Ein Toolkit für den Entwurf softwarebasierter Empfangssysteme

Jan Zöllner, Jörg Robert, Daniel Rother, Mariem Slimani, Institut für Nachrichtentechnik, TU Braunschweig, Braunschweig, Deutschland, {zoellner, robert, rother, slimani}@ifn.ing.tu-bs.de

Kurzfassung

Software-Defined-Radio bezeichnet den Ansatz, Empfangssysteme so weit wie möglich in Software zu realisieren. Dabei ist im Idealfall das Empfangsfrontend mit Antenne, welches das Empfangssignal ins Basisband mischt und in ein Digitalsignal wandelt, die einzige Hardwarekomponente. Die Algorithmen zur Demodulation und Decodierung der Daten werden komplett in Software realisiert. In Verbindung mit einem leistungsfähigen PC ist dies eine flexible und günstige Möglichkeit, um Empfangssysteme zu entwickeln und zu analysieren. Wegen der geringen Entwicklungskosten und der hohen Flexibilität im Vergleich zum Entwurf in Hardware sind Sofware-Defined-Radios insbesondere in der Forschung und Entwicklung zu einem festen Bestandteil geworden. Um den Entwurf softwarebasierter Empfänger zu erleichtern, wurde am IfN der Technischen Universität Braunschweig ein auf C++ basiertes Toolkit entwickelt, das die Anforderungen für die Entwicklung von echtzeitfähigen und datendominierten Systemen für Mehrkernarchitekturen berücksichtigt. Dabei wird das zu entwickelnde System mit Hilfe des Toolkits als Modell aus mehreren Verarbeitungsblöcken mit Ein- und Ausgangsströmen abgebildet. Die Nutzung des Toolkits wird durch eine graphische Oberfläche mit automatischer Codeerzeugung vereinfacht.

1. Einleitung

Durch die rapide Steigerung der Leistungsfähigkeit von Desktop-PCs in den letzten Jahrzehnten hat sich der Zugang zu Rechnern mit hoher Leistungsfähigkeit deutlich vereinfacht. Dies ermöglicht beispielsweise die Simulation komplexer Kommunikationssysteme in Software mit Tools wie MATLAB oder Simulink. Aber auch die Entwicklung im Bereich moderner Empfangssysteme hat sich durch die Anwendung des Software-Defined-Radio (SDR)-Ansatzes maßgeblich verändert. Dabei wird versucht den wesentlichen Teil des Systems in Software abzubilden. Das Empfangsfrontend mit Antenne, welches das Empfangssignal ins Basisband mischt und in ein Digitalsignal wandelt, ist dann die einzige Hardwarekomponente. Dieser Ansatz ist beispielhaft in Abbildung 1 dargestellt. Im Fall eines Echtzeitsystems werden die Basisbanddaten direkt im Arbeitsspeicher verarbeitet, im Fall der Offline-Verarbeitung werden sie zunächst auf einer Festplatte zwischengespeichert und später ohne Echtzeitanforderung decodiert. Wegen der geringen Entwicklungskosten und der hohen Flexibilität im Vergleich zum Entwurf in Hardware, sind SDRs insbesondere in der Forschung und Entwicklung zu einem festen Bestandteil geworden. Weitere Vorteile des SDR-Ansatzes sind eine deutlich reduzierte Entwicklungszeit im Vergleich zur Hardware-Entwicklung, aber auch die Möglichkeit, verschiedene Algorithmen auf Grundlage derselben Eingangsdaten vergleichen zu können.

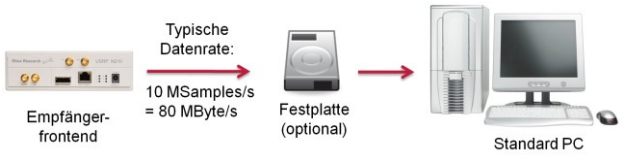

Bild 1 Empfangssystem auf Basis des Software-Defined-Radio Ansatzes

Aus diesen Gründen wurde am Institut für Nachrichtentechnik (IfN) der Technischen Universität Braunschweig auf Grundlage des SDR-Ansatzes ein DVB-C2-Echtzeitempfänger [4] auf Basis von C++ [13] entwickelt. Die Implementierung von Empfängern komplexer Übertragungssysteme wie DVB-C2 erfordert jedoch ein hohes Maß an Erfahrung in Softwaredesign, um effizienten und gut strukturierten Quelltext zu erzeugen und die Entwicklungszeit auf die eigentliche Aufgabe, nämlich die Entwicklung von Empfängeralgorithmen, zu konzentrieren. Dies gilt insbesondere dann, wenn Echtzeitbedingungen bei der Decodierung eingehalten werden sollen, die in der Regel die Nutzung paralleler Prozesse erfordern, um die Mehrkernarchitekturen aktueller Prozessoren auszunutzen. Um den Entwurf softwarebasierter Empfänger zu erleichtern, wurde am IfN ein auf C++ basiertes Toolkit entwickelt, das diese Besonderheiten bei der Entwicklung von Empfangssystemen berücksichtigt, die durch die Verarbeitung großer Datenmengen gekennzeichnet sind (sog. datendominierte Systeme). Das Toolkit bildet das System dabei als Modell aus mehreren Verarbeitungsblöcken ab, die jeweils einer Teilmenge des Systems entsprechen. Die zu verarbeitenden Daten der Verarbeitungsblöcke werden in Paketen gespeichert und über Verbindungen zwischen den Blöcken ausgetauscht. Auf Grundlage des Toolkits wurde der erste mobilfähige DVB-T2-Empfänger entwickelt [5], [6], der im Rahmen des DVB-T2-Modellversuchs in Norddeutschland [7] zur Auswertung der Mobilmessungen eingesetzt wurde.

Der Rest des Papers ist wie folgt gegliedert: In Abschnitt II wird auf die unterschiedlichen Anforderungen an ein SDR-Toolkit eingegangen. Das entwickelte Toolkit und die graphische Oberfläche werden dann ausführlich in Abschnitt III und IV vorgestellt. Anschließend wird das Paper in Abschnitt V zusammengefasst.

2. Anforderungen

Für den Entwurf softwarebasierter Empfangssysteme gelten eine Vielzahl von Anforderungen, die bei der Entwicklung eines SDR-Toolkits gleichermaßen berücksichtigt werden müssen. Diese werden im Folgenden vorgestellt.
Um die zentrale Anforderung der Echtzeitfähigkeit zu erfüllen, sind verschiedene Aspekte zu berücksichtigen. Da sich in den letzten Jahren mit der Sättigung der erreichbaren Taktfrequenzen von aktuellen Prozessoren ein Trend zur Verwendung von Mehrkernarchitekturen ergeben hat, steht beim Erreichen einer hohen Systemperformance datendominierter Systeme insbesondere die Ausnutzung paralleler Prozessorarchitekturen durch Multithreading [8] im Vordergrund. Darüber hinaus werden von aktuellen Prozessoren auch sogenannte SIMD (Single Instruction, Multiple Data) Befehle [9] unterstützt, die in einem Takt mehrere Operationen des gleichen Typs - z.B. eine Addition oder eine Multiplikation - auf unterschiedlichen Daten gleichzeitig ausführen können. Diese Befehlssätze werden unter Namen wie “SSE”, “MMX”, oder “AVX” [10] angeboten. AVX erlaubt dabei durch die Verwendung von 256-Bit breiten Registern die Ausführung von acht 32-Bit Floating-Point-Operationen in einem Taktzyklus. Derartige SIMD-Befehle erlauben insbesondere bei Vektoroperationen große Geschwindigkeitsgewinne und sind ebenfalls bei der Realisierung des Toolkits zu berücksichtigen.
Für den Entwurf des Toolkits existierten zudem Anforderungen, die aus der Historie bisheriger Entwicklungen am IfN gewachsen sind. Da im Rahmen eines am IfN verwendeten Systemsimulators bereits wesentliche Elemente verschiedener DVB-Systeme in C++ implementiert wurden, lag zum einen wegen der gewünschten Wiederverwendung dieser Elemente, aber auch zum anderen wegen der guten Performance die Verwendung von C++, nahe.
Der einfache Umgang mit dem Toolkit bei der Entwicklung von Verarbeitungsblöcken und dem daraus entstehenden Gesamtsystemen ist dabei essentiell. Dementsprechend sollte die Verwendung des Toolkits so generisch wie möglich sein, was zum Beispiel durch den objektorientierten Entwurf mit Mustern (Templates) ermöglicht wird. Der Entwickler sollte darüber hinaus bei der Nutzung des Toolkits mit einer graphischen Oberfläche unterstützt werden.

3. SDR-Toolkit

Im Folgenden werden die technischen Aspekte des C++-Toolkits detailliert beschrieben. Das zu entwickelnde Gesamtsystem wird mit dem Toolkit als Modell aus mehreren Verarbeitungsblöcken mit Ein- und Ausgangsströmen abgebildet. Jeder Block entspricht dabei einer Teilmenge des Modells, zum Beispiel einem bestimmten Algorithmus, wie der Fast-Fourier-Transformation (FFT). Der Datenaustausch zwischen den Blöcken geschieht paketbasiert über gepufferte Verbindungen zwischen den Blöcken. Die Blöcke, Verbindungen und Pakete werden mit Hilfe von abstrakten Template-Klassen in C++ abgebildet, von denen jeder konkrete Block, jede Verbindung oder jedes Paket erbt. Die Mutterklassen beinhalten unter anderem die Mechanismen zur Kommunikation zwischen den Blöcken und stellen viele weitere Funktionalitäten zur Verfügung.
Ziel des Toolkits ist es dabei, die Entwicklung eines Modells deutlich zu vereinfachen, indem die Entwickler sich nicht mehr mit der Kommunikation zwischen den Teilblöcken des Modells auseinandersetzen müssen. Darüber hinaus wird die Wiederverwendbarkeit implementierter Algorithmen vereinfacht, indem klare Schnittstellen zwischen den Blöcken definiert werden (sog. Kapselung).

3.1. Verarbeitungsblöcke

Für die Entwicklung von Verarbeitungsblöcken, im Folgenden auch einfach Blöcke genannt, stellt das Toolkit mehrere Mutterklassen zur Verfügung. Dabei wird zwischen Quellen, Sinken und Verarbeitungsblöcken unterschieden, die sich durch die Anzahl der Eingangs- und Ausgangsports unterscheiden. Eine Quelle hat dabei einen oder mehrere Ausgangsports jedoch keine Eingangsports und wird zum Einlesen von Daten in das Modell genutzt. Beispiele dafür sind Quellblöcke zum Einlesen von Paketen aus Dateien oder über Ethernet. Analog dazu sind Sinken definiert: Sie enthalten lediglich Eingangsports und werden zum Auslesen von Paketdaten aus dem Modell genutzt. Verarbeitungsblöcke enthalten sowohl Eingangs- als auch Ausgangsports und ermöglichen die Verarbeitung der Daten innerhalb des Modells. Ein einfaches Modell, bestehend aus einer Quelle, einem Verarbeitungsblock und einer Sinke, ist beispielhaft in Abbildung 2 dargestellt.

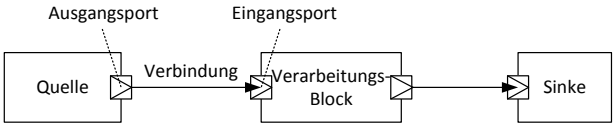

Bild 2 Abbildung eines Modells mit Quelle, Verarbeitungsblock und Sinke, die über Verbindungen an ihren Eingangs- und Ausgangsports Daten austauschen.

Jeder Block enthält einen oder mehrere Threads, die die eigentliche Datenverarbeitung enthalten und innerhalb des Modells parallel ausgeführt werden, so dass Mehrkernarchitekturen optimal ausgenutzt werden können. Als Beispiel für eine Mutterklasse eines Verarbeitungsblocks repräsentiert die Klasse *CProcessingBlock* einen Block mit einem Thread und jeweils einem Eingangs- und Ausgangsport. Weitere Mutterklassen erlauben den Entwurf komplexerer Blöcke. Sämtliche Klassen des Toolkits sind dabei als sog. Templateklassen implementiert, die als Templateparameter den verwendeten Pakettyp übergeben.
Das Beispiel der Implementierung eines einfachen Verarbeitungsblocks, der Pakete mit IQ-Basisbanddaten einliest, sie auf die Leistung 1 normalisiert und anschließend wieder ausgibt, ist in Abbildung 3 dargestellt. Der Pakettyp, der IQ-Basisbanddaten repräsentiert, heißt *CSamplesPacket* und wird in der ersten Zeile für die Definition des Eingangs- und Ausgangspakettyps des Blocks

als Templateparameter übergeben. Die *computeThread* Methode enthält die eigentliche Verarbeitung des Blocks und wird beim Start des Modells aufgerufen. In diesem Beispiel werden in einer Endlosschleife Pakete vom Eingangsport eingelesen, die Pakete anschließend normiert und wieder über den Ausgangsport ausgegeben.

```
class CNormalizeSamples
    : public CProcessingBlock<CSamplesPacket,CSamplesPacket> {

    void computeThread(void) {

        while (true) {
            // read packet from input port
            shared_ptr<CSamplesPacket> InputPacket;
            PacketSource_p->readData(packet_ps);

            // normalize packet and write to output port
            shared_ptr<CSamplesPacket> OutputPacket;
            OutputPacket = InputPacket;
            OutputPacket->normalize();
            writeData(OutputPacket);
        }
    }
};
```

Bild 3 Codebeispiel eines einfachen Verarbeitungsblocks zum Normalisieren von Basisbanddaten (Includes wurden zur Erhöhung der Übersichtlichkeit weggelassen)

Jeder Block erbt darüber hinaus von seiner Mutterklasse zahlreiche Methoden, die jedem Block ein Interface zur Verfügung stellen und vom Modell aufgerufen werden, das den Block beinhaltet. Dies sind zum Beispiel Methoden zum initialisieren, starten und stoppen der Verarbeitung in dem Block sowie Methoden zum Verbinden der Eingangs- und Ausgangsports des Blocks mit anderen Blöcken.

3.2. Verbindungen

Der Datenaustausch zwischen zwei Blöcken wird mit Hilfe von Verbindungen realisiert (siehe Abbildung 2), die im Toolkit durch die *UnidirectionalLink* Klasse abgebildet werden. Verbindungen entsprechen dabei im Wesentlichen einem Puffer nach dem first-in-first-out-(FIFO)-Prinzip, der die zu übertragenden Pakete zwischenspeichert und den Zugriff auf die Pakete mit Methoden für den Lese- und Schreibzugriff ermöglicht. Da von den verbundenen Blöcken aus unterschiedlichen Threads auf den Puffer der Verbindung zugegriffen wird, müssen die Lese- und Schreibvorgänge des Blocks synchronisiert werden. Dies wird mit Hilfe von sog. Mutexen realisiert, die in der boost-Bibliothek [12] enthalten sind. Ein Mutex ist ein Kontrollelement, das den gleichzeitigen Zugriff von verschiedenen Threads unterbindet und so die Synchronisation gewährleistet. Für den Zugriff auf die Pakete des Puffers sind vier Methoden verfügbar, die blockierend oder nicht-blockierend, lesend oder schreibend auf den Puffer der Verbindung zugreifen und aus dem Thread des Blocks aufgerufen werden (siehe Abbildung 3). Bei Schreiboperationen wird dabei immer an den Anfang des Puffers geschrieben, bei Leseoperationen immer das letzte Element des Puffers ausgelesen, so dass die Reihenfolge der übertragenen Pakete nach dem FIFO-Prinzip erhalten bleibt. Im Fall einer blockierenden Lese- oder Schreiboperation wird der Thread aus dem der Zugriff aufgerufen wird, so lange blockiert, bis ein Paket erfolgreich in den Puffer gelesen oder geschrieben wurde. Im Fall einer nicht-blockierenden Operation wird unmittelbar zum Thread zurückgekehrt und mit Hilfe des Rückgabewerts der Methode übergeben, ob die Lese- oder Schreiboperation erfolgreich war.

Die Pakete werden im Puffer der Verbindung mit Hilfe sog. Shared-Pointer gespeichert (siehe Abbildung 3). Shared-Pointer sind Zeiger auf Objekte, in diesem Fall Zeiger auf die Pakete im Puffer, die den Zugriff von mehreren Threads vereinfachen. Shared-Pointer haben die Eigenschaft, die referenzierten Pakete so lange im Speicher zu halten, bis diese von jedem verbundenen Block erfolgreich gelesen wurden. Anschließend werden die Pakete automatisch aus dem Speicher gelöscht. Auf diese Weise werden Pakete nur so lange wie notwendig im Speicher gehalten, ohne dass der Entwickler Einfluss auf die Speicherverwaltung nehmen muss. Darüber hinaus müssen keine Kopieroperationen auf den Paketen vorgenommen werden, die sich negativ auf die Laufzeit auswirken würden.

Der verwendete Puffer ist dabei ein auf der Basis der Vector-Klasse der C++-Standardbibliothek selbst entwickelter Datencontainer. Die Besonderheit dieses Containers im Vergleich zur Vector-Klasse der C++-Standardbibliothek ist dabei, dass sämtliche Daten der Pakete 16-Byte-Aligned im Speicher abgelegt werden. Dies ermöglicht die unmittelbare Manipulation der Paketdaten mit Hilfe von SIMD-Befehlen wie SSE- oder AVX-Befehlen, die ein 16-Byte-Alignement voraussetzen. Andernfalls wäre vor jedem SIMD-Befehl ein Kopiervorgang des Pakets in einen Speicherbereich mit 16-Byte-Alignement notwendig, der unnötige Kopieroperationen erfordern würde.

3.3. Pakete

Die im Modell zu verarbeitenden Nutzdaten werden in Paketen gespeichert, die analog zu den Verarbeitungsblöcken von der Mutterklasse *CPacket* abgeleitet werden. Diese Klasse enthält Variablen, die gemeinsame Eigenschaften aller Pakete repräsentieren. Beispiele hierfür sind eine Paket-ID, die das konkrete Paketformat identifiziert, oder ein Zeitstempel, der den relativen Zeitpunkt der Erzeugung des Pakets seit Programmstart in µs speichert. Darüber hinaus stellt die *CPacket*–Klasse Methoden zur Serialisierung der Pakete ins Binärformat bereit. Diese sind insbesondere für Quellen- und Sinkenblöcke nützlich, um Pakete im Binärformat ein- und auslesen zu können. Anwendungsbeispiele sind das Schreiben von Paketen in eine Datei oder das Versenden von Paketen via Ethernet. Das verwendete Binärformat ist in Abbildung 4 dargestellt. Nach einem fest definierten Synchronisationsheader, der die Synchronisation der Deserialisierung vereinfacht, folgen im Binärformat alle Variablen der Mutterklasse (IDs, Timestamp, …), die Angabe der Länge des Pakets in Byte und anschließend die eigentlichen Nutzdaten, die vom verwendeten Pakettyp abhängen.

Sync. Header (2 Byte)	Packet ID (1 Byte)	Stream ID (1 Byte)	Time-Stamp (8 Byte)	Paketlänge (4 Byte)	Paketdaten (*Paketlänge* Bytes)

Bild 4 Binärformat eines Pakets nach Serialisierung

Abbildung 5 zeigt beispielhaft die Implementierung des von der *CPacket*–Klasse abgeleiteten *CSamplesPacket*-Pakettyps, das IQ-Daten repräsentiert und bereits im Beispiel in Abbildung 3 verwendet wurde. Es enthält eine Variable *rate_d*, die die Abtastrate der Samples des Pakets speichert, sowie zwei Puffer *dataReal_vf* und *dataImag_vf*, die den Real- und Imaginärteil der Samples speichern. Die Länge der Puffer und damit der Pakete kann dabei beliebig gewählt werden. Für jedes Paket können Hilfsfunktionen definiert werden, die Operationen auf den Daten eines Pakets ausführen. Im Fall des *CSamplesPacket* ist dies die *normalize*-Funktion, die die Leistung des Pakets auf 1 normalisiert. Diese wurde bereits im Codebeispiel in Abbildung 3 verwendet.

```
// packet representing base band samples in IQ format
class CSamplesPacket : public CPacket {
public:

    CSamplesPacket(void) : CPacket(SAMPLES_PACKET_ID) {}

    double  rate_d; //!< Sampling rate in samples/s

    //!< real and imaginary part (I and Q)
    buffer<float, alignedAllocator<float, 32> > dataReal_vf;
    buffer<float, alignedAllocator<float, 32> > dataImag_vf;

    //! \brief  Normalizes the power of the packet to 1
    void normalize () {
        // code removed for simplicity
    }
};
```

Bild 5 Codebeispiel eines Paketformats zum Austausch von IQ-Basisbandaten mit Hilfsfunktion zur Normalisierung

3.4. Erzeugung eines Modells

Auf Basis der implementierten Verarbeitungsblöcke und Pakete wird anschließend das Modell des Empfangssystems erstellt. Dazu werden in einer C++-Modelldatei, die die C++ *main*-Methode enthält, die erstellten Blöcke instanziiert, initialisiert und miteinander verbunden. Dies ist in Abbildung 6 am Beispiel eines einfachen Modells mit zwei Blöcken und einer Verbindung dargestellt. Das Modell liest IQ-Basisbanddaten des Pakettyps *CSamplesPackets* vom Empfangsfrontend mit Hilfe des *CTsmwReader*-Blocks ein und schreibt diese ohne weitere Verarbeitung mit Hilfe des *CFileSink*-Blocks in eine Datei. Dies ist der typische Schritt zum Aufzeichnen von IQ-Basisbanddaten eines Empfangsfrontends auf einer Festplatte für die spätere offline-Decodierung.
Dazu werden zunächst beide Blöcke instanziiert und mit Hilfe ihrer *init*-Methode initialisiert. Der *init*-Methode werden die Parameter der Blöcke übergeben, wie zum Beispiel die Empfangsfrequenz des Frontends im Fall des *CTsmwReader*-Blocks. Die *SamplesLink*-Verbindung realisiert den Datenaustausch zwischen den beiden Blöcken und wird mit den *registerInput* und *registerOutput*-Methoden mit den Ports der beiden Blöcke verbunden. Anschließend können die beiden Blöcke mit der *startOperation*-Methode gestartet werden. Das Modell wird nach der Ausführung so lange in der while-Schleife ausgeführt, bis der *TsmwReader*-Block beendet wird. Dies kann mit der *isRunning*-Methode für jeden Block abgefragt werden. Anschließend werden die Blöcke mit der *stopOperation*-Methode angehalten und das Modell wird beendet.

```
int main (int argc, char* argv[]) {

    CTsmwReader TsmwReader;
    TsmwReader.init(frequency);

    CFileSink FileSink;
    FileSink.init(outputFileName);

    CUnidirectionalLink<CSamplesPacket> SamplesLink;
    TsmwReader.registerOutput(SamplesLink);
    FileSink.registerInput(SamplesLink);

    // Start blocks
    FileSink.startOperation();
    TsmwReader.startOperation();

    // Write until TSMW Reader stops
    while (TsmwReader.isRunning()) {
    }

    // Stop blocks
    TsmwReader.stopOperation();
    FileSink.stopOperation();

    return 0;
}
```

Bild 6 Codebeispiel eines einfachen Modells, das IQ-Basisbanddaten vom Empfangsfrontend einliest und als Datei auf die Festplatte schreibt. Includebefehle und Deklarationen der globalen Variablen *frequency* und *outputFileName* wurden der Übersichtlichkeit halber weggelassen.

Analog zum vorgestellten Beispiel können beliebig komplexe Modelle mit zahlreichen Blöcken und Verbindungen erstellt werden. Dabei können Ausgangsports von Blöcken auch mit den Eingängen mehrerer Folgeblöcke verbunden werden. Darüber hinaus sind Rückkopplungen zu früheren Blöcken möglich. Insbesondere für die Erfüllung von Echtzeitanforderungen ist die Aufteilung eines Modells in mehrere Untermodelle interessant, die auf verschiedenen PCs ausgeführt werden können und über die Ethernet-Schnittstelle miteinander kommunizieren. Dies ist exemplarisch für einen vereinfacht dargestellten DVB-T2 Empfänger mit FFT, QAM (Quadrature-Amplitude-Modulation)-Demapper und LDPC (Low-Density-Parity-Checkcode)-Decoder in Abbildung 7 dargestellt. Im oberen Blockdiagramm ist der ursprüngliche Ablauf auf einem einzigen PC dargestellt. Da insbesondere der LDPC-Decoder ein äußerst laufzeitintensiver Block ist, wird dieser mit Hilfe von Netzwerksinken (NS) und Netzwerkquellen (NQ) auf einen zweiten PC ausgelagert, wie im unteren Blockdiagramm dargestellt. Die im Rahmen des Toolkits implementierten Netzwerkblöcke unterstützen dabei sowohl UDP- als auch TCP-Übertragung sämtlicher Pakettypen auf Grundlage der vorgestellten Funktionen zur Serialisierung und Deserialisierung ins Binärformat.

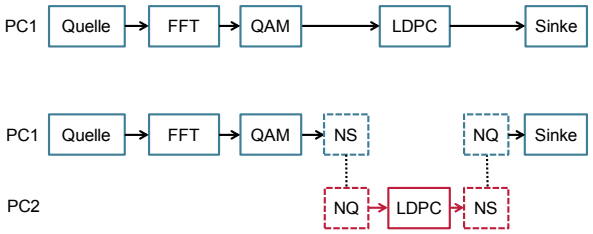

Bild 7 Stark vereinfachtes Beispiel eines DVB-T2-Empfängers (oben). Unten ist die Nutzung von Netzwerkquellen (NQ) und –sinken (NS) für die Realisierung von verteiltem Rechnen dargestellt.

4. Graphische Oberfläche

auf Grundlage des Toolkits einen konkreten Empfänger zu entwickeln, müssen eine Vielzahl von Blöcken und Verbindungen per Hand in Programmquelltext miteinander verknüpft werden (siehe Abbildung 6). Die C++-Modelldatei des aus mehr als 40 Blöcken bestehenden DVB-T2-Messempfängers, die lediglich die Instanziierung, Konfiguration und Verbindung der Blöcke enthält, umfasst beispielsweise knapp 1000 Zeilen. Die Übersichtlichkeit einer solchen Modelldatei ist entsprechend gering und das Debugging kompliziert, weil das Erkennen von Fehlern, wie falsche Verbindungen zwischen Blöcken, schwierig ist. Aus diesem Grund wurde eine graphische Oberfläche auf Grundlage des Qt-Frameworks [11] entwickelt (siehe Abbildung 8), die den graphischen Entwurf eines Empfangssystems auf Grundlage des Toolkits erlaubt. Aus der Block-Bibliothek, die die entwickelten Blöcke enthält, kann dann per Drag-and-Drop ein Modell erstellt, verbunden und konfiguriert werden. Dabei müssen keinerlei Metadaten bezüglich der Blockbibliothek gepflegt werden. Die GUI analysiert sämtliche C++-Quelldateien der Blöcke und liest ihre Eigenschaften wie die Anzahl der Eingangs- und Ausgangsports, die Paketypen der Ports und Variablen für die Konfiguration der Blöcke direkt aus dem C++-Quelltext ein.

Ist das Modell fertig verbunden und konfiguriert, wird mit Hilfe von automatischer Codeerzeugung der C++-Quelltext der Modelldatei (siehe Abbildung 6) generiert und kompiliert. Als Ausgabe entsteht dann eine ausführbare Datei des Modells, die unter Linux oder Windows ausgeführt werden kann. Das Starten des Modells ist direkt aus der GUI heraus möglich, wobei die Konsolenausgabe des Modells in der GUI erfolgt.

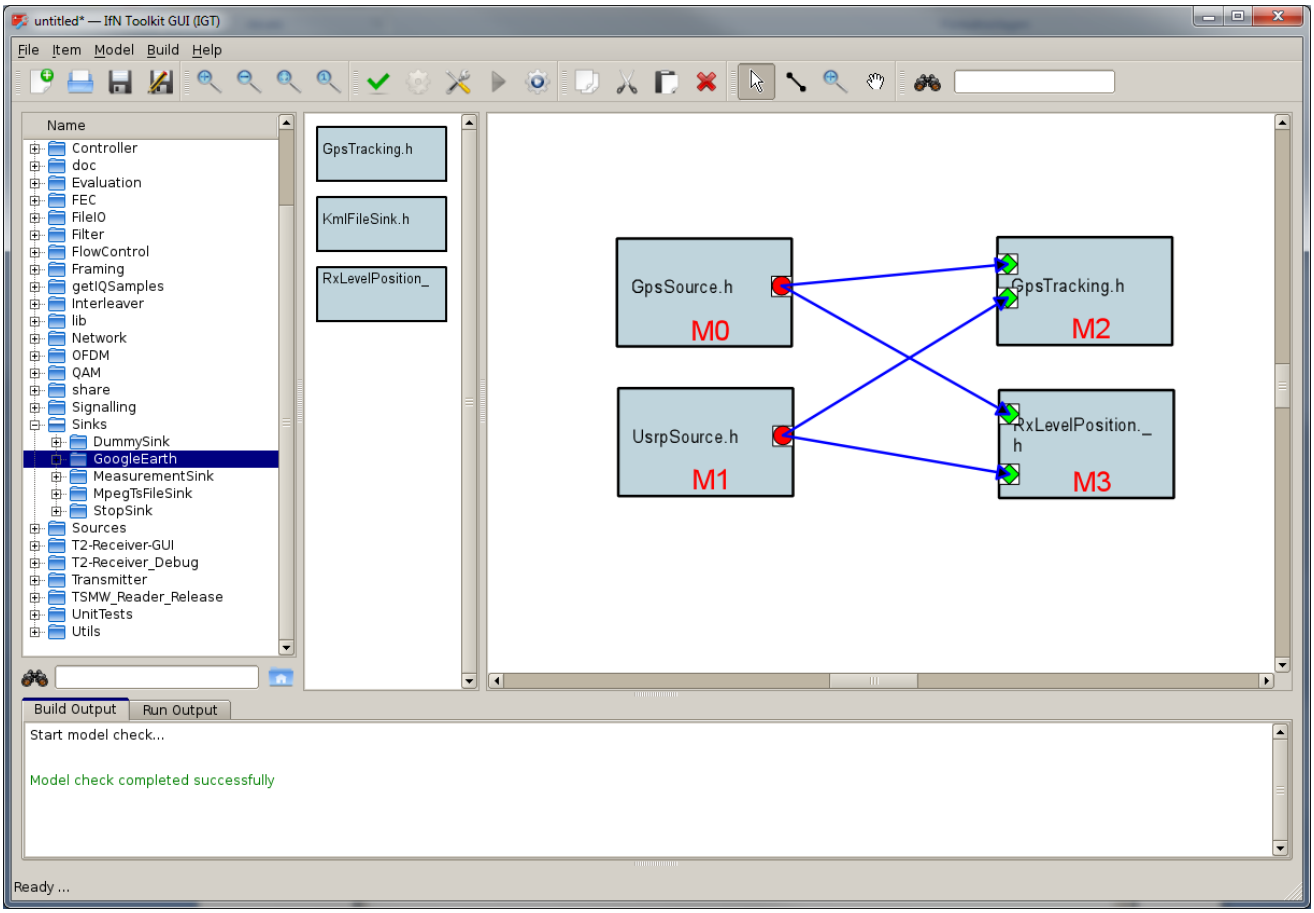

Bild 8 Graphische Oberfläche des Empfänger-Toolkits: Die Baumansicht repräsentiert die Blockbibliothek, die sämtliche Verarbeitungsblöcke organisiert in einer Verzeichnisstruktur enthält. Die Blöcke des ausgewählten Verzeichnisses können per drag-and-drop zum Modell hinzugefügt, verbunden und konfiguriert werden. Der Quelltext des Modells (vgl. Abbildung 5) wird automatisch generiert und anschließend zu einer ausführbaren Datei kompiliert.

5. Zusammenfassung

Um den Entwurf softwarebasierter Empfangssysteme zu erleichtern, wurde ein Toolkit auf Basis von C++ entwickelt, das ein System als Modell mit Hilfe von Blöcken darstellt, zwischen denen paketbasierte Daten über gepufferte Verbindungen ausgetauscht werden. Das Toolkit berücksichtigt Anforderungen wie die Ausnutzung von Mehrkernarchitekturen und SIMD-Streaming-Extensions wie SSE, die für den Entwurf echtzeitfähiger, softwarebasierter Empfangssysteme essentiell sind. Zur Vereinfachung der Benutzung des Toolkits wurde eine graphische Oberfläche entwickelt, die den Entwurf von Modellen durch automatische Codeerzeugung und Kompilierung maßgeblich vereinfacht. Auf Grundlage des vorgestellten Toolkits wurde ein mobilfähiger DVB-T2-Empfänger entwickelt, der z.B. im Rahmen des DVB-T2 Modellversuchs in Norddeutschland zur Auswertung der Mobilmessungen eingesetzt wurde.

Danksagung

Die Autoren danken ihren Kollegen des Instituts für Nachrichtentechnik der Technischen Universität Braunschweig, insbesondere Prof. Dr.-Ing. Ulrich Reimers, für die hilfreichen Kommentare und Anmerkungen.

6. Literatur

[1] U. Reimers: DVB – The Family of International Standards for Digital Video Broadcasting, 2nd edition, Springer, 2005

[2] Digital Video Broadcasting (DVB); Frame structure channel coding and modulation for a second generation digital terrestrial television broadcasting system (DVB-T2), ETSI EN 302 755, V1.1.1, October 2008

[3] Digital Video Broadcasting (DVB); Implementation guidelines for a second generation digital terrestrial television broadcasting system (DVB-T2), ETSI TS 102 831 V1.1.1, 2010

[4] Hasse, P.; Robert, J.: A software-based real-time DVB-C2 receiver, IEEE International Symposium on Broadband and Multimedia Systems (BMSB), 2011

[5] Slimani, M; Robert, J.; Zoellner, J.: A Software-Based Mobile DVB-T2 Measurement Receiver, IEEE International Symposium on Broadband and Multimedia Systems (BMSB), 2012

[6] Slimani, M.; Robert, J.; Zoellner, J.: Softwarebasierter Messempfänger für DVB-T2, Fernseh- und Kinotechnik (FKT), Heft 3/2012, S. 84-87, März 2012.

[7] Terrestrik der Zukunft: Zukunft der Terrestrik, Projektbericht DVB-T2 Norddeutschland, Shaker-Verlag, Aachen, 2012.

[8] Butenhof, D.R.: Programming with POSIX Threads, Addison-Wesley, 1997

[9] Westermann, P.: Exploration of the scalability of SIMD processing for software defined radio, Dissertation, Technische Universität Dortmund, 2010

[10] Intel® C++ Intrinsics Reference, Document Number: 312482-002US, available at www.intel.com

[11] Homepage des Qt Projektes, http://qt-project.org/, abgerufen am 16.10.2012

[12] B. Schäling: The Boost C++ Libraries, XML Press, 2011

[13] B. Stroustrup, The C++ Programming Language, Addison-Wesley Verlag, 2000