5 I/O-Verbindung reaktiver Komponenten - DEC Methode

Im Kapitel zur Problem-Analyse von Embedded Systemen wurde für solche Systeme eine generische domänenorientierte Software-Architektur definiert. Der zentrale Punkt dabei ist, dass die funktionale Software und die Verbindungssoftware streng getrennt entwickelt werden. Die funktionale Software wird als ereignisgetriebene *Reaktive Maschinen* modelliert. Aus den Modellen lässt sich der Source-Code für ausführbare *Reaktiven Komponenten* generieren. Die Verbindung zu den Technischen Prozessen und den *Reaktiven Komponenten* muss mittels Hardware- und Softwareverbindungen realisiert werden. Wir gehen davon aus, dass die Hardwareverbindungen bereits durch installierte Sensoren und Aktoren implementiert sind. Was zu tun bleibt, ist das softwaremässige Einbetten der *Reaktiven Komponenten* auf dem Zielsystem. Die zum Teil modell-basierte Entwicklung dieser Verbindungssoftware (Embedded Connection) ist das Thema dieses Kapitels.

Damit die *Reaktiven Komponenten* ausgeführt werden können, müssen Software-Verbindungen zwischen den Sensor- und Aktor-Schnittstellen und den *Reaktiven Komponenten* konstruiert werden. Diese Software-Verbindungen werden als verkettete Interaktionsfunktionen realisiert, die auf der einen Seite für die Wandlung der Sensordaten in funktional verarbeitbare Formate und für die Detektion aufgetretener Ereignisse zuständig sind. Auf der anderen Seite muss der funktionale Output in die Datenformate der Aktorschnittstellen gewandelt werden. Falls mehrere Reaktive Komponenten im Spiel sind, ist immer auch noch Message-Passing-Interaktion zwischen diesen Komponenten zu implementieren.

DEC - Domain-oriented Embedded Connection. Die domänenorientierte Entwicklung der Verbindungssoftware (*Embedded Connection*) wird durch die DEC-Methode unterstützt. Mit domänenorierten Modellen werden die Schnittstellen der *Reaktiven Maschinen* und damit auch die Virtuelle Interaktion definiert, und über relationale Datenflussmodelle wird spezifiziert, wie die Interaktionsfunktionen die Verbindung zwischen den Prozessorschnittstellen und den *Reaktiven Komponenten* herstellen. Die Entwicklung des DEC-Modells beginnt damit vor der Entwicklung der *Reaktiven Maschinen*. Sobald aber die Virtuelle Interaktion spezifiziert ist, kann mit der Modellierung der funktionalen Software begonnen werden (siehe Entwicklungsprozess im Kapitel Problem Analyse).

Die DEC-Methode wird auf der Eclipse Plattform durch das DEC Tool unterstützt (aktuelles Forschungsprojekt). DEC-Modelle können softwaremässig erfasst und verifiziert werden, und die Implementierung der Verbindungssoftware wird durch Code-Generatoren unterstützt.

5.1 DEC Referenzmodelle

Mit einem Referenzmodell kann für eine Gesamtheit verwandter Problemstellungen generisch ein Lösungskonzept beschrieben werden. Das OSI-Referenzmodell beschreibt zum Beispiel ein Schichtenkonzept für die Software von Kommunikationssystemen.

Ein DEC-Referenzmodell beschreibt für eine Prozessverbindung, wie reale Ereignisse über überwachter Zustandsgrössen erfasst, und wie reale Aktionen mittels gesteuerter Zustandsgrössen erzeugt werden. Die Zustandsgrössen sind über Sensoren und Aktoren an den Rechner gekoppelt. Die übertragene elektronische Information muss auf dem Rechner in *Interaktionsfunktionen* entsprechend analysiert werden, bzw. auszugebende Information muss richtig aufbereitet werden.

Die DEC Methode basiert auf drei verschiedenen Referenzmodellen die beschreiben, wie die Interaktion zwischen Technischen Prozesskomponenten und reaktiven Maschinen konstruiert werden kann. Das LPC-Referenzmodell (*LPC - Local Process Connection*) wird für Prozessverbindungen verwendet, deren Software auf einem Prozessor implementiert werden kann. Das DPC-Referenzmodell (*DPC - Distributed Process Connection*) beschreibt Prozessverbindungen, bei welchen die Sensoren und Aktoren an spezifischen Peripherie-Prozessoren angeschlossen sind. Die Reaktive Maschine läuft auf einem anderen Prozessor. Und schliesslich das VPC-Referenzmodell (*VPC - Virtual Process Connection*) zeigt, wie prozessorbasierte IO-Komponenten benutzt werden, die ein spezifisches API für die Interaktion mit den technischen Prozessen zur Verfügung stellen.

Ein viertes Referenzmodell (*RMC - Reactive Machine Connection*) beschreibt, wie die Interaktion zwischen verteilt implementierten reaktiven Maschinen zu realisieren ist.

5.1.1 LPC - Local Process Connection

Das *LPC-Referenzmodell* beschreibt generisch die Interaktionen und Informationsflüsse zwischen einer technischen Prozesskomponente und der Reaktiven Maschine für den Fall, dass die Sensoren und Aktoren an demjenigen Prozessor angeschlossen sind, auf dem auch die entsprechende Reaktive Maschine läuft.

LPC - Local Process Connection (DEC Reference Model with Local Peripherals)

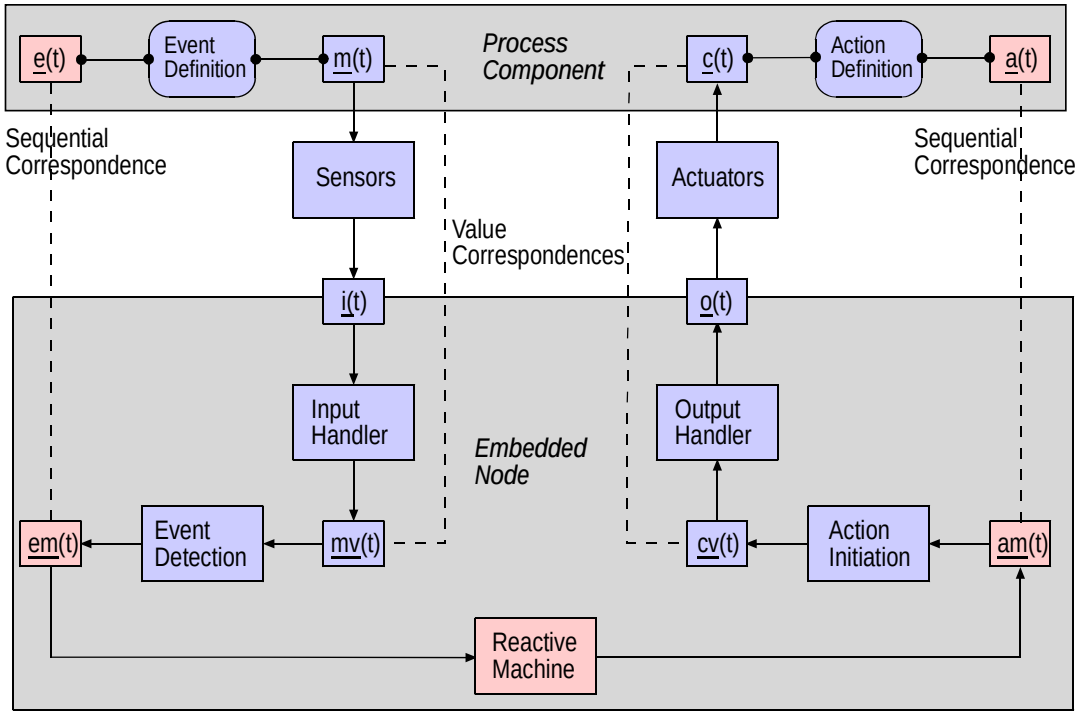


Abb. 47: DEC Referenzmodell für lokale Embedded Process Connections

Notation: $\underline{d}$ bezeichnet ein heterogenes Tupel $(d_1, . , d_n)$ von Symbolen.

Das Referenzmodell kann als Kombination des sog. Inverted 4-Variable-Model von David Parnas und dem rein ereignisbasierten Interaktionsmodell der JSD-Methode gesehen werden (JSD Jackson System Development, Spezifikationsmethode für Informationssysteme). Das Wortes Inverted in der Bezeichnung Inverted 4-Variable-Model hat folgende Bedeutung: Input Handler invertieren die Wandlungen der Sensoren, Output Handler die Wandlungen der Aktoren.

Zustandsgrössen, Schnittstellen, Variablen und Interaktionsfunktionen des Referenzmodells

Mittels Sensoren überwachte Zustandsgrössen werden im Interaktionsmodell als Monitored Quantities m_i und über Aktoren gesteuerte Zustandsgrössen werden als Controlled Quantities c_i bezeichnet.

In der grafischen Darstellung des DEC-Referenzmodells (*Abbildung 47*) sind die Zeitfunktionen $\underline{m}(t)$ der Monitored Quantities und die Zeitfunktionen $\underline{c}(t)$ Controlled Quantities dargestellt. Für jede dieser Zustandsgrössen ist auf dem Rechner als Abbild eine entsprechende Software-Variable zu implementieren (Value Correspondence), bezeichnet als Monitored Variable mv_i und Controlled Variable cv_i. Dargestellt sind die entsprechenden Zeitfunktionen $\underline{mv}(t)$ und $\underline{cv}(t)$. Diese Variablen werden auch als Prozess-Variablen bezeichnet.

Die Input und Output-Schnittstellen $\underline{i}$ und $\underline{o}$ am Rechner sind extern über Sensoren und Aktoren mit den Zustandsgrössen der Prozesse verbunden (dargestellt sind die Zeitfunktionen $\underline{i}(t)$ und $\underline{o}(t)$). Auf der Softwareseite sind die Rechnerschnittstellen über die Interaktionsfunktionen *Input Handler*, bzw. *Output Handler* mit den entsprechenden Prozess-Variablen $\underline{mv}$ und $\underline{cv}$ verbunden.

Weiter bestimmt die Spezifikation *Event Definition* die zu verarbeitenden Ereignisse. $\underline{e}(t)$ bezeichnet eine mögliche Ereignissequenz. Entsprechende Sequenzen von Ereignismeldungen $\underline{em}(t)$ werden auf dem Rechner durch die Interaktionsfunktionen der Komponente *Event Detection* aus den Zeitfunktionen der Monitored Variablen $\underline{mv}(t)$ erzeugt. Die Sequenz ($\underline{ex}(t)$) bezeichnet das zeitliche Triggern der Reaktiven Maschine.

Analog bezeichnet beim Output die Komponente *Action Initiation* die Interaktionsfunktionen, welche die erzeugten Sequenzen von Aktionsmeldungen $\underline{am}(t)$ in Zeitfunktionen der entsprechenden Controlled Variablen umsetzen. Die Aktionen der real produzierten Aktionssequenzen $\underline{a}(t)$ sind durch die *Action Definition* zu spezifizieren.

Zeitgetriebene Steuersysteme, wie sie von Reglersystemen und SPS-basierten Systemen bekannt sind, können als Spezialfall des LPC-Referenzmodells gesehen werden, nämlich als Interaktionsmodelle, in welchem nur periodische Zeitereignisse (Sampling) verarbeitet werden. Die entsprechenden Ereignismeldungen liefern in diesem Fall periodisch die Werte der erfassten Monitored Variablen.

5.1.2 DPC - Distributed Process Connection

Das *DPC-Referenzmodell* beschreibt Prozessverbindungen mit verteilter Peripherie, d. h. das Erfassen der überwachten Zustandsgrössen, bzw. das Einwirken auf gesteuerte Zustandsgrössen erfolgt auf spezifischen Prozessoren, die als *Peripheral Nodes* bezeichnet werden. Die funktionale Verarbeitung detektierter Ereignisse wird hingegen in einer Embedded Unit auf einem zentralen Prozessor ausgeführt, dem *Unit Node*, der typischerweise über einen Feldbus mit den *Peripheral Nodes* verbunden ist. Es ist auch möglich, das derselbe Prozessor für gewisse Technische Prozesse als *Peripheral Node* dient, und für andere als *Unit Node*.

DPC - Distributed Process Connection (DEC Reference Model with Distributed Peripherals)

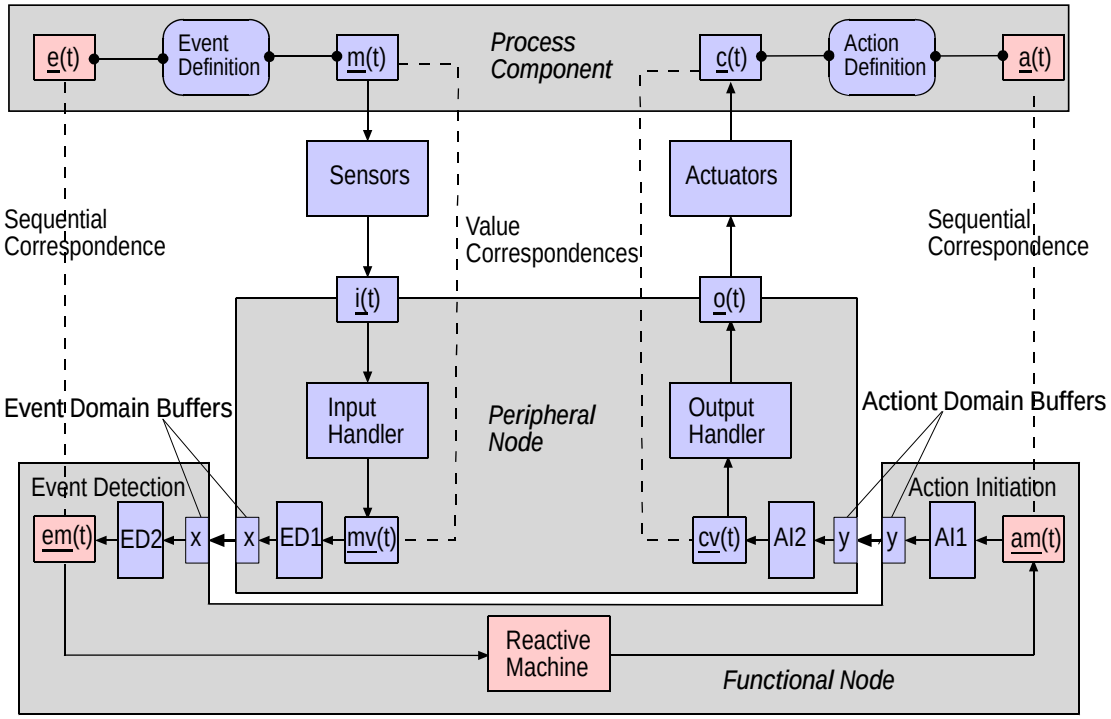


Abb. 48: DEC Referenzmodell für Systeme mit verteilter Peripherie

Für Prozessverbindungen mit verteilter Peripherie muss das LPC-Referenzmodell erweitert werden, weil die Interaktionsfunktionen *Event Detection* und *Action Initiation* in der Regel auf zwei oder mehrere Prozessoren verteilt werden und damit nicht mehr als zeitlich atomare Funktionen spezifiziert werden können. Diese Interaktionsfunktionen sind je als zwei Funktionen zu realisieren, die über ein bestimmtes Feldbussystem kommunizieren. Im DEC-Interaktionsmodell müssen die Links zu den Kommunikationsschnittstellen spezifiziert und die Datentypen für die übertragene Information festgelegt werden (Event und Action Domain Interfaces).

Das DPC-Referenzmodell kommt auch zur Anwendung, wenn anstatt Peripherie-Prozessoren *Input Tasks* und *Output Tasks* (Real-Time Tasks) verwendet werden, die asynchron zur Task mit der funktionalen Verarbeitung (Reaktive Maschine) auf demselben Prozessor ausgeführt werden. Diese Task wird als *Unit Task* (Embedded Unit) bezeichnet. Anstatt eines Kommunikationssystems sind jetzt *Message Passing Queues* zu verwenden, welche die *Input Tasks* mit der *Unit Task*, bzw. die *Unit Task* mit den *Output Tasks* verbinden. Die *Event Detection* und *Action Initiation* sind jetzt entsprechend auf die *Input Tasks* und die *Unit Task*, bzw. auf die *Unit Task* und die *Output Tasks* verteilt.

Auch wenn der Input per Interrupt erfasst wird, ist das DPC-Referenzmodell relevant. Eine *Interrupt Subroutine* entspricht einer asynchronen *Input Task*.

5.1.3 VPC - Virtual Process Connection

Das *VPC-Referenzmodell* beschreibt Prozessverbindungen, die bereits fabrizierte *IO-Komponenten* (*Deployed Components*) verwenden. *IO-Komponenten* sind Embedded-Komponenten, die bereits mit den Sensoren und Aktoren gewisser Prozesse interagieren. In der Regel benutzen sie einen eigenen Prozessor. Solche Komponenten werden typischerweise über einen Feldbus mit dem zentralen Teil eines Embedded Systems verbunden. Für die Embedded Entwicklung steht in der Regel ein API (Application Programming Interface) zur Verfügung, über welches *Notifications* (Antworten) empfangen und *Commands* (Befehle) abgesetzt werden können. Solche Meldungen betreffen meistens das Verhalten der entsprechenden Peripheriegeräte, können sich aber auch direkt auf die angeschlossenen Technischen Prozesse beziehen.

Am meisten verbreitet sind IO-Komponenten für Antriebe, welche generische Grundfunktionen für den Betrieb von Motoren anbieten. *IO-Komponenten* werden aber auch im Eigenbau entwickelt (HW oder SW), z. B. um bei einer Produktfamilie die Anbindung an die Basisprozesse als wiederverwendbare Ressource bereitzustellen.

VPC - Virtual Process Connection (DEC Reference Model with Deployed IO-Components)

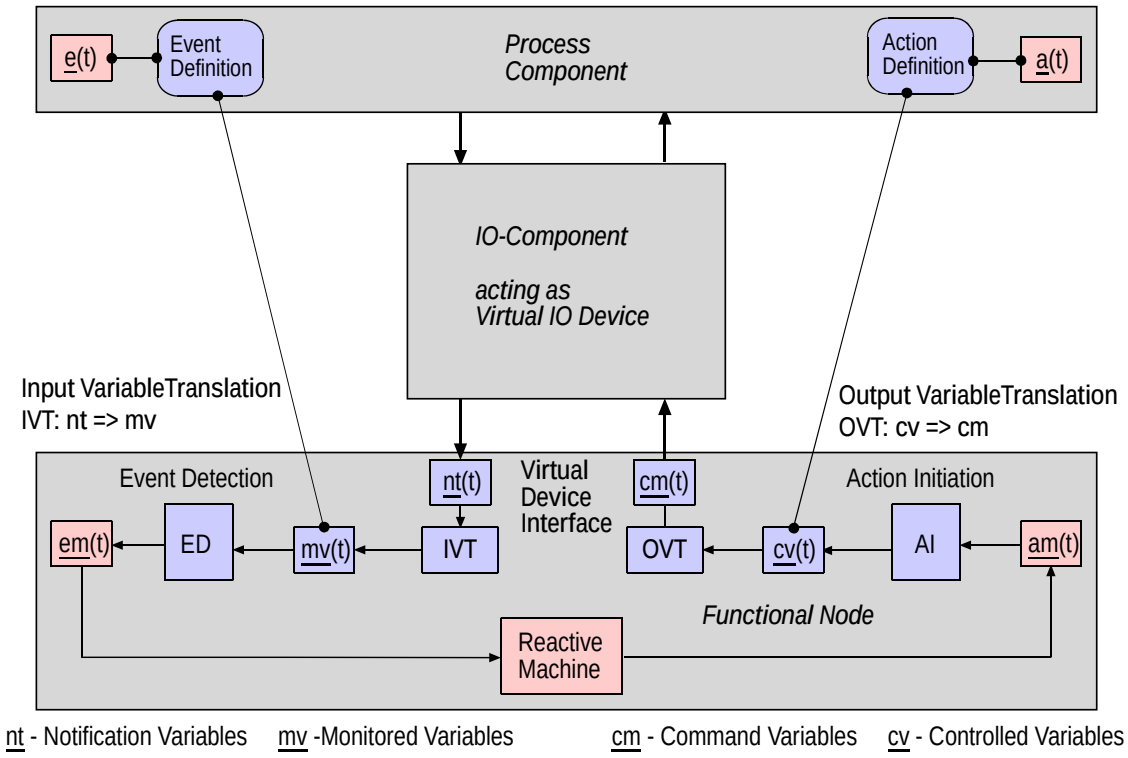


Abb. 49: DEC Referenzmodell für Systeme mit IO-Komponenten

Die Input-Schnittstellen in *Abbildung 49* sind durch Zeitfunktionen *nt(t)* von *Notification Variables* beschrieben, und die Output-Schnittstellen durch Zeitfunktionen *cm(t)* von C*ommand Variables.*

Bei der Verwendung von *IO-Komponenten* sind die Signalschnittstellen der IO-Komponente nicht bekannt. Auch der Bezug zu den realen Prozessvariablen ist in der Regel nicht direkt einsehbar. Trotzdem müssen *Event* und *Action Messages* spezifiziert werden, damit mit den Reaktiven Maschinen interagiert werden kann. Zudem muss festgelegt werden, wie die entsprechenden Events über das API detektiert, bzw. die Actions bewirkt werden können.

Es ist klar, dass vom API ausgegangen werden muss. Man definiert zuerst *Event* und *Action Messages* aus der Sicht des API's, indem die unterstützten *Notifications* und *Commands* analysiert werden. Typisch ist zum Beispiel, dass über einen bestimmten *Notification Type* unterschiedliche Events erfasst werden können. Diese Events müssen aufgrund der zur Verfügung stehenden *Notification*

Variables detektiert werden können. Analog sind die Actions über *Command Variables* eines *Command Type* zu erzeugen.

Damit sind auch die *Monitored* und *Controlled Variables* der Technischen Prozesse aus der Sicht des API's zu definieren. Im DEC Tool kann der Bezug zwischen diesen Prozessvariablen und API Variablen formal durch eine entsprechende *Variable Translations* definiert werden.

5.1.4 RMC - Reactive Machine Connection

Das *RMC-Referenzmodell* beschreibt, wie in einem System zwischen verschiedenen Reaktiven Maschinen eines Embedded Systems Interaktion erzeugt wird. Diese Interaktion wird als asynchrones Message Passing modelliert und zählt auch zur virtuellen Ineraktion, in welche die Reaktiven Maschinen involviert sind. Die entsprechenden Schnittstellen der Reaktiven Maschinen werden als *Input Ports* und *Output Ports* definiert. Für jeden Port werden die zu übertragenden *Messages* festgelegt, und für jede *Message* ist der Datentyp für die zu übertragende Information zu spezifizieren.

Reaktive Maschinen laufen in nebenläufigen *Embedded Units* eines Embedded Systems, d. h. sie werden in verschiedenen Tasks auf verschiedenen Prozessoren ausgeführt. Die Interaktion zwischen den Reaktive Maschinen kann dementsprechend standardmässig als asynchrones Message Passing zwischen Tasks implementiert werden.

Reaktive Maschinen desselben Prozessors interagieren über *lokale Kanäle (Message Passing Queues)*, die man softwarebasiert implementiert. Reaktive Maschinen verschiedener Prozessoren sind über *globale Kanäle* verbunden, für die bei der Implementierung ein Kommunikationssystem (z. B. Feldbus) verwendet werden muss.

Das *DEC* Referenzmodell für interagierende Reaktive Maschinen (*Abbildung 50*) ist sehr einfach. Das hängt damit zusammen, dass die gesamte Interaktion Teil des zu entwickelnden Embedded Systems ist und keine Übereinstimmungen zu einer vorgegebenen technischen Prozessumgebung gefunden werden müssen.

RMC - Reactive Machine Connection (DEC Reference Model for Interacting Reactive Machines)

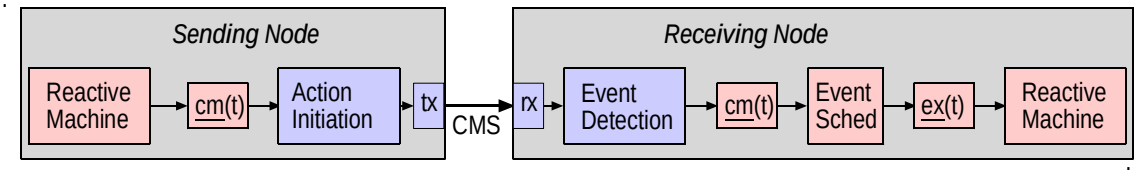


Abb. 50: DEC Referenzmodell für kommunizierende reaktive Maschinen

In *Abbildung 50* ist die Interaktion zwischen zwei Reaktiven Maschinen dargestellt, die auf zwei unterschiedlichen Prozessoren implementiert sind. Die durch eine Reaktive Maschine erzeugten *Communication Messages* (cm(t)) werden von einem *Communication System* (CMS) zum Prozessor mit der empfangenden Reaktiven Maschine übertragen.

Die initiierten Aktionen werden hier als Aktionen verstanden, die beim Kommunikationssystem die Übertragung der entsprechenden Nachricht auslösen. Das Empfangen dieser Nachrichten auf dem empfangenden Prozessor sind Kommunikationsereignisse, die als Events detektiert werden.

5.2 Domänenorientierte Spezifikation der Verbindungssoftware

Ein DEC-Modell ist ein Datenflussmodell der *Embedded Connection.* Dabei werden die Schnittstellenvariablen für die Interaktionsfunktionen und die Reaktive Maschinen definiert. Beziehungen zwischen Schnittstellen spezifizieren den Datenfluss, und sie bestimmen damit, wie die Interaktionsfunktionen als Komponenten zu implementieren sind. Das Datenflussmodell und die Interaktionsfunktionen ergeben zusammen eine ausführbare Datenflussmaschinen (Data Flow Machines), welche auf dem Zielsystem die Prozessorschnittstellen mit den Reaktiven Komponenten verbinden.

Kompositionelle Struktur eines Embedded System

Ein mit DEC entwickeltes Embedded System besteht im allgemeinen aus mehreren Prozessoren mit mehreren *Embedded Units*. Von jeder Embedded Unit ist die *Reaktive Maschine* über ein Bündel von *Process Connections* mit der realen Umgebung verbunden. Jede *Process Connection* besteht aus einem *Input Channel* und/oder einem *Output Channel*, die als einfache Datenflussmaschinen wirken. Dazu kommen noch Verbindungen zwischen den Reaktiven Maschinen selbst. Die Embedded Units eines Prozessors laufen in der Regel als preemptive Real-Time Tasks mit verschiedenen Zykluszeiten. Die Process Connections einer Embedded Unit werden hingegen alternierend (non-preemptiv) ausgeführt. Eine Ausnahme bilden hier interrupt-getriebene Input Handler.

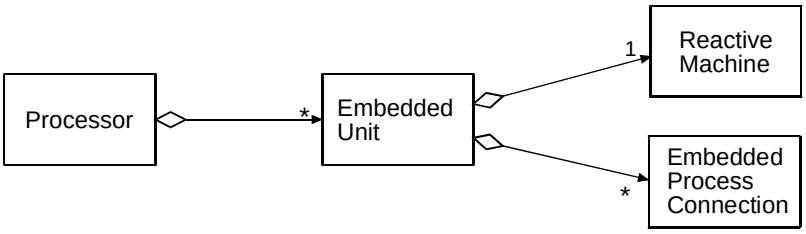


Abb. 51: UML Darstellung der Aggretation von Embedded Units und Process Connections

Die *Process Connections* einer *Embedded Unit* können alle Referenzmodelle unterstützen, das heisst sowohl *Local Process Connections (LPC), Distributed Process Connections (DPC)* als auch *Virtual Process Connections (VPC).*

Datenflussmaschinen für eine Embedded Unit auf einem Prozessor

Die *Reactive Machine* einer *Embedded Unit* wird mit parallelen *Input Channels* und *Output Channels* mit den Sensor- und Aktor-Schnittstellen des Prozessors verbunden.

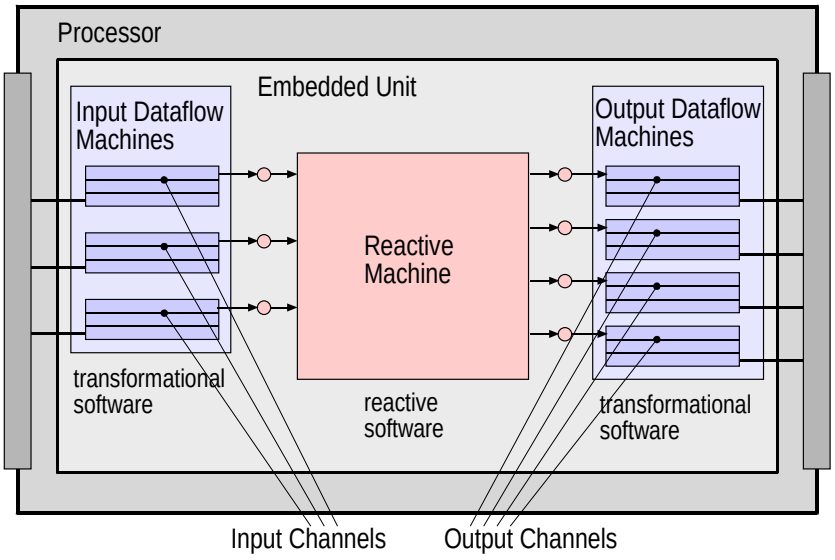


Abb. 52: Datenflussmaschinen und Reaktive Maschine einer Embedded Unit

In *Abbildung 52* ist der einfache Fall eines Embedded Systems dargestellt, das aus einem Prozessor mit nur einer Embedded Unit besteht. Bei der domänen-orientierten Entwicklung der *Embedded Connection* wird die Architektur der Verbindungssoftware durch Eigenschaften der realen technischen Umgebung bestimmt (Korrespondenzprinzip). Dementsprechend bestehen die Input- und Output-Datenflussmaschinen aus parallelen einfacheren Datenflussmaschinen, die als *Input Channels*, bzw. als *Output Channels* bezeichnet werden. Jeder *Input* oder *Output Channel* überträgt die Information, die zu genau einer technischen Prozesskomponente gehören. Bei der Übertragung werden aus Sensorsignalen für detektierte Ereignisse *Event Messages* für die Reactive Machine erzeugt, und für abgesetzte *Action Messages* werden die entsprechenden Aktorsignale aktualisiert. *Input* und *Output Channels* realisieren damit die Softwareseite von Verbindungen zu Technischen Prozessen (*Embedded Connection*). Diese Verbindungen werden auf drei Layern spezifiziert (siehe unten). Die entstehende *3-Tier Architecture* ist vergleichbar mit Schichtenarchitekturen von Kommunkationssystemen. Für jeden Input oder Output Channel werden auf jedem Layer Variablen definiert, die einen Bezug zu entsprechenden Grössen in der realen Umgebung haben (Korrespondenzen im DEC- Referenzmodell). Der Datenfluss in einem Kanal wird durch entsprechende Interaktionsfunktionen erzeugt, welche die verschiedenen Ebenen verbindet, ähnlich einem Protokoll Stack eines Kommunikationssystems.

Zwei formale Spezifikationsebenen

Das DEC Tool unterstützt die DEC Methode, indem die Architektur und die Interaktionspfade der Verbindungssoftware formal als Relationen erfasst werden können. Die Interaktionsfunktionen hingegegen werden direkt codiert. Diese Programmierung wird durch kontextsensitive Editoren unterstützt, welche die spezifizierten Schnittstellen als Tags zur Verfügung stellen siehe. Das Tool kann Spezifikationen auf Vollständigkeit und Konsistenz überprüfen und ausführbaren Code für das Zielsystem erzeugen.

Ausführung einer Embedded Unit - Unit Controller

Der generierte Code für die Datenflussmaschinen besteht aus strukturierten Schnittstellenvariablen und Prozeduren, welche die Interaktionsfunktionen enthalten. Dazu kommt der Code der Reaktiven Maschine, die über eine prozedurale Schnittstelle (*function pointers*) zu aktivieren ist. Was noch fehlt ist der Unit Controller, der als aktiver Thread die Ausführung der Embedded Unit steuert. Dabei werden die Datenfluss Maschine und die Reaktive Maschine durch Prozeduraufrufe (*function calls*) gezielt aktiviert werden. Der Code des Unit Controller kann nicht generiert werden. Bei der Strukturierung und Programmierung des Unit Controller müssen oft Echtzeitbedingungen berücksichtigt werden, das heisst es sind Anforderungen an Antwortzeiten zu beachten. Zudem muss garantiert werden können, dass die anderen Embedded Units auf demselben Prozessor genügend Ausführungszeit bekommen. Man trifft hier auf einen neuen Problembereich, dem Real-Time Scheduling.

Wie ein Unit Controller zu entwerfen ist, dass dies überhaupt möglich ist, wird im Kapitel 6 (Scheduling) behandelt. Wie garantiert werden kann, dass alle Embedded Units auf einem Prozessor genügend Prozessorzeit erhalten ist Thema des nächsten Kapitels (Real-Time Scheduling Konzepte).

5.2.1 Channel-Struktur bei Local Process Connection (LPC)

Das Referenzmodell für *Local Process Connection* ist das primäre Basis-Referenzmodell. Es ist der Ausgangspunkt für die Festlegung der internen Interaktionsstruktur der Input und Output Channel. Das Referenzmodell für *Distributed Process Connection* ist eine Erweiterung, und das Referenzmodell für *Virtual Process Connection* ist ein Einschränkung diese Referenzmodells. Die entsprechenden Channel-Strukturen werden in Abschnitt 5.2.2 und Abschnitt 5.2.3 erläutert.

LPC Channel-Struktur

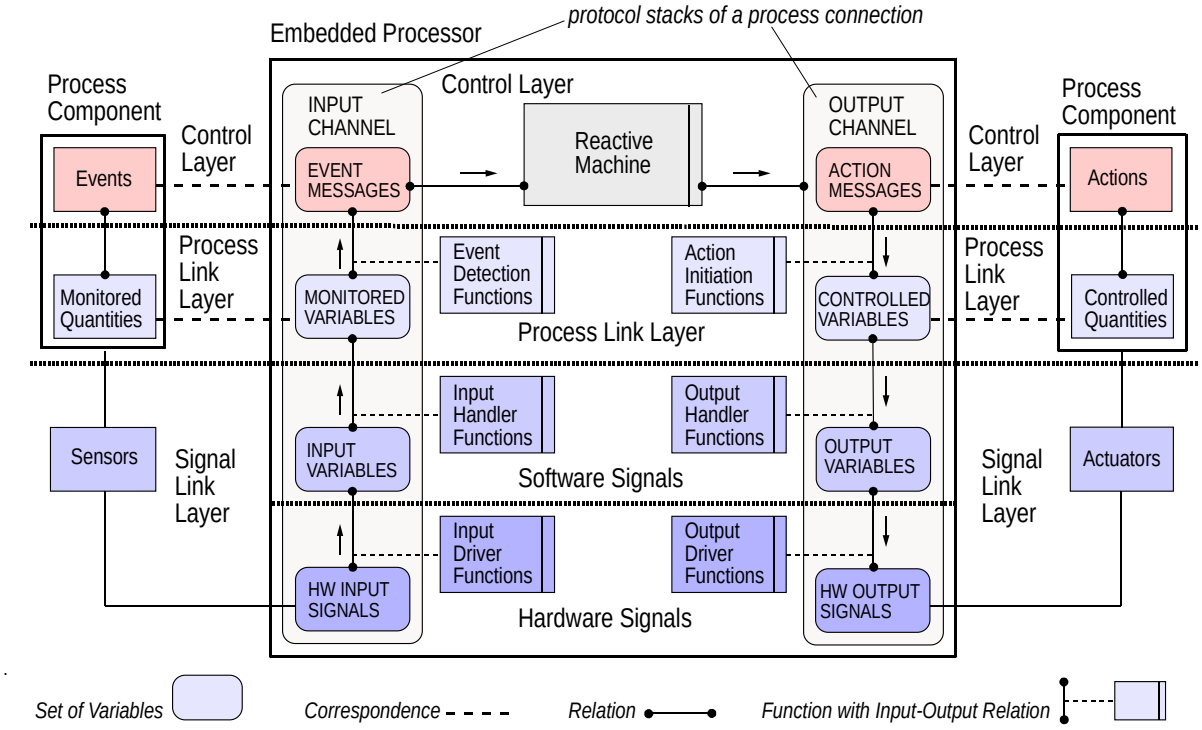


Abb. 53: 3-Schichten Architektur bei lokaler Prozessinteraktion

Die in *Abbildung 53* mit Grossbuchstaben bezeichneten Mengen von Schnittstellen-Variablen werden im DEC Tool definiert. Die gezeichneten Verbindungen solcher Mengen stellen Beziehungen dar, die als Relationen spezifiziert werden. Diese Beziehungen definieren bei den angehängten Interaktionsfunktionen die Input- und Output-Schnittstellen.

Control Layer

Im *Control Layer* wird die Reaktive Maschine ausgeführt (ereignis-basierte virtuelle Interaktion mit den Technischen Prozessen). Mit der DEC Methode werden die Schnittstellen dieser Software Komponente definiert und dem entsprechenden Modellierungswerkzeug (CIP Tool) zur Verfügung gestellt. Die Inputschnittstellen sind als EVENT MESSAGES spezifiziert, die für real auftretende Events der entsprechenden technischen Prozesskomponente erzeugt werden. Die Outputschnittstellen sind ACTION MESSAGES, die reale Actions initiieren. Die entsprechenden Software-Schnittstellen für die Reaktive Maschine werden durch das DEC Tool generiert.

Bemerkung. Ein Input/Output Channel eines CIP Modells entspricht der Reduktion eines Input/Output Channels auf den *Control Layer* und wird im DEC Tool als INPUT/OUTPUT MESSAGE CHANNEL bezeichnet.

Process Link Layer

Im *Process Link Layer* werden in Input Channels virtuell die Werte von realen *Monitored Quantities* übertragen und in den entsprechend definierten MONITORED VARIABLES festgehalten. Analog

findet in Output Channels eine virtuelle Übertragung der Werte von CONTROLED VARIABLES in entsprechende reale *Controlled Quantities* der technischen Prozesskomponente statt.

Durch Beziehungen zwischen MONITORED VARIABLES und EVENT MESSAGES wird festgelegt, welche Rolle diese Prozessvariablen bei der Detektion aufgetretener Events haben, und mit Beziehungen zwischen CONTROLLED VARIABLES und ACTION MESSAGES wird die Rolle bei der Initiierung von Actions bestimmt.

Die *Event Detection Functions* erzeugen als Dienst für den *Control Layer* die entsprechenden EVENT MESSAGES, und für jede ACTION MESSAGE erzeugt die zugehörige *Action Initiation Function* die neuen Werte für die entsprechenden CONTROLLED VARIABLES.

Signal Link Layer

Der *Signal Link Layer* ist die unterste Interaktionsschicht. Er wird wie üblich in zwei Sublayers aufgeteilt. Im *Software Signal Link Layer* werden als INPUT und OUTPUT SIGNAL INTERFACES Variablen definiert, die den Prozessorschnittstellen für externen Signale im *Hardware Signal Link Layer* entsprechen. Diese Schnittstellen sind elektronisch mit den Sensoren und Aktoren verbunden.

Die *Input Handler Functions* im *Hardware Signal Link Layer* erzeugen als Dienst für den *Process Link Layer* die aktuellen Werte der zugeordneten MONITORED VARIABLES. Dabei müssen für den Zugriff auf die HW-Schnittstellen die entsprechenden *Input Driver Functions* des *Hardware Signal Link Layer* benutzt werden. Eine *Input Handler* Function invertiert gewissermassen die Interaktionswandlung des entsprechenden Sensors.

Analog erzeugen *Output Handler Functions* die aktuellen Werte der Signalschnittstellen aus den Werten der zugeordneten CONTROLLED VARIABLES. Auch hier müssen für den Zugriff auf die HW-Schnittstellen die entsprechenden *Output Driver Functions* des *Hardware Signal Link Layer* benutzt werden. Eine *Output Handler* Function kann ebenso als Inversion der Interaktionswandlung des entsprechenden Aktors gesehen werden.

Anwendung des domänenorientierten Korrespondenzprinzip

Entscheidend bei der Spezifikation der Verbindungssoftware sind die domänenorientierten Übereinstimmungen (Korrespondenzen) mit der realen Prozessumgebung. Beim Erstellen des Datenfluss-Modells wird man in der Regel von den realen mit Sensoren überwachten, bzw. mit Aktoren gesteuerten Zustandsgrössen ausgehen und für das Embedded System die entsprechenden MONITORED VARIABLES und CONTROLLED VARIABLES definieren.

Um die notwendigen EVENT und ACTION MESSAGES definieren zu können ist anschliessend aufgrund einer Analyse der funktionalen Anforderungen zu bestimmen, auf welche Events reagiert werden muss, und welche Actions erzeugbar sein müssen. Weiter ist festzuhalten, über welche MONITORED VARIABLES die entsprechenden Events zu detektieren sind, und über welche CONTROLLED VARIABLES die spezifizierten Aktionen bewirkt werden können.

Die Interaktionspfade sind schliesslich zu vervollständigen, indem für die MONITORED und CONTROLLED VARIABLES spezifiziert wird, über welche Signalschnittstellen die entsprechenden Sensoren und Aktoren am Prozessor angebunden sind.

5.2.2 Channel-Struktur bei Distributed Process Connection (DPC)

Bei Technischen Prozessen, die über spezifische *Input* und *Output Nodes* mit dem *Unit Node* verbunden sind, auf welchem die Reactive Machine läuft, muss die Embedded Connection entsprechend verteilt implementiert werden. Die Verteilung macht sich nur im *Process Link Layer* bemerkbar. Dabei müssen die *Event Detection* und die *Action Initiation* Funktionen, und damit auch der entsprechend *Input* und *Output Channel* verteilt implementiert werden. Für die Kommunikationsschnittstellen (Feldbuskommunikation), sind entsprechende *Communication Variables* zu definieren. Diese entsprechen zum Teil den bereits spezifizierten *Monitored Variables*, bzw. *Controlled Variables* und können mit neu definierten *Auxiliary Variables* erweitert werden.

DPC Input Channel Struktur

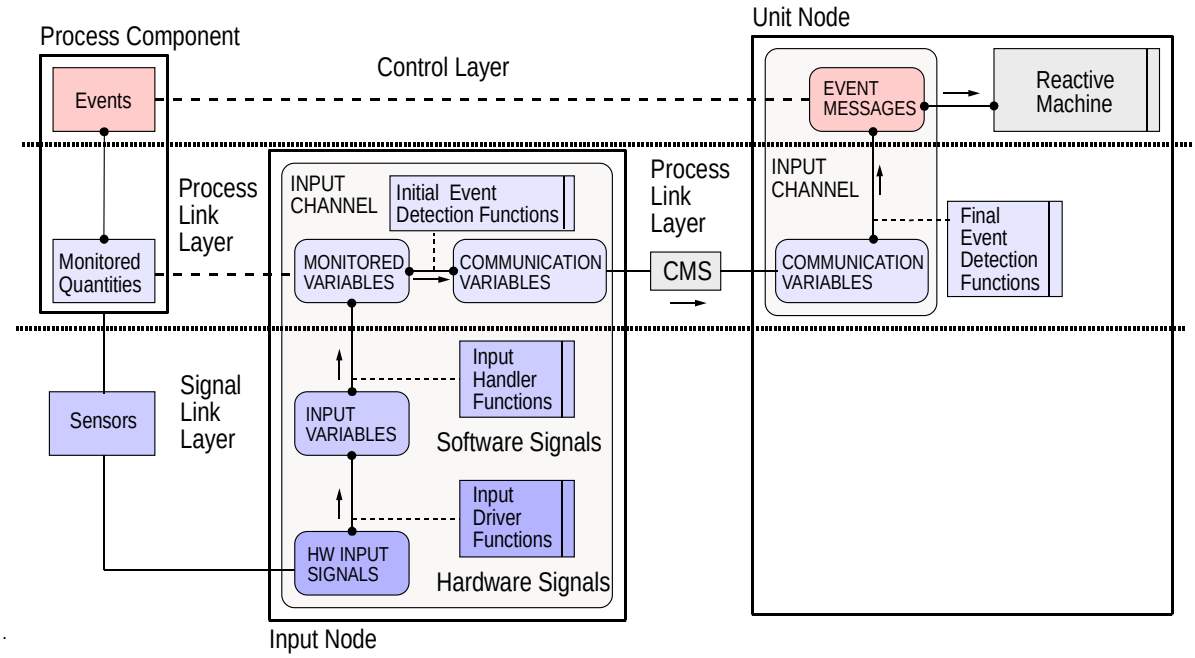


Abb. 54: 3-Schichten Architektur bei verteilter Input-Prozessinteraktion

Die *Event Detection Functions* eines verteilten Input Channels werden in *Initial Event Detection Functions* auf dem *Input Node* und *Final Event Detection Functions* auf dem *Unit Node* aufgeteilt.

In der Regel initiiert die *Initial Event Detection Functions* nur eine Kommunkationsmeldung, wenn sich eine Monitored Variable verändert hat, die für die Detektion aufgetretener Ereignisse verwendet wird (*Detection Variable)*. Die Veränderung wird in einer *Auxiliary Variable* mitgegeben. Die andern Monitored Variablen, die zur selben *Event Domain* gehören, werden als Kopien der Kommunkationsmeldung mitgegegeben. Die *Final Event Detection Functions* auf dem *Unit Node* bestimmt dann, welcher *Event* aufgetreten ist und erzeugt die entsprechende *Event Message* für die *Reactive Machine*.

Es sind aber auch extremere Aufteilungen möglich. Zum Beispiel *Remote Polling*. Dabei werden periodisch die Signale eingelesen und als aufbereitete Monitored Variables dem Unit Node gesendet. Eine *Final Event Detection Function* führt dann die vollständige Event Detection aus. Der andere Extremfall ist die vollständige Event Detektion auf dem *Input Node*. In diesem Fall entspricht die Kommunkationsmeldung bereits einer *Event Message*, welche die Event Message ID in einer enum Variablen mitträgt.

DPC Output Channel Struktur

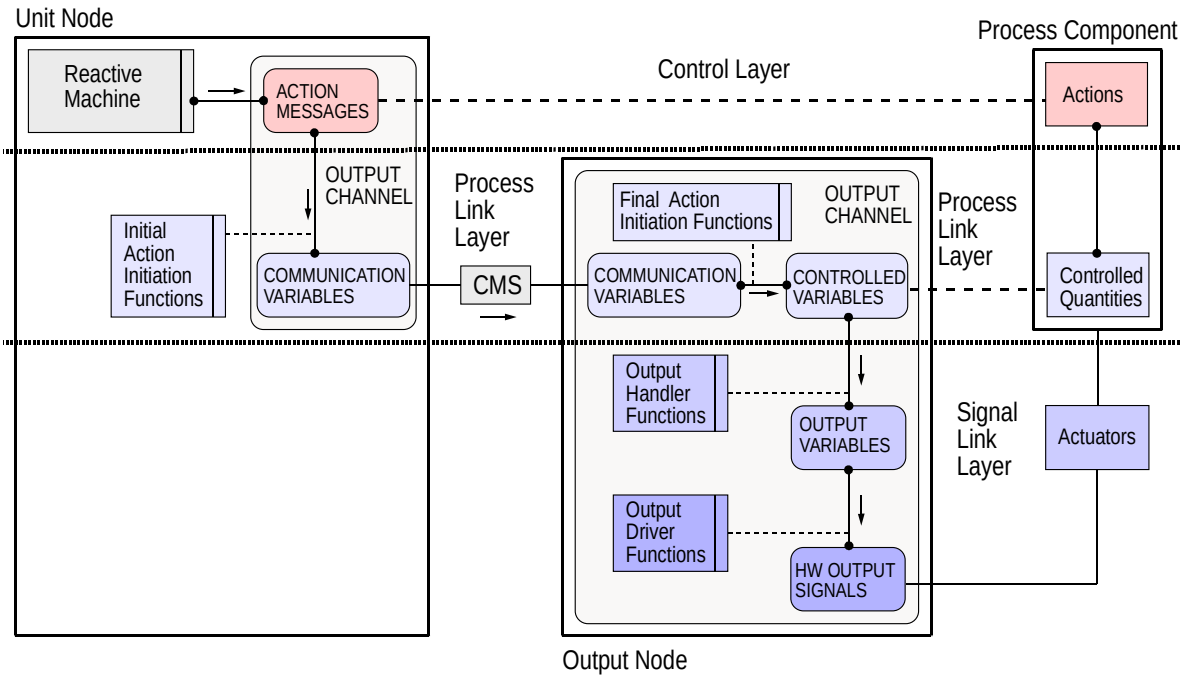


Abb. 55: 3-Schichten Architektur bei verteilter Output-Prozessinteraktion

Die *Action Initiation Functions* eines verteilten Input Channels werden in *Initial Action Initiation Functions* auf dem *Unit Node* und *Final Action Initiation Functions* auf dem *Output Node* aufgeteilt. Dabei werden als *Communication Variables* oft Kopien der *Controlled Variables* verwendet, die bereits korrekt aktualisiert sind. In diesem Fall führt eine *Initial Action Initiation Function* bereits die gesamte Aktionsinitiierung durch.

Meistens werden auf dem Output Node als Optimierung (Bypass) die *Output Handler Functions* in der entsprechenden *Final Action Initiation Function* integriert, um eine unnötige Zwischenspeicherung (Buffer) der aktualisierten *Controlled Variables* zu vermeiden. Diese Optimierung wird auch durch das *DEC Tool* unterstützt, das heisst es sind keine separaten *Output Handler Functions* zu erstellen.

Input und Output Tasks auf einem Prozessor anstatt Input und Output Nodes

Werden anstatt Peripherie-Prozessoren *Input Tasks* und *Output Tasks* verwendet, die asynchron zur *Unit Task* (Reaktive Maschine) auf demselben Prozessor ausgeführt werden, ist in *Abbildung 54* und *Abbildung 55* lediglich das Kommunikationssystem (CMS) durch eine Message Passing Queue (MPQ) zu ersetzen.

5.2.3 Channel-Struktur bei Virtual Process Connection (VPC)

Werden externe IO-Komponenten verwendet, entfällt der *Signal Link Layer*. Die embedded Interaktion findet vollständig zwischen den Schnittstellen des Kommunikationssystems, die als API (*Virtual Device Interface*) zur Verfügung stehen, und der Reaktiven Maschine statt. Dafür müssen zusätzlich die Variablen der ankommenden *Notifications* und der abgesetzen *Commands* mit den *Monitored* und *Controlled Variablen* der involvierten *Process Connections* in Bezug gesetzt werden.

VPC Channel-Struktur

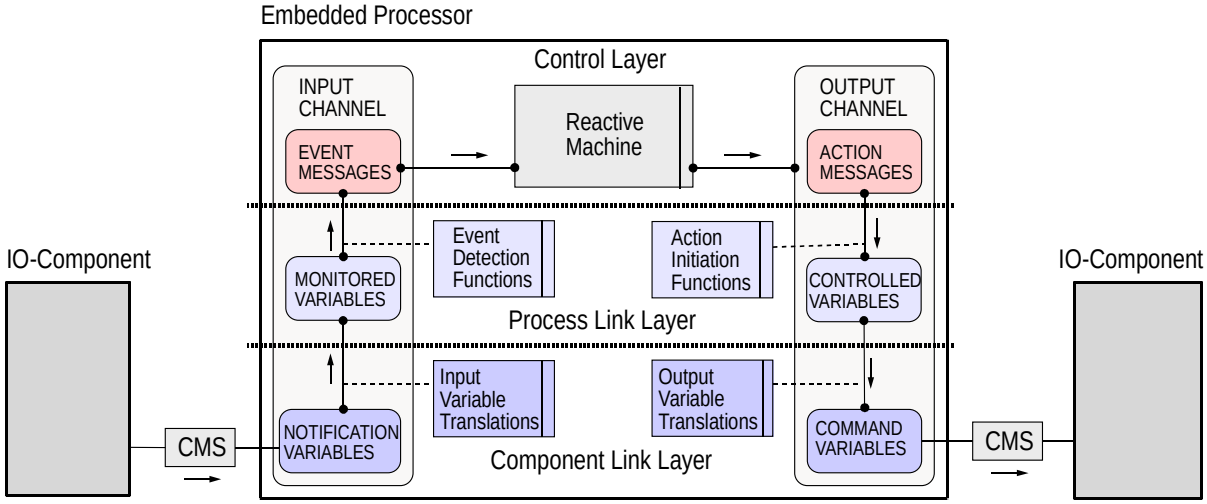


Abb. 56: 2-Schichten Architektur bei der Verwendung von IO-Komponenten

Die dargestellten Mengen von *Notification Variables* und *Command Variables* sind die Kommunikationsvariablen, die im API definiert sind. Diese Variablen sind mit den Applikationsspezifischen *Monitored Variables* und *Controlled Variables* in Beziehung zu setzen. Die entsprechende Abbildung wird durch *Input Variable Translations* und *Output Variable Translations* implementiert. Diese Funktionen werden vom DEC Tool automatisch erzeugt, wenn die Beziehungen zwischen *Notification Variables* und *Monitored Variables*, bzw. zwischen den *Controlled Variables* und *Command Variables* vollständig und bezüglich Datentypen konsistent spezifiziert sind.

Die Problematik bei der Entwicklung liegt beim Festlegen der Beziehungen zwischen *Monitored Variables* und *Event Messages*, bzw. zwischen *Controlled Variables* und *Action Messages*, weil die Korrespondenzen zu den entsprechenden Grössen in der "fremden" realen Prozessumgebung nicht immer einfach zu finden sind.

5.2.4 Channel-Struktur bei Reactive Machine Connection (RMC)

Die Channels zwischen Reaktiven Maschinen werden als *Communication Channels* bezeichnet. Die Struktur solcher Channel ist einfach und unproblematisch. Es müssen lediglich spezifizierte Communication Messages mit dem spezifizierten *Communication System* übertragen werden.

RMC Channel-Struktur

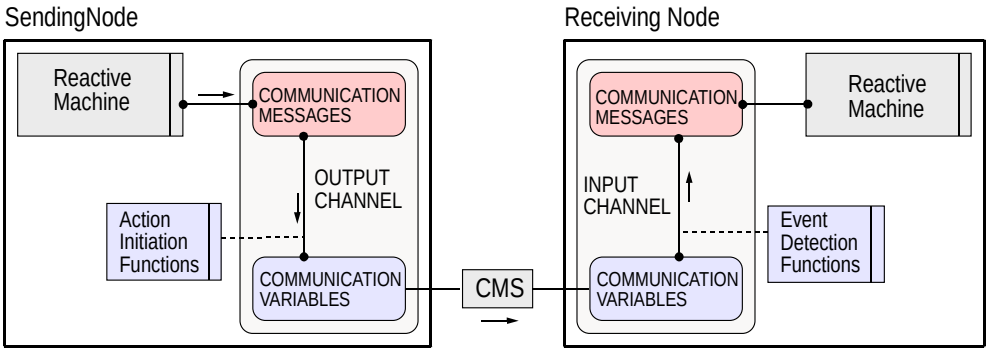


Abb. 57: Implementierung eines Communication Channels

In *Abbildung 57* sind die interaktionspfade für einen Communication Channel dargestellt, der zwei Reaktive Maschinen verbindet, die auf zwei verschieden Prozessoren implementiert sind. Entsprechende Action Initiation Functions sorgen auf der Senderseite dafür, dass die durch die Reaktive Maschine erzeugten Messages der Outputschnittstelle des *Communication Systems* übergeben werden. Auf der Empfangsseite erzeugen die entsprechenden Event Detection Functions aus den ankommendenden Daten die Messages für die empfangende Reaktive Maschine.

Werden zwei Reaktive Maschinen auf demselben Prozessor über eine Communication Channel verbunden, ist in *Abbildung 57* lediglich das Kommunikationssystem (CMS) durch eine Message Passing Queue (MPQ) zu ersetzen.

5.3 Beispiel einer DEC-Spezifikation - SlidingGate

Ein gesteuertes Schiebetor wird mit einem Drehgeber überwacht. Wenn das Sensorrad eine bestimmte Stellung erreicht (Rotationsmarke), wird eine binäres Signal auf *high* gesetzt, wird diese Stellung verlassen, fällt das Signal auf *low*. Damit können regelmässige Positionsschritte überwacht werden. Um die Richtung eines Positionsschrittes zu erfassen steht ein weiteres binäres Signal zur Verfügung.

Weiter wird hardwaremässig bei jedem erfolgten Positionsschritt die Zeit eines Hardware-Timers in einem bestimmten Register des Prozessors abgelegt. Mit diesen Zeitwerten kann die Geschwindigkeit des Tors ermittelt und geregelt werden.

Die Ansteuerung des Motors erfolgt über zwei binäre Signale für die Aktivierung und die Drehrichtung. Zusätzlich kann die Motorleistung variiert werden (digitaler Wertebereich).

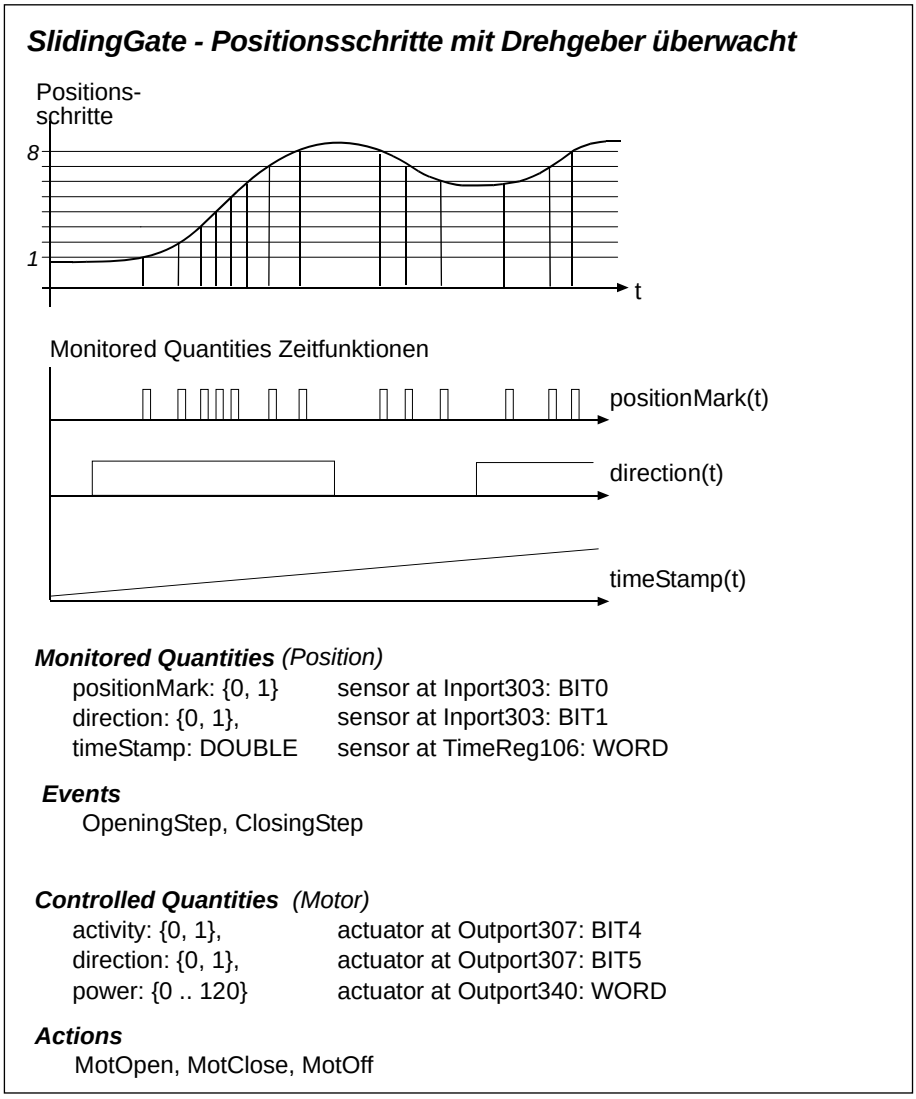


Abb. 58: Zustandsgrössen mit Events und Actions eines Schiebetors mit Drehgeber

Es ist nur das Verlassen der Positionsmarke als Ereignis zu detektieren, dabei soll aber unterschieden werden, ob sich das Tor öffnet oder schliesst. Das Erreichen der Positionsmarke wird nicht als Ereignis verarbeitet. Für die Steuerung des Motors sind drei Aktionsmeldungen notwendig.

Bemerkung. Die folgende Darstellung der DEC-Spezifikation für die Torsteuerung entspricht einem generierten Modell-Report des DEC Tool für die UNIT *GateControl*. Die PROCESS CONNECTION *SlidingGate* ist als einzige Prozessverbindung in dieser Embedded Unit eingebunden.

DEC Spezifikation der UNIT GateControl

```
CONSTANTS
addrPort303      u16      #define addrPort303 0x303
addrPort307      u16      #define addrPort307 0x307
addrPort340      u16      #define addrPort340 0x340
addrTmReg106     u16      #define addrTmReg106 0x106
usPerTick        u16      #define usPerTick 200
PFactor          u16      #define PFactor 8
DATATYPES
u8    typedef unsigned char u8;
u16   typedef unsigned short int u16;
u32   typedef unsigned long int u32;
FUNCTION TYPES
readBytePort     u8 readBytePort(u16 addr);
readWordReg      u16 readWordReg(u16 addr);
writeBytePort    void writeBytePort(u16 addr, u8 val);
writeWordPort    void writeWordPort(u16 addr, u16 val);
```

Process Connections

```
PROCESS CONNECTION SlidingGate
    INPUT CHANNEL
        MONITORED VARIABLES
        positionMark: u8,
        direction: u8,
        timeStamp: u32
        EVENT DOMAIN PositionSteps
            EVENT MESSAGES OpeningStep, ClosingStep
            DETECTION VARIABLE: positionMark
            INDICATION VARIABLES: direction
            ATTRIBUTE VARIABLES: timeStamp
    OUTPUT CHANNEL
        CONTROLLED VARIABLES
        activity: u8,
        direction: u8,
        power: u16
        ACTION DOMAIN MotorActivity
            ACTION MESSAGES MotOpen, MotClose, MotOff
            ACTUATION VARIABLES: activity, direction
            ATTRIBUTE VARIABLES: power
```

EVENT und ACTION DOMAINS sind MESSAGE-Gruppen einer PROCESS CONNECTION.

Pro EVENT DOMAIN werden für ihre EVENT MESSAGES die Rollen der MONITORED VARIABLES bei der Event Detection spezifiziert:

DETECTION VARIABLE - Bei einer Wertänderung ist möglicherweise ein Event aufgetreten.

INDICATION VARIABLES - Mit solchen Variablen wird bestimmt, welcher Event aufgetreten ist.

ATTRIBUTE VARIABLES - Die Werte werden der EVENT MESSAGE als Daten mitgegeben.

Entsprechend werden pro ACTION DOMAIN für ihre ACTION MESSAGES die Rollen der CONTROLLED VARIABLES bei der Action Initiation festgelegt:

ACTUATION VARIABLES - Die ACTION MESSAGE bestimmt die Werte dieser Variablen.

ATTRIBUTE VARIABLES - Die Werte für diese Variablen stammen aus den Daten der ACTION MESSAGE (message attributes).

Ressources

```
PROCESSOR Mcu_ST10
    MODULE ModuleSG
        SIGNAL INPUT INTERFACES
        Inport303: u8       AT: addrPort303      ACCESS: readBytePort
        TimeReg106: u16     AT: addrTmReg106     ACCESS: readWordReg
        SIGNAL OUTPUT INTERFACES
        Outport307: u8      AT: addrPort307      ACCESS: writeBytePort
        Outport340: u16     AT: addrPort340      ACCESS: writeWordPort
```

Bemerkung

Elemente einer Signal-Schnittstelle:

aVariable: aDatatype - Signal-Softwarevariable mit Datentyp.

AT: anAddress - Adresse der entsprechenden HW/SW-Schnittstelle.

ACCESS: aFunctionType - Prototyp der Zugriffsfunktion (driver function).

Architecture

```
UNIT GateControl
    MONITORED VARIABLE BUFFERS fastInputBlock
    CONTROLLED VARIABLE BUFFERS fastOutputBlock
    UNIT NODE: Mcu_ST10
    PROCESS CONNECTIONS
        PROCESS CONNECTION: SlidingGate
            INPUT CHANNEL TYPE: LOCAL
            OUTPUT CHANNEL TYPE: LOCAL
            MONITORED VARIABLE BUFFER: fastInputBlock
            CONTROLLED VARIABLE BUFFER: fastOutputBlock
            INPUT SIGNAL LINKS
            positionMark:   Inport303_0   POINTER    /* BIT 0 */
            direction::     Inport303_1   MACRO      /* BIT 1*/
            timeStamp:      TimeReg106    VARIABLE
            OUTPUT SIGNAL LINKS
            activity:       Outport307_4  POINTER    /* BIT 4 */
            direction:      Outport307_5  MACRO      /* BIT 5 */
            power:          Outport340    VARIABLE
```

Bemerkungen

Für die INPUT und OUTPUT CHANNELS einer UNIT werden für die Prozessvariablen MONITORED VARIABLE BUFFERS und CONTROLLED VARIABLE BUFFERS festgelegt. Diese Datenblöcke oder Queues können bei der Erstellung der Ausführungssteuerung eine Rolle spielen (siehe nächstes Kapitel).

SIGNAL LINK Optimierungsoptionen

Falls die Signalvariablen direkt als *Monitored Variablen* oder *Controlled Variablen* verwendet werden können (bei trivialer Transformation), kann mit Pointer- oder Direktzugriffen gearbeitet werden:

VARIABLE - *Monitored* oder *Controlled Variable* haben eigenen Speicherplatz (keine Optimierung)

POINTER - *Monitored* oder *Controlled Variable* ist Pointer auf die Signalvariable.

MACRO - Direkter Zugriff auf die Signalvariable mit Macro **`#define aMonVar() aSignalVar`**

SlidingGate - DEC-Datenflussmodell (ohne Datentypen)

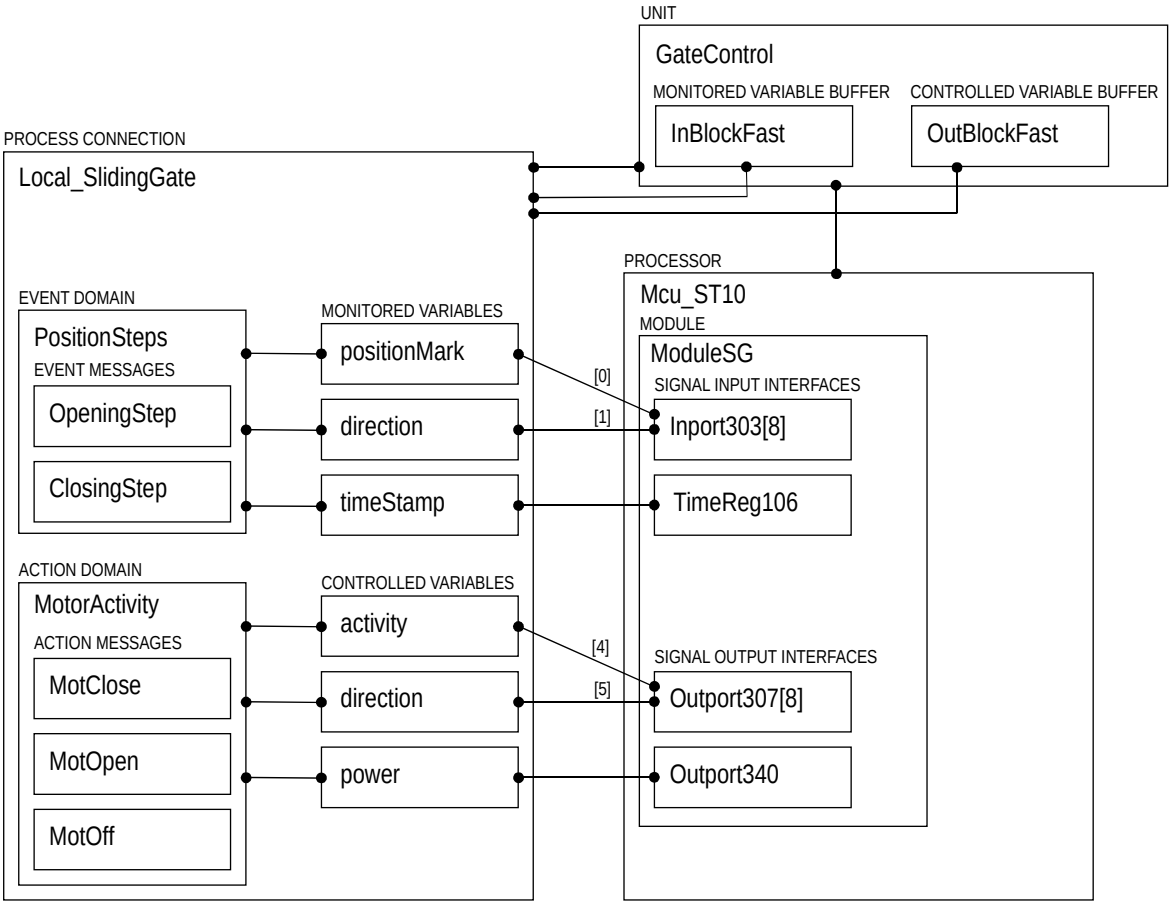


Abb. 59: DEC Datenflussmodell für die Process Connection SlidingGate

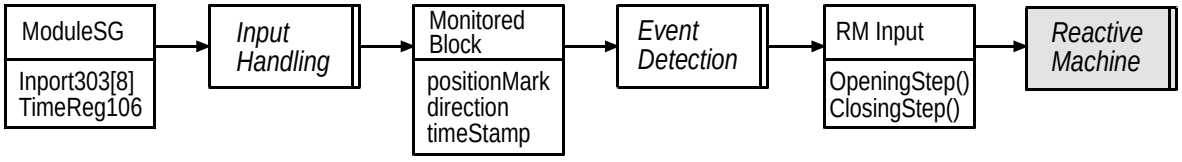
Die grafische Darstellung zeigt die verlinkten DEC-Objekte der DEC-Spezifikation für die Torsteuerung. Die Spezifikation definiert das Datenflussmodell für einen *Input* und einen *Output Channel*.

5.4 Interaktionsfunktionen

Jede Datenflussmaschine besteht aus den spezifizierten Schnittstellenvariablen und Interaktionsfunktionen, welche als Transformationen den Datenfluss erzeugen. Das Datenflussmodel definiert für jede Interaktionsfunktion die Input- und Outputschnittstellen. Das Erstellen der Interaktionsfunktionen kann damit im DEC Tool durch sogenannte *kontextsensitive Editoren* unterstützt werden.

Beispiel SlidingGate

Local_SlidingGate: Input Datenflussmaschine



Local_SlidingGate: Output Datenflussmaschine

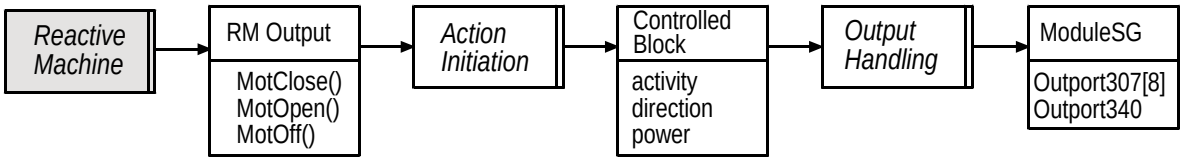


Abb. 60: Datenflussmaschinen für die Process Connection SlidingGate

DEC Tool: Kontextsensitive Editoren für Interaktionsfunktionen

Eine in einem kontextsensitiven Editor editierte Interaktionsfunktion (compound statement) besteht aus Tags und geschriebenem C-Code. Die Tags repräsentieren spezifizierte DEC-Objekte. Sie stehen in Listen zur Verfügung und können mit *Drag and Drop* in den Editor gezogen werden.

Kontextsensitiver Editor für eine Event Detection Function

PROCESS CONNECTION SlidingGate

MONITORED VARIABLES I

- positionMark
- positionMark_old
- direction
- timeStamp

```
if (positionMark != positionMark_old){
    if (!positionMark){
        if (direction)
            OpeningStep(timeStamp);
        else
            ClosingStep(timeStamp);
    }
    positionMark_old = positionMark;
}
```

EVENT MESSAGES I

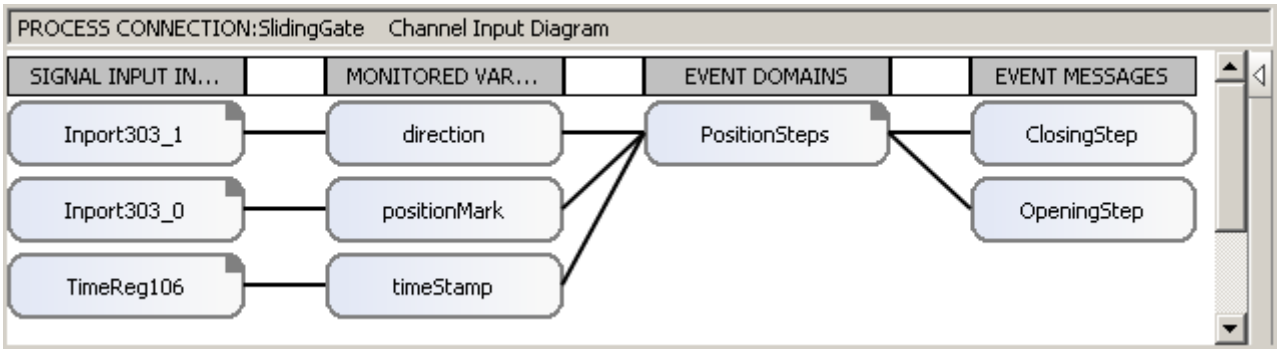
- Openingstep()
- Closingstep()

Channel Diagramme

Im DEC Tool erzeugte *Channel Diagramme* visualisieren die Spezifikation der Input und Output Datenflussmaschinen der *Process Connections*.

Markierte Boxen sind kontext-definierende Objekt für Interaktionsfunktionen. Aus dem Menü einer solchen Box lässt sich der entsprechende kontextsensitive Editor öffnen.

Input Channel Diagram der Process Connection SlidingGate

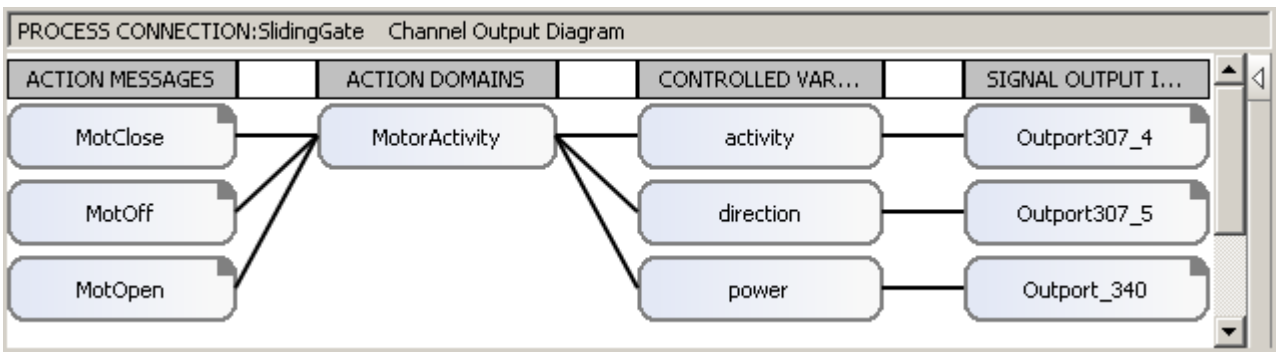


SIGNAL INPUT INTERFACES - MONITORED VARIABLES - EVENT DOMAINS - EVENT MESSAGES

SIGNAL INPUT INTERFACES sind kontext-definierende Objekte für *Input Handler Functions*

EVENT DOMAINS sind kontext-definierende Objekte für *Event Detection Functions*

Output Channel Diagram der Process Connection SlidingGate



ACTION MESSAGES - ACTION DOMAINS - CONTROLLED VARIABLES - SIGNAL OUTPUT INTERFACES

ACTION MESSAGES sind kontext-definierende Objekte für *Action Initiation Functions*

SIGNAL OUTPUT INTERFACES sind kontext-definierende Objekte für *Output Handler Functions*

Beispiele von Interaktionsfunktionen

Eine in einem kontextsensitiven Editor editierte Interaktionsfunktion (compound statement) besteht aus Tags und geschriebenem C-Code (Tagged Text). Die Tags repräsentieren spezifizierte DEC-Objekte. In den folgenden Beispielen sind die Tags durch **`fetten Text`** dargestellt. Der edierte C-Code ist `normaler Programm Text`.

Mit dem *expand all* Befehl können die Tags expandiert werden. Es erscheint der C-Code, der bei der Codegenerierung entstehen würde. Die Variablennamen der generierten Datenstrukturen sind immer Kombinationen von DEC-Objektnamen der DEC Spezifikation.

Event Detection Function für die Event Domain PositionSteps

Erläuterung der Funktion

1. Deklaration der statischen Variable *direction_old* (letzter Wert von direction). *positionMark_old* wird automatisch deklariert, weil *positionMark* die Detection Variable ist.
2. Prüfen, ob die Detection Variable *positionMark* den Wert verändert hat (Pointer auf Inport).
3. Prüfen, ob die Indication Variable *direction* gleich geblieben ist (Inport-Zugriff mit Macro).
4. Prüfen, ob der neue Wert von *positionMark* auf *low* ist (Ist die negative Flanke aufgetreten?).
5. Falls die Indication Variable *direction* auf high ist, Event Message *OpeningStep* triggern; andernfalls Event Message *ClosingStep* triggern.

   Der Parameter dieser Funktionsaufrufe referenzieren die Attribute Variable *timeStamp*.
6. Variablen *direction_old* und *positionMark_old* aktualisieren.

Bemerkung.
positionMark_MASK und *direction_MASK* sind Masken für das Bit entsprechenden Port-Signals.

```
{
   static u8 direction_old;
   if ((*positionMark^positionMark_old) & positionMark_MASK){
      if((({direction()^direction_old) & direction_MASK) == 0){
          if ((*positionMark & positionMark_MASK) == 0){
              if (direction() & direction_MASK){
                  OpeningStep(&temp_OpeningStep);
              }
              else {
                  ClosingStep(&temp_ClosingStep);
              }
          }
      }
      direction_old = direction();
      positionMark_old = *positionMark;
}
```

expanded

```
{
   static u8 direction_old;
   if ((*InBlockFast.ECH_SlidingGate.positionMark^positionMark_old) & MASK_0){
      if(((ModuleSG.Inport303^direction_old) & MASK_1) == 0){
          if ((*InBlockFast.ECH_SlidingGate.positionMark & MASK_0) == 0){
              if (ModuleSG.Inport303 & MASK_1){
                  IN_LocalPeriphery.ECH_SlidingGate.OpeningStep
                                      (&InBlockFast.ECH_SlidingGate.timeStamp);
              }
              else {
                  IN_LocalPeriphery.ECH_SlidingGate.ClosingStep
                                      (&InBlockFast.ECH_SlidingGate.timeStamp);
              }
```

```
            }
        }
        direction_old = ModuleSG.Inport303;
        positionMark_old = *InBlockFast.ECH_SlidingGate.positionMark;
    }
}
```

Action Initiation Function für die Action Message MotOpen

Erläuterung der Funktion

1. *activity* Bit setzen (Pointer auf Outport).
2. *direction* Bit setzen (Outport-Zugriff mittels Macro)
3. Parameter-Wert der Action Message in die Attribut Variable *power* kopieren.

```
{
    *activity |= activity_MASK;
    direction() |= direction_MASK;
    power = MSG_power;
}
```

expanded

```
{
    *OutBlockFast.ACH_SlidingGate.activity |= MASK_4;
    ModuleSG.Outport307 |= MASK_5;
    OutBlockFast.ACH_SlidingGate.power = data_->power;
}
```

Input Handler Function für das Signal Input Interface TimeReg106

Erläuterung der Funktion

1. Time-Register einlesen.
2. Register-Wert skalieren und in die Attribute Variable *timeStamp* kopieren.

```
{
    TimeReg106 = readWordReg( addrTimeReg106 );
    timeStamp = usPerTick * TimeReg106;
}
```

expanded

```
{
    ModuleSG.TimeReg106 = readWordReg( addrTimeReg106 );
    InBlockFast.ECH_SlidingGate.timeStamp = usPerTick * ModuleSG.TimeReg106;
}
```

Output Handler Function für das Signal Output Interface Outport340

Erläuterung der Funktion

1. Attribute Variable *power* skalieren und in die Interface Variable *Outport340* kopieren.
2. Wert der Interface Variable auf den Port schreiben.

```
{
    Outport340 = PFactor * power;
    writeWordPort( addrPort340, Outport340 );
}
```

expanded

```
{
    ModuleSG.Outport340 = PFactor * OutBlockFast.ACH_SlidingGate.power;
    writeWordPort( addrPort340, ModuleSG.Outport340 );
}
```

5.5 Codegenerierung mit dem DEC Tool

Die formale Entwicklung der Datenflussmaschinen mit dem DEC Tool verläuft semantisch auf zwei unterschiedlichen Ebenen.

1. Die Datenflussmaschinen-Architektur wird im DEC Tool ausgehend von den Zustandsgrössen der externen Prozesse, der zu verarbeitenden Ereignisse und der zu erzeugenden Aktionen durch formale relationale Modelle spezifiziert. Damit werden alle Interaktionspfade spezifiziert, und für jede Interaktionsfunktion ist der Kontext durch eine entsprechende Input-Output-Relation definiert.

2. Die auszuführenden elementaren Interaktionsfunktionen werden direkt durch Programmierung in kontextsensitiven Editoren des DEC Tool als *Compound Statements* implementiert. Die Input- und Output-Schnittstellen dieser Programmkomponenten stehen als Tags zur Verfügung, die in den Programmtext gezogen werden können.

Code Generierung

Sowohl die Interaktionsmodelle als auch der editierte Code der Interaktionsfunktionen können automatisch verifiziert werden. Aus vollständigen Spezifikationsmodellen kann der Code für das Schnittstellengerüst der Interaktionsfunktionen generiert werden. Dabei werden die elementaren Interaktionsfunktionen, die in den kontextsensitiven Editoren als *Compound Statements* zu erstellen sind, in entsprechenden generierten *Interaktionsprozeduren* eingebunden. Damit wird die Ausführung gleichartiger Interaktionsfunktionen in Prozeduren zusammengefasst, die vom entsprecheden *Unit Controller* aufgerufen werden können.

Die Organisation der Interaktionsfunktionen in Prozeduren ist durch die Strukturierung der DEC Spezifikation bestimmt.

Zum Beispiel für das Input Handling können SIGNAL INPUT INTERFACES in verschiedenen MODULES zusammengefasst werden. Die pro MODULE generierte *Input Handler Procedure* führt dann die entsprechenden *Input Handler Functions* (1 Compound Statement pro Input Variable) in einer Sequenz aus. Falls ein MODULE *Bit-Variablen* enthält (Zugriff über Bit-Masken), werden die Funktionen, welche die Ports einlesen (*HW Interface Access*), nur einmal am Anfang der Prozedur aufgerufen.

Oder für die Event Detection sind die MONITORED VARIABLES in verschieden MONITORED VARIABLE BUFFERS zu organisieren. Wenn zum Beispiel für eine Unit sowohl lokale und als auch verteilte Prozessverbindungen spezifiziert sind, müssen die MONITORED VARIABLES entsprechend in mindestem einem MONITORED VARIABLE BLOCK BUFFER und in einem MONITORED VARIABLE QUEUE BUFFER angelegt werden. Für die Event Detection werden dann zwei entsprechende Event Detection Procedures generiert. Die Prozedur, die auf den Datenblock zugreift, führt alle Event Detection Functions in einer Sequenz aus. Die Prozedur hingegen, welche die Queue abarbeitet, führt die Event Detection Functions selektiv in entsprechenden Switch/Case Statements aus.

Ausführungssteuerung des generierten Codes

Die Ausführung der Datenflussmaschinen ist damit noch nicht vollständig definiert. Zum Teil ist sie bereits durch die generierten Interaktionsprozeduren bestimmt (Reihenfolge der Ausführung der Interaktionsfunktionen). Die Ausführungsreihenfolge der Interaktionsprozeduren hingegen wird erst bei der Programmierung der *Unit Controller* der Embedded Units festgelegt (non-preemptives Real-Time Scheduling der Ereignisverarbeitung). Weiter sind bei verteilter Prozessinteraktion die Interaktionsprozeduren, welche das Erfassen des Signal-Inputs und das Erzeugen des Signal-Outputs betreffen, in einem *Processor Controller* des entsprechenden Peripherieknoten aufzurufen. Das gilt auch für Interaktionsprozeduren, die für das Communication Handling zuständig sind.