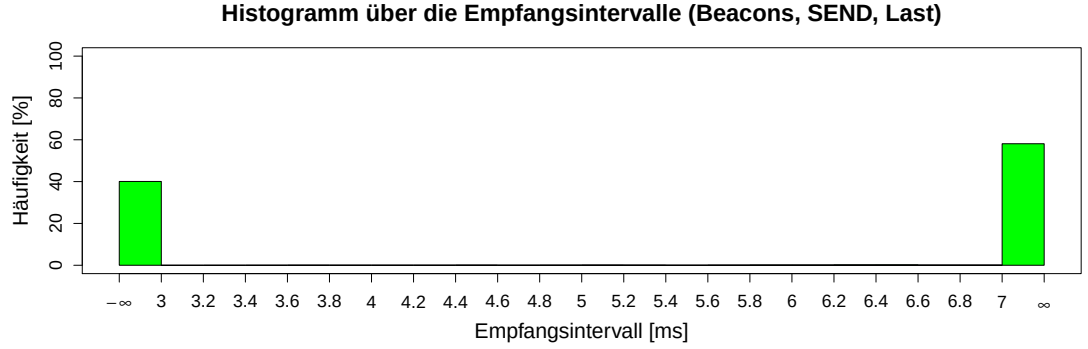


(d) Synchronisation mit BBS, Rahmenversand mit SEND_AT, mit Last.

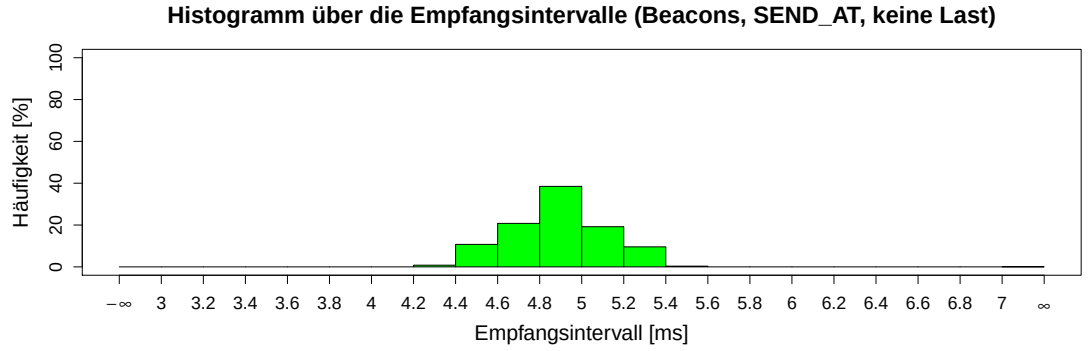
Abbildung 6.11.: Empfangsintervall in Abhängigkeit der Übertragungsmodi von Datenrahmen und Last bei Synchronisation BBS.



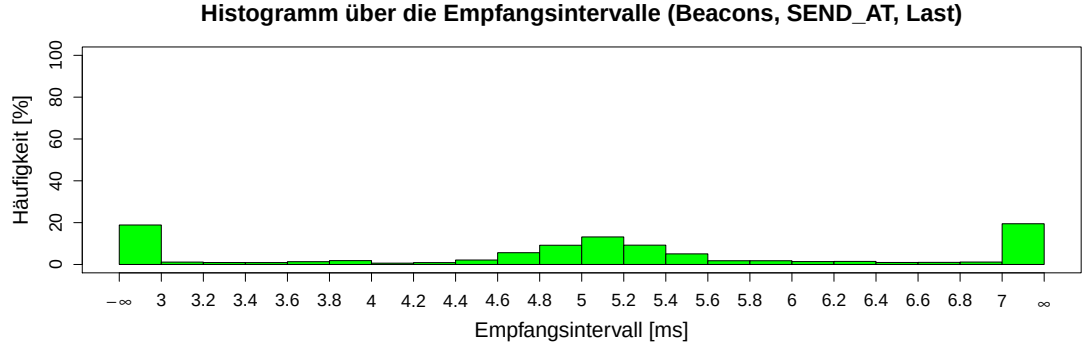
(a) Beacon-basierte Synchronisation, Rahmenversand mit SEND, ohne Last.



(b) Beacon-basierte Synchronisation, Rahmenversand mit SEND, mit Last.



(c) Beacon-basierte Synchronisation, Rahmenversand mit SEND_AT, ohne Last.



(d) Beacon-basierte Synchronisation, Rahmenversand mit SEND_AT, mit Last.

Abbildung 6.12.: Empfangsintervall in Abhängigkeit der Übertragungsmodi von Datenrahmen und Last bei Synchronisation mit Beacons.

Als nächstes wird der Einfluss der Übertragungsmodi näher betrachtet. Entsprechend den Erwartungen sind die Empfangsintervalle bei Verwendung der *SEND_AT*-Funktion näher an den geforderten 5 ms-Intervallen (vgl. Abbildungen 6.11a und 6.11c). Die Begründung liegt in dem Sendeintervall beider Knoten: Wird *SEND_AT* verwendet, entspricht das Sendeintervall beider Knoten exakt den spezifizierten 10 ms. D.h. wenn die beiden Sendeknoten mit einer hohen Genauigkeit synchronisiert sind, reicht dies bereits aus, damit die Empfangsintervalle nahe den gewünschten 5 ms liegen. Wird hingegen *SEND* verwendet, d.h. die Übertragung wird direkt aus dem SDL-System heraus gesteuert, leiden die Sendeintervalle der Knoten unter Schwankungen aufgrund der Bearbeitungszeiten der Sendeaufträge innerhalb des Systems.

Die erzeugte Last hat den größten Einfluss auf die Empfangsintervalle der Rahmen. Bei BBS ist dies deutlich in Abbildung 6.11b zu erkennen. Hier liegen die Empfangsintervalle in etwa 40% der Fälle außerhalb des Intervalls [3 ms,7 ms]; ein Ergebnis, das ohne Last deutlich anders aussieht (vgl. Abbildung 6.11a). Noch schlechter sind die Ergebnisse bei der beacon-basierten Synchronisation. Abbildung 6.12b zeigt, dass nahezu gar keine Rahmen mit einem Abstand aus dem Intervall [3 ms,7 ms] empfangen werden. Dass die Empfangsintervalle bei der beacon-basierten Synchronisation deutlicher von den geforderten 5 ms abweichen als bei der Synchronisation mit BBS, liegt an den Auswirkungen der Last. Bei BBS wirkt sich Last nur auf den Versand von Datenrahmen aus und hier auch nur, wenn auf die Verwendung von *SEND_AT* verzichtet wird. Die Synchronisationsgenauigkeit ist unabhängig von der Lastsituation. Bei der beacon-basierten Synchronisation leidet bei Last hingegen sowohl die Synchronisationsgenauigkeit als auch der Versand der Datenrahmen.

Zeigen die Histogramme in Abbildungen 6.11 und 6.12 noch die Häufigkeiten, in denen einzelne Empfangsintervalle gemessen werden, fasst Abbildung 6.13 das durchschnittliche, maximale und minimale Empfangsintervall in Abhängigkeit der einzelnen Faktoren zusammen. Entsprechend der Symmetrie der Histogramme stimmt das durchschnittliche Empfangsintervall in allen Fällen nahezu exakt mit dem geforderten 5 ms-Intervall überein. Die besten Ergebnisse hinsichtlich des kleinsten Jitters liefert die Kombination (BBS, *SEND_AT*, *keine Last*). Hier ist das Maximum nur 146 μs größer und das Minimum nur 31 μs geringer als das geforderte Intervall von 5 ms. Besonders negativ auffallend sind die Kombinationen (BBS, *SEND*, *Last*), (Beacons, *SEND_AT*, *Last*) und (Beacons, *SEND*, *Last*), bei denen die minimalen Empfangsintervalle sogar teilweise bis $-45\,ms$ gingen (Hinweis: Bei großen/kleinen Empfangsintervallen wurde die x-Achse gestaucht). Ein negatives Empfangsintervall bedeutet hierbei, dass der Datenrahmen von Knoten 2 vor dem entsprechenden Datenrahmen von Knoten 0 empfangen wurde, d.h. es wurde noch nicht einmal die Sendereihenfolge erhalten.

Verlustraten von Datenrahmen

Die reine Übertragungszeit eines Datenrahmens berechnet sich anhand der Länge des Rahmens auf dem Medium (18 Bytes + 8 Bytes = 26 Bytes) und der Übertragungsrate des Transceivers (250 kbps) mit 832 μs. Bei einer reservierten Slot-Länge von 5 ms ist daher bei einer ausreichend genauen Synchronisation genügend Spielraum, um Kollisionen zwischen Datenrahmen vollständig zu vermeiden. Die teilweise sehr starken Abweichungen von dem nominellen Empfangsintervall von 5 ms, die oben analysiert wurden, lassen allerdings vermuten, dass die Ungenauigkeiten der Synchronisation und die starken Abweichungen von dem Sendezeitpunkt in manchen Fällen dennoch zu Kollisionen zwischen Datenrahmen führten.

Abbildung 6.14 zeigt die Ergebnisse aller acht durchgeführten Experimente. Es ist erkennbar, dass die Verlustrate bei allen Experimenten ohne Last sehr gering war. Dies wird auch von Ab-

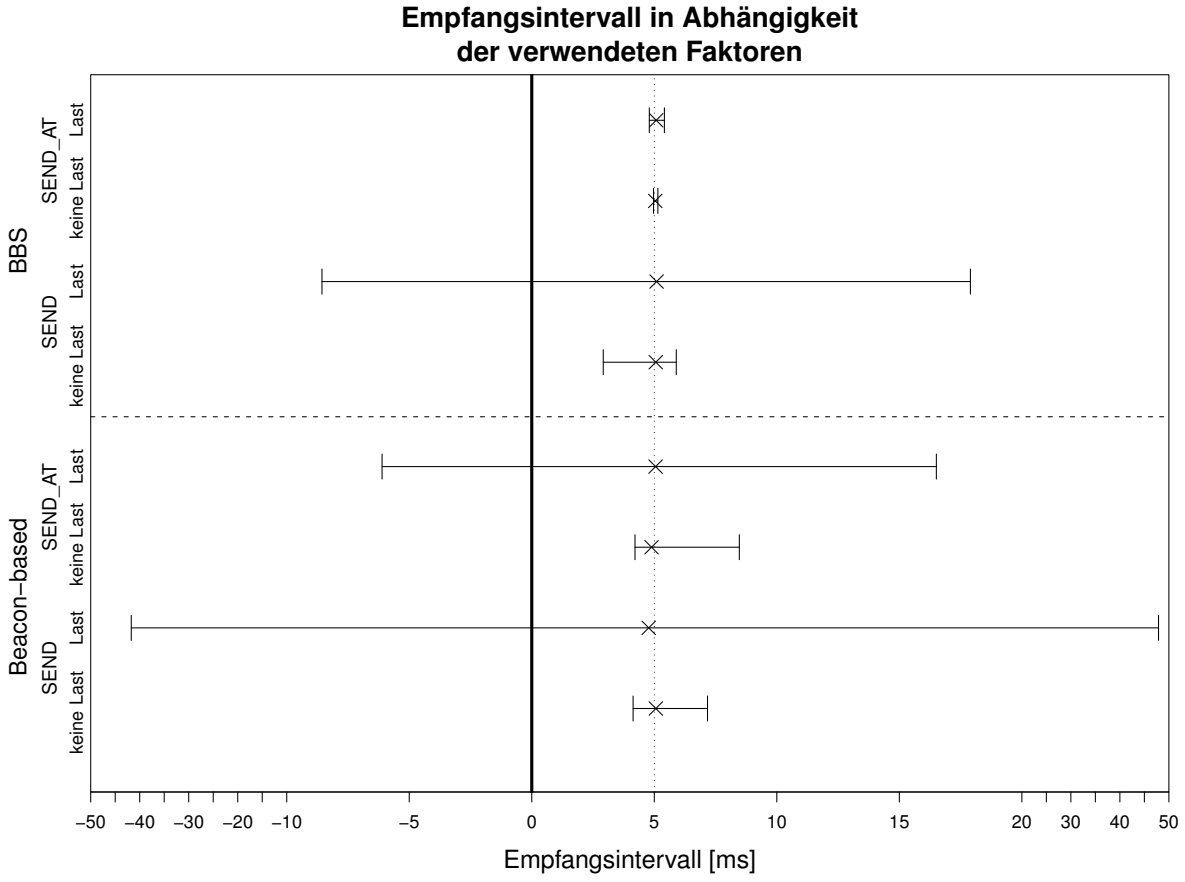


Abbildung 6.13.: Vergleich des durchschnittlichen, minimalen und maximalen Empfangsintervalls in Abhängigkeit von Synchronisation, Last und Versandart.

bildung 6.13 bekräftigt, da sich in allen Experimenten ohne Last das minimale/maximale Empfangsintervall nicht mit dem nächsten bzw. vorherigen Sendevorgang überschneidet[27].

Weniger erfreulich sind die Ergebnisse bei vorhandener Last. Einzige Ausnahme stellt hierbei die Kombination aus *Last*, *SEND_AT* und BBS zur Synchronisation dar, deren Verlustrate vergleichbar mit den Experimenten ohne Last ist. Die erhöhten Verlustraten in den anderen Fällen mit Last lassen sich auf das stark variierende Sendeintervall zurückführen, das nicht verhindert, dass Sendezeitpunkte disjunkt voneinander sind. Besonders negativ auffallend sticht die Kombination (BBS, *SEND*, *Last*) hervor, bei welcher die Verlustrate bei 19% liegt. Es ist schwer herauszufinden, warum die Verlustrate gerade bei dieser Kombination höher ist als beispielsweise bei der Kombination (Beacons, *SEND*, *Last*), insbesondere weil die Empfangsintervalle bei zweitgenannter Kombination deutlich stärker vom 5 ms-Sollwert abweichen (vgl. Abbildungen 6.12b und 6.11b). Man kann nur mutmaßen, dass die erzeugte Last einem für die Kombination ungünstigen Muster entsprach, welches Kollisionen förderte. Eine andere Möglichkeit wäre, dass der CC2420-Treiber durch die Integration des BBS-Treibers in *SEnF* in irgendeiner Form negativ beeinflusst wird und der BBS-Treiber zum Beispiel die „falschen“ Hardware-Interrupts filtert (vgl. Abschnitt 5.2.4). Zur genauen Analyse dieses Phänomens sind weitere Evaluationen notwendig.

[27] Da das Sendeintervall in den Fällen ohne *SEND_AT*-Funktion und ohne Last kleineren Schwankungen unterworfen ist, lassen sich Kollisionen auf Basis von Abbildung 6.13 nicht vollständig ausschließen.

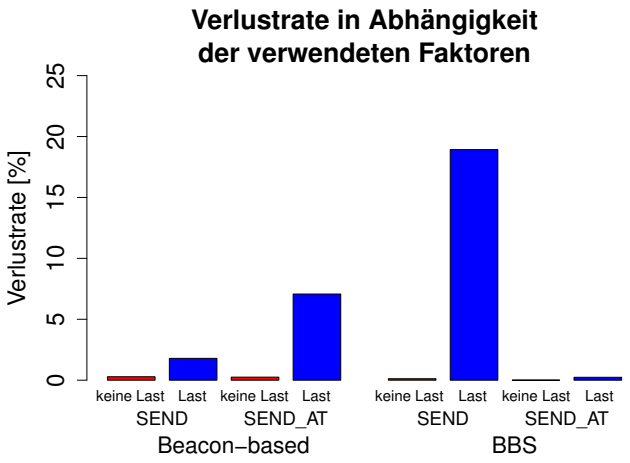


Abbildung 6.14.: Verlustrate von Datenrahmen in Abhängigkeit von Synchronisation, den Übertragungsmodi und der Lastsituation.

Interessant erscheint in Abbildung 6.14 auch, dass die Kombination (Beacon, *SEND_AT*, *Last*) eine höhere Verlustrate aufweist als (Beacon, *SEND*, *Last*). Basierend auf den Schwankungen und der minimalen/maximalen Abweichung des Empfangsintervalls von den nominellen 5 ms (vgl. Abbildungen 6.13, 6.12b und 6.12d), erscheint diese Tatsache zunächst nicht sehr plausibel. Die Begründung ist vermutlich in den Ursachen von Kollisionen bei der beacon-basierten Synchronisation zu finden: Werden Datenrahmen mit *SEND_AT* versendet, reicht es aus, dass die Synchronisation der zwei Sendeknoten „unglücklich“ ungenau ist. Dies passiert, wenn ein Sendeknoten während des Empfangs von Beacons Last abarbeitet, während der andere Sendeknoten nichts zu tun hat. Ist die Synchronisation in einem Macro-Slot erst einmal ungünstig, gehen in dem Macro-Slot viele Datenrahmen durch Kollisionen verloren, da beide Sendeknoten die Sendeintervalle aufgrund der *SEND_AT*-Funktion einhalten. In dem Fall von *SEND_AT* reicht also eine ungünstige Synchronisationsungenauigkeit für eine erhöhte Rahmenverlustrate aus. Werden Datenrahmen hingegen mit *SEND* übertragen, dann hängt die Rahmenverlustrate von zwei Faktoren ab: Einerseits von der Synchronisationsungenauigkeit und andererseits von der vorhandenen Last während dem von dem Knoten errechneten Sendezeitpunkt, an dem ein SDL-Timer dem Knoten signalisiert, den Datenrahmen zu senden.

Es ist übrigens zu bemerken, dass bei Last und der Verwendung von *SEND* nicht alle Rahmenverluste auf Kollisionen auf dem Medium zurückzuführen sind, sondern die Übertragung vieler Rahmen bereits vor dem Übertragungsbeginn scheitert. Hintergrund dieses Problems ist, dass der SDL-Agent, der die Sendeaufträge erteilt, bei Last nicht in dem vorgesehenen 10 ms-Sende-Intervall ausgeführt werden kann. Es kommt zu einer Verzögerung der Ausführung, die unter Umständen so groß ist, dass der 10 ms-Timer des Agenten bereits mehrfach abgelaufen wäre. In diesem Fall würde der Agent bei seiner Ausführung einen Timer mit einem Zeitpunkt in der Vergangenheit aufziehen, der wiederum sofort eine Transition aktiviert. Als Folge sendet der Agent bei seiner Ausführung in kurzen Abständen Sendeaufträge an das SDL-Environment. Wenn das SDL-Environment nun ausgeführt wird, werden alle SDL-Signale direkt nacheinander bearbeitet. Der erste Sendeauftrag bewirkt den Start einer neuen Übertragung. Der zweite Auftrag kann allerdings nicht direkt danach bearbeitet werden, da der Transceiver (bzw. der SPI-Bus) noch mit der Übertragung des ersten Signals beschäftigt ist. Dem SDL-System wird dies mit einer negativen Bestätigung mitgeteilt, wodurch es ihm möglich ist, auf den Fehlschlag

zu reagieren und eventuell eine Neuübertragung zu initiieren. Allerdings wäre dies in diesem Fall im Widerspruch zu der Übertragung von Datenrahmen zu reservierten Zeitpunkten.

Unabhängig von der Verlustrate weichen die Empfangsintervalle bei *SEND* im Allgemeinen deutlich stärker von den geforderten 5 ms ab als bei *SEND_AT*. Daher sollte die Verlustrate nicht darüber hinwegtäuschen, dass *SEND_AT* die bessere Alternative ist und gerade in Systemen mit unvorhersehbaren Lastsituationen verwendet werden sollte. Zusammen mit einer genauen Synchronisation, wie sie zum Beispiel mit BBS oder einer exakteren beacon-basierten Synchronisation (Zeitstempeln von Beacons, Verwendung von *SEND_AT* für Beacons) möglich ist, sind hierbei die besten Ergebnisse zu erwarten.

Ungenauigkeit durch Quarze

Es ist bekannt, dass die Synchronisationsungenauigkeit im Verlauf eines Macro-Slots aufgrund von unterschiedlicher Genauigkeit der Hardware-Quarze steigen kann. In diesem Abschnitt wird untersucht, ob dieser Effekt auch bei den vorliegenden Messdaten erkennbar ist. Bei einem Resynchronisationsintervall von 1 s ist die Zeitspanne, in der dieser Effekt auftreten kann, relativ gering. Das Wachstum der Synchronisationsungenauigkeit aufgrund der Differenz der Uhrenraten (*clock skew*) sollte in diesem Zeitfenster 80 μs nicht überschreiten [GK08] (siehe auch Anhang C.3). Da Datenrahmen nur in einem Zeitfenster von 800 ms übertragen werden, sollte die zusätzliche Ungenauigkeit zwischen dem ersten und letzten Messwert sogar 64 μs nicht überschreiten.

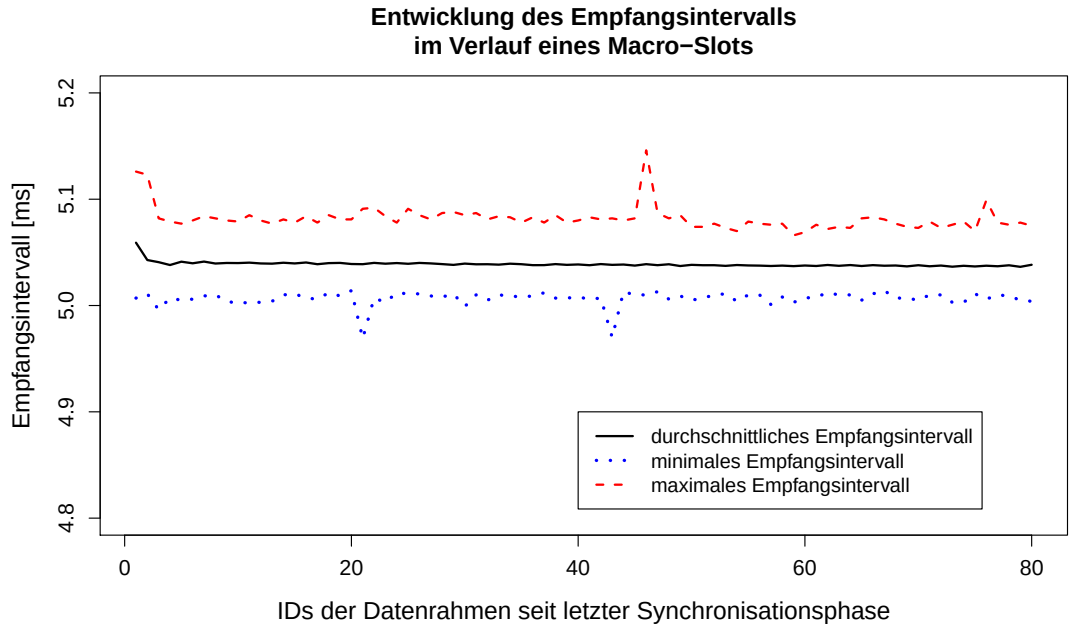


Abbildung 6.15.: Verlauf der Genauigkeit des Empfangsintervalls während eines Macro-Slots. Entsprechend dem Protokollverhalten umfasst die x-Achse ein Zeitintervall von 800 ms. Die Ergebnisse stammen aus dem Experiment mit den Faktoren (BBS, *SEND_AT*, *keine Last*).

Abbildung 6.15 zeigt den Verlauf der Empfangsintervalle innerhalb eines Macro-Slots unter Verwendung der Messdaten des besten Experiments: BBS zur Synchronisation, *SEND_AT* zur Übertragung der Datenrahmen, keine zusätzliche Last innerhalb des Systems. Die x-Achse ist mit den Sequenznummern der Datenrahmen innerhalb eines Macro-Slots beschriftet, deren Empfangsintervalle anschließend auf der y-Achse abgebildet werden. Beispielsweise werden bei $x = 0$ die

Ergebnisse angezeigt, die etwa 100 ms nach Ende der Synchronisationsphase gemessen werden; bei $x = 79$ werden hingegen die Ergebnisse etwa 900 ms nach Ende der Synchronisationsphase gezeigt. Die dargestellten Ergebnisse beinhalten sowohl das durchschnittliche (arithmetische Mittel) als auch das minimale und maximale Empfangsintervall, welche aus den Messdaten über alle 1800 Macro-Slots (30-minütiges Experiment) gewonnen wurden.

In Abbildung 6.15 fällt zunächst auf, dass das durchschnittliche Empfangsintervall etwas höher ist als die geforderten 5 ms. Bezüglich der Synchronisationsgenauigkeit bedeutet dies, dass in der Regel Knoten 2 seinen Tick etwas später wahrnimmt als Knoten 0 (siehe Formel 6.1). Diese Tatsache könnte einerseits darauf hinweisen, dass die Bearbeitungsverzögerungen durch Hardware-Interrupts genauer bestimmt werden müssten[28], oder andererseits bedeuten, dass Knoten 2 das SOF-Bit des Tick-Rahmens entsprechend der tolerierten Verzögerung bei der Erkennung einer Medienbelegung d_{maxCCA} in der Regel erst verspätet wahrnimmt.

Der für diesen Abschnitt eher relevante Verlauf der Synchronisationsgenauigkeit innerhalb eines Macro-Slots lässt hingegen nicht auf eine Differenz der Uhrenraten schließen. Das maximale und minimale Empfangsintervall sind zwar keineswegs konstant, sondern besitzen sogar ein paar Peaks, aber es ist keine Tendenz im Verlauf des Macro-Slots zu erkennen. Würden sich die Uhrenraten der beiden Sendeknoten deutlich unterscheiden, dann wäre dies in Form von fallenden oder steigenden minimalen und/oder maximalen Empfangsintervallen im Verlauf eines Macro-Slots zu erkennen. Um die Differenz der Uhrenraten allerdings im Detail zu analysieren, müssten längere Macro-Slots betrachtet und die Synchronisationsungenauigkeit müsste mit adäquateren Messmethoden untersucht werden.

Genauigkeit von *SEND_AT*

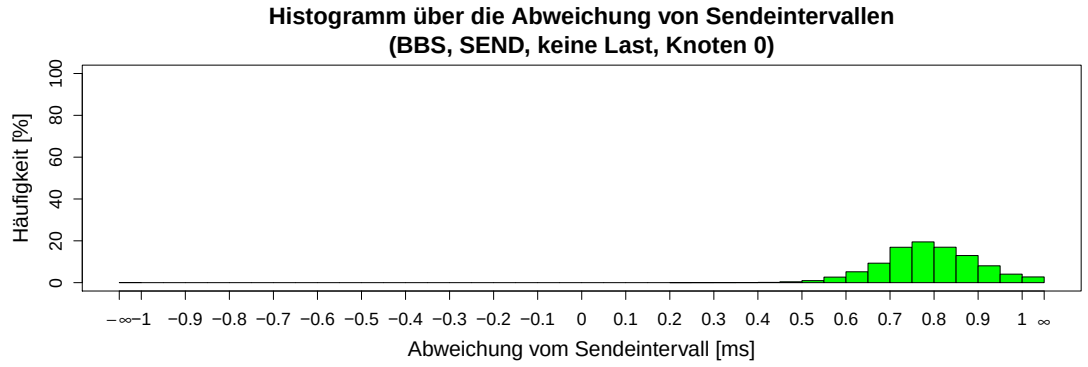
Bei den bisherigen Betrachtungen lag der Fokus auf dem Empfangsintervall zwischen den Datenrahmen der beiden Sender. In diesem Abschnitt wird untersucht, wie konstant das Sendeintervall eines Knotens ist. Hierzu wird ebenfalls das dem Sendeintervall entsprechende Empfangsintervall auf dem Empfangsknoten gemessen, welches im Idealfall 10 ms beträgt. Im Folgenden interessiert die Streuung um diesen Idealwert. Entsprechend Formel 6.1 lässt sich die nun betrachtete Streuung wie folgt berechnen (siehe auch Abbildung 6.10):

$$d'_{i,seqId} \quad = \quad t_{i,seqId} - t_{i,0} - seqId \cdot 10ms \quad , \quad 1 \leq seqId \leq 79, i \in 0,2 \tag{6.2}$$

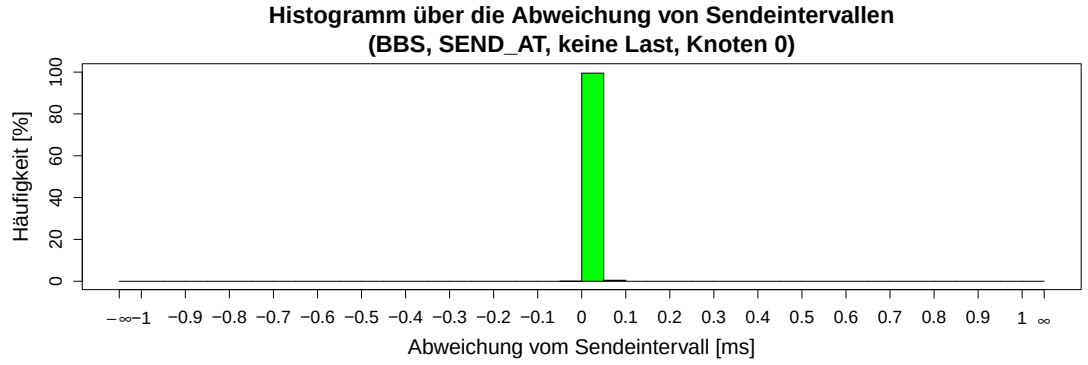
Hierbei ist $t_{i,seqId}$ der Empfangszeitpunkt des Rahmens, der von Knoten i mit der Sequenznummer $seqId$ gesendet wurde. $t_{i,0}$ ist der Empfangszeitpunkt des ersten von Knoten i gesendeten Rahmens eines Macro-Slots. Die 10 ms entsprechen dem geforderten Sendeintervall eines Knotens. Der Sollwert von $d'_{i,seqId}$ ist 0 ms.

Die Auswertung von $d'_{i,seqId}$ beschränkt sich auf Experimente, in denen BBS zur Synchronisation verwendet wurde und keine zusätzliche Last im System erzeugt wurde. Abbildung 6.16 zeigt die entsprechenden Ergebnisse sowohl für Knoten 0 als auch für Knoten 2 in Form von Histogrammen. Die Ergebnisse verdeutlichen die Vorteile von *SEND_AT*, da bei beiden Knoten die Abweichung von den nominellen 10 ms bei der Verwendung von *SEND_AT* deutlich geringer ausfällt. Bei den vorgestellten Messergebnissen wurde unter Verwendung von *SEND_AT* eine maximale

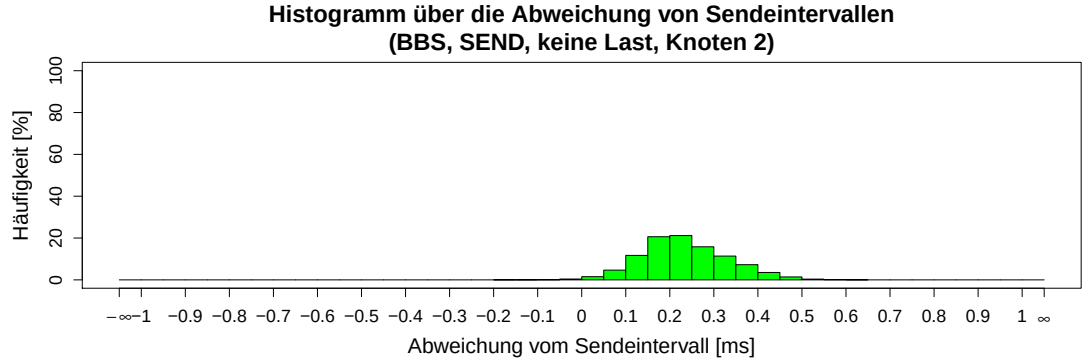
[28] Da Knoten 0 die Master-Rolle und Knoten 2 die Slave-Rolle einnimmt, entstehen auf beiden Knoten unterschiedliche Verzögerungen durch die Abarbeitung des BBS-Protokolls und die verschiedenartigen Interrupts. Diese Differenzen schaden der Synchronisationsgenauigkeit und müssen empirisch oder zum Beispiel durch Zählen von Instruktionen kompensiert werden [GK08].



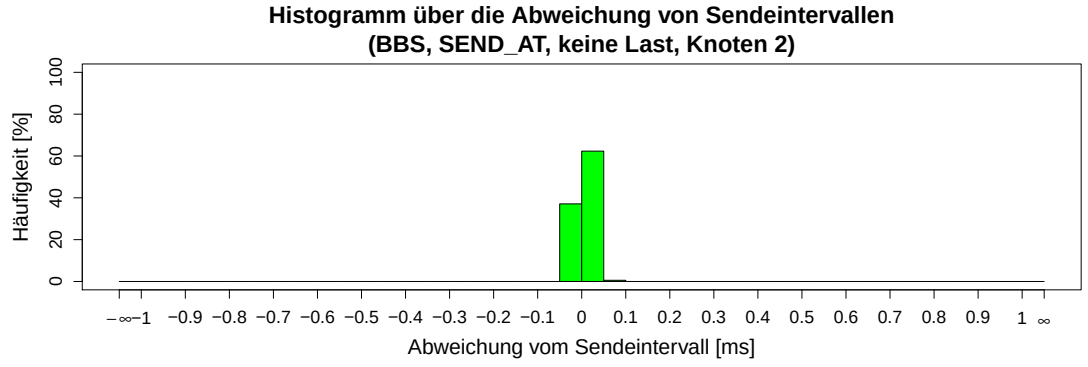
(a) Synchronisation mit BBS, Rahmenversand mit SEND, ohne Last, Knoten 0.



(b) Synchronisation mit BBS, Rahmenversand mit SEND_AT, ohne Last, Knoten 0.



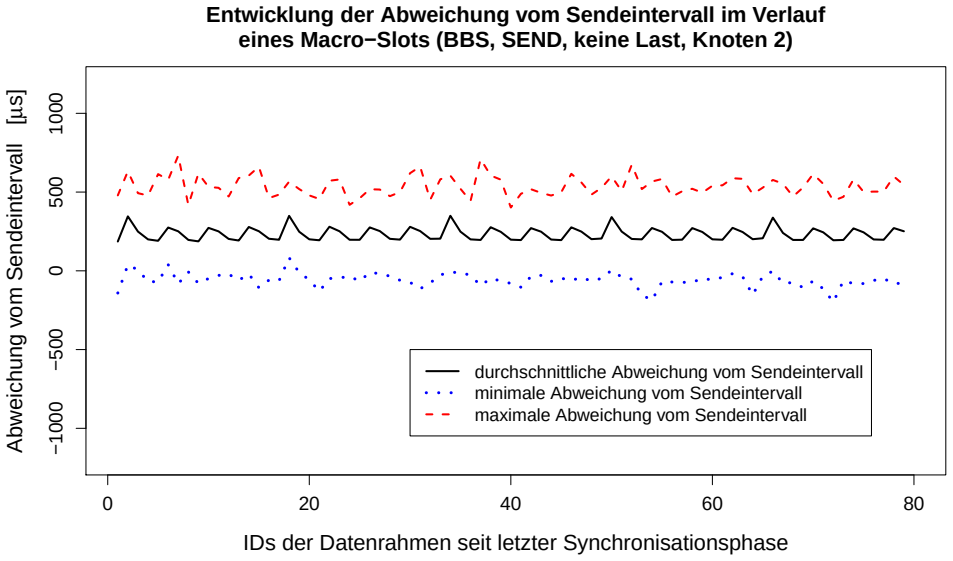
(c) Synchronisation mit BBS, Rahmenversand mit SEND, ohne Last, Knoten 2.



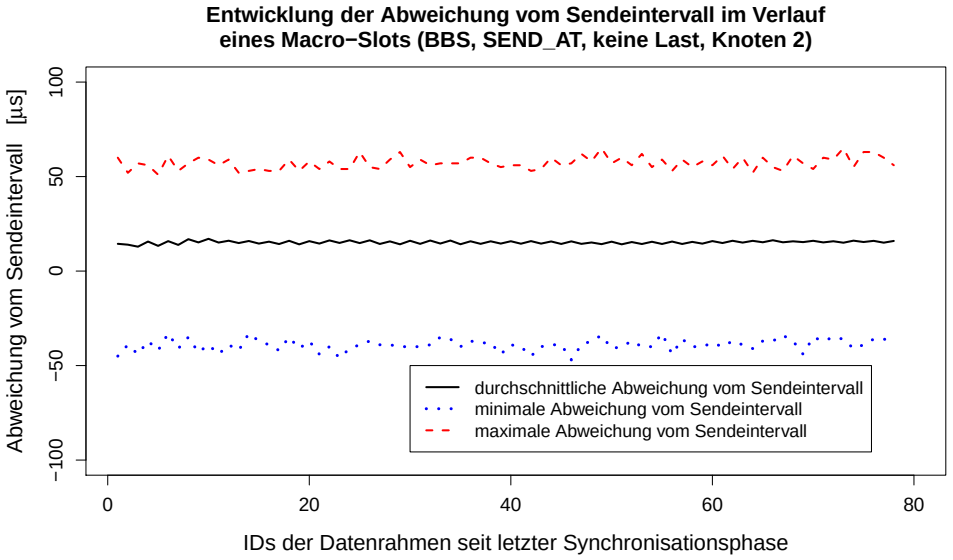
(d) Synchronisation mit BBS, Rahmenversand mit SEND_AT, ohne Last, Knoten 2.

Abbildung 6.16.: Sendeintervalle in Abhängigkeit der Übertragungsmodi von Datenrahmen für Knoten 0 und Knoten 2. Synchronisation erfolgt mit BBS ohne zusätzlicher Last innerhalb des Systems.

absolute Abweichung beider Knoten von 69 μs gemessen. Es fällt außerdem auf, dass die Histogramme keine Symmetrie um den Sollwert von 0 ms aufweisen. Im Falle von *SEND_AT* lässt sich dies damit erklären, dass zwischen dem Auftreten eines Timer-Interrupts und dem Neusetzen des Hardware-Timers etwas Zeit vergeht. Diese Zeitspanne ist jedoch so gering, dass sie kaum in den entsprechenden Abbildungen 6.16b und 6.16d in Form einer Rechtsverschiebung der Peaks auffällt. Bei der Kombination (BBS, *SEND*, *keine Last*, Knoten 0) ist die Antisymmetrie deutlich stärker zu erkennen. Da sich $d'_{i,seqId}$ immer auf den Empfang des ersten Rahmens bezieht, besagt die Grafik, dass der zeitliche Abstand zwischen dem ersten Rahmen und den folgenden Rahmen in der Regel größer ist als die geforderten Vielfachen von 10 ms. Diese Ergebnisse lassen sich möglicherweise mit Optimierungen durch *Caches* in der Laufzeitumgebung *SdlRE* erklären. Eine exakte Betrachtung erfordert allerdings eine detailliertere Auswertung.



(a) Synchronisation mit BBS, Rahmenversand mit SEND, ohne Last, Knoten 2.



(b) Synchronisation mit BBS, Rahmenversand mit SEND_AT, ohne Last, Knoten 2.

Abbildung 6.17.: Abweichungen vom Sendeintervall in Abhängigkeit der Übertragungsmodi von Datenrahmen. Es werden die Ergebnisse aus den Messreihen gezeigt, in denen BBS zur Synchronisation verwendet wurde und keine zusätzliche Last vorlag.

Des Weiteren wurde analysiert, wie sich das Sendeintervall der Knoten innerhalb eines Macro-Slots entwickelt. Hierzu zeigen die Abbildungen 6.17a bzw. 6.17b den Verlauf der Abweichungen von den Sendeintervallen während eines Macro-Slots für Knoten 2 bei Verwendung von *SEND* und *SEND_AT* (Hinweis: y-Achsen sind unterschiedlich skaliert). Die Abbildungen zeigen sowohl die durchschnittlichen als auch die minimalen und maximalen Abweichungen vom Sendeintervall. Bei Verwendung von *SEND* sind die durchschnittlichen Abweichungen vom Sollwert 0 ms deutlich größer, wodurch die Erkenntnisse bezüglich der Antisymmetrie des Histogrammes aus Abbildung 6.16c bestätigt werden. Ebenso ist die Spanne zwischen maximalem und minimalem Sendeintervall mit etwa 600 μs relativ groß, was eine Bestätigung der Streuung aus Abbildung 6.16c darstellt. Bei Verwendung von *SEND_AT* ist die durchschnittliche Abweichung des Sendeintervalls deutlich näher an den nominellen 0 ms (15 μs vs. 250 μs bei *SEND*). Sowohl die Spanne zwischen maximalem und minimalem Sendeintervall (etwa 100 μs) als auch die Schwankungen der ermittelten durchschnittlichen Abweichung fallen ebenfalls sehr viel geringer aus. Dies zeigt die Stärke von *SEND_AT*, die nicht unter dem Jitter leidet, der bei *SEND* aus der Ausführung eines SDL-Systems und der enthaltenen Agenten resultiert. Die Verwendung von *SEND_AT* ermöglicht dadurch die sehr exakte Einhaltung von Sendeintervallen.

6.3.2 Multi-Hop-Synchronisation mit BBS

BBS zeichnet sich insbesondere durch die Unterstützung von Multi-Hop-Topologien aus, in welchen durch die kollisionsresistente Ausbreitung von Tick-Rahmen eine Synchronisation innerhalb fester Zeitschranken erreicht wird. Dieser Abschnitt überprüft die Implementierung hinsichtlich der Kollisionsresistenz und Multi-Hop-Unterstützung.

6.3.2.1 Beschreibung des Szenarios

Das Szenario besteht aus einem Experiment, das die korrekte Funktionalität von BBS ohne etwaigen Versand von regulären Datenrahmen zur Aufgabe hat. Im Fokus steht die Rate der erfolgreichen Resynchronisationsphasen.

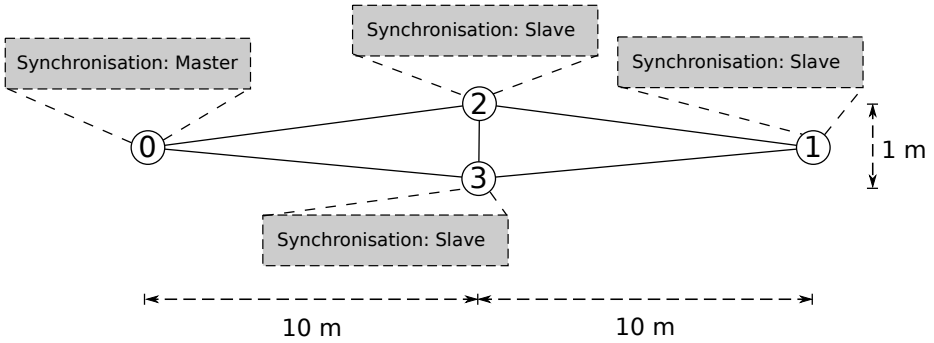


Abbildung 6.18.: Multi-Hop-Topologie zur Evaluation der Multi-Hop-Fähigkeit des BBS-Treibers.

Der Versuchsaufbau erfolgte ebenfalls auf einem Waldparkplatz, um externe Einflüsse zu minimieren. Die verwendete Topologie ist in Abbildung 6.18 dargestellt und besteht aus einem Master-Knoten (Knoten 0) und drei Slave-Knoten. Messdaten wurden bei Knoten 1 für eine spätere Auswertung gesammelt. Zu den gesammelten Messdaten gehörten unter Anderem die Assoziation des Referenzticks mit der lokalen SDL-Zeit und die Rundennummer, in welcher der Knoten einen Tick-Rahmen empfing. Knoten 0 und Knoten 1 wurden mit einem Abstand von 20 m so

angeordnet, dass sie sich nicht in Sendereichweite voneinander befanden, d.h. im Regelfall empfing Knoten 1 keine Tick-Rahmen von Knoten 0. Knoten 2 und 3 wurden zwischen Knoten 0 und 1 positioniert, um eine Konnektivität zwischen Knoten 0 und 1 herzustellen.

Der Aufbau überprüfte daher zwei Aspekte von BBS: Die Multi-Hop-Fähigkeit, ohne die Knoten 1 in der Regel unsynchronisiert bliebe, und die Kollisionsresistenz, da Knoten 1 im Regelfall Tick-Rahmen empfängt, die von Knoten 2 und Knoten 3 gleichzeitig übertragen wurden. Obwohl das Netzwerk nur einen Durchmesser von zwei Hops besitzt, wurde $n_{maxHops} = 5$ festgelegt, damit bei der Auswertung der gewonnenen Messdaten zwischen den Rundennummern, in welchen Knoten 1 den Tick-Rahmen empfing, unterschieden werden konnte.

6.3.2.2 Ergebnisse

Die Auswertung der Messdaten legt den Fokus auf zwei Gesichtspunkte: Die Zuverlässigkeit, mit welcher Knoten 1 Tick-Rahmen empfängt, und die Abweichung zwischen neu propagierten Ticks und erwarteten Ticks, die Knoten 1 jeweils aus dem alten Tick und der Macro-Slot-Länge berechnet.

Zuverlässigkeit

Eine Synchronisation stellt eine zwingende Voraussetzung für Anwendungen und andere Protokolle dar und ist daher entscheidend für den Erfolg eines Netzwerks. Durch Setzen von $n_{maxHops}$ auf einen größeren Wert als er eigentlich für die Topologie unter idealen Bedingungen notwendig wäre, kann nun zwischen dem Grad an Zuverlässigkeit differenziert werden.

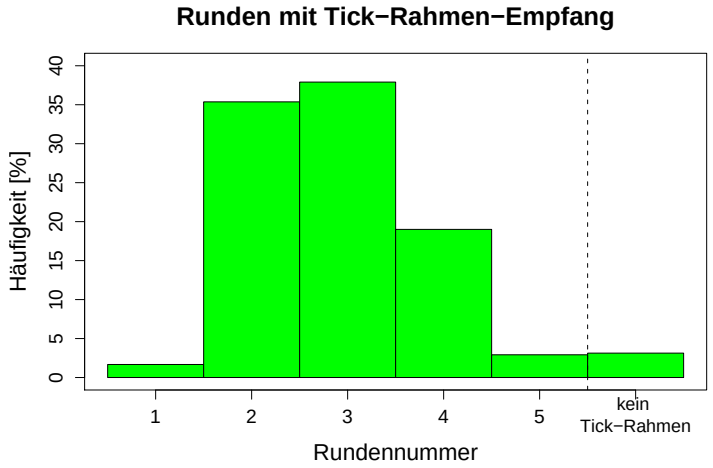


Abbildung 6.19.: Häufigkeitsverteilung von empfangenen Rundennummern.

Abbildung 6.19 zeigt in einem Histogramm die Häufigkeiten der Synchronisationsrunden, in denen sich Knoten 1 durch den Erhalt eines korrekten Tick-Rahmens auf den vom Master-Knoten definierten Referenztick synchronisierte. Das Histogramm bezieht sich dabei auf den ersten korrekt empfangenen Tick-Rahmen innerhalb einer Synchronisationsphase. Da Knoten 1 nach Erhalt eines korrekten Tick-Rahmens in den eventuell folgenden Synchronisationsrunden ebenfalls Tick-Rahmen sendete, war der empfangene Tick-Rahmen außerdem der einzige, den der Knoten innerhalb der Synchronisationsphase empfing. Ebenso enthält das Histogramm die Häufigkeit von Synchronisationsrunden, in denen Knoten 1 keinen Tick-Rahmen erhält und sich folglich

nicht auf den neuen Referenztick des Master-Knotens synchronisiert. Unter idealen externen Bedingungen und einer vollständig korrekten Implementierung käme es zu keinen Verlusten von Tick-Rahmen und entsprechend der Topologie von Abbildung 6.18 würde Knoten 1 in Runde 2 immer einen korrekten Tick-Rahmen empfangen.

Das Histogramm in Abbildung 6.19 lässt erkennen, dass die Synchronisation in 97% aller Fälle erfolgreich verlief und nur in etwa 3% der Synchronisationsphasen keine Tick-Rahmen empfangen wurden. Auffallend ist ebenfalls, dass Knoten 1 trotz der Entfernung von 20 m in 2% der Fälle korrekte Tick-Rahmen von Knoten 0 empfing (Knoten 0 war der einzige Knoten, der Tick-Rahmen mit Rundennummer 1 sendete). Dies weist darauf hin, dass sich Knoten 2 nicht permanent außerhalb der Sendereichweite von Knoten 0 befand und zumindest teilweise eine Kommunikation mit Black Bursts möglich war.

Negativ fällt in der Abbildung die Verteilung der erfolgreichen Synchronisationsrunden auf die Runden 2, 3 und 4 auf. Insbesondere liegt die größte Häufigkeit nicht bei Synchronisationsrunde 2 (etwa 35%), wie entsprechend dem Versuchsaufbau zu erwarten wäre, sondern bei Synchronisationsrunde 3 (etwa 38%). Die Ursache könnte zum Beispiel in externen Störquellen liegen, welche die Erkennungsrate von Black Bursts negativ beeinflusste, oder in einer fehlerhaften oder ungenauen Implementierung des BBS/CC2420-Treibers. Da sich der CC2420-Treiber durch Erweiterungen und Bugfixes ständig entwickelt und sich das Zeitverhalten der Treiber-Implementierung dadurch verändern kann, spricht vieles für die zweite Ursache. Die genauen Ursachen für die Abweichungen von den Erwartungen lassen sich allerdings ohne weitere Betrachtungen nicht feststellen.

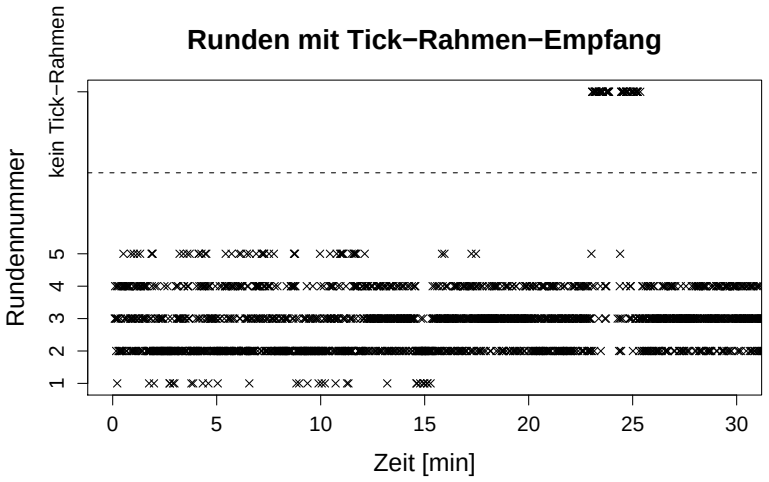


Abbildung 6.20.: Zeitliche Entwicklung der Synchronisationsrunden, in denen Tick-Rahmen empfangen wurden.

Anhand von Abbildung 6.20 lässt sich der zeitliche Verlauf der Synchronisationsrunden, in denen Knoten 1 einen korrekten Tick-Rahmen empfing, über die 30-minütige Dauer des Experiments im Detail betrachten. Hierbei fällt auf, dass sich die erfolglosen Synchronisationsphasen nicht über die komplette Dauer des Experiments verteilten, sondern erst nach etwa 25 min geballt auftraten. Diese Tatsache könnte ein Indiz dafür sein, dass sich ein nicht-betrachteter Faktor, wie zum Beispiel eine externe Störquelle oder die begrenzte Kapazität des Imote2-Akkus, zeitweise in einem Maße verschlechterte, so dass er einen Einfluss auf die Messdaten hatte. Ohne genauere Erfassung dieser Faktoren kann man allerdings nur mutmaßen. Die Ergebnisse sind

jedoch ein Indiz dafür, dass bei einer drahtlosen Kommunikation selbst bei stationären Knoten starke Schwankungen in den Messergebnissen zu erwarten sind.

Abweichungen von erwarteten Ticks

Ein Vorteil von BBS ist die niedrige und deterministisch beschränkte Synchronisationsungenauigkeit. Die Synchronisationsungenauigkeit zwischen zwei Knoten ist dabei als Offset zwischen den lokal wahrgenommenen Referenzticks beider Knoten definiert. In der vorliegenden Topologie und unter Verwendung des CC2420-Transceivers errechnet sich die maximale Synchronisationsungenauigkeit zwischen Knoten 0 und Knoten 1 auf 336 μs (siehe Anhang C.3). Die Beschränktheit der Synchronisationsgenauigkeit hat allerdings nicht nur Auswirkungen auf den Offset zwischen den wahrgenommenen Ticks, sondern hat ebenfalls Einfluss auf die Zeitpunkte, zu denen ein Slave-Knoten den Empfang eines Tick-Rahmens erwartet. Dadurch wirkt sich die Synchronisationsgenauigkeit ebenfalls auf die maximal mögliche Abweichung zwischen dem erwarteten Tick, der basierend auf dem letzten Tick und der Dauer eines Macro-Slots berechnet wird, und dem neuen Tick nach einer erfolgreichen Resynchronisation aus.

Ist Slave-Knoten 1 synchronisiert und assoziiert seine lokale Zeit $t_{1,tick}$ mit dem Referenztick, dann erwartet der Knoten den nächsten Tick zum Zeitpunkt $t'_{1,tick} = t_{1,tick} + d_{macro}$. Durch Ungenauigkeit in der Synchronisation muss der nächste Tick aus Sicht von Knoten 1 nicht exakt bei $t'_{1,tick}$ erfolgen, sondern kann in dem Intervall $[t'_{1,tick} - d_{maxOffset}, t'_{1,tick} + d_{maxOffset}]$ liegen. Die deterministisch beschränkte Synchronisationsungenauigkeit $d_{maxOffset}$ stellt dabei sicher, dass der nächste Tick tatsächlich in diesem Intervall liegt.

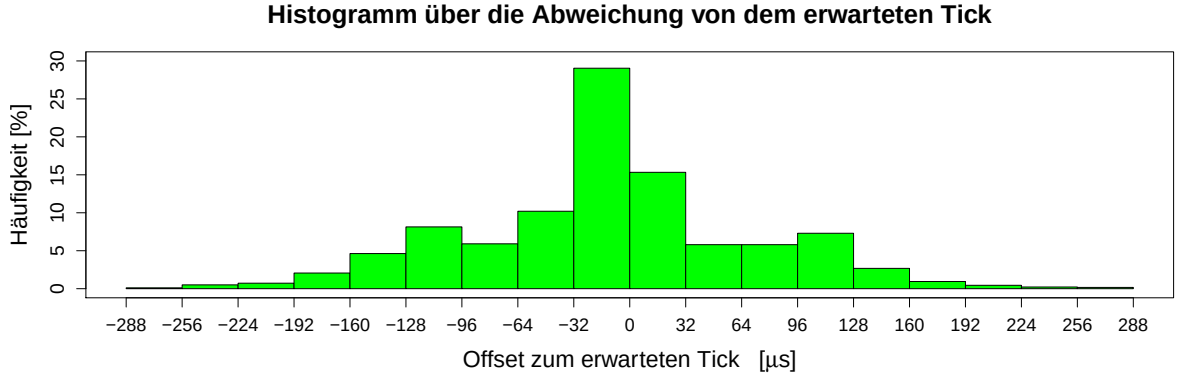


Abbildung 6.21.: Abweichung des über BBS propagierten Ticks von dem erwarteten Tick.

Abbildung 6.21 zeigt für Knoten 1 den Offset zwischen erwartetem Tick und tatsächlich empfangenem Tick in Form eines Histogramms. Es ist zu erkennen, dass in fast 30% der Fälle der nächste Tick in dem Intervall $[t'_{1,tick} - 32\,\mu s\,, t'_{1,tick}]$ liegt und damit nur minimal früher ist als der errechnete Zeitpunkt des Ticks. In mehr als 87% der Fälle liegt der nächste Tick in dem Intervall $[t'_{1,tick} - 128\,\mu s\,, t'_{1,tick} + 128\,\mu s\,]$ und weicht damit maximal 128 μs von dem erwarteten Tick ab. Die absolute maximale Abweichung von dem erwarteten Tick liegt bei 287 μs bzw. -279 μs, d.h. der nächste Tick wird 287 μs bzw. -279 μs später/früher wahrgenommen als berechnet. Damit ist der maximale Offset zum erwarteten Tick in dem berechneten Bereich $[t'_{1,tick} - 336\,\mu s\,, t'_{1,tick} + 336\,\mu s\,]$, wodurch der deterministischen Grenze der Synchronisationsgenauigkeit nicht widersprochen wird.

Des Weiteren stellen die Ergebnisse in Abbildung 6.21 einen Beleg dar, dass die empfangenen Rundennummern korrekt in die Berechnung der Referenzticks eingingen und insbesondere

auch kein verfälschter Tick-Rahmen irrtümlicherweise zur Berechnung des Referenzticks verwendet wurde. Wäre dies nämlich der Fall, dann wäre die maximale Abweichung von dem errechneten Tick deutlich größer als 287 μs bzw. -279 μs, da eine Synchronisationsrunde bei der verwendeten Konfiguration eine zeitliche Dauer von knapp 5 ms benötigte (siehe Anhang C.4).

Es sei darauf hingewiesen, dass diese Ergebnisse nur indirekt etwas über die Synchronisationsungenauigkeit zwischen zwei Knoten aussagen. Bei zwei Slave-Knoten, die den gleichen Tick-Rahmen empfangen, sind die Ergebnisse ein Indiz für die Güte der Synchronisation, da beide Knoten den Tick mit identischem Vorgehen (Zeitpunkt des CCA-Interrupts, Abziehen von Berechnungsaufwand, etc.) mit einem lokalen Zeitpunkt assoziieren. Über die Synchronisationsgenauigkeit zwischen Sender eines Tick-Rahmens und entsprechendem Empfänger sagen die Ergebnisse hingegen nichts aus. Hierzu müsste eine andere Methodik verwendet werden.

6.3.3 Fazit

Die Evaluation des BBS-Treibers zeigt bereits das Potential der BBS-Implementierung hinsichtlich der Genauigkeit der Synchronisation, welche insbesondere unabhängig von einer möglichen Last innerhalb des SDL-Systems ist. Dieser Vorteil schafft die Grundlage, um den vorgestellten BBS-Treiber als szenariounabhängigen Basisdienst für Anwendungen und andere Kommunikationsprotokolle zu verwenden. In einem der verwendeten Evaluationssysteme mit TDMA stellt BBS diese Vorteile unter Beweis und zeigt die Wichtigkeit der Synchronisationsgenauigkeit hinsichtlich der Rahmenkollisionsrate. Ebenso hat sich in den evaluierten Systemen die Verwendung von *SEND_AT* bewährt, die der Last-Anfälligkeit der Versandzeitpunkte entgegenwirkt.

Die Ergebnisse der durchgeführten Experimente zeigen aber auch einige Probleme der Implementierung von BBS auf. Gerade die Zuverlässigkeit der Übertragung von Tick-Rahmen, die in der Multi-Hop-Topologie weniger gute Ergebnisse als erhofft lieferte, erfordert die fortwährende Suche nach Ursachen. Es muss herausgefunden werden, ob ausschließlich externe Einflüsse für die Ergebnisse verantwortlich waren oder ob Fehler in der Implementierung ihren Teil dazu beigetragen haben. Andererseits konnten sich in diesem Zusammenhang die Maßnahmen zur Erhöhung der Robustheit von BBS und hier insbesondere das permanente Senden von Tick-Rahmen nach einmal erfolgter Synchronisation auszeichnen (siehe Abschnitt 5.1.4.5), ohne die die Synchronisation noch häufiger fehlgeschlagen wäre.

7. KAPITEL

Zusammenfassung & Ausblick

In der vorliegenden Masterarbeit wurden die Probleme von SDL hinsichtlich der Spezifikation zeitkritischen Verhaltens untersucht und Lösungen zur Verbesserung des Laufzeitverhaltens vorgestellt. Im Wesentlichen lassen sich die Ergebnisse der Arbeit in zwei Teile untergliedern: Die Umsetzung eines Frameworks für Planungsverfahren, das in die SDL-Laufzeitumgebung *SdlRE* integriert wurde und dem Entwickler eines SDL-Systems die Wahl zwischen verschiedenen Schedulingstrategien eröffnet, und die Realisierung von *Black Burst Synchronization* (BBS), die einen Kompromiss zwischen einer effizienten/hardwarenahen und modellgetriebenen Entwicklung darstellt.

Das Framework für Planungsverfahren entstand aus der Notwendigkeit, den Indeterminismus, den der SDL-Standard an vielen Stellen zulässt, zu reduzieren. Dieser Indeterminismus hat einerseits den Vorteil, dass er von Implementierungsdetails, wie zum Beispiel Ausführungszeiten von Agenten oder dem Grad an Parallelität, abstrahiert, und die Voraussetzungen für eine plattformunabhängige Spezifikation des Systemverhaltens schafft. Andererseits ist es dadurch mit SDL nicht möglich, Aussagen über das Zeitverhalten des Systems bei der Ausführung zu treffen oder ein bestimmtes Laufzeitverhalten zu erzwingen. Eine Überprüfung, ob Fristen eingehalten werden, ist basierend auf der SDL-Spezifikation somit nicht möglich.

Daher wurde neben den vorhandenen Planungsverfahren ein weiteres Verfahren in das Framework integriert, das die Ausführung von Agenten nach statischen Prioritäten steuert und damit eine Bewertung bezüglich der Dringlichkeit von Aktivitäten zulässt. Das Verfahren ist konform zum SDL-Standard, nutzt aber die Freiheiten, die der Standard an vielen Stellen lässt, um zeitkritische Aktivitäten in der Ausführung zu bevorzugen. Dadurch verringert es den konzeptionellen Abstand zwischen der Ebene von SDL, deren Ausführung auf asynchronen ASMs basiert, und der Implementierung auf realen Plattformen. Die Wahl des Planungsverfahrens und die Vergabe der Prioritäten folgt dabei dem modellgetriebenen Entwicklungsgedanken, indem der Entwickler das Verfahren und die Prioritäten direkt in der SDL-Spezifikation festlegt. Hierzu wurde Time-critical-SDL (TC-SDL) entwickelt, eine Erweiterung von SDL mit Annotationen, die es dem Entwickler ermöglicht, zwischen Planungsverfahren zu wählen und Prioritäten flexibel auf der Ebene von SDL-Blöcken, SDL-Prozessen oder SDL-Services zu vergeben.

Mit der Realisierung von BBS wurde in der Arbeit die Umsetzung eines Synchronisationsprotokolls mit hohen Anforderungen bezüglich der zeitlichen Ausführung vorgestellt. Nach der Diskussion über Probleme, denen BBS in einem realen Umfeld mit Rauschen und Interferenzen gegenübersteht, wurden Maßnahmen zur Erhöhung der Robustheit beschrieben. Für die Realisierung des Protokolls schied eine vollständige Spezifikation auf SDL-Ebene aus, da auch mit

dem vorgestellten prioritätsbasierten Planungsverfahren die von BBS geforderten Genauigkeiten und Geschwindigkeiten nicht eingehalten werden können. Das Protokoll wurde daher in zwei Komponenten realisiert: Einem Treiber auf SDL-Ebene, der ein komfortables Konfigurationsinterface zur Verfügung stellt, und einem Treiber im SDL-Environment-Framework *SEnF*, der effizient in C programmiert wurde, um den Zeitanforderungen von BBS gerecht zu werden.

Zur Validierung der entwickelten Maßnahmen wurden experimentelle Evaluationen mit der Imote2-Plattform durchgeführt. In einer ersten Serie von Experimenten wurde das neu eingeführte prioritätsbasierte Planungsverfahren funktional getestet und die Ergebnisse mit anderen Planungsverfahren verglichen. Die Ergebnisse verdeutlichen bereits das große Potential des prioritätsbasierten Planungsverfahrens, dessen Overhead verglichen mit anderen Verfahren zwar etwas höher ausfällt, aber im Gegenzug die Wartezeiten hochpriorer Agenten deutlich reduziert und einen entscheidenden Beitrag zur Vorhersagbarkeit von SDL-Systemen leistet.

Mit einer weiteren Serie von Experimenten wurde die Realisierung von BBS getestet. Hierzu wurde BBS in einem Single-Hop-Szenario mit einem beacon-basierten Synchronisationsverfahren verglichen, welches vollständig in SDL spezifiziert wurde. Die Ergebnisse zeigen, dass ein SDL-System mit einem Synchronisationsprotokoll, das einen äußerst zeitkritischen Vorgang darstellt, auf die Unterstützung einer hardwarenahen Treiberkomponente angewiesen ist, da sich Ausführungsverzögerungen in dem System insbesondere in Lastsituation unvorhersehbar verhalten können. In einem weiteren Experiment wurde BBS in einem Multi-Hop-Szenario evaluiert. Die gewonnenen Ergebnisse demonstrieren sowohl die Multi-Hop-Fähigkeit des Synchronisationsprotokolls als auch die Kollisionsresistenz der Tick-Rahmen, zeigen allerdings noch einige Schwächen der Implementierung hinsichtlich der Zuverlässigkeit auf.

Mit der Implementierung des Frameworks für Planungsverfahren wurde erst der Grundstein gelegt, um weitere Schedulingstrategien für das Ausführungsmodell von SDL zu realisieren. Das prioritätsbasierte Verfahren zeigt bereits die Vorteile eines für zeitkritische Systeme adäquateren Verfahrens, stellt aber keinesfalls das Optimum bezüglich der Planung von zeitkritischen Aktivitäten dar. Als Kandidat für eine zukünftige Erweiterung des Frameworks eignet sich unter Anderem das bekannte und erprobte *Rate Monotonic*-Verfahren. Hierbei muss allerdings abgewogen werden, in welchem Ausmaß der SDL-Sprachumfang bei Verwendung entsprechender Planungsverfahren unterstützt wird, da viele Verfahren für Systeme mit dynamischer Prozess-Instanziierung nur bedingt geeignet sind. Ebenso soll zukünftig eine feingranularere Wahl zwischen Planungsverfahren umgesetzt werden, um in unterschiedlichen Systemteilen verschiedene Schedulingstrategien verwenden zu können. Dadurch wäre eine klare Differenzierung zwischen „zeitkritischen“ Komponenten und Komponenten „durchschnittlicher Priorität“ in einem System möglich. Des Weiteren ist die Ergänzung des Frameworks mit präemptiven Planungsverfahren ein Thema, welches in zukünftigen Arbeiten zu diskutieren ist.

Für die Implementierung von BBS gilt es, die Effizienz und Zuverlässigkeit zu steigern. Gerade die Evaluation in der Multi-Hop-Topologie hat gezeigt, dass bezüglich der Zuverlässigkeit noch Schwächen existieren, deren Ursachen durch weitere Experimente identifiziert und behoben werden müssen. Außerdem fehlen Experimente zur Bestimmung der Synchronisationsgenauigkeit, um zu überprüfen, ob die berechneten Grenzen in der Implementierung eingehalten werden oder ob zusätzliche Optimierungen notwendig sind.

Des Weiteren ist es erstrebenswert, die aus dieser Arbeit entstandenen Ergebnisse in zukünftigen Systemen einzusetzen. Dies könnte sowohl zu einer Qualitätsverbesserung der bestehenden Verfahren beitragen als auch neue Anforderungen definieren, die in der Weiterentwicklung der Verfahren Beachtung finden.

A. ANHANG

Werkzeuge des modellgetriebenen Entwicklungsprozesses mit SDL

Die in dieser Arbeit vorgestellten Ergebnisse dienen der Unterstützung eines Entwicklungsprozesses, bei dem die SDL-Spezifikation im Mittelpunkt der gesamten Entwicklung steht [Got07]. Ein solcher Entwicklungsprozess profitiert von den Möglichkeiten der automatisierten Verarbeitung der SDL-Spezifikationen, anhand derer Implementierungen für unterschieden Plattformen generiert werden können. Im Kontext dieser Arbeit tragen folgende Werkzeuge hierzu ihren Teil bei:

- *IBM Tau/IBM Rational SDL Suite* [IBMar]: Mit Hilfe von IBM Tau können SDL-Spezifikationen graphisch erzeugt und bearbeitet werden. Des Weiteren bietet die Suite Unterstützung zur syntaktischen und (eingeschränkten) semantischen Analyse der Spezifikation. Die Suite wird innerhalb des Entwicklungsprozesses außerdem zum Konvertieren zwischen der graphischen Repräsentation von SDL (SDL/GR) und der textuellen Darstellungsform (SDL/PR) verwendet.
- *ConTraST* [FGW06]: *ConTraSt* ist ein Werkzeug zur Umwandlung von SDL-Spezifikationen (SDL/PR) nach C++. Das Werkzeug entstand in der AG Vernetzte Systeme [Netar]. Im Zuge dieser Arbeit wurde *ConTraST* um die von TC-SDL eingeführten Annotationen erweitert.
- *SDL Runtime Environment* (SdlRE): *SdlRE* ist eine Laufzeitumgebung für SDL. Seine Aufgaben bestehen unter Anderem in dem *Schedulen* und *Dispatchen* von Agenten eines SDL-Systems. Außerdem ist *SdlRE* für die Kommunikation zwischen den Agenten verantwortlich. *SdlRE* wurde ebenfalls in der AG Vernetzte Systeme entwickelt [Netar]. In der vorliegenden Arbeit wurde *SdlRE* um ein Framework für die Wahl zwischen verschiedenen Planungsverfahren ergänzt, in welches ein Verfahren mit statischen Prioritäten integriert wurde.
- *SDL Environment Framework* (SEnF) [FGJ+05]: Das *SDL Environment Framework*, oder kurz *SEnF*, ist eine Ansammlung von Treibern, die den Hardwarezugriff kapseln. Das Framework entstand ebenfalls in der AG Vernetzte Systeme [Netar] und bildet die Brücke zwischen dem SDL-Environment und der realen Hardwareplattform. Zusammen mit *SdlRE* übernimmt *SEnF* die Aufgaben, die üblicherweise ein Betriebssystem erledigt. Aktuell werden die Plattformen MICAz, Imote2, PC (Linux/Windows) und der Netzwerksimulator *ns+SDL* [BGK08] unterstützt, wobei sich die Auswahl der verfügbaren Hardware zwi-

schen den Plattformen unterscheidet. In der vorliegenden Arbeit wurde *SEnF* um einen virtuellen Treiber für die Imote2-Plattform erweitert, welcher die Protokollfunktionalität von BBS implementiert.

ConTraST, *SdlRE* und *SEnF* sind ebenfalls auf der beiliegenden CD zu finden (siehe Anhang D).

B. ANHANG

TC-SDL: Parameter der Planungsverfahren

Im Folgenden werden die mit TC-SDL eingeführten Annotationsmöglichkeiten zusammengefasst.

B.1 Annotationen auf Systemebene

Innerhalb des Kopf-Symbols des SDL-Systems werden grundlegende Entscheidungen bezüglich des gewünschten Planungsverfahrens getroffen. Allgemein ist zu beachten, dass ein Text innerhalb des Kopf-Symbols nicht als Kommentar gesehen wird, sodass der Entwickler die Annotationen manuell in eine /* ... */-Umgebung setzen muss.

B.1.1 Syntax

Die erlaubte Syntax ist in Form einer kontextfreien Grammatik unter Verwendung der Backus-Naur-Form gegeben:

Nichtterminal-Symbole:

{⟨sched_annotation⟩, ⟨sched_param_list⟩, ⟨key_val_pair⟩,
⟨sched_nat_key⟩, ⟨natural⟩,⟨sched_pos_key⟩, ⟨pos_natural⟩,
⟨sched_strategy_key⟩, ⟨sched_strategy_val⟩,⟨digit⟩, ⟨pos_digit⟩ }

Terminal-Symbole:

{0,1,2,3,4,5,6,7,8,9,default,non-optimized,static-prorities,
strategy,levels,priority,env-signal_in-threshold,
env-signal_out-threshold,env-agent-threshold,
SCHEDULING, :, =, ;, ε }

Start-Symbol:

⟨sched_annotation⟩

Regeln:

```
⟨sched_annotation⟩      ::=  SCHEDULING : ⟨sched_param_list⟩ |
                             ε
⟨sched_param_list⟩      ::=  ⟨key_val_pair⟩ ; ⟨sched_param_list⟩ |
                             ⟨key_val_pair⟩ |
                             ε
⟨key_val_pair⟩          ::=  ⟨sched_nat_key⟩ = ⟨natural⟩ |
                             ⟨sched_pos_key⟩ = ⟨pos_natural⟩ |
                             ⟨sched_strategy_key⟩ = ⟨sched_strategy_val⟩
⟨sched_pos_key⟩         ::=  levels |
                             env-signal_in-threshold |
                             env-signal_out-threshold |
                             env-agent-threshold
⟨sched_nat_key⟩         ::=  priority
⟨sched_strategy_key⟩    ::=  strategy
⟨sched_strategy_val⟩    ::=  default |
                             non-optimized |
                             static-prorities
⟨pos_natural⟩           ::=  ⟨pos_digit⟩ | ⟨pos_digit⟩ ⟨natural⟩
⟨natural⟩               ::=  ⟨digit⟩ | ⟨digit⟩ ⟨natural⟩
⟨digit⟩                 ::=  0 | ⟨pos_digit⟩
⟨pos_digit⟩             ::=  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Aus Gründen der Übersichtlichkeit geht die Grammatik nicht auf Groß-/Kleinschreibung, Leerzeichen und Zeilenumbrüche ein. Bis auf das Schlüsselwort `SCHEDULING` können alle Terminal-Symbole sowohl groß als auch klein geschrieben werden. Ebenfalls spielt die Anzahl an Leerzeichen und Zeilenumbrüchen zwischen den Tokens keine Rolle.

Bei der Generierung des Quellcodes kontrolliert *ConTraST* die Korrektheit der Syntax. Falsche Annotationen führen zu einer Warnmeldung während des Generierungsschritts.

B.1.2 Schlüssel und Werte

Folgende Aufstellung fasst die möglichen/sinnvollen Werte der Parameter und deren Bedeutung zusammen:

Schlüssel:	`strategy`
Wertebereich:	{ `non-optimized`, `default`, `static-prorities` }
Bedeutung:	Festlegung des Planungsverfahrens.

Schlüssel:	`levels`
Wertebereich:	$\mathbb{N} \setminus \{0\}$
Bedeutung:	Anzahl an Prioritätsstufen.

Schlüssel:	`priority`
Wertebereich:	$\mathbb{N}$
Bedeutung:	Die Verwendung dieses Parameters im Kopf-Symbol auf Systemebene legt die Standardpriorität von Agenten fest, falls keine explizite Priorität (an SDL-Prozess, SDL-Block, ...) zugewiesen wird. Dieser Wert muss kleiner sein als die Anzahl an Prioritätsstufen. Die höchste Priorität entspricht dem Wert 0.

Schlüssel:	`env-signal_in-threshold`
Wertebereich:	$\mathbb{N} \setminus \{0\}$
Bedeutung:	Maximale Anzahl an Signalen, die in der Input-Warteschlange des SDL-Environments liegen dürfen, bevor das Environment ausgeführt werden muss.

Schlüssel:	`env-signal_out-threshold`
Wertebereich:	$\mathbb{N} \setminus \{0\}$
Bedeutung:	Maximale Anzahl an Ereignissen, die einen Interrupt auslösen und ein SDL-Signal erzeugen können, bevor das SDL-Environment ausgeführt werden muss.

Schlüssel:	`env-agent-threshold`
Wertebereich:	$\mathbb{N} \setminus \{0\}$
Bedeutung:	Maximale Anzahl an Ausführungen von SDL-Agenten innerhalb des SDL-Systems bevor das SDL-Environment ausgeführt werden muss.

Bei genauerer Betrachtung ergeben nicht alle Schlüssel-Kombinationen auch einen Sinn. Zum Beispiel ist eine Angabe von `priority` bedeutungslos, wenn das nicht-optimierte Planungsverfahren (`strategy=non-optimized`) verwendet wird. Folgende Tabelle zeigt die sinnvollen Kombinationen.

	`strategy = non-optimized`	`strategy = default`	`strategy = static-priorities`
`levels`			✓
`priority`			✓
`env-signal_in-threshold`		✓	✓
`env-signal_out-threshold`		✓	✓
`env-agent-threshold`		✓	✓

ConTraST prüft bei der Generierung zwar, ob syntaktisch korrekte Annotationen spezifiziert wurden, nicht aber, ob die Kombinationen auch sinnvoll sind. Überflüssige Annotationen werden zur Laufzeit ignoriert, ohne eine Warn- oder Fehlermeldung zu erzeugen. Andernfalls müsste ein Entwickler, der testweise ein anderes Planungsverfahren verwenden wollte, die „falschen" Annotationen entfernen oder auskommentieren, was im Widerspruch zu dem Ziel steht, das Planungsverfahren schnell wechseln zu können. Allerdings gibt *ConTraST* bei semantischen Fehlern, die zu Laufzeitfehlern führen können (zum Beispiel wenn die Standardpriorität höher ist als die Anzahl an Prioritätsstufen), Warnmeldungen aus.

B.2 Annotationen auf Block- und Prozessebene

Bisher wird lediglich die Zuordnung von Prioritäten unterstützt. Durch weitere Planungsverfahren werden auch andere Parameter denkbar (relative Fristen, Ausführungshäufigkeit, ...). Da allerdings zur Zeit nur Prioritäten zugewiesen werden können, wird auf eine kontextfreie Grammatik verzichtet und stattdessen eine informelle Beschreibung geliefert.

Die Definition einer Priorität erfolgt innerhalb eines SDL-Kommentars und wird nur bei dem prioritätsbasierten Planungsverfahren beachtet. Die Syntax hierzu lautet:

```
SCHEDULING:
  priority = ⟨priority_level⟩;
```

Wie bereits bei den Annotationen auf Systemebene wird bei der Verwendung `priority` innerhalb der Systemspezifikation nicht zwischen Groß- und Kleinschreibung unterschieden. Ebenso sind Leerzeichen und das abschließende Semikolon optional. Erhält `priority` einen kleinen Wert, entspricht dies einer hohen Priorität. Der Wert von `priority` muss kleiner sein als die Anzahl an Prioritätsstufen, die im System-Kopf mit `level` spezifiziert wurde. Prinzipiell ist die Vergabe von Prioritäten optional.

Die Zuordnung zwischen einer Priorität und einer SDL-Entität erfolgt mit der von SDL vorgesehenen gestrichelten Kommentarbox. Prioritäten können mit folgenden Entitäten verknüpft werden:

- Service-Typen und Service-Instanzen
- Prozess-Typen und Prozess-Instanzen
- Block-Typen und Block-Instanzen

Die Priorität eines SDL-Agenten wird zur Laufzeit bestimmt und hängt davon ab, welchen in dem Agenten enthaltenen und den Agenten enthaltenden Entitäten eine Priorität zugewiesen wurde (siehe Abschnitt 4.2.5). SDL-Agentenmengen und Linkagenten werden grundsätzlich mit der höchsten Priorität ausgeführt. Allerdings werden diese beiden Arten von ASM-Agenten in der Regel durch entsprechende Optimierungen nicht explizit in der Planung berücksichtigt (siehe Abschnitt 4.2.2).

C. ANHANG

Konfigurationsparameter von BBS

Dieser Abschnitt gibt eine Definition aller Bezeichner, die im Zusammenhang mit BBS in der Arbeit verwendet werden. Außerdem werden alle Konfigurationsparameter des BBS-SDL-Treibers mit Wertebereich und *Default*-Wert aufgeführt. Darüber hinaus fasst der Abschnitt die notwendigen Formeln zur Berechnung der Synchronisationsungenauigkeit und der Konvergenzzeit (Dauer der Synchronisationsphase) zusammen.

C.1 Verwendete Bezeichner

In Kapitel 5 der Arbeit tauchen folgende Bezeichner im Zusammenhang mit BBS auf. Sie stimmen weitgehend mit den Bezeichnern in [GK08] und [Kuh09] überein.

C.1.1 Szenariospezifische Parameter

Bezeichner	**Erklärung**
d_{macro}	Dauer eines Macro-Slots. Der Wert definiert das Resynchronisationsintervall von BBS.
$d_{syncPhase}$	Dauer der (Re-)Synchronisationsphase (Konvergenzzeit)
$n_{maxHops}$	Maximaler Netzwerkdurchmesser in Hops.
d_{round}	Dauer einer BBS-Synchronisationsrunde.
$d_{roundInterval}$	Zeitlicher Abstand zwischen zwei aufeinanderfolgenden Synchronisationsrunden. Der Parameter wurde neu eingeführt, um eine flexiblere Verteilung von Synchronisationsrunden innerhalb eines Macro-Slots zu ermöglichen.
$d_{maxOffset}$	Maximale Synchronisationsungenauigkeit am Ende eines Macro-Slots.

Hinweis: Manche der aufgeführten Parameter sind ebenfalls hardwareabhängig.

C.1.2 Hardwarespezifische Parameter

Bezeichner	Erklärung
$d_{accessTx}$	Umschaltzeit des Transceivers vom Empfangs- in den Sendemodus. Für den CC2420-Transceiver gilt $d_{accessTx} = 192\,\mu$s .
$d_{accessRx}$	Umschaltzeit des Transceivers vom Sende- in den Empfangsmodus. Für den CC2420-Transceiver gilt $d_{accessRx} = 192\,\mu$s , wobei $128\,\mu$s zusätzlich notwendig sind, bis der Transceiver den gültigen Status des Mediums erkennt.
d_{BB_0} , d_{BB_1}	Physikalische Sendedauer eines Black Bursts ohne Beachtung von Umschaltzeiten. Für einen rezessiven Black Burst gilt $d_{BB_0} = 0\,\mu$s . Ein dominanter Black Burst benötigt bei der aktuellen Implementierung (nicht-optimiertes Rahmen-Format) auf dem CC2420-Transceiver $d_{BB_1} = 256\,\mu$s .
$d_{masBurst}$	Dauer eines Black Bursts bei Verwendung der master-basierten BBS-Version. Der Parameter beachtet neben der physikalischen Dauer d_{BB} die Umschaltzeiten des Transceivers. In der Implementierung für den CC2420-Transceiver gilt: $d_{masBurst} = d_{BB_1} + d_{accessTx} + d_{accessRx} + d_{bb_processing} = 840\,\mu$s . ($d_{bb\_processing}$ erlaubt zur Zeit mit $200\,\mu$s relativ viel Zeit für die Bearbeitung, hier sind weitere Optimierungen möglich)
$d_{decBurst}$	Dauer eines „dezentralen“ Black Bursts. Da bisher nur die Implementierung des master-basierten Verfahrens umgesetzt wurde, wird nicht näher auf den Parameter eingegangen.
d_{maxCCA}	Maximale Erkennungsverzögerung einer Medienbelegung, die abhängig vom CCA-Mechanismus des Transceivers ist. Für den CC2420-Transceiver gilt $d_{maxCCA} = 128\,\mu$s .
$r_{clockSkewMax}$	Maximale Abweichung zweier Hardware-Quarze in ppm. Für die verwendete Hardware gilt $r_{clockSkewMax} = 40\,ppm$.

Die konkreten Werte sind dem Datenblatt des CC2420-Transceivers entnommen [Chi07].

C.2 Konfigurationsparameter

Der SDL-Teil des BBS-Treibers bietet den Dienstnutzern ein Interface zur szenariospezifischen Konfiguration des BBS-Protokollverhaltens. Teil des Interfaces sind die beiden folgenden Signale:

- SetParameter(<key>, <value>)
- ResetParameter(<key>)

Der Signalparameter <key> ist dabei vom Typ BBS_CONFIG_KEY, der Parameter <value> vom Typ BBS_CONFIG_VALUE. Die beiden Signale sprechen über den Signalparameter <key> einen spezifischen BBS-Konfigurationsparameter an. Das Signal SetParameter setzt diesen Konfigurationsparameter auf den neuen Wert <value>. ResetParameter setzt den Konfigurationsparame-

ter zurück auf den *Default*-Wert. Die vorhandenen Konfigurationsparameter und deren *Default*-Werte sind weiter unten aufgeführt.

Zu beachten ist, dass der BBS-Treiber Konfigurationsparameter unterschiedlichen Typs besitzt. Beispielsweise lassen sich der Netzwerkdurchmesser $n_{maxHops}$, ein Integer, und d_{macro}, eine Duration, konfigurieren. Daher wäre es eigentlich am sinnvollsten, wenn der Datentyp von <value>, d.h. BBS_CONFIG_VALUE, eine `CHOICE` wäre [Int99]. Da aber `CHOICE` zur Zeit nicht von *ConTraST* unterstützt wird, ist BBS_CONFIG_VALUE als STRUCT definiert. Dies ist zwar ineffizienter bezüglich des Speicherverbrauchs, stellt aber eine bessere Alternative dar als das Definieren mehrerer parameterspezifischer Signale zum Setzen neuer BBS-Konfigurationsparameter.

Der Datentyp BBS_CONFIG_VALUE sieht deshalb folgendermaßen aus:

```
NEWTYPE BBS_CONFIG_VALUE STRUCT
  role BBS_ROLE;
  mode BBS_MODE;
  dur Duration;
  i Integer;
ENDNEWTYPE;
```

```
NEWTYPE BBS_ROLE
LITERALS
  MASTER,
  SLAVE,
  MASTER_IF_REQUIRED
ENDNEWTYPE;
```

```
NEWTYPE BBS_MODE
LITERALS
  MASTER,
  DECENTRALIZED,
  HYBRID
ENDNEWTYPE;
```

Abbildung C.1.: Der Datentyp BBS_CONFIG_VALUE wird im Signal SetParameter als Signalparameter verwendet, um einen Konfigurationsparameter von BBS zu setzen.

In Variablen vom Typ BBS_CONFIG_VALUE ist zu einem Zeitpunkt abhängig von dem neu zu setzenden Konfigurationsparameter, d.h. von <key>, genau ein Element der STRUCT von Interesse. Wird zum Beispiel ein Parameter vom Typ Duration neu gesetzt (beispielsweise d_{macro}), ist dur das relevante Element der STRUCT.

Folgende Aufstellung listet alle Schlüssel der unterstützten BBS-Konfigurationsparameter, die Wertebereiche sowie die *Default*-Werte auf. Die Auflistung enthält außerdem die Elemente aus BBS_CONFIG_VALUE, die bei Setzen des entsprechenden Konfigurationsparameters von Interesse sind.

Schlüssel:	MODE	**Element**:	mode
Default:	MASTER	**Wertebereich**:	BBS_MODE

Der Parameter MODE legt die zu verwendende BBS-Version fest: MASTER, DECENTRALIZED oder HYBRID. Zur Zeit unterstützt die Implementierung nur den Wert MASTER.

Schlüssel:	HOPS	**Element**:	i
Default:	3	**Wertebereich**:	Integer

HOPS legt den Netzwerkdurchmesser fest und entspricht damit $n_{maxHops}$.

Schlüssel:	ROLE	**Element**:	role
Default:	MASTER_IF_REQUIRED	**Wertebereich**:	BBS_ROLE

Der Parameter ROLE definiert die Rolle, die der Knoten einnehmen soll: MASTER, SLAVE oder MASTER_IF_REQUIRED. Wird der Parameter auf MASTER_IF_REQUIRED gesetzt, übernimmt der Knoten die Master-Rolle, wenn er kein bestehendes Netzwerk findet oder der bestehende Master ausfällt. Siehe auch Parameter STARTUP_DELAY.

Schlüssel:	RESYNCH_INTERVAL	**Element**:	dur
Default:	1000 ms	**Wertebereich**:	Duration

RESYNCH_INTERVAL legt die Länge des Macro-Slots fest und entspricht d_{macro}.

Schlüssel:	REQUIRED_VALID_TICKS	**Element**:	i
Default:	5	**Wertebereich**:	Integer

Ein Knoten, der gerade erst aktiviert wurde, synchronisiert sich nicht direkt nach Erhalt eines gültigen Tick-Rahmens auf den assoziierten Tick. Stattdessen wartet er, bis REQUIRED_VALID_TICKS gültige Tick-Rahmen empfangen wurde. Bei dem ersten Tick-Rahmen kann sich der Knoten ausschließlich auf das Paritätsbit verlassen. Bei weiteren Tick-Rahmen prüft er anhand des Empfangszeitpunkts und der empfangenen Rundennummer ebenfalls, ob der empfangene Tick-Rahmen erwartet wurde.

Schlüssel:	TOLERABLE_TICK_MISSES	**Element**:	i
Default:	5	**Wertebereich**:	Integer

Der Parameter TOLERABLE_TICK_MISSES legt die Anzahl an Synchronisationsphasen fest, die ein Slave-Knoten ohne Erhalt von Tick-Rahmen wartet, bevor er nach einem neuen Master-Knoten sucht bzw. sich selbst nach Warten einer von seiner ID abhängigen Zeitspanne zum Master deklariert. Der zweite Fall geschieht nur, wenn die Rolle des Knotens mit MASTER_IF_REQUIRED konfiguriert ist. Weitere Informationen befinden sich in Abschnitt 5.1.4.2.

Schlüssel:	TOLERABLE_UNEXPECTED_TICKS	**Element**:	i
Default:	5	**Wertebereich**:	Integer

Der Parameter spezifiziert die Anzahl an Tick-Rahmen, die zu unerwarteten Zeitpunkten, d.h. außerhalb der Synchronisationsphase, empfangen werden dürfen, bevor eine Fusionierung durchgeführt wird. Nähere Informationen sind in Abschnitt 5.1.4.3 zu finden.

Schlüssel:	STARTUP_DELAY	**Element**:	i
Default:	5	**Wertebereich**:	Integer

Der Parameter STARTUP_DELAY spezifiziert die Anzahl an Macro-Slots, die ein Knoten mit der Rolle MASTER_IF_REQUIRED auf ein bereits synchronisiertes Netzwerk wartet, bevor er sich selbst zum Master-Knoten ernennt.

Schlüssel:	ROUND_INTERVAL	**Element**:	dur
Default:	0 ms	**Wertebereich**:	Duration

Der Parameter ROUND_INTERVAL entspricht $d_{roundInterval}$. Wird er auf einen Wert $> 0\,ms$ gesetzt, verteilen sich die einzelnen BBS-Runden in dem spezifizierten Intervall auf den Macro-Slot.

Die folgenden zwei Beispiele verdeutlichen die Verwendung des Interfaces zur Konfiguration von BBS:

- SetParameter(HOPS, value) mit
  DCL value BBS_CONFIG_VALUE;
  value!i := 5;
  setzt den Netzwerkdurchmesser auf 5 Hops.
- ResetParameter(ROLE) setzt die Funktion des Knotens zurück auf den *Default*-Wert MASTER_IF_REQUIRED.

C.3 Synchronisationsungenauigkeit

Die Synchronisationsungenauigkeit für die master-basierte Version von BBS errechnet sich wie folgt [GK08]:

$$\begin{aligned} d_{maxOffset} &= d_{offsetBase} + d_{offsetClockSkew} \\ &= n_{maxHops} \cdot d_{maxCCA} + 2 \cdot r_{clockSkewMax} \cdot (n_{missedTicks} + 1) \end{aligned}$$

Hierbei ist $d_{offsetBase}$ die initiale Ungenauigkeit nach Abschluss einer erfolgreichen Synchronisationsphase und $d_{offsetClockSkew}$ die Ungenauigkeit durch unterschiedlich schnell laufende Hardware-Quarze, die im Verlauf eines Macro-Slots steigt. $d_{offsetBase}$ hängt ausschließlich von dem Netzwerkdurchmesser $n_{maxHops}$ und der Verzögerung bei der Erkennung der Medienbelegung d_{maxCCA} ab und ist daher konstant. $d_{offsetClockSkew}$ ist neben der Ungenauigkeit der Quarze $r_{clockSkewMax}$, von $n_{missedTicks}$ abhängig, der Anzahl an Synchronisationsphasen, welche seit letzter erfolgreicher Synchronisation ohne Erhalt eines Tick-Rahmens verstrichen sind. Im Regelfall ist $n_{missedTicks} = 0$ und die Ungenauigkeit am Ende des Macro-Slots ist ausschließlich um $2 \cdot r_{clockSkewMax}$ erhöht.

C.4 Konvergenzzeit

Die Dauer der Synchronisationsphase (Konvergenzzeit) $d_{syncPhase}$ muss bei der Planung von Macro-Slots berechnet werden. Allgemein zu beachten ist, dass die Konvergenzzeit $d_{syncPhase}$ in der Macro-Slot-Dauer d_{macro} enthalten ist. Der für die Anwendung nutzbare Anteil des Macro-Slots ist demnach $d_{macro} - d_{syncPhase}$.
Mit den Hardware-Parametern aus Abschnitt C.1 lässt sich $d_{syncPhase}$ für die master-basierte Version von BBS folgendermaßen bestimmen [GK08]:

$$\begin{aligned} d_{syncPhase} &= d_{maxOffset} + n_{maxHops} \cdot d_{round} \\ &= d_{maxOffset} + n_{maxHops} \cdot (n_{bb} \cdot d_{masBurst} + d_{tickProcessing}) \\ &= d_{maxOffset} + n_{maxHops} \cdot ((1 + ld(n_{maxHops} + 1) + 1) \cdot d_{masBurst} + d_{tickProcessing}) \\ &= d_{maxOffset} + n_{maxHops} \cdot ((1 + ld(n_{maxHops} + 1) + 1) \cdot 840\,\mu\text{s} + 600\,\mu\text{s}) \end{aligned}$$

Die Bearbeitungszeit eines Frames $d_{tickProcessing}$ ist mit 600 μs zur Zeit ziemlich großzügig in der Implementierung angegeben. Hier besteht also weiteres Optimierungspotential.

Interessant ist, dass die Synchronisationsungenauigkeit $d_{maxOffset}$ ebenfalls eine Rolle bei der der Dauer der Synchronisationsphase spielt. Dies liegt daran, dass Slave-Knoten theoretisch den Anfang der Synchronisationsphase $d_{maxOffset}$ früher wahrnehmen können als der Master-Knoten. Wie zuvor allerdings berechnet, ist $d_{maxOffset}$ abhängig von der Anzahl an erfolglosen Synchronisationsphasen, ein Faktor, der zur Laufzeit wechseln kann und im Widerspruch zu einem statisch konfigurierten Macro-Slot steht. Um auf „Nummer sicher“ zu gehen, sollte bei der Planung des Macro-Slots der *worst case* der Synchronisationsungenauigkeit angenommen werden, der sich als Produkt aus $d_{maxOffset}$ und dem BBS-Konfigurationsparameter TOLERABLE_TICK_MISSES berechnet. Da allerdings $d_{maxOffset}$ bereits für sich einen *worst case* berechnet,

weil einerseits die maximale Verzögerung bei der Erkennung der Medienbelegung d_{maxCCA} und andererseits der maximale Netzwerkdurchmesser $n_{maxHops}$ in die Berechnung eingeht, und die Synchronisation in der Praxis deutlich bessere Genauigkeiten erzielt (siehe [GK08]), sollte die Beachtung von $2 \cdot d_{maxOffset}$ bei der Planung des Macro-Slots in der Praxis ausreichen.

D. ANHANG

CD

Die CD enthält das vorliegende Dokument im pdf-Format, alle in der Arbeit verwendeten SDL-Systeme und Messdaten, sowie die Werkzeuge *ConTraST*, *SdlRE* und *SEnF*.

Inhalt

- **Arbeit**: Latex-Quellen, Abbildungen, Literaturverzeichnis, Tabellen, sowie das generierte pdf-Dokument.
- **Evaluation**: Messdaten und R-Skripte zur Erzeugung der Plots aus Kapitel 6.
- **SDL**: SDL-Systeme, die in der Arbeit verwendet wurden. Die Betrachtung der Systeme benötigt eine Installation von IBM Tau.
  - **bbs_beispiel**: Kleines SDL-Testsystem, das die Verwendung von BBS anhand eines Beispiels zeigt.
  - **bbs_evaluation**: Alle SDL-Systeme, die für die Evaluation des BBS-Treibers verwendet wurden. Der Ordner enthält die Systeme, die BBS zur Synchronisation verwenden, als auch die Systeme mit beacon-basierter Synchronisation. Die Systeme, die BBS zur Synchronisation verwenden, basieren zu großen Teilen auf dem BBS-Beispielsystem.
  - **scheduling_beispiel**: Mehrere kleine Beispielsysteme, die den Funktionsumfang von TC-SDL zeigen:
    - ResumeSuspend veranschaulicht das Deaktivieren von Agenten mit kleiner Priorität.
    - EnvironmentTest gibt ein Beispiel zur Beeinflussung der Planung bezüglich der Ausführung des Environments.
    - PingPong ist ein sehr einfaches Beispiel, das die Auswirkungen der unterschiedlichen Planungsverfahren zeigt.
    - Testsystem gibt ein erweitertes Beispiel für die prioritätsbasierte Planung, welches die Möglichkeiten der Vergabe von Prioritäten an die unterschiedlichen SDL-Objekte (SDL-Prozess-Typ,SDL-Prozess-Instanz,SDL-Service-Typ,...) aufzeigt.
  - **scheduling_evaluation**: SDL-System, das zur Evaluation der Planungsverfahren verwendet wurde.

  - **scheduling_sdlreTestsystem**: Mit von TC-SDL eingeführten Annotationen erweitertes SDL-System, welches zur Evaluation der Laufzeitumgebung bereits vor diese Master-Arbeit in der AG Vernetzte Systeme entwickelt wurde [Netar]. Das System nutzt einen großen Teil des Sprachumfangs von SDL (Vererbung, dynamische Prozesserzeugung, ...) und eignet sich daher gut, um die richtige Zuordnung von Prioritäten zu Agenten zu testen.

  Des Weiteren enthält der Ordner:

  - **common**: Hilfsprozeduren (Ausgabe von Statistiken über UART, etc.), die in mehreren SDL-Systemen benutzt werden.
  - **config**: Java-Klassen mit den Konfigurationen für alle Knoten, die zur Erstellung der Beispiel- und Evaluationssysteme benötigt werden. Die Java-Klassen müssen gemäß den Anforderungen des SDL-Config-Interfaces weiterverarbeitet werden.

- **Werkzeuge**: Die Toolchain zur Erzeugung eines lauffähigen SDL-Systems:
  - **ConTraST**: Generierung von C++-Code aus SDL/PR-Dateien.
  - **SdlRE**: Laufzeitumgebung zur Ausführung von ASM-Agenten. Der Ordner enthält das Framework von Planungsverfahren und das prioritätsbasierte Verfahren, welche in Kapitel 4 vorgestellt wurden.
  - **SEnF**: Plattformspezifische Treiber und Unterstützung der Laufzeitumgebung (Imote2, Linux, ...). Hier ist der BBS-SEnF-Treiber aus Kapitel 5 zu finden.

Literaturverzeichnis

[ÁDL+03] ÁLVAREZ, JOSÉ M., MANUEL DÍAZ, LUIS LLOPIS, ERNESTO PIMENTEL und JOSÉ M. TROYA: *Integrating Schedulability Analysis and Design Techniques in SDL*. Real-Time Systems, 24(3):267–302, 2003.

[Apaar] APACHE SOFTWARE FOUNDATION: *HTTP Server*. `http://httpd.apache.org/`, 2010.

[ASSC02] AKYILDIZ, IAN F., WEILIAN SU, YOGESH SANKARASUBRAMANIAM und ERDAL CAYIRCI: *Wireless sensor networks: a survey*. Computer Networks, 38(4):393–422, 2002.

[BB90] BAUSE, FALKO und PETER BUCHHOLZ: *Protocol Analysis Using a Timed Version of SDL*. In: QUEMADA, JUAN, JOSÉ A. MAÑAS und ENRIQUE VÁZQUEZ (Herausgeber): *FORTE*, Seiten 239–254. North-Holland, 1990.

[BB93] BAUSE, FALKO und PETER BUCHHOLZ: *Qualitative und Quantitative Analysis of Timed SDL Specifications*. In: GERNER, NINA, HEINZ-GERD HEGERING und JOACHIM SWOBODA (Herausgeber): *Kommunikation in Verteilten Systemen*, Informatik Aktuell, Seiten 486–500. Springer, 1993.

[BCG09] BECKER, PHILIPP, DENNIS CHRISTMANN und REINHARD GOTZHEIN: *Model-Driven Development of Time-Critical Protocols with SDL-MDD*. In: REED, RICK, ATTILA BILGIC und REINHARD GOTZHEIN (Herausgeber): *14th International SDL Forum*, Band 5719 der Reihe *Lecture Notes in Computer Science*, Seiten 34–52. Springer, 2009.

[BGK+00] BOZGA, MARIUS, SUSANNE GRAF, ALAIN KERBRAT, LAURENT MOUNIER, IULIAN OBER und DANIEL VINCENT: *SDL for Real-Time: What is Missing?* In: SHERRATT, EDEL (Herausgeber): *SAM*, Seiten 108–121. VERIMAG, IRISA, SDL Forum, 2000.

[BGK07] BECKER, PHILIPP, REINHARD GOTZHEIN und THOMAS KUHN: *MacZ – A Quality-of-Service MAC Layer for Ad-hoc Networks*. In: *HIS '07: Proceedings of the 7th International Conference on Hybrid Intelligent Systems*, Seiten 277–282. IEEE Computer Society, Sept. 2007.

[BGK08] BECKER, PHILIPP, REINHARD GOTZHEIN und THOMAS KUHN: *Model-driven Performance Simulation of Self-organizing Systems with PartsSim*. Praxis der Informationsverarbeitung und Kommunikation, Seiten 45–50, 1/2008.

[BGM+01] BOZGA, MARIUS, SUSANNE GRAF, LAURENT MOUNIER, IULIAN OBER, JEAN-LUC ROUX und DANIEL VINCENT: *Timed Extensions for SDL*. In: REED, RICK und JEANNE REED (Herausgeber): *SDL Forum*, Band 2078 der Reihe *Lecture Notes in Computer Science*, Seiten 223–240. Springer, 2001.

[BH93] BRÆK, ROLV und ØYSTEIN HAUGEN: *Engineering Real Time Systems*. Prentice Hall, 1993.

[BK86] BERGSTRA, J A und J W KLOP: *Conditional rewrite rules: Confluence and termination*. J. Comput. Syst. Sci., 32(3):323–362, 1986.

[BS03] BÖRGER, EGON und ROBERT STÄRK: *Abstract State Machines - A Method for High-Level System Design and Analysis*. Springer, 2003.

[But05] BUTTAZZO, GIORGIO C.: *Hard real-time computing systems*. Springer, New York, NY, 2. ed. Auflage, 2005.

[CGK09] CHRISTMANN, DENNIS, REINHARD GOTZHEIN und THOMAS KUHN: *Multi-hop Clock Synchronization in Wireless Ad-Hoc Networks*. In: *Proceedings of Workshops on Mobile Ad-Hoc Networks (WMAN 2009), Kassel, Germany*, March 2009.

[Chi07] CHIPCON / TEXAS INSTRUMENTS: *CC2420 datasheet*. `http://focus.ti.com/lit/ds/symlink/cc2420.pdf`, 2007. Revision SWRS041b.

[Chr09] CHRISTMANN, DENNIS: *Duty Cycling in drahtlosen Multi-Hop-Netzwerken*. Technischer Bericht 372/09, TU Kaiserslautern, 2009.

[Cinar] CINDERELLA APS: *Cinderella*. `http://www.cinderella.dk/`, 2010.

[Cro07] CROSSBOW: *MICAz Datasheet*. `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf`, 2007. Revision B.

[Cro09] CROSSBOW: *Imote 2 Datasheet*. `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Imote2_Datasheet.pdf`, 2009.

[DHMC95] DIEFENBRUCH, MARC, JÖRG HINTELMANN und BRUNO MÜLLER-CLOSTERMANN: *QSDL: Sprache und Werkzeuge zur Leisungsanalyse von SDL-Systemen*. In: *Tagungsband des 5. GI/ITG-Fachgespräch: Formale Beschreibungstechniken für verteilte Systeme*, june 1995.

[DHMC97] DIEFENBRUCH, MARC, JÖRG HINTELMANN und BRUNO MÜLLER-CLOSTERMANN: *QUEST Performance Evalution of SDL System*. In: IRMSCHER, KLAUS, CHRISTIAN MITTASCH und KLAUS RICHTER (Herausgeber): *MMB (Kurzbeiträge)*, Seiten 126–132. TU Bergakademie Freiberg, 1997.

[Dij65] DIJKSTRA, EDSGER WYBE: *Cooperating Sequential Processes*. Technischer Bericht, Technological University, Eindhoven, The Netherlands, 1965.

[EGE02] ELSON, JEREMY, LEWIS GIROD und DEBORAH ESTRIN: *Fine-Grained Network Time Synchronization Using Reference Broadcasts*. In: *OSDI*. Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, MA, USA, 2002.

[EGG+01] ESCHBACH, ROBERT, UWE GLÄSSER, REINHARD GOTZHEIN, MARTIN VON LÖWIS und ANDREAS PRINZ: *Formal Definition of SDL-2000 - Compiling and Running SDL Specifications as ASM Models*. J. UCS, 7(11):1024–1049, 2001.

[Ele10] ELEKTRONIK PRAXIS: *Neutrino Safe Kernel ist jetzt SIL3-Zertifiziert*. `http://www.elektronikpraxis.vogel.de/embedded-computing/articles/274156/?icmp=aut-artikel-artikel-50`, 20.7.2010, letzter Aufruf 4.8.2010.

[FGJ+05] FLIEGE, INGMAR, ALEXANDER GERALDY, SIMON JUNG, THOMAS KUHN, CHRISTIAN WEBEL und CHRISTIAN WEBER: *Konzept und Struktur des SDL Environment Framework (SEnF)*. Technischer Bericht 341/05, TU Kaiserslautern, 2005.

[FGW06] FLIEGE, INGMAR, RÜDIGER GRAMMES und CHRISTIAN WEBER: *ConTraST – A Configurable SDL Transpiler and Runtime Environment*. In: GOTZHEIN, REINHARD und RICK REED (Herausgeber): *System Analysis and Modeling: Language Profiles, 5th International Workshop, SAM 2006, Kaiserslautern, Germany, May 31 – June 2, 2006, Revised Selected Papers*, Band 4320 der Reihe *Lecture Notes in Computer Science*, Seiten 216–228. Springer, 2006.

[Fli09] FLIEGE, INGMAR: *Component-based Development of Communication Systems*. Doktorarbeit, University of Kaiserslautern, 2009.

[Ger97] GERLICH, RAINER: *Tuning development of distributed real-time systems with SDL: Current experience and future issues*. In: CAVALLI, ANA R. und AMARDEO SARMA (Herausgeber): *SDL Forum*, Seiten 85–100. Elsevier, 1997.

[GHJV09] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston [u.a.], 37. print. Auflage, 2009.

[GK08] GOTZHEIN, REINHARD und THOMAS KUHN: *Decentralized Tick Synchronization for Multi-Hop Medium Slotting in Wireless Ad Hoc Networks Using Black Bursts*. In: *Proceedings of the Fifth Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, SECON 2008, June 16-20, 2008, Crowne Plaza, San Francisco International Airport, California, USA*, Seiten 422–431. IEEE, 2008.

[Got07] GOTZHEIN, REINHARD: *Model-driven with SDL – Improving the Quality of Networked Systems Development (Invited Paper)*. In: *Proceedings of the 7th International Conference on New Technologies of Distributed Systems (NOTERE 2007), Marrakesh, Morocco*, Seiten 31–46, June 4-8 2007.

[GP05] GRAF, SUSANNE und ANDREAS PRINZ: *Time in State Machines*. In: *Abstract State Machines*, Seiten 217–232, 2005.

[GS88] GOODENOUGH, J. B. und L. SHA: *The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks*. In: *IRTAW '88: Proceedings of the second international workshop on Real-time Ada issues*, Seiten 20–31, New York, NY, USA, 1988. ACM.

[HE00] HAARTSEN, JAAP C. und ERICSSON RADIO SYSTEMS B. V.: *The Bluetooth radio system*. IEEE Personal Communications, 7:28–36, 2000.

[Hoa74] HOARE, C. A. R.: *Monitors: an operating system structuring concept*. Commun. ACM, 17(10):549–557, 1974.

[IBMar] IBM: *Rational SDL Suite*. `http://www-01.ibm.com/software/awdtools/sdlsuite/`, 2010.

[IEE99] IEEE STD. 802.11: *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE Computer Society, 1999.

[IEE03] IEEE STD. 802.15.4: *Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*. IEEE Computer Society, New York, NY, USA, Oct. 2003.

[Int99] INTERNATIONAL TELECOMMUNICATION UNION (ITU): *ITU-T Recommendation Z.100 (11/99): Specification and Description Language (SDL)*. `http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf`, 1999.

[Int00] INTERNATIONAL TELECOMMUNICATION UNION (ITU): *ITU-T Recommendation Z.100 Annex F: Formal Semantics Definition*, 2000.

[Int04] INTERNATIONAL TELECOMMUNICATION UNION (ITU): *ITU-T Recommendation Z.120 (04/2004): Message Sequence Charts (MSC)*. `http://www.itu.int/ITU-T/2005-2008/com17/languages/Z120.pdf`, 2004.

[Jai91] JAIN, RAJ: *The Art of Computer Systems Performance Analysis*. WILEY Professional Computing, 1991.

[KdI07] KUHN, THOMAS und JOSÉ IRIGON DE IRIGON: *An experimental evaluation of black burst transmissions*. In: ZOMAYA, ALBERT Y. und SHERALI ZEADALLY (Herausgeber): *MOBIWAC*, Seiten 163–167. Proceedings of the Fifth ACM International Workshop on Mobility Management & Wireless Access, MOBIWAC 2007, Chania, Crete Island, Greece, October 2007.

[KF98] KOLLOCH, THOMAS und GEORG FÄRBER: *Mapping an Embedded Hard Real-Time Systems SDL Specification to an Analyzable Task Network - A Case Study*. In: MUELLER, FRANK und AZER BESTAVROS (Herausgeber): *LCTES*, Band 1474 der Reihe *Lecture Notes in Computer Science*, Seiten 156–165. Springer, 1998.

[Kop91] KOPETZ, HERMANN: *Event-Triggered Versus Time-Triggered Real-Time Systems*. In: KARSHMER, ARTHUR I. und JÜRGEN NEHMER (Herausgeber): *Operating Systems of the 90s and Beyond*, Band 563 der Reihe *Lecture Notes in Computer Science*, Seiten 87–101. Springer, 1991.

[Kop97] KOPETZ, HERMANN: *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[Kuh09] KUHN, THOMAS: *Model Driven Development of MacZ – A QoS Medium Access Control Layer for Ambient Intelligence Systems*. Doktorarbeit, University of Kaiserslautern, 2009.

[LEH00] LEBLANC, PHILIPPE, ANDERS EK und THOMAS HJELM: *Telelogic SDL and MSC Tool Families*. In: *Telektronikk 4.2000, Languages for Telecommunication Applications*. Telenor, 2000.

[LL73] LIU, C. L. und JAMES W. LAYLAND: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. J. ACM, 20(1):46–61, 1973.

[MT00] MITSCHELE-THIEL, ANDREAS: *Engineering with SDL - Developing Performance-Critical Communication Systems*. John Wiley & Sons, 2000.

[Netar] NETWORKED SYSTEMS GROUP: *Homepage*. `http://vs.cs.uni-kl.de`, 2010.

[NS01] NEHMER, JÜRGEN und PETER STURM: *Systemsoftware – Grundlagen moderner Betriebssysteme*. dpunkt.verlag Heidelberg, 2001.

[OK01] OBER, IULIAN und ALAIN KERBRAT: *Verification of Quantitative Temporal Properties of SDL Specifications*. In: *SDL '01: Proceedings of the 10th International SDL Forum Copenhagen on Meeting UML*, Seiten 182–202, London, UK, 2001. Springer-Verlag.

[OMG10] OMG UNIFIED MODELLING LANGUAGE SPECIFICATION: *Version 2.3*, 2010. www.uml.org.

[Praar] PRAGMADEV: *Real Time Developer Studio*. `http://www.pragmadev.com/`, 2010.

[Pre10] PRESSEECHO: *Echtzeit-Ethernet in der Automation*. `http://www.presseecho.de/vermischtes/PE12801301279387.htm`, 26.7.2010, letzter Aufruf 4.8.2010.

[Pri00] PRINZ, ANDREAS: *Formal Semantics of Specification Languages*. In: *Telektronikk 4.2000, Languages for Telecommunication Applications*. Telenor, 2000.

[PST07] PRINZ, ANDREAS, MARKUS SCHEIDGEN und MERETE SKJELTEN TVEIT: *A Model-Based Standard for SDL*. In: GAUDIN, EMMANUEL, ELIE NAJM und RICK REED (Herausgeber): *SDL Forum*, Band 4745 der Reihe *Lecture Notes in Computer Science*, Seiten 1–18. Springer, 2007.

[PvL03a] PRINZ, ANDREAS und MARTIN VON LÖWIS: *Engineering the SDL Formal Language Definition*. In: NAJM, ELIE, UWE NESTMANN und PERDITA STEVENS (Herausgeber): *FMOODS*, Band 2884 der Reihe *Lecture Notes in Computer Science*, Seiten 47–63. Springer, 2003.

[PvL03b] PRINZ, ANDREAS und MARTIN VON LÖWIS: *Generating a Compiler for SDL from the Formal Language Definition*. In: REED, RICK und JEANNE REED (Herausgeber): *SDL Forum*, Band 2708 der Reihe *Lecture Notes in Computer Science*, Seiten 150–165. Springer, 2003.

[Ree00] REED, RICK: *The Evolution of SDL-2000*. In: *Telektronikk 4.2000, Languages for Telecommunication Applications*. Telenor, 2000.

[Ree09] REED, RICK: *SDL-2010 Route Map (TAF08)*. `http://www.sdl-forum.org/ftp/pub/SDL-2010/T09-SG17-090916-TD-PLEN-0469!!MSW-E.doc`, september 2009.

[Resar] RESEARCH, MICROSOFT: *AsmL: Abstract State Machine Language*. `http://research.microsoft.com/en-us/projects/asml/`, 2010.

[San00] SANDERS, RICHARD: *Implementing from SDL*. In: *Telektronikk 4.2000, Languages for Telecommunication Applications*. Telenor, 2000.

[SRL87] SHA, LUI, RAGUNATHAN RAJKUMAR und JOHN P. LEHOCZKY: *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*. (CMU-CS-87-181), 1987.

[SRL90] SHA, LUI, RAGUNATHAN RAJKUMAR und JOHN P. LEHOCZKY: *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*. IEEE Trans. Computers, 39(9):1175–1185, 1990.

[SSH92] SANDER, PETER, WOLFFRIED STUCKY und RUDOLF HERSCHEL: *Automaten, Sprachen, Berechenbarkeit*. Teubner, Stuttgart, 1992.

[Sta88] STANKOVIC, JOHN A.: *Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems*. Computer, 21(10):10–19, 1988.

[Sta92] STANKOVIC, JOHN A.: *Real-Time Computing*, 1992.

[Tan09] TANENBAUM, ANDREW S.: *Moderne Betriebssysteme*. Pearson Studium, 3. aktualisierte Auflage Auflage, 2009.

[The10] THE H OPEN: *Concurrent RedHawk Embedded real-time Linux version 5.4 released*. `http://www.h-online.com/open/news/item/Concurrent-RedHawk-Embedded-real-time-Linux-version-5-4-released-874295.html`, 2.12.2009, letzter Aufruf 4.8.2010.

[Zöb08] ZÖBEL, DIETER: *Echtzeitsysteme*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2008.