Beschreibung der Plattformabhängigkeit eingebetteter Applikationen mit Dienstmodellen

Simon Barner[1], Andreas Raabe[2], Christian Buckl[2] und Alois Knoll[1]

[1]{barner,knoll}@in.tum.de, [2]{raabe,buckl}@fortiss.org

Abstract: Modellbasierte Entwicklung ist ein zunehmend populärer Ansatz, um der wachsenden Komplexität eingebetteter Anwendungen zu begegnen. Eine besondere Herausforderung besteht in der Berücksichtigung nicht-funktionaler und zeitlicher Anforderungen (z.B. Kommunikation, Safety, Energieverbrauch), deren Erfüllung ein nahtloses Zusammenspiel der Hardware, der verschiedenen Ebenen der Systemsoftware und der Anwendungssoftware bedingt. Die vorliegende Arbeit schlägt vor, die Abhängigkeiten der Hardware/-Software-Schichten mit einem *Dienstmodell* zu beschreiben. Dieses Modell umfasst einerseits die Kapselung funktionaler Abhängigkeiten zwischen verschiedenen Systemebenen (Peripherietreiber, Taskausführung, Kommunikation, etc.); andererseits beinhaltet es die Spezifikation nicht-funktionaler Eigenschaften in Form von Garantien durch den Dienstanbieter sowie in Form von Anforderungen durch die Bindung an einen Dienst. Beide Aspekte kommen bei der Verfeinerung einer Applikation bis hin zu einer plattformspezifischen Implementierung zum Tragen. Die spezifizierten nicht-funktionalen Abhängigkeiten können dabei als Ausgangspunkt für Analysen dienen (z.B. Mapping, Scheduling), deren Ergebnisse für die automatische Erzeugung von Programmcode genutzt werden können. Diese Arbeit präsentiert neben einem geeigneten Metamodel auch eine Taxonomie von Diensten für die Entwicklung eingebetteter Systeme und validiert den vorgeschlagenen Ansatz in einer Fallstudie.

1 Einleitung

Mehr als 90 Prozent der verkauften Prozessoren kommen in eingebetteten Anwendungen zum Einsatz [Bro05]. Im Bereich der eingebetteten Systeme ist die Variantenvielfalt und Heterogenität der verfügbaren Hardware sowie zugehöriger Abstraktionsschichten, Middlewares und Betriebssysteme eine zentrale Problemstellung, die durch die dadurch bedingte Zunahme des Anteils plattformabhängiger Software verschärft wird [EMD08]. Modellbasierte Softwareentwicklung ist ein möglicher Ansatz, der beschriebenen Komplexität durch Abstraktion und Hierarchiebildung zu begegnen. Da bei eingebetteten Systemen die Berücksichtigung nicht-funktionaler Anforderungen eine zentrale Rolle spielt und deren Erfüllbarkeit eng mit der Plattform verwoben ist, muss die Plattform bei der Systemmodellierung mit in Betracht gezogen werden. Hierbei kann grob zwischen leichtgewichtigen Ansätzen wie etwa der Definition von Profilen für die *Unified Modelling Language* (UML) und schwergewichtigen Varianten in Form von domänenspezifischen Sprachen (DSLs) unterschieden werden [PHG+09].

Die vorliegende Arbeit verfolgt das Ziel, eine problemorientierte Sicht auf den Entwurf eingebetteter Systeme zu bieten und umfasst daher sowohl Metamodelle zur Beschrei-

bung der Schichtenarchitektur der Plattform (Hardware, Abstraktionsschichten, Middleware, OS) als auch der Applikation. Hierbei steht die Beschreibung des Zusammenwirkens der unterschiedlichen Schichten durch ein *Dienstmodell* im Vordergrund, mit dem sowohl funktionale Abhängigkeiten als auch nicht-funktionale Anforderungen und Garantien beschrieben werden können.

Der Rest dieser Arbeit ist wie folgt aufgebaut: Nach einer Motivation der Untersuchung von Dienstmodellen in Abs. 2 geben wir eine Zusammenfassung des aktuellen Stands der Technik in Abs. 3. Anschließend beschreiben wir die Schichtenarchitektur der Systemmodellierung in Abs. 4. In Abs. 5 definieren wir neben einem geeigneten Metamodell auch eine Taxonomie jener Dienste, die für die Entwicklung eingebetteter Anwendungen relevant sind. In Abs. 6 werden die vorgeschlagenen Konzepte im Rahmen einer Fallstudie auf ihre Anwendbarkeit geprüft und in Abs. 7 nochmals zusammengefasst und bewertet.

2 Anwendbarkeit von Dienstmodellen

Im Folgenden wird erläutert, welche Beiträge der vorgeschlagene Ansatz zu verschiedenen Phasen des Entwicklungsprozesses liefert.

Auslegung des Gesamtsystems: Eingebettete Systeme kommen in verschiedensten Anwendungsdomänen mit ebenso vielfältigen Anforderungen zum Einsatz. Bei der Auslegung des Gesamtsystems sind insbesondere drei Komponenten zu unterscheiden: Die zugrundeliegende Hardware, die verwendete Middleware und die zu implementierenden Applikationen. Zwischen den Ebenen der Hardware/Software-Schichten kann – mit gewissen Einschränkungen – Funktionalität verschoben werden. So überwiegen etwa bei hohen Stückzahlen die Hardwarekosten, wohingegen andernfalls die Entwicklungskosten entscheidend sind, so dass zwischen dem Einkaufspreis und der von der Hardware/Middleware-Kombination angebotenen Funktionalität abgewogen werden muss. Das vorgeschlagene Dienstmodell erlaubt eine schrittweise Verfeinerung der Spezifikation im Hinblick auf Lokalisierung von Funktionalität und Zusicherung nichtfunktionaler Eigenschaften.

Analyse nicht-funktionaler Eigenschaften: Ein weiteres häufig anzutreffendes Anwendungsszenario ist der Einsatz von commercial-of-the-shelf (COTS) Plattformen. Hier ist vom Systemarchitekten vor allem eine Auswahl aus den angebotenen Hardwarealternativen zu treffen, wobei insbesondere auch die Eigenschaften der jeweils verfügbaren Middlewares zu beachten ist. Eine besondere Schwierigkeit stellt hier die Berücksichtigung jener nicht-funktionalen Anforderungen dar, die entweder auf verschiedenen Ebenen der Hardware/Software-Schichten realisiert werden können, oder die sogar mehrere Ebenen involvieren. Eine zentrale Voraussetzung für die Analyse der Eignung einer Plattformkombination ist demnach die geeignete Darstellung ihrer nicht-funktionalen Zusicherungen sowie die Zuordnung nach deren Ursprung.

Software-Synthese: Im diesem Bereich sind zwei weitgehend unabhängige Konzepte umzusetzen. Einerseits muss Funktionalität (je nach Granularität der Anwendungsbeschreibung bspw. Komponenten, Anweisungen, Tasks) jeweils geeigneten Hardwarekomponenten zugewiesen werden (*Bindung*). Dies bedingt insbesondere eine automatisierte Analyse nicht-funktionaler Eigenschaften sowie die Darstellbarkeit dieser Bindung. Andererseits

muss die Verwendung automatisch auflösbar sein, also die entsprechende Funktion auf der dafür vorgesehenen Hardware ausgeführt werden können. Hierzu ist die Bereitstellung der entsprechenden Code-Templates, Compileraufrufen u.Ä. notwendig.

3 Stand der Technik

Wie von Passerone et al. ausgeführt [PHG+09] ist modellbasierte Entwicklung ein geeigneter Ansatz, der wachsenden Komplexität eingebetteter Anwendungen zu begegnen. Aufgrund der inhärenten wechselseitigen Abhängigkeit von Ausführungsplattform und Applikation geben wir einen Überblick über existierende Ansätze, mit denen diese präzise beschrieben werden kann und die insbesondere zur Modellierung der Interaktion zwischen den verschiedenen Systemschichten geeignet sind.

Kontrakte (*contracts*) sind ein vielfach verfolgter Ansatz, der auf die Zusicherung funktionaler Eigenschaften zwischen *Eiffel*-Modulen mit Hilfe von Annahmen und Garantien (`require` und `ensure`) zurückgeht [Mey92]. Der Ansatz von Beugnard et al. macht sich dieses Konzept zunutze, um Abhängigkeiten zwischen Komponenten zu beschreiben. Der Fokus liegt dabei auf der Betrachtung syntaktischer Aspekte (IDLs), des Verhaltens (Prä- und Postbedingungen), der Synchronisation zur Koordination von Operationen sowie der Dienstgüte (*quality of service*) [BJPW99]. In einer Retroperspektive [BJP10] stellen die Autoren sowohl die Bedeutung von Interaktionen (zwischen Komponenten sowie von Komponenten mit der Umwelt) als auch die zentrale Rolle nicht-funktionaler Anforderungen heraus. Beide Aspekte werden u.a. im HRC-Komponentenmodell [Dam05] berücksichtigt, das verschiedene Modellierungssichten mit Hilfe von Kontrakten realisiert [BCF+08]. Neben der formalen Definition von Kontrakten durch Annahmen (*assumptions*: Anforderungen von Komponenten an ihre Umgebung) und Versprechen (*promises*: bei Erfüllung der Annahmen garantierte Eigenschaften) wird mit CSL (*contract specification language*) eine Patternsprache zur Spezifikation von Echtzeitkontrakten vorgeschlagen [GBC+08]. Im Projekt FRESCOR[1] werden ähnliche Aspekte behandelt, allerdings liegt der Fokus auf der Aushandlung von Kontrakten zur Laufzeit (Scheduling, Kommunikation).

Bei *DOL* (*Distributed Operation Layer)* [TBHH07] werden Applikationen als Prozess-Netzwerk-Modell dargestellt. Bei der Plattformmodellierung werden dabei sowohl die Topologie der Hardware als auch die Schichten der Systemsoftwares berücksichtigt. Der Fokus der Arbeit liegt auf der Performanzanalyse auf Systemebene, auf deren Basis Applikationen auf die jeweilige Plattform abgebildet werden.

In *SysCOLA* [WHHW09], einer Kombination der formalen COLA-Komponentensprache und SystemC, kommen COLA-Modelle für die Beschreibung der Applikation sowie von Topologie und Features der abstrakten Plattform zum Einsatz. Das Hauptziel der Arbeit liegt darin, einen virtuellen Prototyp der Ausführungsplattform zur Verfügung zu stellen, mit dem das zu entwickelnde System simuliert werden kann. Hierzu kommt eine Plattform-Abstraktionsschicht (VPAL: *virtual plattform abstraction layer*) zum Einsatz, die allerdings nur sehr grundlegende Dienste zur Verfügung stellt (Kommunikation, zustandsbehaftete Tasks).

[1] `http://www.frescor.org/`

Hardware-dependent Software (HdS) beschreibt den Softwareanteil der Hardware-/Software-Schichten eingebetteter Anwendungen wie etwa BSPs (*board support packages*), HALs (*hardware abstraction layers*) oder Treiber, der nicht nur einen Großteil der Komplexität der Systeme ausmacht, sondern auch wesentlichen Einfluss auf deren Eigenschaften hat [EMD08]. Daher stellt HdS Modelle zur Spezifikation der HW/SW-Schnittstelle (bspw. spezielle Prozessorinstruktionen, memory-mapped I/O) sowie von Kommunikation und Ausführungskontexten zur Verfügung. Während HdS somit zwar Mechanismen zur detaillierten Darstellung des Hardwarezugriffs durch die Software bietet, ist aufgrund der fehlenden Mechanismen zur Beschreibung der Applikation keine ganzheitliche Systemmodellierung möglich.

Das *MaCC*-Framework [PKMM10] ist eine Modellierungssprache zur strukturellen Beschreibung von Systemen. Auf der gewählten Abstraktionsebene werden bspw. Prozessoren, DMA-Controller, Speicher und deren Verbindungen wie etwa Busse, Punkt-zu-Punkt-Verbindungen, Registerfile, etc. strukturell beschrieben. Das Ziel ist es dabei, die Entwicklung von Source-Level-Codeoptimierungen zu unterstützen, die vom Wissen über die Speicherhierarchie profitieren (etwa: Nutzung von Scratchpads), wozu das Framework das Mapping zwischen verschiedenen Adressräumen unterstützt. Der Fokus liegt darauf, einen datenbankartigen Zugriff auf die Systembeschreibung zu bieten, aus der für die Optimierung benötigte Größen abgefragt werden können (bspw. Energie, Latenz, Durchsatz).

SysML [OMG10] wird zur Beschreibung komplexer Systeme eingesetzt, die u.a. aus Hardware und Software bestehen können und baut dazu auf einer Teilmenge von UML 2 auf. SysML bietet dabei allerdings mit sog. *allocations* nur einen sehr generischen Mechanismus zur Beschreibung der Beziehung zwischen verschiedenen Hierarchieebenen. AADL [FGH06] verfolgt ähnliche Ziele, indem es Modellierungskonzepte zur Beschreibung und Analyse von Systemarchitekturen in Form von Komponenten und deren Interaktion bietet, wozu eine Reihe von Abstraktionen von Software-, Hardware- und Systemkomponenten zur Verfügung gestellt werden.

4 Überblick über die Modellierung

Mehrschichtige Modelle sind ein Ansatz zur Beschreibung von eingebetteten Anwendungen. Mit ihnen lassen sich neben der eigentlichen Anwendung und den verschiedenen Schichten der Ausführungsplattform insbesondere deren wechselseitige Abhängigkeiten erfassen (vgl. Abs. 3).

Im Folgenden wird ein geeignetes Metamodell zur Beschreibung von eingebetteten Systemen skizziert, das neben verschiedenen Komponenten-Metamodellen zur Beschreibung der Systemschichten, über ein Dienst-Metamodell verfügt. Auf letzteres wird in Abs. 5 detailliert eingegangen wird.

Das **abstrakte Komponenten-Metamodell** ist die Basis zur Beschreibung der topologischen Struktur von Systemkomponenten. Es umfasst *Komponenten*, deren Schnittstelle zur Umgebung durch *Ports* repräsentiert wird. Die strukturelle Relation zwischen Komponenten wird mit Hilfe von an Ports gebundene *Kanäle* dargestellt. *Hierarchiebildung* ermöglicht es, bestehende Komponentenmodelle zu kapseln und wiederzuverwenden.

Das **Applikations-Metamodell** stellt die erste Gruppe der Verfeinerungen des abstrak-

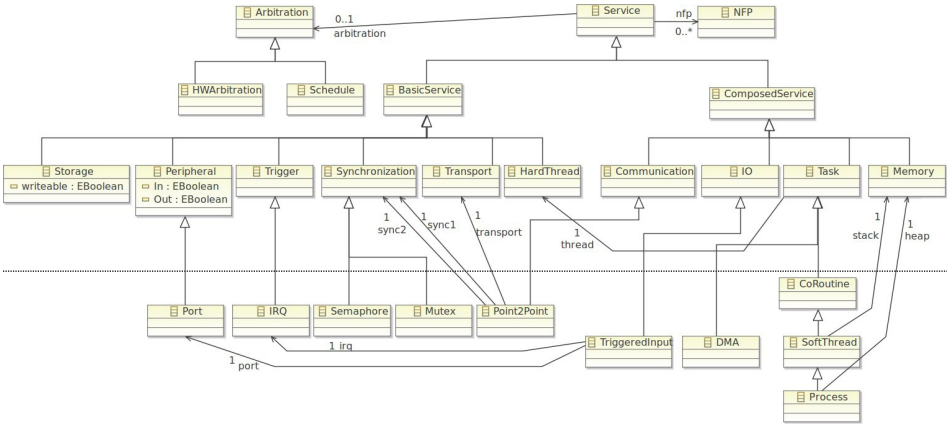

Abbildung 1: Metamodell und Taxonomie von Diensten sowie einfache Beispiele (unten).

ten Komponenten-Metamodells dar und dient zur Spezifikation von funktionalen Eigenschaften. Beispiele sind Datenflussmodelle wie SDF oder Automatenmodelle wie SFC oder FSM. Details zu beiden Typen finden sich in [BGBK08].

Plattform-Metamodelle sind eine Verfeinerung des abstrakten Komponenten-Metamodells zur Spezifikation der Ausführungsplattform. Dabei dient das **topologische Hardware-Metamodell** dazu, die Einzelkomponenten einer Hardware-Plattform darzustellen und deren Topologie zu erfassen. Es umfasst daher Basisklassen zur Repräsentation von Prozessoren und Kernen, Bussen, Netzwerken, Speichern sowie Sensoren und Aktuatoren, die zur Beschreibung einer konkreten Hardware entsprechend verfeinert werden. Darüber hinaus werden im Plattformmodell die Komponenten des Softwarestapels beschrieben, etwa Prozesse einer Middleware, Module einer Hardwareabstraktionsschicht oder Treiber eines Betriebssystems.

5 Dienste

Um die Abhängigkeiten der im vorangegangenen Abschnitt erwähnten Ebenen der HW/SW-Schichten darstellbar zu machen, schlagen wir im Folgenden ein Dienstmodell vor. Die reine Darstellung dieser Abhängigkeiten nimmt Anleihen am Konzept der Kontrakte (vgl. [Mey92, BJPW99, BCF+08, GBC+08]), sowie am HRC-Modell [Dam05]. Die besondere Neuerung unseres Ansatzes ist, dass zusätzlich zur reinen Darstellung der Abhängigkeit die Funktionalität tieferliegender Schichten durch bereitgestellte Codetemplates zugänglich gemacht wird. Dies kann als Interface-Kapselung interpretiert werden. Da nun jede Komponente ihre Interfacedefinition samt Codetemplate in Form eines Dienstes zur Verfügung stellt, ermöglicht dies die Implementierung eines völlig hardwareunabhängigen Codegenerators. Im Folgenden wird ein Metamodell zur Beschreibung solcher *Dienste* vorgestellt und eine entsprechende Taxonomie angegeben.

Dienst-Metamodell. Dienste dienen dazu, die Beziehung zwischen Komponenten zu beschreiben, die in unterschiedlichen Schichten des Systems beschrieben sind (siehe Abb. 1). Sie werden in unserem Metamodell durch die Klasse `Service` repräsentiert. Komponenten können einerseits eine Menge von Diensten anbieten. Andererseits können sie Dienste

anfordern, und so ihre Implementierung auf diesen abstützen. Die Realisierung dieser Beziehungen erfolgt durch eine geeignete Aggregation bzw. Komposition (nicht abgebildet).

Falls ein von einer Komponente angebotener Dienst `P` mit einem von einer Komponente einer anderen Modellierungsebene angeforderten Dienst `R`, kompatibel ist, so können beide aneinander gebunden werden (`ServiceBinding`). Dabei ist die syntaktische Verträglichkeit gewährleistet, falls der Typ von `R` ein Supertyp von `P` ist, d.h. falls der angebotene Dienst den Anforderungen entspricht bzw. noch weiter verfeinert ist. Eine Bindung repräsentiert somit die durch die Verwendung eines angebotenen Dienstes dargestellte funktionale Abhängigkeit der `R`-Komponente von der zugrundeliegenden `P`-Komponente. Zur Realisierung dieser funktionalen Beziehung sind Diensten geeignete Codetemplates zugeordnet, wie bspw. ein Treiberaufruf durch eine Applikationskomponente, die so einen Zugriff auf eine Hardwarekomponente durchführt.

Abb. 1 zeigt die vorgeschlagene Taxonomie von Diensten zusammen mit einigen einfachen Beispielen. Um die semantische, sowie die nicht-funktionale Kompatibilität von zu bindenden Diensten nach unterschiedlichen Gesichtspunkten wie etwa Echtzeit, Kommunikation, Safety, Energieverbrauch, etc. beschreiben zu können, wird jedem Dienst eine Beschreibung seines Angebots bzw. seiner Forderung zugeordnet. Hierzu können einerseits direkt formale Spezifikationen wie etwa temporale Logik zur Definition von zeitlichen Bedingungen verwendet werden. Gafni et al. haben für zeitliche Eigenschaften einen vielversprechenden Ansatz vorgestellt, wie mit Hilfe einer Menge von vordefinierten Pattern eine formale, aber dennoch menschenlesbare Codierung der Bedingungen angegeben werden kann [GBC+08]. Daher erscheint es sinnvoll, diesen Ansatz auch auf andere durch Dienste darstellbare Eigenschaften zu übertragen und geeignete Pattern zu definieren. Nicht-funktionale Eigenschaften von angebotenen Diensten sowie Anforderungen an Dienste werden in Abb. 1 durch die Referenz `nfp` repräsentiert. Dazu werden Verfeinerungen der Klasse `NFP` für die Beschreibung verschiedener Aspekte benötigt, die in geeigneten Darstellungen unterschiedlicher Ausdrucksmächtigkeit ausgedrückt werden können. Jedem Diensttyp können so unterschiedliche nicht-funktionale Angebote / Forderungen zugeordnet sein. Typische Beispiele sind: Durchschnittliche Latenz, maximale Latenz, Durchsatz, Fehlerwahrscheinlichkeit und Energieverbrauch.

Dienst-Taxonomie. Basisdienste (`BasicService`) stellen eine Schnittstelle zu grundlegenden, nur in Hardware realisierbaren Primitiven dar (vgl. Abs. 4), die sich wie folgt einteilen lassen:

- `Storage` abstrahiert verschiedenartige Speicher (RAM, Flash, etc.).
- `Peripheral` fasst die Eigenschaften von Ein-/Ausgabegeräten wie etwa Ports oder A/D bzw. D/A-Wandlern zusammen.
- `Trigger` stellen eine Abstraktion von Ereignisquellen dar (Beispiele: Interrupts wie Timer, Peripherieeinheiten, etc.).
- `Synchronization` ist eine Basisklasse für Synchronisationsdienste, die direkt von der Hardware zur Verfügung gestellt werden (z.B.: Semaphore, Mutexe, ...).
- `Transport` beschreibt Dienste von Hardwarekomponenten, mit denen Daten im System bewegt werden können (Bsp.: Busse, NoC, ...).
- `HardThread` abstrahiert Berechnungsdienste, die hardwareseitig durch einen Programmzähler und einen Registersatz zur Verfügung gestellt werden.

Die eigentliche Taxonomie von Diensten ist jedoch auf zusammengesetzten Diensten (`ComposedService`) definiert. Diese ordnen sich einer der folgenden Gruppen zu: `Communication`, `IO`, `Task` oder `Memory`. Ableitungen dieser Klassen fassen Basisdienste zusammen und können so komplexere, entweder in Software oder Hardware realisierte Dienste beschreiben. Diese explizite Modellierung der Fähigkeiten/Forderungen der HW/SW-Schichten ermöglicht es, die Abhängigkeiten zwischen diesen Schichten in der Modellierung zu erfassen. Hierbei wird nur deren statische „verwendet"-Beziehung erfasst. Anhand der Verhaltensbeschreibung des zusammengesetzten Dienstes muss die dynamische Verträglichkeit der Dienstbindung mit der Applikation analysiert sowie für geeignete Arbitrierung gesorgt werden (s.u.).

Falls ein Dienst auf verschiedenen Systemebenen realisiert werden kann, und er daher durch verschiedene Ausprägungen des gleichen abstrakten Dienstes repräsentiert wird, kann sowohl eine nachträgliche Verfeinerung bzw. Verschiebung einer Dienstbindung erfolgen. Ein Beispiel hierfür ist der Einsatz eines Betriebssystems während einer schrittweisen Verfeinerungen des Designs, dessen Dienste anstelle des direkten Hardwarezugriffs verwendet werden.

Im unteren Teil von Abb. 1 sind einige einfache Beispiele abgeleiteter Dienste dargestellt. So setzt sich bspw. eine Point-to-Point Kommunikation aus Synchronisationen und einem Datentransport zusammen. Eine `CoRoutine` ist ein Task ohne eigenen Kontext, besitzt aber einen Abarbeitungsfaden `HardThread`. Ein `SoftThread` hat darüber hinaus einen eigenen Stack, während ein `Process` auch einen Heap besitzt. In dieser Darstellung ist ein `DMA` lediglich ein (Kopier-)`Task`. `Port`, `IRQ`, `Semaphore` und `Mutex` sind einfache Dienste, welche Spezialisierungen von Basisdiensten darstellen.

Über die reine Darstellung der funktionalen und nicht-funktionalen Kompatibilität und der tatsächlichen Bindung hinaus stellen Dienste der jeweils höheren Schicht im Stapel Codetemplates zur Verfügung, die den Zugriff auf die unterliegende Komponente ermöglichen. Beispiele hierfür sind die Implementierungen von Zugriffsfunktionen für Pins eines Ports oder die Berechnung von physikalischen Adressen wie in [PKMM10] vorgeschlagen.

Arbitrierung. Da in dieser Arbeit nebenläufige Applikationsmodelle betrachtet werden, muss der Zugriff auf Dienste, die Ressourcen der Ausführungsplattform kapseln, geregelt werden. Eine Möglichkeit besteht darin, dass die Hardware geeignete Arbiter (`HWArbitration`) bereit stellt. Falls dies nicht der Fall ist, muss der Zugriff auf die Ressource durch geeignete Ausführungspläne (`Schedule`) geregelt werden (statisch oder dynamisch). Dabei ist zu beachten, dass die gewählte Arbitrierungsstrategie konsistent mit den durch den jeweiligen Dienst beschriebenen nicht-funktionalen Anforderungen sein muss (z.B. worst-case response time).

6 Fallstudie

Das Ziel dieses Abschnittes ist es, die in dieser Arbeit eingeführten Konzepte an einem Beispiel zu validieren. In Abb. 2 ist hierzu das Modell eines eingebetteten Systems dargestellt, dessen Hauptbestandteile das Plattformmodell (unten), ein Anwendungsmodell (oben, rot umrahmte Bereiche) sowie die durch die jeweiligen Dienste spezifizierten wechselseitigen Anforderungen und Garantien sind.

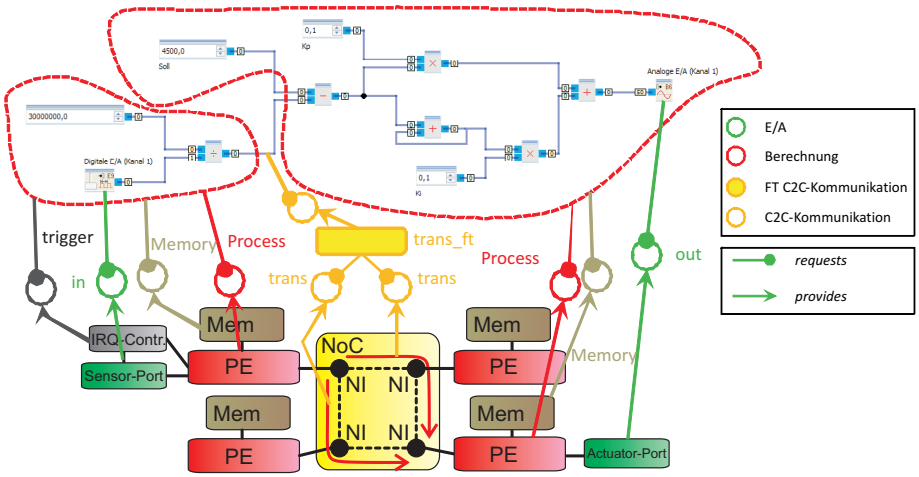

Abbildung 2: Exemplarisches 4-Kern-System mit Anbindung an Sensorik und Aktuatorik.

Die **Hardwareplattform** besteht dabei aus 4 Rechenkernen (PE: *processing element*), die jeweils über lokale Speicher (Mem) verfügen, und die untereinander über ein NoC verbunden sind, auf das sie über ein geeignetes Netzwerkinterface (NI) zugreifen. Eines der PEs ist dabei an einen Eingangsport und einen Interrupt-Controller angebunden; ein anderes verfügt über einen Ausgangsport, über den ein Aktuator angesteuert wird. Wie in Abs. 5 ausgeführt, bieten die Plattformkomponenten geeignete Verfeinerungen der folgenden Basisdienste an, die durch gleichfarbige Pfeile repräsentiert werden.

- Sensorik- und Aktuatorik-Ports: `Peripheral`
- Interrupt-Controller: `Trigger`
- Kommunikationspfade im NoC: `Transport`
- Rechenkerne: `HardThread`

Im Beispiel sind die Fähigkeiten der Hardware durch die Dienste einer **Middleware** bzw. eines **Betriebssystems** weiter abstrahiert:

- `Memory`, `Process`: Ausführungskontext für die gewählte Partitionierung des Anwendungsmodells (s.u.).
- `Communication`: Kommunikation zwischen den Anwendungskomponenten.

Im oberen Teil der Grafik ist das Datenflussmodell einer **Anwendung** dargestellt, in der ein PI-Regler realisiert ist (je ein Sensor und Aktuator), der gemäß einer gegebenen Partitionierung auf zwei Kernen ausgeführt wird.

Im Folgenden wird die Modellierung der **Abhängigkeit der Anwendung von der Ausführungsplattform** erläutert, die in der Abbildung durch Linien, die in gefüllten Kreisen enden, dargestellt werden. Die eigentliche Dienstbindung, d.h. die Erfüllung einer Anforderung durch einen vom der Plattform zur Verfügung gestellten Dienst, wird durch ungefüllte Kreise dargestellt.

Allen Komponenten des Applikationsmodells ist gemein, dass sie einen Berechnungsdienst (hier: `Prozess`) sowie Speicher (`Memory`) benötigen. Die Bindung der durch

die Anwendung angeforderten Plattformdienste stellt ein Mapping der Datenfluss-Komponenten auf Rechenkerne und deren zugehörigen lokalen Speicher dar. Da einem Kern mehrere Komponenten zugewiesen werden können, wird zur zeitlichen Arbitrierung der `HardThread`s ein Schedule benötigt. Im Falle des betrachteten synchronen Datenfluss-Modells kann dieser Schedule statisch vorberechnet werden; andere Applikationsmodelle sind dagegen auf die Schedulingdienste eines Betriebssystems angewiesen.

Die Kommunikation zwischen den Komponenten des Anwendungsmodells wird über entsprechende Plattformdienste realisiert. Lokale Kommunikation kann bspw. über einen an den jeweiligen Kern angebundenen Speicher realisiert werden. Die Inter-Core-Kommunikation, die durch das Mapping der Anwendung nötig wird, soll im Beispiel fehlertolerant ausgeführt sein. Sie wird hier durch eine Middleware realisiert, die sich den Kommunikationsdiensten zweier unabhängiger Pfade im NoC bedient. Bei der Implementierung der Applikation auf einer Plattform, die fehlertolerante Kommunikation in Hardware realisiert, kann dieser Schritt entfallen und die Anforderung der Software direkt erfüllt werden.

Der Zugriff auf die Peripherie erfolgt über entsprechende `IO`-Dienste. Im Beispiel signalisiert der Sensor über einen Interrupt, dass ein neuer Wert anliegt. Daher wird ein `Trigger` verwendet, um eine Interrupt-Service-Routine an den entsprechenden Kern zu binden.

7 Zusammenfassung und Ausblick

Der Beitrag dieser Arbeit liegt in der Einführung und Definition eines Dienstmodells, das zur Spezifikation der Abhängigkeiten zwischen den verschiedenen Schichten der Systemmodellierung verwendet werden kann. Dabei kann eine Bindung zwischen einer anfordernden und einer bietenden Komponente zustande kommen, falls diese syntaktisch kompatibel sind. Hierzu muss der Nachweis für die in den Diensten spezifizierten nichtfunktionalen Anforderungen erbracht und für eine geeignete Arbitrierung gesorgt werden. Ein wesentlicher Aspekt ist dabei, dass Dienste nicht nur die Verträglichkeit von Bindungen modellieren, sondern durch die Bereitstellung von Codetemplates auch deren Realisierbarkeit gewährleisten. Dies ermöglicht die Implementierung eines vollständig hardwareunabhängigen Codegenerators. Es ist geplant, die Skalierbarkeit des Ansatzes im Rahmen mehrerer realer Anwendungsfälle zu überprüfen.

Ein möglicher Anknüpfungspunkt an das in dieser Arbeit vorgeschlagene Framework ist die Integration von formalen Techniken zur Analyse der Verträglichkeit von Dienstbindungen. Ein anderer betrifft die Darstellbarkeit der dabei involvierten nicht-funktionalen Anforderungen, deren Lesbarkeit und Anwendbarkeit eine zentrale Rolle für die Akzeptanz der Methodik spielt. Hierbei scheint die Definition von patternbasierten Sprachen zur Beschreibung weiterer Aspekten (neben zeitlichen [GBC+08]) von Interesse.

Literatur

[BCF+08] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone und Christos Sofronis. Multiple Viewpoint Contract-Based Specification and Design. In Frank de Boer, Marcello Bonsangue, Susanne Graf und Willem-Paul de Roever, Hrsg., *Formal Methods for Components and Objects*, Jgg. 5382 d. *Lecture Notes in Computer Science*, Seiten 200–225. Springer Berlin / Heidelberg, 2008.

[BGBK08] Simon Barner, Michael Geisinger, Christian Buckl und Alois Knoll. EasyLab: Model-Based Development of Software for Mechatronic Systems. In *IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications*, Seiten 540–545, Beijing, China, Oktober 2008.

[BJP10] Antoine Beugnard, Jean-Marc Jezéquel und Noël Plouzeau. Contract aware components, 10 years after. *Electronic proceedings in theoretical computer science*, 37:1 – 11, Oktober 2010.

[BJPW99] Antoine Beugnard, Jean-Marc Jezequel, Noël Plouzeau und Damien Watkins. Making components contract aware. *Computer*, 32(7):38 –45, Juli 1999.

[Bro05] Manfred Broy. Automotive software and systems engineering. In *Proceedings of the 3rd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '05, Seiten 143 – 149, Washington, DC, USA, Juli 2005. IEEE Computer Society.

[Dam05] Werner Damm. Controlling Speculative Design Processes Using Rich Component Models. In *Proceedings of the Fifth International Conference on Application of Concurrency to System Design*, Seiten 118–119, Washington, DC, USA, Juni 2005. IEEE Computer Society.

[EMD08] Wolfgang Ecker, Wolfgang Müller und Rainer Dömer, Hrsg. *Hardware-dependent Software – Principles and Practice*. Springer, 2008.

[FGH06] Peter H. Feiler, David P. Gluch und John J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical Note CMU/SEI-2006-TN-011, Carnegie Mellon University, Februar 2006.

[GBC+08] Vered Gafni, Albert Benveniste, Benoit Caillaud, Susanne Graf und Bernhard Josko. D.2.5.4 Contract Specification Language (CSL). Bericht, SPEEDS project, 2008. SPEEDS deliverable D.2.5.4.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25:40–51, Oktober 1992.

[OMG10] Object Management Group. OMG. Systems Modeling Language 1.2, Juni 2010.

[PHG+09] Roberto Passerone, Imene Ben Hafaiedh, Susanne Graf, Albert Benveniste, Daniela Cancila, Arnaud Cuccuru, Sebastien Gerard, Francois Terrier, Werner Damm, Alberto Ferrari, Leonardo Mangeruca, Bernhard Josko, Thomas Peikenkamp und Alberto Sangiovanni-Vincentelli. Metamodels in Europe: Languages, Tools, and Applications. *IEEE Design & Test of Computers*, 26:38–53, Mai 2009.

[PKMM10] Robert Pyka, Felipe Klein, Peter Marwedel und Stylianos Mamagkakis. Versatile system-level memory-aware platform description approach for embedded MPSoCs. *SIGPLAN Not.*, 45:9–16, April 2010.

[TBHH07] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid und Kai Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, Seiten 29–40, Washington, DC, USA, Juli 2007. IEEE Computer Society.

[WHHW09] Zhonglei Wang, Andreas Herkersdorf, Wolfgang Haberl und Martin Wechs. SysCOLA: a framework for co-development of automotive software and system platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, Seiten 37–42, New York, NY, USA, Juli 2009. ACM.