

Eingebettete Systeme als Architektur mechanistischer Modelle

Architekturorientierte Modellkonstruktion mit CIP Tool®

Hugo Fierz

Zürcher Hochschule Winterthur, Postfach 805, CH-8401 Winterthur, Switzerland
hugo.fierz@zhwin.ch

Eingebettete Systeme wirken als reaktive Maschinen auf die umgebenden physikalischen Prozesse, und als Maschinen sollten solche Systeme auch entworfen werden. Fast alle Real-Time-Erweiterungen allgemeiner Entwicklungsmethoden verwenden deshalb Zustandsmaschine um das reaktive Verhalten eines Systems zu modellieren. Dieser mechanistische Ansatz wird dem Problem gerecht, aber nur selten dient er auch als Konstruktionsprinzip. Für die Definition der Interaktion zwischen den Zustandsmaschinen werden anstelle von Interaktionsmodellen bereits Codekonstrukte der Implementationssprache verwendet, meist in Form von Prozeduraufrufen. Dieser Aufsatz zeigt, wie sich ein ganzheitlicher mechanistischer Modellierungsansatz auf explizite Interaktionsmodelle abstützen kann. Kompositionelle Maschinenmodelle werden aus Komponenten aufgebaut, die in einem Architekturmodell über explizite Interaktionskonnektoren verbunden sind. Zum Schluss wird das Tool-basierte Modelling Framework der CIP-Methode vorgestellt, wo diese Entwurfskonzepte beispielhaft umgesetzt sind.

Die Hauptaufgabe eines eingebetteten Systems ist das Steuern physikalisch aktiver Prozesse der Systemumgebung. Um solche Prozesse steuern zu können muss ständig über Sensoren das aktuelle Verhalten dieser Prozesses erfasst und mittels Aktoren auf diese eingewirkt werden. Hier liegt auch der tiefere Grund für die mechanistische Natur eingebetteter Systeme, die sich von üblichen Informationssystemen durch ihr reaktives Verhalten unterscheiden. Beim Systementwurf spielen damit die kausalen Zusammenhänge zwischen erfassten Ereignissen und erzeugten Aktionen eine zentrale Rolle, d. h. Ursache und Wirkung von Systemreaktionen müssen für den Entwickler in jedem Fall nachvollziehbar sein.

Im ersten Teil dieses Aufsatzes gehen wir auf die reaktive Natur eingebetteter Systeme ein. Eine vergleichende Charakterisierung mit Problemstellungen aus klassischen IT-Bereichen zeigt, warum eingebettete Systeme grundsätzlich anders zu entwerfen sind. Steuerkomponenten modellieren wir wie üblich mittels Zustandsmaschinen. Eine nähere Betrachtung der Problemstruktur eingebetteter Systeme macht aber klar, dass dieser mechanistische Ansatz mit formalen Interaktionsmodellen vervollständigt werden muss.

Der zweite Teil erläutert, wie architekturorientierte Softwaretechniken bei der Konstruktion mechanisti-

scher Entwurfsmodelle eingesetzt werden. Hier treffen sich moderne Software-Paradigmen und herkömmliche Entwurfskonzepte für reale Maschinen. Der dritte Teil zeigt, wie die diskutierten Entwurfskonzepte umgesetzt werden. Als Beispiel einer Tool-basierten Realisierung präsentiert der letzte Teil das Modelling Framework der CIP-Methode.

1 DIE MECHANISTISCHE NATUR EINGEBETTETER SYSTEME

Eingebettete Systeme unterscheiden sich von anderen Computeranwendungen in erster Linie dadurch, dass sie direkt auf eine physikalische Umgebung einwirken. Im Gegensatz zu Computeranwendungen aus der IT-Welt wird der Computer nicht als Werkzeug, Rechner oder Datenverarbeitungsinstrument benutzt, sondern als elektronische Maschine, welche die umgebenden physikalischen Prozesse steuert.

In diesem Abschnitt charakterisieren wir die Natur von Embedded Problemen und diskutieren den wesentlichen Unterschied zu Anwendungen, die wir als *Informationssysteme* bezeichnen. Dazu zählen wir unter anderem Datenverarbeitungssysteme, On-Line Dienste oder Softwarewerkzeuge. Anschliessend begründen wir den mechanistischen Entwurfsansatz für

eingebettete Systeme und grenzen diesen zu den Ansätzen allgemeiner Software-Entwicklungsmethoden ab.

1.1 Merkmale eingebetteter Systeme

Unter einem eingebetteten System versteht man ein Computersystem, das fester Bestandteil eines Gerätes oder einer Anlage ist und für das Gesamtsystem bestimmte funktionale und leistungsmässige Anforderungen erfüllt. Typische Beispiele sind automatisierte Haushaltgeräte, Verkehrssteuerungen, Transportsysteme oder verteilte Produktionsanlagen.

Die zu steuernde Umgebung besteht immer aus einer Anzahl physikalischer Prozesse. Das können mechanische, chemische, elektronische oder hydraulische Prozesse sein, wie z. B. die Komponenten eines Liftsystems, die Freiheitsgrade einer Waschmaschine oder die elektrischen Verbindungen einer Schaltzentrale. Wie bei anderen Computeranwendungen gehören zur Umgebung auch Bediener, welche die Steuerung in Gang setzen und beeinflussen können. Dazu kommen Anzeigen, die Informationen zum aktuellen Systemzustand liefern.

Im Gegensatz zu den Bedienern haben die physikalischen Prozesse keine Intelligenz (nicht-kognitive Prozesse). Das Verhalten dieser Prozesse ist durch ihre physikalischen dynamischen Eigenschaften bestimmt. Im Unterschied zu einem Bediener "schert" sich ein physikalischer Prozess nie um das Verhalten des Computers. Diese Umstände sind der Hauptgrund für die besonderen Anforderungen an eingebettete Systeme:

- Harte Echtzeitbedingungen
- Fehlertoleranz
- Betriebssicherheit bei Fehlverhalten
- Systemtest an Simulationsmodellen
- Extreme Umgebungsbedingungen

Der wesentliche Unterschied zu Informationssystemen liegt aber im Verhalten eingebetteter Systeme, und es ist genau dieser Umstand, der für solche Systeme spezifische Entwicklungsmethoden notwendig macht. Die Steuerung physikalischer Prozesse verlangt eine ständige Interaktion mit diesen Prozessen. Ein eingebettetes System muss i. a. immer bereit sein, auf asynchrone und periodische Ereignisse mit entsprechenden Einwirkungen auf die externen Prozesse zu reagieren. Solche Systeme werden als *reaktiv* bezeichnet. Die Interaktion mit den parallel aktiven Prozessen lässt sich normalerweise nur in den

Griff bekommen, indem das eingebettete System aus Einheiten aufgebaut wird, die jeweils für die Steuerung eines einzelnen Prozesses zuständig sind. Damit ergibt sich aber als weitere Aufgabe die ständige Koordination der Steuereinheiten. Die Koordination muss so gestaltet sein, dass die gesteuerten Prozesse die angeforderte Gesamtfunktionalität der Anlage erbringen. Die Lösung dieser Aufgabe bestimmt grundlegend die Softwarequalität des entwickelten Systems. Sie ist deshalb bereits im Entwurf auf robuste Architekturmodelle abzustützen.

Bei der Koordinierung der Steuereinheiten zeigt sich ein weiterer entscheidender Unterschied zu Informationssystemen. Bei eingebetteten Systemen bestimmt die aktive Umgebung, wann und wie schnell das Rechnersystem zu reagieren hat. Bei interaktiven Informationssystemen ist das gerade umgekehrt. Hier bestimmt der Rechner, wann welche Eingaben gemacht werden können, und wie lange das Erbringen von Informationsdiensten dauert.

1.2 Mechanistische Entwurfsmodelle

'General-Purpose'-Methoden und -Notationen, wie zum Beispiel Structured Analysis [1] oder UML [2], sind ohne Ausnahme für Informationssysteme entwickelt worden. Vereinfacht ausgedrückt, ist die Aufgabe solcher Systeme das Verwalten von Daten und das Erbringen von Informationsdiensten. Oft kommen komplexe Berechnungen und intelligentes Datenmanagement dazu. Solche Systeme werden als *transformationell*, bzw. als *interaktiv* bezeichnet. Methodisch basiert die Softwareentwicklung solcher Systeme im wesentlichen auf prozeduraler und Datenabstraktion. Für die Systemstrukturierung steht *Stepwise Refinement* im Vordergrund, und Systemfunktionen werden meistens als hierarchische Komposition von Diensten realisiert. Als Interaktionsmechanismen zwischen Objekten oder Modulen dienen Prozeduraufrufe. Prozeduren eignen sich hervorragend für den Informationsaustausch zwischen Entitäten verschiedener Abstraktionsebenen.

Eingebettete Systeme sind anders. Die Hauptaufgabe solcher Systeme ist nicht, Daten zu verwalten oder komplexe Berechnungen auszuführen, sondern eine Anzahl physikalischer Prozesse zu steuern. Wegen der ständigen externen und internen Interaktionen der verschiedenen Steuereinheiten ist es naheliegend, solche Systeme als reaktive Maschinen zu entwerfen. Auch Maschinenmodelle abstrahieren, wie Programmiersprachen, vom Verhalten des Mikropro-

zessors, der ja selbst eine physikalische Maschine ist. Der Unterschied zu Programmiersprachen ist die Art der Abstraktion. Für die Beschreibung der einzelnen Freiheitsgrade eines Maschinenmodells werden typischerweise elementare Zustandsmaschinen verwendet. Die Koordination dieser Teilmaschinen wird durch Übertragung von Stimuli oder spezifizierte Zustandsabhängigkeiten realisiert. Im Unterschied zur prozeduralen Abstraktion findet hier die Interaktion zwischen Komponenten statt, deren Verhalten auf derselben Abstraktionsebene beschrieben wird (Peer-to-Peer Interaktion). Der Zweck der Interaktion ist eine bestimmte Kooperation der Komponenten zu erzeugen, die das gewünschte Gesamtverhalten des Maschinenmodells erbringt.

Die Interaktion zwischen den Maschinenkomponenten kann im allgemeinen nicht hierarchisch strukturiert werden. Externe Ereignisse für einzelne Steuerkomponenten können einen direkten Einfluss auf das gesamte Systemverhalten haben, und selbst zwischen elementaren Komponenten muss mit zyklischen Abhängigkeiten gerechnet werden.

Wie für reale Maschinen ist Komposition das grundlegende Konstruktionsprinzip. Ausgehend von den zu steuernden externen Prozessen wird man zuerst elementare Steuerkomponenten definieren, die jeweils für die Interaktion mit einem bestimmten externen Prozess zuständig sind. In weiteren Entwicklungsschritten wird für die Koordination der Steuerkomponenten gesorgt. Dazu müssen meistens weitere Komponenten eingeführt werden, um Strukturkonflikte zwischen den bereits bestehenden Steuerkomponenten zu vermeiden.

Auch 'General-Purpose'-Methoden berücksichtigen die mechanistische Natur eingebetteter Systeme, indem sie sog. Real-Time Erweiterungen in Form von Zustandsmaschinenmodellen anbieten, wie zum Beispiel SDRS [3] oder RT-UML [2]. Was bei diesen Erweiterungen jedoch fehlt sind explizite Interaktionsmodelle, mit deren Hilfe die Kooperation der Zustandsmaschinen auf derselben Abstraktionsebene wie die Maschinenmodelle beschrieben werden können. Für RT-UML wird diese Problematik am Schluss von Teil 3 diskutiert (siehe *Beispiele aktueller Real-Time-Modelle*).

2 KOMPONENTEN UND ARCHITEKTUR

Der Komponentenbegriff wird in der Softwareentwicklung sowohl für vorfabrizierte Module (Units of Deployment) als auch für Bausteine (Units of Con-

struction) kompositionell konstruierter Systeme verwendet. Im Abschnitt 2.1 diskutieren wir verschiedene Aspekte und Verwendungsarten von Komponenten. Im Zusammenhang mit mechanistischen Entwurfsmodellen interessiert uns hier in erster Linie die architekturorientierte Komposition von Komponenten, auf die wir im Abschnitt 2.2 näher eingehen.

2.1 Komponenten

Was ist eine Komponente? In der Literatur findet man die verschiedensten Antworten, und an komponentenorientierten Workshops sind Komponentendefinitionen oft Anlass zu ausgiebigen Diskussionen. Hier vier Beispiele von Komponentendefinitionen:

Eine Komponente ist ein Teil eines Ganzen.
(Duden)

Eine Software-Komponente ist ein vorfabriziertes selbstkonsistentes Modul. (J. Sametinger)

A software component is a unit of composition with contractual interfaces and explicit context dependencies only. (C. Szypersky)

A component is a static abstraction with plugs. (O. Nierstrasz)

Software-Komponenten sind offenbar Systemteile, die nur über explizite Schnittstellen mit ihrer Umgebung verbunden werden können.

Das Kompositionsprinzip

Entscheidend für komponentenorientiertes Konstruieren ist das Kompositionsprinzip:

Beim Verbinden einer Komponente mit ihrer Umgebung wird die Komponente nie verändert.

Das Kompositionsprinzip kann damit direkt als Kriterium dienen um zu entscheiden, ob ein System aus Komponenten aufgebaut ist. Sowohl modulare als auch objektorientierte Programme verletzen zum Beispiel das Kompositionsprinzip. Objekte werden über Methodenaufrufe verknüpft. Das Server-Objekt stellt zwar eine Schnittstelle (*Provided Interface*) in Form von Public Methoden zur Verfügung, der Methodenaufruf muss aber im Innern des Client-Objekts definiert werden. Was beim Client-Objekt fehlt, ist ein sog. *Used Interface*, das mit dem *Provided Interface* des Server-Objekts zu verbinden wäre.

Um Verknüpfungen zwischen Komponenten herzustellen, sind im allgemeinen explizite Verbindungselemente notwendig, die als *Konnektoren* bezeichnet werden. Die Ausdruckskraft objektorientierter Programmiersprachen erlaubt natürlich, Ob-

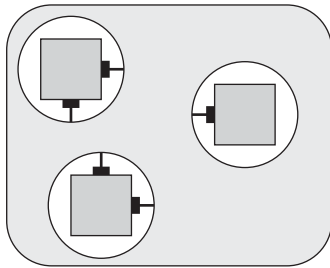
jekte zu definieren, welche die Rolle von Konnektoren übernehmen, wie zum Beispiel Event Adapter Objekte für JavaBeans.

Oft müssen bei Systemänderungen sowohl das Komponentenverhalten als auch die Verbindungsstruktur verändert werden. Die Unabhängigkeit von Komponenten und Verbindungen erweist sich hier als fundamentale Eigenschaft für die Software-Qualität und ist der tiefere Grund für die robuste Flexibilität komponentenbasierter Systeme.

Zur Verwendung von Komponenten

Es gibt grundsätzlich zwei Arten, wie Komponenten in einem System verwendet werden können: Integration oder Komposition.

Integration

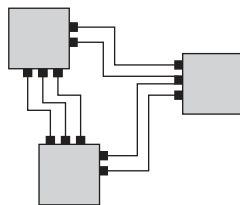


Figur 1. Komponenten integriert in einer Applikation

Komponenten werden als ausführbare *Black-Boxes* in einer Applikation eingebunden. Diese Art der Verwendung ergibt sich, wenn Komponenten als vorfabrizierte Module erstanden und als Standardbausteine eingesetzt werden (Units of Deployment).

Komposition mittels "Verdrahtung"

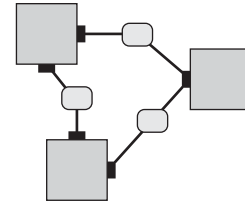
Komponenten mit elementaren Schnittstellen werden mittels entsprechenden elementaren Verbindungselementen verbunden. Diese Konstruktionstechnik ist z. B. notwendig, um elektrische Bauteile zusammenzusetzen (reale Verdrahtung).



Figur 2. Verdrahtete Komponenten

Auf diese Art werden auch Hardwarekomponenten zusammengesetzt, im Design mit VHDL und bei der Implementation in integrierten Schaltungen. Ein bekanntes Software-Beispiel sind Simulink-Modelle, welche die Übertragung von Float-Werten zwischen Funktionsblöcken mit Signal-Linien darstellen.

Architekturorientierte Komposition



Figur 3. Konnektoren mit Interaktionssemantik

Komponenten werden über ihre Schnittstellen mit spezifischen Konnektoren verbunden. Jeder Konnektortyp realisiert einen bestimmten Interaktionsmechanismus. Spezifikationsprachen, die diese kompositionelle Softwaretechnik unterstützen, bezeichnet man als *Architectural Description Languages* (ADL) [5, 6]. Mittels Konnektoren wird die Software-Architektur eines Systems explizit beschrieben. Komponenten werden oft als sogenannte *White-Boxes* verwendet, das heißt beim Zusammensetzen ist das Innere der Komponenten sichtbar, wird dabei aber nicht verändert. Auch Programmsysteme definieren Interaktion zwischen Programmteilen (procedure call, shared variable access). Der Unterschied zu ADL's ist, dass die Interaktionsverbindungen und damit die Architektur nur implizit definiert sind.

Architekturorientierte Komposition ist die mächtigste kompositionelle Softwaretechnik, weil zusätzlich zur Komponentenabstraktion die Interaktion zwischen Komponenten explizit und formal beschrieben wird. Wir gehen im nächsten Abschnitt detaillierter auf diese Konstruktionstechnik ein, weil sie ideal den mechanistischen Entwurf eingebetteter Systeme unterstützt.

Ein aktuelles Beispiel architekturorientierter Softwarekomposition ist die Installation von Gerätetreibern unter Linux. Treibermodule können im laufenden System im Virtual Filesystem Switch (VFS) eingeklinkt werden. Der VFS ist der Konnektor zu den virtuellen Geräten, die der Programmierer in Linux als abstrakte Device Files anspricht.

2.2 Architekturorientierte Komposition (Architectural Composition)

Architekturorientierte Komposition [5, 6] ist die reinste Form komponentenorientierter Entwicklung. Ein System wird als Komposition von Komponenten gebaut, indem Komponenten mittels Konnektoren verbunden werden.

Komponenten



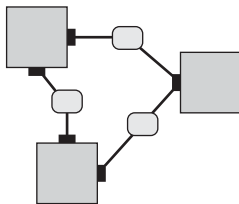
Komponenten haben lokale Zustände und ein spezifisches Eigenverhalten. Sie sind der Ort, wo Funktionen und Berechnungen ausgeführt werden. Für die Verknüpfung mit Konnektoren stellen Komponenten entsprechende Schnittstellen zur Verfügung, die als *Ports* bezeichnet werden (shared events, shared states).

Konnektoren



Konnektoren sind wie Komponenten 'First Class Entities'. Ihre Aufgabe ist es, Interaktion zu erzeugen, das heißt Wirkung und Daten zwischen Komponenten zu übertragen. Verschiedene Konnektortypen unterstützen dabei unterschiedliche Interaktionsmechanismen. Zudem sind Konnektoren der Ort, wo die Relationen zwischen Komponentenschnittstellen definiert sind.

Konfiguration



Eine Konfiguration definiert Verknüpfungen zwischen Komponenten und Konnektoren. Eine Verbindung zwischen Komponenten wird typisch in zwei Schritten erstellt:

1. Verknüpfen eines Konnektors mit Komponentenports. Dabei müssen die Port-Typen dem Typ des Konnektors entsprechen.
2. Spezifikation der Interaktionsrelation, das heißt die Ereignisse oder Zustände der verbundenen Ports müssen zueinander in Beziehung gesetzt werden.

Kompositionalität

Die Bedeutung einer Komposition ergibt sich aus der Bedeutung der Teile und den Kompositionsregeln. Auf architekturorientierte Komposition angewendet heißt das:

Das Verhalten einer Komposition ist bestimmt durch das Eigenverhalten der Komponenten und die Interaktionsmechanismen der verwendeten Konnektoren.

Die Kompositionalitätseigenschaft wirkt sich entscheidend auf die Verständlichkeit komplexer Systembeschreibungen aus, weil die Bedeutung eines einzelnen Systemteils nie vom Kontext abhängt.

3 DIE ARCHITEKTUR VON REAKTIVEN MASCHINENMODELLEN

Die Interaktion zwischen Komponenten realer Maschinen wird von Natur aus durch physikalische Konnektoren übertragen. Die physikalische Realität lässt gar keine andere Konstruktionstechnik zu. Um zum Beispiel zwischen den Teilen einer mechanischen Maschine Interaktion zu erzeugen sind reale Verbindungselemente notwendig, die mechanische Kräfte übertragen (Kupplungen, Zahnräder, Keilriemen). Auf analoge Weise werden wir unsere reaktiven Maschinen bauen.

Wir wollen nicht Rechnerprogramme schreiben, deren Abläufe bestimmte Reaktionen implementieren, sondern reaktive Maschinen konstruieren, die auch rechnen können.

Dieser Teil beschreibt, wie die architekturorientierten Konstruktionsprinzipien beim Bau reaktiver Maschinen anzuwenden sind. Am Schluss diskutieren wir bei drei bekannten Modellierungstechniken, wie weit sie die vorgeschlagenen Entwurfskonzepte realisieren.

3.1 Interaktionskonnektoren für Zustandsmaschinen

Für einen ganzheitlichen mechanistischen Modellierungsansatz ist es naheliegend, reaktive Maschinenmodelle mittels Komponenten und Konnektoren aufzubauen. Als elementare Maschinenkomponenten bieten sich klar Zustandsmaschinen (FSM, Finite State Machine) an. Für die Interaktionskonnektoren müssen jedoch spezifische Konnektortypen definiert werden, die erlauben, Stimuli zu übertragen und Zustandsabhängigkeiten zu erzwingen. Stimulusübertragung erinnert an physikalische Systeme, die Wirkung mittels Energiequanten übertragen. Zustands-

abhängigkeiten entsprechen eher statischen Kraftfeldern, welche die Bewegungsfreiheit einschränken.

Als Komponenten von Embedded Modellen verwenden wir *erweiterte* Zustandsmaschinen (EFSM, Extended Finite State Machine). Die Erweiterungen reiner Zustandsmaschinen bestehen aus lokalen Variablen, Operationen und Bedingungen, damit diese Maschinen auch rechnen und Daten verarbeiten können. Die Erweiterungen gelten als primitive Komponenten (Computational Primitives), die in Form von Code-Konstrukten in die Zustandsmaschinen eingebunden werden.

Die Interaktionsmodelle sind für die reinen Zustandsmaschinen definiert, so dass bereits mit reinen Zustandsmaschinen selbstkonsistente und ausführbare Systemmodelle gebildet werden können. Um die Komposition erweiterter Zustandsmaschinen zu unterstützen, werden auch die Interaktionsmodelle erweitert. Stimuli können z. B. Daten übertragen, und Zustandsabhängigkeiten können zu Variablenabhängigkeiten erweitert werden.

Die Komposition der Zustandsmaschinen erfolgt über spezifische Verbindungselemente, die, wie oben beschrieben, als Konnektoren bezeichnet werden.



Figur 4. Konnektor mit Interaktionsrelation

Der Typ eines Konnektor bestimmt, wie die verbundenen Komponenten interagieren. Dazu müssen die Schnittstellen der verbundenen Zustandsmaschinen zueinander in Beziehung gesetzt werden, z. B. durch Abbilden gesendeter Stimuli in empfangene.

3.2 Synchroner und asynchroner Komposition von Verhaltensmodellen

Beim Zusammensetzen von Komponenten mit zeitlichem Verhalten muss zwischen synchroner und asynchroner Komposition unterschieden werden. Das gilt für allgemeine Softwarekomponenten gleich wie für Zustandsmaschinen.

Bei asynchroner Komposition gibt es a priori keine Gleichzeitigkeitsbeziehungen zwischen den Komponenten. Jede Komponente hat Ihre eigene Ausführungsgeschwindigkeit. Synchronisation kann nur über spezifische Interaktionsmechanismen erzeugt werden. Asynchrone Komposition ist notwendig, um verteilte Systeme beschreiben zu können.

Bei synchroner Komposition agieren die Kompo-

nenten in global synchronen Schritten (Lock Step). Auf ein Ereignis kann eine oder können mehrere Zustandsmaschinen gleichzeitig reagieren, während die anderen bis zum Abschluss der Reaktion blockiert sind. Für die Modellierung reaktiver Systeme bringt synchrone Komposition gegenüber asynchroner grosse Vorteile, weil der Lock-Step-Mechanismus viele unnötige Race-Conditions verhindert.

Bei der Definition von Interaktionsmechanismen muss natürlich festgelegt werden, ob ein Mechanismus für synchrone oder asynchrone Komponenten zu verwenden ist. Wichtig ist, dass für beide Kompositionsarten Interaktionsmodelle zur Verfügung stehen, damit problemspezifisch die beste Lösung gefunden werden kann.

3.3 Beispiele aktueller Real-Time-Modelle

Wir diskutieren bei drei bekannten Modellierungstechniken, wie weit architekturorientierte Komposition unterstützt ist.

RT-UML – teilw. mechanistisch, teilw. OO

RT-UML Systeme [4] bestehen aus Objekten, deren Verhalten mit Zustandsmaschinen modelliert ist. Die Interaktion zwischen den Objekten basiert jedoch auf OO-Prozeduraufrufen (implizite Interaktionsverbindungen). Diese Vermischung von Modellierungs- und Programmierkonstrukten verunmöglicht kompositionelle Gesamtmodelle. Diese Entwurfstechnik wird oft als *Visual Programming* bezeichnet.

Statecharts – mechanistisch, keine Komponenten

Klassische Statecharts-Modelle [7] sind hierarchisch aus synchronen Subcharts (EFSM) aufgebaut. Als Interaktionsmodelle sind Stimulusübertragung und Zustandsabhängigkeiten unterstützt. Die Interaktion zwischen Subcharts wird aber lokal in den involvierten Subcharts in Form von Inskriptionen spezifiziert. Subcharts sind damit keine Komponenten, und die Interaktion zwischen Subcharts kann nicht durch Architekturbeschreibungen spezifiziert werden.

SDL – mechanistisch u. Komponentenarchitektur

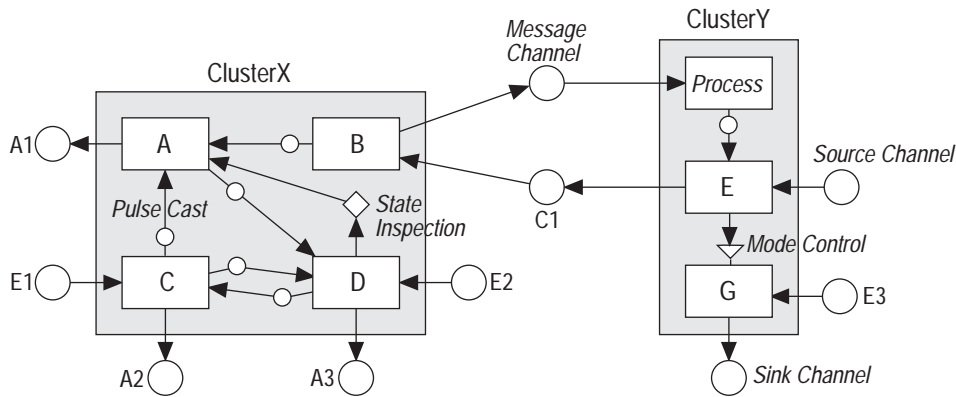
Mit SDL [8] wird ein System mittels asynchroner Prozesse (EFSM) spezifiziert. Als Interaktionsmodell dienen fifo-Kanäle (message queues), die in einem Architekturmodell mit den Prozessen verbunden werden. Damit unterstützt SDL vollumfänglich die vorgeschlagenen Entwurfskonzepte. Hingegen ist bei SDL keine synchrone Komposition von Prozessen möglich, wie das für die Modellierung reaktiver Systemmodelle notwendig ist.

4 CIP-METHODE – ARCHITEKTURORIENTIERTE MECHANISTISCHE ENTWURFSMODELLE

CIP-Modelle [9, 10, 11] werden mit CIP Tool® konstruiert (CIP, Communicating Interacting Processes). Die Interaktionsverbindungen zwischen Prozessen (EFSM) erzeugt man grafisch in entsprechenden

Architektur-Editoren. Interaktionsstrukturen können automatisch analysiert werden, und für vollständige Modelle generiert das Tool ausführbare Softwarekomponenten für die Zielsysteme (C, Java).

4.1 CIP-Modelle im Überblick



Figur 5. Architektur eines allgemeinen CIP-Modells

Ein CIP-Modell besteht aus asynchronen Clustern, die aus synchronen Prozessen (Zustandsmaschinen) zusammengesetzt sind (Figur 5).

CIP stellt vier spezifische Interaktionsmechanismen zur Verfügung: *Message Passing*, *Pulse Cast*, *State Inspection* und *Mode Control*. Alle Prozesse eines Modells können asynchron Messages austauschen (*Message Passing*). Innerhalb eines Clusters bewirkt *Pulse Cast* synchrone Kettenreaktionen. *State Inspection* erlaubt es, Zustandsabhängigkeiten zwischen Prozessen zu spezifizieren. Mit *Mode Control* können die Betriebsarten (Modi) eines Prozesses durch andere Prozesse bestimmt werden.

Wir gehen in diesem Aufsatz nur auf Modelle mit reinen Zustandsmaschinen (FSM) ein. Allgemeine CIP-Modelle werden mittels erweiterter Zustandsmaschinen (EFSM) gebaut. Die Erweiterungen (*Computational Primitives*) bestehen aus lokalen statischen Variablen, Operation und Bedingungen. Auch die Interaktionsmechanismen werden entsprechend erweitert: Stimuli können Daten übertragen und Prozesse können über prozedurale Schnittstellen Variablen anderer Prozesse inspizieren.

Prozessports (für reine Zustandsmaschinen)

Prozessports sind Komponentenschnittstellen, die mit entsprechenden Interaktionskonnektoren zu verknüpfen sind.

Stimulusübertragung

Inport1, Inport2, ...

Jeder Port empfängt von einem *Message Passing* Konnektor (Channel) Meldungen, die im Prozess Zustandsübergänge auslösen.

Outport1, Outport2, ...

Jeder Port übergibt einem *Message Passing* Konnektor die Meldungen, die in Zustandsübergängen in den Output geschrieben worden sind.

ImpulsePort

Der Port empfängt i. a. von mehreren *Pulse Cast* Konnektoren Impulse, die im Prozess Zustandsübergänge auslösen können.

OutputPort

Der Port übergibt einem *Pulse Cast* Konnektor die in Zustandsübergängen erzeugten Outputpulse.

Zustandsabhängigkeiten

Gate1, Gate2, ...

Ein *State Inspection* Konnektor setzt den boolschen Wert des Gate.

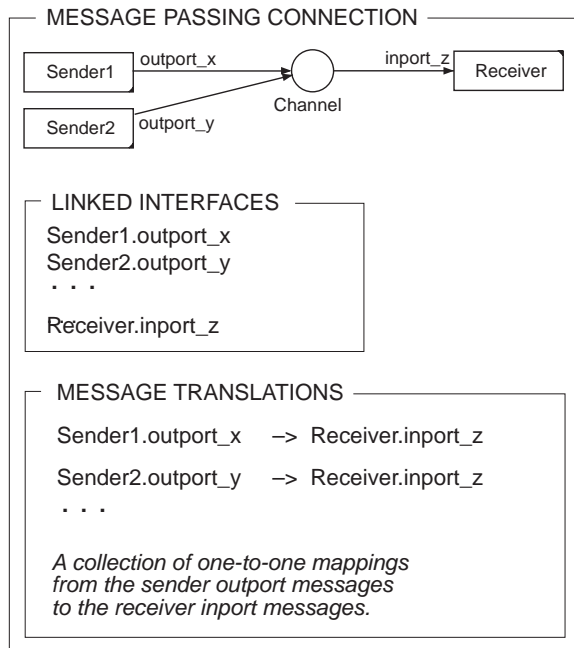
ModePort

Ein *Mode Control* Konnektor setzt den aktiven Modus des Prozesses.

StatePort

State Inspection und *Mode Control* Konnektoren lesen den aktuellen Zustand des Prozesses.

4.2 Message Passing



Alle Prozesse eines CIP-Modells können asynchron Messages austauschen. Messages sind Stimuli,

die in Zustandsübergängen gesendet und empfangen werden. Die Konnektoren für *Message Passing* Interaktion heissen *Channels* und stellen fifo-gepufferte Kommunikationsverbindungen dar. Periphere Kanäle modellieren die Verbindung von und zu der realen Umgebung. Die Implementation eines Kanals bestimmt, wann die älteste Message an den Empfänger übertragen wird. Dieser muss jede empfangene Meldung erwarten, andernfalls muss ihr Auftreten als Kontextfehler behandelt werden (Exception).

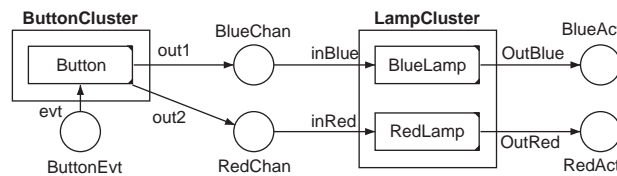
Ein Kanal verbindet Sender-Outports mit einem Empfänger-Inport. Die Interaktion ist durch ein-eindeutige Zuordnung der Sender- zu den Empfänger-meldungen spezifiziert (Message Translation).

Message Passing Beispiel

Im folgenden CIP-Modell wird der *Button* Prozess durch reale Button-Ereignisse getriggert (*Down*, *Up*), um die Meldungen *Start* und *End*, bzw. *Trigger* an die beiden Lampenprozesse zu senden. Als Reaktion schalten die Lampenprozesse ihre Lampe ein und aus. Die blaue Lampe brennt doppelt so häufig wie die rote. Die Übertragung der Meldungen an die Lampenprozesse ist asynchron, dass heisst es ist zeitlich nicht bestimmt, wann die Lampen ein und ausgeschaltet werden.

SYSTEM SimpleMessagePassing

COMMUNICATION NET SwitchLampNet



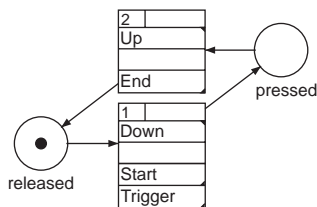
MESSAGE TRANSLATION

Button.out1 → BlueLamp.inBlue
 Start → On
 End → Off

Button.out2 → RedLamp.inRed
 Trigger → Change

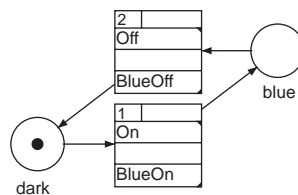
CLUSTER ButtonCluster

PROCESS Button

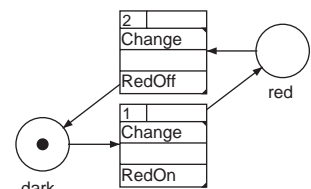


CLUSTER LampCluster

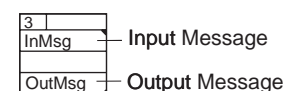
PROCESS BlueLamp



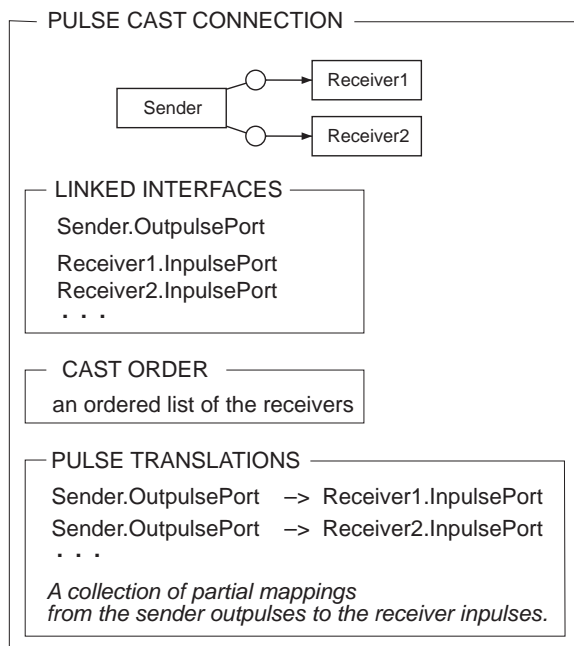
PROCESS RedLamp



Legende:



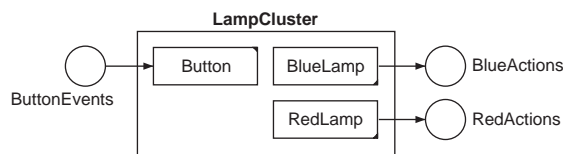
4.3 Pulse Cast



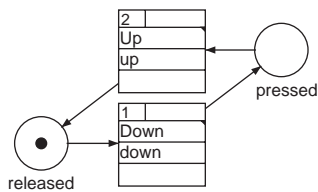
Die Prozesse eines Clusters wechselwirken über *Pulse Cast* Interaktion. Pulse sind Stimuli, die in Zu-

SYSTEM SimplePulseCast

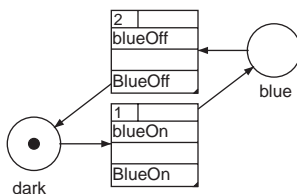
COMMUNICATION NET SourcesAndSinks



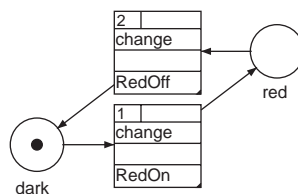
PROCESS Button



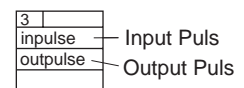
PROCESS BlueLamp



PROCESS RedLamp



Legende:



standsübergängen gesendet und empfangen werden. Ein Puls muss nicht erwartet werden. Die durch Pulse Cast bewirkte Kettenreaktion wird immer durch eine Message eines Channels ausgelöst. Im Cluster kann erst wieder eine Message empfangen werden, wenn die Kettenreaktion beendet ist. Dieses Verhalten, das als *Run-to-Completion* Semantik bezeichnet wird, verhindert Race-Conditions zwischen Pulsen in Interaktionsketten und anstehenden Messages.

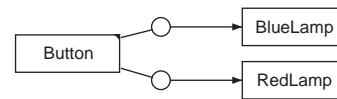
Die Interaktion ist durch partielle Abbildungen (eine pro Empfänger) der Sender Outpulse auf Empfänger Impulse spezifiziert (Pulse Translation).

Pulse Cast Beispiel

Die Funktionalität des folgenden CIP-Modells ist gleich wie im obigen Message Passing Beispiel, die Betätigung eines Knopfes bewirkt das Ein- und Ausschalten von zwei Lampen. Der Unterschied liegt im zeitlichen Verhalten. Als Folge von jedem *Down*-Ereignis wird ein *down*-Puls abgesetzt, der unmittelbar eine Reaktion beider Lampenprozesse hervorruft. Ein *Up*-Ereignis bewirkt hingegen nur bei *BlueLamp* eine Zustandsänderung. *RedLamp* ist während dieser Zeit blockiert (Run-to-Completion Semantik).

CLUSTER LampCluster

PULSE CAST NET



CAST ORDER

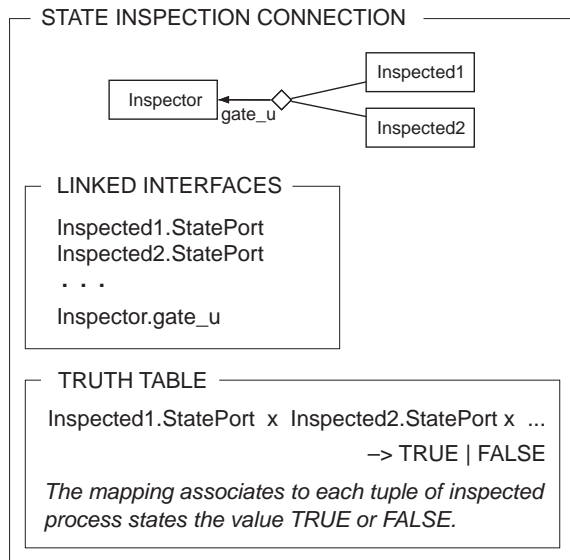
BlueLamp, RedLamp.

PULSE TRANSLATION

SENDER Button

down -> BlueLamp.blueOn,
RedLamp.change
up -> BlueLamp.blueOff

4.4 State Inspection



Die Reaktion eines Empfängers auf eine Message oder einen Puls kann über *State Inspection* Interaktion von den Zuständen anderer Prozesse desselben Clusters abhängen.

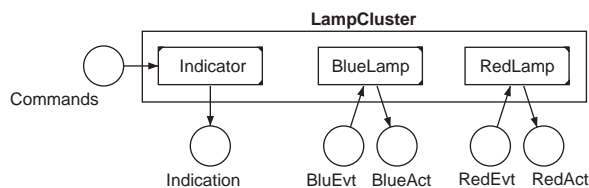
Ein Inspektionskonnektor wird mit einem Gate des Inspizienten und mit den inspizierten Prozessen verknüpft (*StatePort*). Die Interaktion wird durch eine Wahrheitstabelle spezifiziert (N inspizierte Prozesse bedingen eine N-dimensionale Tabelle). Die Wahrheitstabelle definiert eine boolesche Funktion, die den Zustand (True, False) des angeschlossenen Gate bestimmt. Gates werden in non-deterministischen Verzweigungen von Transitionstrukturen als Bedingungen verwendet.

State Inspection Beispiel

Im folgenden CIP-Modell werden alle Prozesse von aussen getriggert, jede der beiden Lampen hat einen eigenen Knopf. Als Reaktion auf den Befehl *Indicate* aktiviert der *Indicator*-Prozess eine bestimmte Zeit lang eine Anzeige, wenn mindestens eine der beiden Lampen brennt; andernfalls wird ein *Beep* erzeugt. Die Reaktion auf den *Indicate*-Befehl im *idle*-Zustand ist vom Gate *indGate* abhängig (SWITCH Konstrukt). Der Gate-Zustand wird durch die State Inspection Verbindung aufgrund der spezifizierten Truth Table gesetzt.

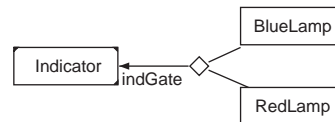
SYSTEM SimpleInspection

COMMUNICATION NET SourcesAndSinks



CLUSTER LampCluster

INSPECTION NET

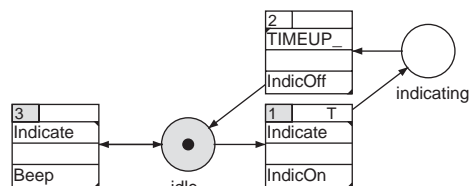


TRUTH TABLE

INSPECTOR Indicator
GATE indGate
RESPONDERS BlueLamp, RedLamp
TRUE <- (blue, dark)
 (blue, red)
 (dark, red)
FALSE <- (dark, dark)

PROCESS Indicator

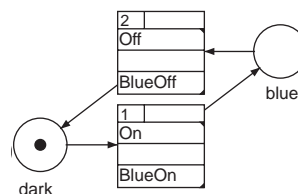
GATE indGate



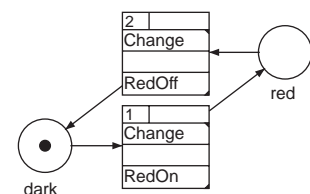
SWITCH

STATE idle MESSAGE Indicate
TRANSITION 1 / GATE indGate
TRANSITION 3 / ELSE_

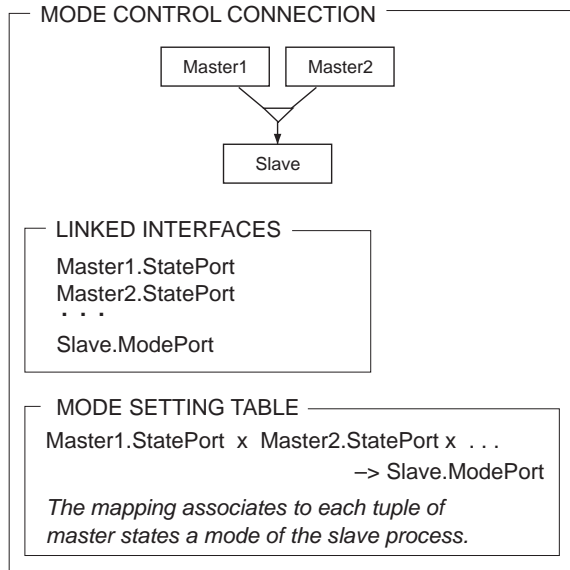
PROCESS BlueLamp



PROCESS RedLamp



4.5 Mode Control



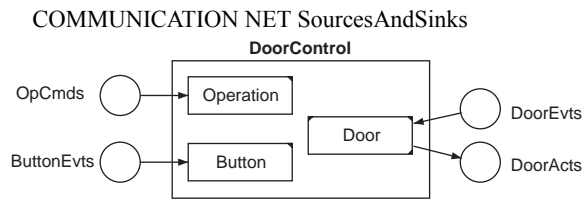
Varianten des Prozessverhaltens sind durch verschiedene Modi spezifiziert. Jeder Modus ist durch eine separate Transitionsstruktur definiert. Mittels *Mode Control* Interaktion können andere Prozesse desselben Clusters jederzeit einen Moduswechsel induzieren. Der aktive Modus des beeinflussten Prozesses ist jeweils durch die aktuellen Zustände der steuernden Prozesse bestimmt. Die Ports für einen Mode Control Konnektor sind damit implizit gegeben (*ModePort*, bzw. *StatePorts*).

Die Zustandsabhängigkeit wird durch eine Mode Setting Tabelle spezifiziert (N Steuerprozesse bedingen eine N-dimensionale Tabelle). Die Tabelle definiert für jede Zustandskombination den aktiven Modus des gesteuerten Prozesses.

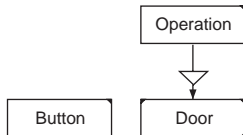
Mode Control Beispiel

Im folgenden CIP-Modell wird eine Türe auf zwei Arten gesteuert. Im Normalbetrieb öffnet sie sich auf Knopfdruck und schliesst dann wieder. Bei Alarm wird die Türe in jedem Fall geöffnet und dann offen gehalten. (Modell-Erläuterung auf der nächsten Seite)

SYSTEM SimpleModeControl



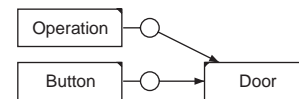
MODE CONTROL NET



MODE SETTING

SLAVE Door
MASTER Operation
userControl <- regular
forcedOpening <- exception

PULSE CAST NET

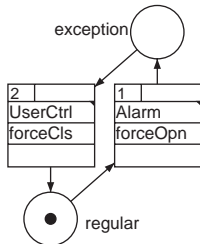


PULSE TRANSLATIONS

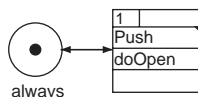
SENDER Button
doOpen -> Door.doOpen

SENDER Operation
forceCls -> Door.forceCls
forceOpn -> Door.forceOpn

PROCESS Operation

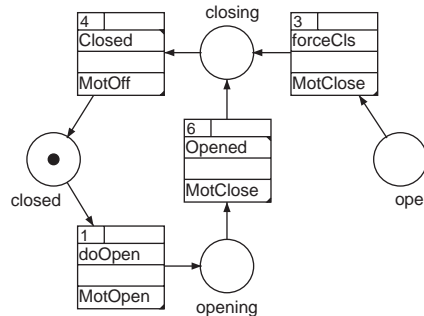


PROCESS Button

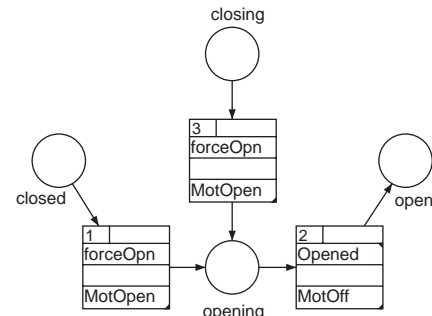


PROCESS Door

MODE userControl



MODE forcedOpening



Der Prozess *Operation* ist für die Umschaltung in die entsprechenden Modi verantwortlich. Wenn zum Beispiel die *Alarm*-Message bei diesem Prozess den Zustandsübergang von *regular* zu *exception* erzeugt, wechselt der inaktive *Door*-Prozess unmittelbar vom *userControl* zum *forcedOpening*-Modus. Der vom *Operation*-Prozess erzeugte *forceOpn*-Puls bewirkt anschliessend durch Pulse Cast Interaktion im neuen Modus des *Door*-Prozesses das Öffnen der Türe, falls diese geschlossen oder gerade am Schliessen ist.

5 SCHLUSSWORT

Dieser Aufsatz befasst sich mit der Konstruktion funktionaler Modelle für eingebettete Systeme. Auf das methodische Vorgehen bei der Modellentwicklung wurde jedoch nicht eingegangen. Dazu sind in [9, 12] und in Kapitel 4 von [10] prozessorientierte Entwicklungsprinzipien vorgeschlagen.

Ein weiteres wichtiges, auch nicht behandeltes Thema, ist die Konstruktion der Softwareverbindung zwischen dem Modell-Code und der oft heterogenen Prozessperipherie (Interface Devices). Für Beiträge zu diesem Problembereich im Zusammenhang mit den vorgestellten Konstruktionsprinzipien verweisen wir auf die grundlegenden Arbeiten von HO. Trutmann [13, 14] und auf die Kapitel 3 und 7 von [10].

Zusammenfassung

Der Aufsatz schlägt architekturorientierte Maschinenmodelle als Entwurfsmittel für eingebettete Systeme vor. Praktisch alle Embedded-Methoden verwenden Zustandsmaschinen, um Steuerabläufe zu modellieren. Was aber fast immer fehlt ist eine kompositionelle Modellierungstechnik, die erlaubt mittels einfacher Zustandsmaschinen komplexere reaktive Maschinenmodelle zu konstruieren.

Die hier präsentierte architekturorientierte Kompositionstechnik setzt die von Architekturbeschreibungssprachen (Architectural Description Languages) verwendeten Konstruktionsprinzipien ein. Diese gehen von Komponenten aus, die mit Hilfe expliziter Verbindungselemente, sog. Konnektoren, zusammengesetzt werden. Die Konnektoren haben die Aufgabe, Interaktion zwischen den Komponenten zu erzeugen.

Die architekturorientierte Konstruktion reaktiver Maschinenmodelle verlangt, dass sowohl für Komponenten als auch für Konnektoren entsprechende Metamodelle definiert sind. Für die Modellierung reaktiver Komponenten bieten sich erweiterte Zu-

standsmaschinen an. Von den Interaktionsmodellen fordern wir, dass sie Stimulusübertragung und Zustandsabhängigkeiten zwischen Zustandsmaschinen erzeugen. Bei der Definition der Interaktionsmechanismen (Konnektorsemantik) muss zudem festgelegt werden, ob die entsprechenden Konnektoren für synchron oder asynchron kooperierende Zustandsmaschinen einzusetzen sind.

Als methodische Realisierung der diskutierten Entwurfskonzepte stellen wir das werkzeugbasierte Modelling Framework der CIP-Methode vor. CIP unterstützt sowohl synchrone als auch asynchrone Komposition erweiterter Zustandsmaschinen. Alle Zustandsmaschinen eines CIP-Modells haben die Möglichkeit, asynchron über *Message Passing* Konnektoren (Message Queues) zu kommunizieren. Synchrone Zustandsmaschinen können mittels drei verschiedener Konnektortypen verbunden werden: *Pulse Cast* Interaktion modelliert synchrone Stimulusübertragung. *State Inspection* Interaktion erzwingt Zustandsabhängigkeiten, und *Mode Control* Interaktion bestimmt bei Zustandsmaschinen mit mehreren Modi den aktiven Modus.

Schlussfolgerungen

Architekturorientierte Softwarekonstruktion wirkt sich positiv auf den ganzen Software Lifecycle aus. Der Gewinn aus der konsequenten Anwendung des architekturorientierten Kompositionsprinzips sind Modelle, deren Robustheit an reale Maschinen erinnern, die aber die Flexibilität von Software besitzen. Kompositionell aufgebaute Systeme mit expliziten Architekturbeschreibungen sind verständlicher und einfacher zu ändern als ablaforientierte Programmsysteme. Die Gefahr unerwünschter Seiteneffekte wird stark vermindert, da Änderungen in Komponenten wegen der Konnektoren von andern Komponenten abgeschirmt sind. Aus denselben Gründen können einzelne Komponenten problemlos als wiederverwendbare Bausteine eingesetzt werden.

Information Hiding ist das leitende Prinzip in der Softwareentwicklung, wenn mit prozeduraler Abstraktion gearbeitet wird. Bei eingebetteten Systemen kann diese Technik nur beschränkt eingesetzt werden, weil sie hierarchische Strukturierung der Systemfunktionen voraussetzt. Mit architekturorientierter Komposition kann aber das Geheimnisprinzip rein konstruktiv realisiert werden. Das zeigt sich unter anderem darin, dass es nie möglich ist, aus der Beschreibung einer Komponente herzuleiten, mit welchen Komponenten sie interagiert.

LITERATUR

1. E. Yourdon.
Modern Structured Analysis.
Yourdon Press, Prentice-Hall, London 1989.
2. G. Booch, J. Rumbaugh and I. Jacobson.
The Unified Modelling Language User Guide.
Addison-Wesley Longman, 1999.
3. P. T. Ward and J.M. Mellor.
Structured Development for Real-Time Systems.
Yourdon Press, Prentice-Hall, New Jersey 1985.
4. B. P. Douglass.
Real-Time UML.
Addison Wesley Longman Inc. 1998.
5. M. Shaw.
Procedure Calls are the Assembly Language of Software Interconnection.
Studies of Software Design, ICSE'93 Workshop, LNCS Vol. 1078, Springer Verlag Berlin, pages 17-32, 1996.
6. M. Shaw and D. Garlan.
Software Architecture. Perspectives on an Emerging Discipline.
Prentice Hall, April 1996.
7. D. Harel.
Statecharts: A Visual Formalism for Complex Systems.
Science of Computer Programming, Vol. 8, pages 231-274, 1987.
8. O. Faergemand (ed.).
SDL '93: Using Objects.
Proc. of the 6th SDL Forum. North-Holland, 1993.
9. H. Fierz.
The CIP Method: Component- and Model-Based Construction of Embedded Systems.
Software Engineering – ESEC/FSE'99, LNCS Vol. 1687, Springer Verlag Berlin, pages 374-391, 1999.
10. H. Fierz et al.
Domain-Oriented Development of Embedded Systems.
Chapters of a methodical book in construction. The CIP Modelling Framework is treated in chapter 5 and 6.
<http://www.ciptool.ch>.
11. CIP Tool®.
CIP System AG, Solothurn, Switzerland.
<http://www.ciptool.ch>.
12. D. K. Hammer.
Process-Oriented Development of Embedded Systems: Modeling Behavior and Dependability.
Third Workshop on Object-Oriented Real-Time Dependable Systems WORDS, pages 57-65, 1997.
13. HO. Trutmann.
Well-Behaved Applications Allow for More Efficient Scheduling.
24th IFAC/IFIP Workshop on Real-Time Programming. Dagstuhl, Saarland pages 69-74, 1999.
14. HO. Trutmann.
Separate Connection and Functionality is the Pivot in Embedded System Design.
Dissertation ETH No. 13891,
TIK-Schriftenreihe Nr. 39. Swiss Federal Institute of Technology (ETH), 2000.