4 Entwicklung reaktiver Modelle - DPC Methode

Wir basieren die Entwicklung der Steuerfunktionen auf kooperierende Zustandsmaschinen. Dabei stellt sich aber die Frage, wie das reaktive Modell zu strukturieren ist. Objekt-orientiert oder durch funktionale Top-Down-Dekomposition der Steuerfunktionen? Wieviele Zustandsmaschinen sind zweckmässig? Wir werden sehen, dass der domänenorientierte Entwicklungsansatz auch hier eine Antwort gibt. Ausgangspunkt muss das Verhalten der Technischen Prozesse sein. Das Verhalten der Sensoren und Aktoren darf hingegen keine Rolle spielen, weil bei der Definition der Modellschnittstellen von der realen Interaktionsverbindung abstrahiert wurde (Virtuelle Interaktion).

Im Abhängigkeitsgrafen des allgemeinen domänenorientierten Entwicklungsprozesses (Unterkapitel 2.4) ist die Entwicklung der Steuerfunktionen lediglich durch zwei Knoten dargestellt: *Reactive Model* und davon abhängig *Reactive Machine. Reactive Model* stellt den Teilprozess dar, in welchem das reaktive Modell der Steuerfunktionen entwickelt wird. Dieser Modellierungsprozess ist das Thema dieses Kapitels. Der Knoten *Reactive Machine* stellt den ausführbaren Quellcode des reaktiven Modells dar. Der Entwicklungsschritt vom reaktiven Modell zum ausführbaren Code ist vollständig automatisiert, wenn das reaktive Modell mit dem CIP Tool erstellt wird (Codegenerierung).

Im Unterkapitel 4.1 werden die Grundlagen des methodischen Modellierungsprozesses erläutert. Ausgangspunkt sind die Technischen Prozesse, welche die Basis für die Strukturierung des reaktiven Steuermodells liefern. Das hier eingeführte dynamische Übereinstimmungsprinzip verlangt für jeden Technischen Prozess einen entsprechenden Prozessagenten im reaktiven Modell. Diese Prozessagenten bilden als dynamische Interaktionsschnittstelle die Basiskomponenten für das kooperative Steuermodell.

Unterkapitel 4.2 beschreibt den domänenorientierten Konstruktionsprozess für reaktive embedded Modelle. Dabei wird das dynamische Übereinstimmungsprinzip angewendet, um für Technische Prozesse entsprechende Prozessagenten zu spezifizieren. In weiteren Entwicklungsschritten werden dann die kooperativen Steuerfunktionen konstruiert.

Unterkapitel 4.3 erläutert am Beispiel *DoorControlSystem* die einzelnen Entwicklungsschritte des domänenorientierten Modellierungsprozesses. Insbesondere wird gezeigt, wie in mehreren iterativen Entwicklungsschritten ein kompatibler Prozessagent entsteht, der auch das Verhalten in Bezug auf externe Race Conditions korrekt modelliert.

In Unterkapitel 4.4 wird ein typisches Verhaltensmuster von steuerbaren Prozessen erläutert und nochmals auf die Problematik externer Race Conditions (kritische Aktionsinitiierungen) eingegangen.

Unterkapitel 4.5 erläutert schliesslich, wie für ein CIP-Modell ausführbare Software-Komponenten generiert werden können, und wie diese mit der I/O-Software zu verknüpfen sind. Mit einem simplen ausführbaren Codebeispiel (main program) wird gezeigt, wie eine generierte Komponente mit Tastatureingaben und Bildschirmausgaben verbunden werden kann. Die Entwicklung der I/O-Software (Verbindungssoftware) für Embedded Systeme ist aber Thema des nächsten Kapitels.

Bemerkung. Die DPC Methode setzt nicht einen bestimmte Spezifikations- oder Modellierungssprache voraus. Die Methode kann auch ohne Modellierungswerkzeug angewendet werden.

Als Modellierungssprache wird in diesem Kapitel der im vorhergehenden Kapitel erläuterte grafische CIP-Formalismus verwendet. CIP wurde bereits in den neunziger Jahren entwickelt um die domänen-orientierte Entwicklung von Steuermodellen zu unterstützen.

4.1 Übereinstimmung von System- und Prozessverhalten

Die zentrale Aufgabe eines Embedded Systems ist das Überwachen und Steuern der Prozesse der Systemumgebung. Ausgangspunkt bei der domänenorientierten Entwicklung funktionaler Modelle ist die Spezifikation der virtuellen Interaktion zwischen Technischen Prozessen und der zu modellierenden reaktiven Maschine. Dabei ist es notwendig, die einzelnen Technischen Prozessen mit entsprechenden Komponenten der reaktiven Maschine zu verbinden. Diese Komponenten sind vorerst gegenseitig unabhängige Zustandsmaschinen, die wir als *Prozessagenten* bezeichnen. In der Komponentenliteratur werden solche Zustandsmaschinen als *Interface Automata* bezeichnet. Zwischen den Technischen Prozessen und den entsprechenden Agenten entsteht so eine Verhaltensbeziehung, welche die Grundlage für das funktionale Modell bildet. Das Verhalten jedes Agenten muss dabei mit dem Verhalten des verbundenen Technischen Prozesses übereinstimmen, das heisst jeder erzeugte Stimulus muss vom Partner erwartet werden. Nur mit solchen dynamischen Übereinstimmungen wird es möglich sein, eine ständige Interaktion zwischen Prozessen und reaktiver Maschine zu unterhalten. Das ist ähnlich wie in zuverlässig kommunizierenden Systemen, wo solche Bedingungen durch sequentielle Protokolle spezifiziert werden müssen.

Mit den Prozessagenten wird ein dynamisches Schnittstellenmodell definiert, das die Basis für die Entwicklung der reaktiven Steuerfunktionalität des Embedded Systems bildet. Das Modell für das kollektive reaktive Gesamtverhalten des Embedded Systems entsteht in anschliessenden Entwicklungsschritten, in denen weitere Zustandsmaschinen eingeführt werden, die mit den Prozessagenten interagieren. Diese rein funktionalen Komponenten sind dafür verantwortlich, dass die gesteuerten Technischen Prozesse korrekt kooperieren.

Verhaltenseigenschaften Technischer Prozesse

Der Entwurf der Steuerfunktionen hängt sowohl von den funktionalen Anforderungen wie auch von den intrinsischen dynamischen Eigenschaften der zu steuernden Prozesse ab. Das intrinsische Verhalten dieser Prozesse ändert selten während der Lebenszeit (Software Life Cycle) des Embedded Systems; was oft ändert. sind die funktionalen Anforderungen. Der domänenorientierte Entwicklungsansatz geht deshalb von den dynamischen Eigenschaften der Technischen Prozesse aus. Dieser Ansatz befolgt ein fundamentales methodisches Softwarentwicklungskonzept, das Michael Jackson folgendermassen beschreibt: “Good methods place inventions as late as possible in the development”.

Nehmen wir zum Beispiel ein Lift-System. Ein einfache dynamische Eigenschaft der Liftkabine ist, dass sie immer zuerst den zweiten Stock in Aufwärtsrichtung oder den dritten Stock in Abwärtsrichtung verlassen haben muss, bevor sie den zweiten Stock erreichen kann. Das heisst, dem Ereignis *ReachThirdFloor* gehen immer entweder das Ereignis *LeaveSecondFloor* oder *LeaveForthFloor* voraus. Die möglichen Ereignissequenzen bewirken entsprechende Meldungssequenzen an der Schnittstelle der Reaktiven Maschine. Wir bezeichnen solche Sequenzen von Ereignismeldungen als *gültige* Input-Sequenzen, und jede Ereignismeldung einer solchen Sequenz als *gültiger* Input.

Wenn ein Technischer Prozess durch eine reaktive Maschine gesteuert wird, hängen die möglichen Ereignisse zusätzlich von den Aktionen ab, die von der reaktiven Maschine bewirkt werden. In unserem Liftbeispiel verlässt die Kabine den ersten Stock in Aufwärtsrichtung nur, wenn der Motor vorwärts dreht. Das Ereignis *ReachThirdFloor* kann deshalb nur nach dem Ereignis *LeaveSecondFloor* auftreten, wenn vorgängig die Aktion *MotorRaise* von der reaktiven Maschine bewirkt worden ist; die Aktion *MotorScend* würde entsprechend zum Ereignis *ReachFirstFloor* führen.

Dass die auftretenden Ereignissequenzen im allgemeinen von den erzeugten Aktionen abhängen ist typisch für Embedded Systeme. Es ist gerade dieser spezielle Problemcharakter, der den primären Zweck von Embedded Systemen reflektiert, nämlich das Ausüben ganz bestimmter Einwirkungen auf die beobachteten Prozesse. Ein Hauptziel bei der Entwicklung eingebetteter Systeme muss deshalb sein, diese dynamischen Abhängigkeiten genau zu verstehen und die Konstruktion reaktiver Maschinen entsprechend auf die Sequenzen gültiger Ereignismeldungen zu basieren.

4.1.1 Das dynamische Übereinstimmungsprinzip

Mit dem dynamische Übereinstimmungsprinzip wird bei der Entwicklung von Embedded Systemen dafür gesorgt, dass die Technischen Prozesse und die reaktive Maschine kompatibles Interaktionsverhalten haben.

Wie bereits vorgängig beschrieben, bauen wir reaktive Maschinen, die aus nebenläufigen Clustern kooperierender Zustandsmaschinen bestehen. Jeder Cluster ist eine sequentielle Maschine, die durch Ereignismeldungen getriggert wird. Die Meldungssequenzen werden durch verschiedene Technische Prozesse verursacht. Eine gültige Inputsequenz für einen Cluster ist deshalb eine Verflechtung gültiger Inputsequenzen, die vom den einzelnen Technischen Prozessen erzeugt werden. Um diesen Strom von Inputstimuli zu entflechten ist es naheliegend, im Cluster als Empfänger entsprechende Zustandsmaschinen zu verwenden. Diese Zustandsmaschinen bezeichnen wir als *Prozessagenten*. Jeder Prozessagent ist virtuell mit dem entsprechenden Technischen Prozess verbunden. Damit erhalten wir bereits eine statische Übereinstimmung zwischen Komponenten der Umgebung und des Embedded Systems, d. h. jeder Technische Prozess hat im System einen entsprechenden Prozessagenten als Interaktionspartner (*Abbildung 35*).

Externe parallele Ereignissequenzen	*Sequenz der Ereignismeldungen für die reaktive Maschine*	*Entflechtete Ereignismeldungen*
ProcessA: A1, A2		AgentA: A1, A2
ProcessB: B1, B2, B3	B1, A1, B2, C1, B3, A2	AgentB: B1, B2, B3
ProcessC: C1		AgentC: C1

Abb. 35: Statische Übereinstimmung: Technische Prozesse und Prozessagenten

Zusätzlich zu dieser statischen Übereinstimmung muss eine dynamische Übereinstimmung der Prozessagenten mit den Technischen Prozessen verlangt werden. Solche dynamischen Übereinstimmungen kennt man von kommunizierenden Systemen. Zuverlässige Kommunikationsprotokolle werden typischerweise mittels dynamisch übereinstimmenden Zustandsmaschinen spezifiziert:

> Zwei kommunizierende Zustandsmaschinen haben übereinstimmendes Verhalten, wenn genau die von der einen Zustandsmaschine erzeugbaren Meldungssequenzen von der anderen empfangen werden können.

Wenn wir dynamische Übereinstimmung von Prozessagent und verbundenem Prozess verlangen, können wir nur Bedingungen an die dynamischen Eigenschaften des Agenten stellen, die dynamischen Eigenschaften des Technischen Prozesses sind nämlich bereits durch seine physikalischen Eigenschaften bestimmt:

> Das Verhalten eines Prozessagenten stimmt mit dem Verhalten des verbundenen Technischen Prozesses überein, wenn der Agent genau die vom Technischen Prozess erzeugbaren Sequenzen von Ereignismeldungen empfangen kann, und wenn der Agent nur Aktionssequenzen initiiert, die für den Technischen Prozess gültig sind.

Die Stimulussequenzen, die ein Agent empfangen und produzieren kann sind durch seine Transitionsstruktur bestimmt (Zustandsübergangsdiagramm). Bei der Entwicklung der Transitionsstruktur ist deshalb von den intrinsischen dynamischen Eigenschaften des verbundenen Technischen Prozesses auszugehen.

Während den initialen Entwicklungsschritten einer reaktiven Maschine wird man deshalb eine Menge gegenseitig noch unabhängiger Prozessagenten spezifizieren. Diese Agenten bilden ein dynamisches Schnittstellenmodell für die reaktive Maschine. Das Schnittstellenmodell garantiert, dass die zu entwickelnden funktionalen Lösungen nur auf gültigen Ereignissequenzen basieren, und dass nur gültige Aktionssequenzen erzeugt werden. Die gesamte reaktive Maschine wird in anschliessenden Entwicklungsschritten konstruiert, indem weitere Zustandsmaschinen eingeführt werden, die

mit den Prozessagenten so interagieren, dass die funktionalen Anforderungen an das Embedded System erfüllt werden.

Trotz übereinstimmendem Verhalten können ungültige Ereignismeldungen nicht ausgeschlossen werden. Ungültige Ereignisse können zum Beispiel auftreten, wenn sich die Technischen Prozesse nicht wie vorgesehen verhalten, oder wenn Sensor- oder Übertragungsfehler auftreten. Ein dynamisch übereinstimmender Prozessagent wird auf eine ungültige Ereignismeldungen nicht mit einem Zustandsübergang reagieren. Trotzdem sollten solche Meldungen nicht ignoriert werden. Bei einer implementierten reaktiven Maschine erwarten wir deshalb, dass eine ungültige Ereignismeldung eine sogenannte *Context Error Exception* auslöst. Der Exception-Handler kann dann zum Beispiel ein spezifisches Fehlerereignis generieren, das in der Folge die reaktive Maschine für eine Fehlerbehandlung triggert. Der dadurch angestossene Error-Handler, eine weitere Zustandsmaschinenkomponente der reaktiven Maschine, kann dann eine geeignete Änderung des reaktiven Verhaltens bewirken zum Beispiel indem in einen *Rescue Mode* gewechselt wird.

Oft wird die Transitionsstruktur eines Prozessagenten so erweitert, dass er auf bestimmte ungültige Ereignismeldungen mit einer direkten Fehlerbehandlung reagieren kann. Durch solche Strukturerweiterungen werden die betroffenen ungültigen Ereignismeldungen gültig, obwohl sie als inkorrekte Meldungen behandelt werden. Mit solchen Erweiterungen des dynamischen Schnittstellenmodells werden Fehlerbehandlungsfunktionen gewissermassen zu einem Teil der modellierten Systemfunktionalität.

Periodische Ereignisse

Zeitgetriebene Beobachtung der Technischen Prozesse basiert auf Abtastereignissen (sampling events), die repetitiv mit gleichem zeitlichen Abstand entsprechender Steueralgorithmen anstossen. Die abgetasteten Werte werden mit der Ereignismeldung als Parameter mitliefert (event attributes). Die Steueralgorithmen sind als Erweiterungen in der Zustandsmaschine (EFSM) des Prozessagenten integriert. Die auszuführenden Systemreaktionen werden durch Berechnung sogenannter Stellwerte ermittelt, die der bewirkten Aktion als Parameter mitgegeben werden (action attributes). Auch die Entwicklung von Steueralgorithmen basiert auf den dynamischen Eigenschaften des gesteuerten Prozesses. Solche Eigenschaften werden aber typischerweise mit Differentialgleichungen beschrieben. Für die Entwicklung der Steueralgorithmen ist die klassischen Regeltechnik zuständig. Unser Begriff von diskreter dynamischer Übereinstimmung wird hier trivial, denn es gibt pro Prozess nur einen einzigen Ereignistyp, das Abtastereignis, und es kann nur eine einzige Ereignissequenz auftreten, die zeitlich periodische Folge von Abtastereignissen.

4.1.2 Dynamische Übereinstimmung von Prozessagenten mit Prozessmodellen

Wir illustrieren die dynamische Modellierung von Schnittstellen mit vier Beispielen, die sich alle auf einen einfachen Lift beziehen. Die ersten zwei Beispiele sind bezüglich der dynamischen Übereinstimmung trivial; sie zeigen lediglich zwei typische Fälle, in denen der Prozessagent und das Prozessmodell identisches Verhalten haben. Das dritte Beispiel zeigt, dass dynamische Übereinstimmung im allgemeinen Fall nicht einfach durch Kopieren des Prozessmodells erhalten werden kann; das Prozessmodell und der entsprechende Prozessagent haben im allgemeinen unterschiedliche dynamische Strukturen. Das vierte Beispiel geht auf das ernsthafte Problem von *Race Conditions* ein, die aufgrund von Interaktionskonflikten zwischen Technischem Prozess und Prozessagent auftreten können.

In allen vier Beispielen ist der Technische Prozess ein simpler Lift mit zwei Stockwerken. Das relevante Verhalten des Technischen Prozesses wird auch mit einer Zustandsmaschine dargestellt. Auftretende Ereignisse sind hier als Output-Stimuli notiert, während die durch den Agenten bewirkte Aktionen als Input-Stimuli erscheinen. Für die grafische Darstellung der Prozessmodelle wurde die klassische FSM-Notation gewählt. Die Prozessagenten sind mit der CIP-Notation spezifiziert.

Beispiel 1 – Beobachten der Position eines Lifts mit zwei Stockwerken

Prozessmodell *Prozessagent*

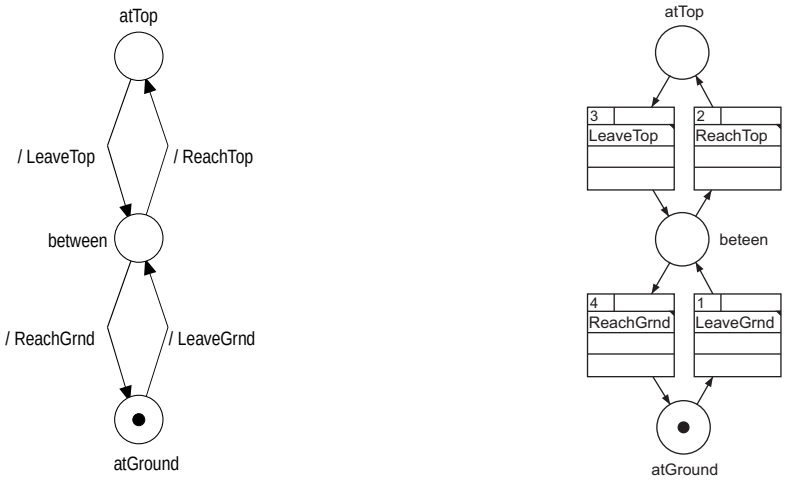


Abb. 36: Technischer Prozess als Sender und Prozessagent als Empfänger

Im ersten Beispiel (*Abbildung 36*) muss das System Information über Liftbewegungen rapportieren; der Liftmotor ist von Hand gesteuert. Die Spezifikation eines Prozessagenten mit übereinstimmendem Verhalten ist hier trivial, weil seine Transitionsstruktur gleich gewählt werden kann wie die des Prozessmodells. Mittels Ereignismeldungen wird lediglich der Zustand des Agenten nachgeführt, das heisst die dynamische Struktur des Prozessagenten ist gleich wie die des Prozessmodells.

Beispiel 2 – Steuern des Liftmotors

Prozessmodell *Prozessagent*

Abb. 37: Prozessagent als Sender und Technischer Prozess als Empfänger

Im zweiten Beispiel (*Abbildung 37*) steuert das System den Liftmotor. Die Steuerung benutzt zum Beispiel einen vorgegebenen statischen Zeitplan (Schedule). Der Prozessagent kann wieder dieselbe dynamische Struktur wie das Prozessmodell haben. Im Vergleich zum ersten Beispiel ist jetzt der Prozessagent dem Prozessmodell zeitlich immer voraus.

Bemerkung: Prozessagenten werden in der Kontextmodellierungsphase spezifiziert und sind im allgemeinen durch unvollständige Zustandsmaschinen darzustellen. Der Prozessagent der Motorsteuerung hat zum Beispiel noch keine Input Stimuli, welche die Zustandsmaschine treiben. Das Zustandsdiagramm wird erst in einer weiteren Entwicklungsphase vervollständigt, in der die Interaktion mit andern Zustandsmaschinen der reaktiven Maschine definiert werden.

Beispiel 3 – Steuern der Liftkabine

Prozessmodell *Prozessagent*

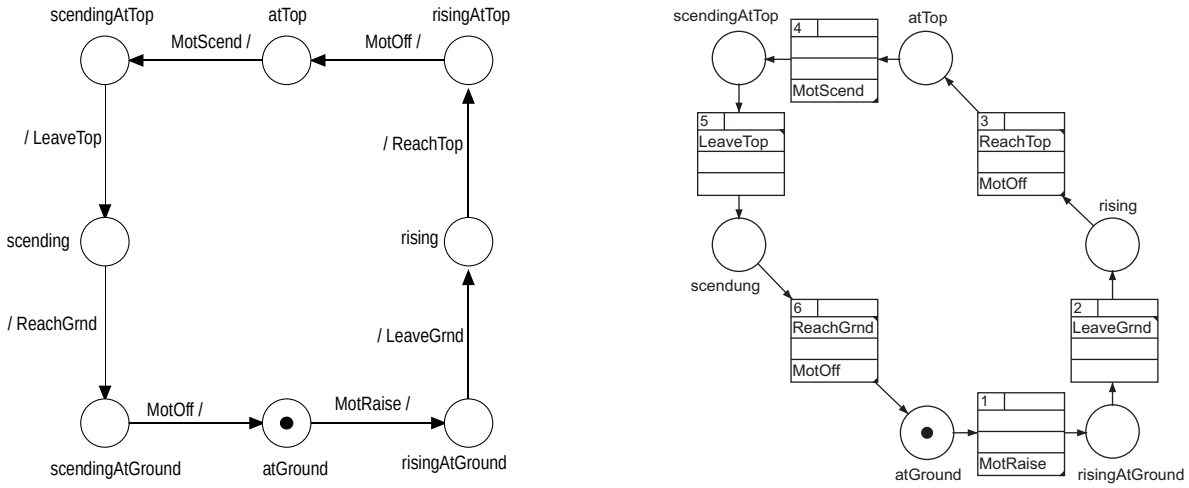


Abb. 38: Prozess und Prozessagent mit bidirektionaler Interaktion

Das dritte Beispiel *Abbildung 38* zeigt in vereinfachter Form eine übliche Liftsteuerung. Für die Steuerung des Motors muss die Liftposition überwacht werden. Damit hängt die Initiierung der Motoraktionen vom Verhalten des Technischen Prozesses ab. Die notwendige Information über den Liftzustand erfolgt über entsprechend erzeugte Ereignismeldungen.

Im diesem Beispiel ist es nicht mehr möglich, den Prozessagenten als Kopie des Prozessmodells zu definieren. Bezüglich den auftretenden Ereignissen ist seine Reaktion immer zeitlich verzögert, hingegen bezüglich den bewirkten Aktionen ist der Agent immer dem Prozessmodell zeitlich voraus. Es gibt zum Beispiel keinen Zustand *risingAtTop* beim Prozessagenten. Der Zustand *risingAtTop* des Prozessmodells ist notwendig, weil das Ereignis und die Aktion aufgrund der asynchronen Stimulusübertragung nicht gleichzeitig stattfinden kann. Im Gegensatz dazu wird die Aktion *MotOff* durch den Agenten direkt in der Transition initiiert, die durch die Ereignismeldung *ReachTop* getriggert wird, das heisst die Initiierung Aktion erfolgt synchron in der Reaktion auf die Ereignismeldung. Die zeitliche Ordnung der Übertragung der Stimuli während eine Liftbewegung vom unten nach oben ist in *Abbildung 39* durch ein *Message Sequence Diagram* dargestellt.

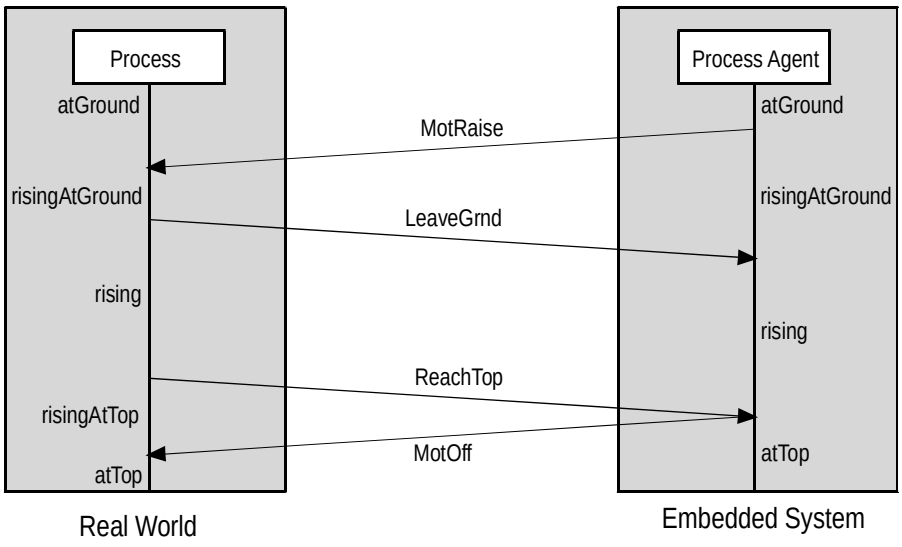


Abb. 39: Message Sequence Diagram einer Liftbewegung

Die dynamische Übereinstimmung des Agenten mit dem Prozessmodell ist in diesem Beispiel leicht zu verifizieren. Die Aktionen werden immer zu Zeiten initiiert, zu denen keine Ereignismeldung unterwegs sein kann. Damit sind die Sequenzen von Ereignissen und Aktionen beim realen Prozess gleich wie die entsprechenden Sequenzen von Ereignismeldungen und Aktionsinitiierungen beim Agenten.

Solange dass Aktionen nur dann initiiert werden, wenn der Technische Prozess in einem stationären Zustand ist, oder als Reaktion auf ein Ereignismeldung des gesteuerten Prozesses, ist die dynamische Übereinstimmung einfach zu erhalten. Es kann in diesen Fällen angenommen werden, dass die Sequenz von Ereignissen und Aktionen mit der Sequenz von Ereignismeldungen und Aktionsinitiierungen übereinstimmt. Diese Annahme geht aber davon aus, dass die Übertragungszeit der Stimuli entscheidend kleiner ist als der zeitliche Abstand zweier sich folgender Ereignisse.

Beispiel 4 – Asynchrones Anhalten der Liftkabine

Prozessagenten mit dynamischer Übereinstimmung zum Technischen Prozess zu spezifizieren wird schwierig, sobald die reaktive Maschine *asynchron* auf einen Technischen Prozess einwirken muss, typischerweise als Reaktion auf einen Stimulus (Pulse) eines anderen Prozessagenten, der durch eine Ereignismeldung eines anderen Technischen Prozesses getriggert wurde. Wir möchten deshalb im vierten Liftbeispiel den aufsteigenden Lift zu einem beliebigen Zeitpunkt aufgrund einer Ereignismeldung eines anderen Prozesses anhalten können, z. B. weil ein Notschalter betätigt wurde.

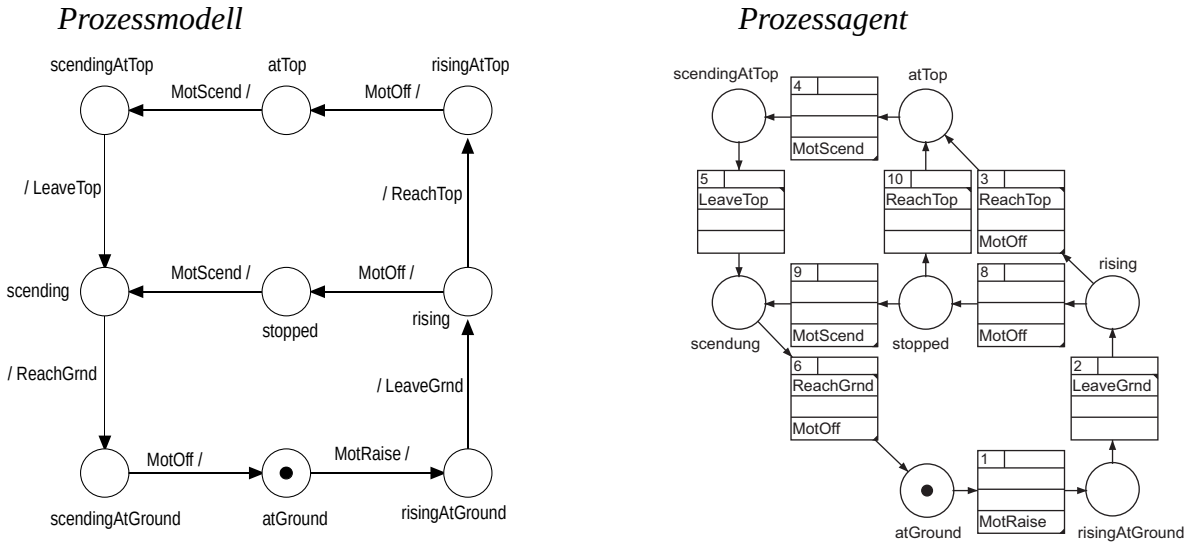


Abb. 40: Prozess und Prozessagent mit bidirektionalem Interaktionskonflikt

Die Erweiterung des Technischen Prozessmodells und des Prozessagenten ist in *Abbildung 40* dargestellt. Wir nehmen an, der Lift habe keine mechanische Trägheit, das heisst der aufsteigende Lift bewegt sich nicht mehr, sobald der Motor ausgeschaltet worden ist.

Der erweiterte Agent kann jetzt im Zustand *rising* durch eine andere Zustandsmaschine getriggert werden um die Aktion *MotOff* zu bewirken. Diese Aktion wird in der Transition in den Zustand *stopped* initiiert.

Obwohl der Lift keine Trägheit hat, müssen wir nach dem Stoppen des Motors während einer kurzen Zeit noch die Ereignismeldung *ReachTop* erwarten. Der Grund für diese Race Condition ist die asynchrone Stimulusübertragung zwischen Technischem Prozess und Prozessagent. In unserem Beispiel ist es deshalb möglich, dass das Ereignis *ReachTop* auftritt bevor die *MotOff* Aktion in der Realität ausgeführt worden ist.

Der Grund für die Race Condition ist ein bidirektionaler Interaktionskonflikt zwischen dem Technischem Prozess und dem Prozessagenten. Der Konflikt kann gut mit einem Message Sequence Diagram gezeigt werden (*Abbildung 41*). Die *ReachTop* und die *MotOff* Message können sich während

der Übertragung kreuzen. Das Diagramm zeigt auch, dass die Zeitordnung des *ReachTop* Ereignisses und der *MotOff* Aktion beim Prozess und die Zeitordnung von Ereignismeldung und Aktionsinitiierung beim Agenten verschieden sein können.

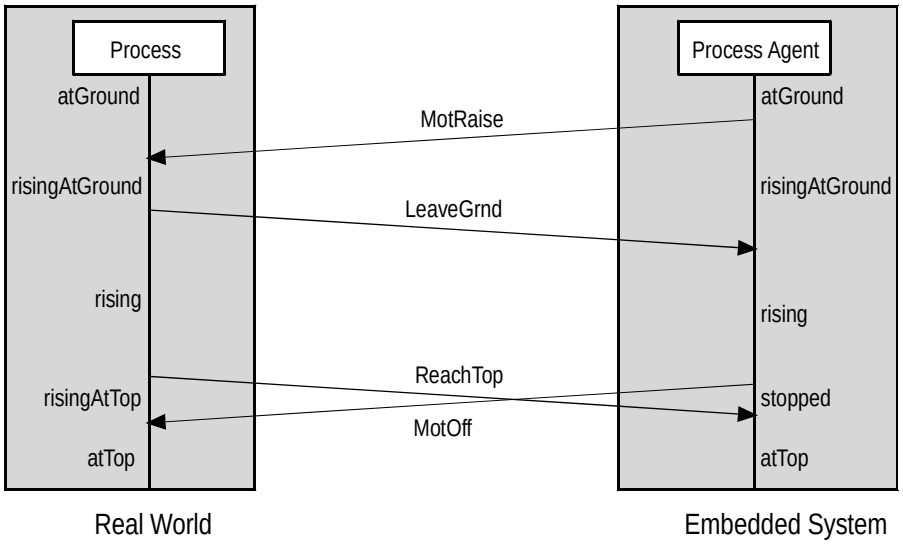


Abb. 41: Message Sequence Diagram einer spät unterbrochenen Liftbewegung

Eine ähnliche Race Condition wäre in unserem Beispiel möglich, wenn die Trägheit der Liftkabine in Betracht gezogen würde. Der Sinn des Beispiels war jedoch zu zeigen, dass bereits die bidirektionale asynchrone Interaktion einen Trägheitscharakter hat, der zu nicht-voraussagbarem Verhalten führen kann.

Das Problem der ereignisgesteuerten asynchronen Beeinflussung technischer Prozesse ist sehr ernst zu nehmen. Ein Hauptgrund dafür ist, dass das Auftreten von Race Conditions beim Testen unwahrscheinlich ist. Obwohl solche Race Condition das Resultat selten auftretender Verhaltenskoinzidenz ist, kann der produzierte Effekt katastrophale Folgen haben. Es ist deshalb notwendig, bereits bei der Erstellung der Verhaltensmodelle auf diese Problematik sensibilisiert zu sein.

Im Unterkapitel 4.2 zur domänenorientieren Modellierung wird in einem Fallbeispiel (DoorControl-System) nochmals auf die “Race Condition”-Problematik eingegangen.

Klassische Objektorientierte Modellierungsansätze und Embedded Systeme

Bei der Entwicklung einer Softwarelösung von einem Modell der Umgebung auszugehen ist ein bekanntes Konzept. Objektorientierte Methoden zum Beispiel beginnen bei der Entwicklung mit einem Objektmodell der betroffenen realen Domäne. Das Domänenmodell wird in der Folge dann als Modell für die Softwareanwendung verwendet, das heisst das gleiche Modell ist sowohl Modell der Realität als auch Modell der Software. Mit einem gemeinsamen Modell zu arbeiten funktioniert gut, so lange die entwickelte Applikation nur die Geschichte realer Objekte nachvollziehen muss (wie bei unserem ersten Liftbeispiel), was typisch der Fall ist für Desktop-und Business-Applikationen. Die Aufgabe solcher Systeme ist meistens, die erfasste Information zu verarbeiten und Anfragen über die Resultate zu beantworten.

Sobald aber die modellierten realen Objekte von der Software auch beeinflusst werden müssen, so wie das für Embedded Systeme typisch ist, kann der aus der Desktop-Welt stammende objektorientierte Modellierungsansatz nicht mehr angewendet werden. Es ist dann nicht mehr möglich, mit einem gemeinsamen Modell für Realität und Software zu arbeiten, weil die Software-Objekte bezüglich erfasster Ereignisse zeitlich im Rückstand, bezüglich den zu produzierenden Aktionen aber zeitlich voraus sind. Diese Zeitunterschiede sind zwar immer klein, aber sobald “Closed Loop”-Interaktion im Spiel ist, muss mit Race Conditions gerechnet werden, die zu nicht voraussehbaren Fehlern führen können.

4.2 Domänenorientierte Modellierung

Mit formalen Verhaltensmodellen wie Zustandsmaschinen wird automatisch formal in einer bestimmen Abstraktionsebene gearbeitet. Zusätzlich zu solchen „Leitplanken“ sind methodische Entwicklungskonzepte notwendig, damit die Entwicklung von Lösungen nachvollziehbar wird. Nur so können Lösungen verständlich sein.

Bei der Entwicklung von CIP-Modellen wird vom Verhalten der Technischen Prozesse ausgegangen werden. Ziel ist, ein embedded Modell mit dynamischer Übereinstimmung zu den Technischen Prozessen zu konstruieren (siehe dynamisches Übereinstimmungsprinzip in 4.1.2). Das domänenorientierte Vorgehen führt zu verständlichen Modellen, da die Bedeutung der Modellelemente und Modellkonstrukte aufgrund der dynamischen Übereinstimmung mit den Technischen Prozessen letztlich immer auf reale Phänomene dieser Prozesse zurückgeführt werden kann. Es ist naheliegend, dass sich die realitätsbezogene Modellsemantik unmittelbar auf Qualitätsmerkmale wie Zuverlässigkeit, Wartbarkeit und Änderungsfreundlichkeit auswirkt.

4.2.1 Modellierungsprozess

Im Abhängigkeitsgrafen des allgemeinen domänenorientierten Entwicklungsprozesses (Unterkapitel 2.4) erscheint die Entwicklung der Steuerfunktion für eine Unit lediglich als Knoten, der mit *Reactive Model* bezeichnet ist. Weiter ist aus dem Abhängigkeitsgrafen ersichtlich, dass für die Generierung der ausführbaren Komponente *Reactive Machine7* nur das *Reactive Model benötigt wird.*

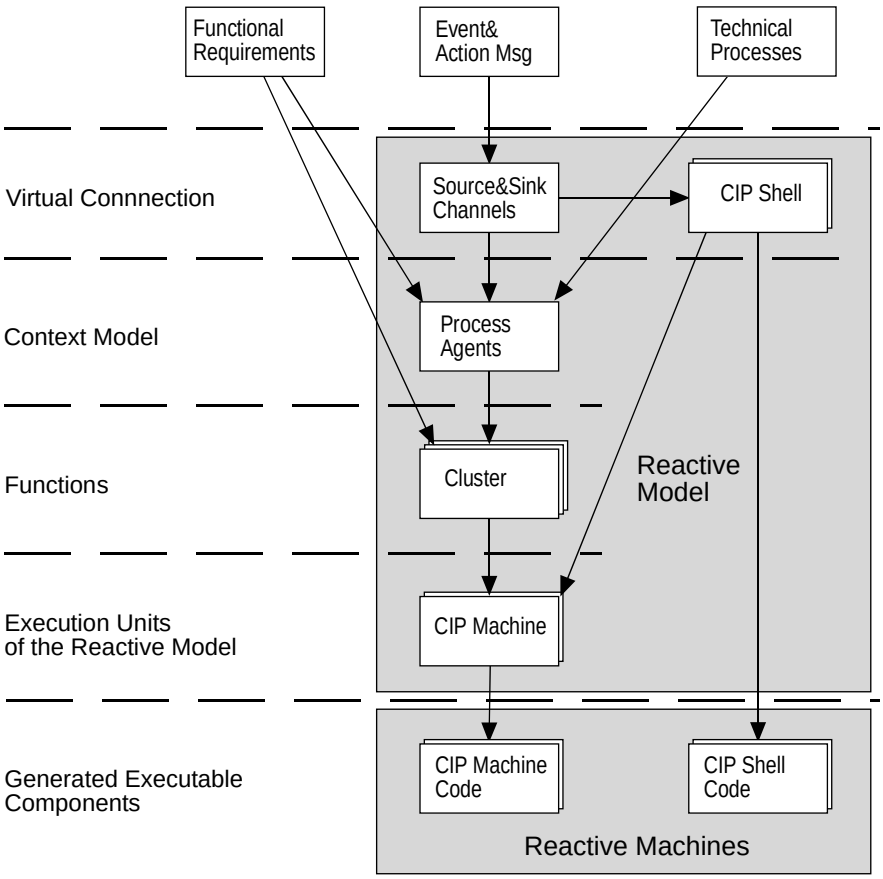


Abb. 42: Modellierungsprozess - Teilprozess des domänenorientierten Entwicklungsprozesses

Die Konstruktion des reaktiven Modells ist ein Teilprozess des gesamten Entwicklungsprozesses, dargestellt in *Abbildung 42*. Um diesen Modellierungsprozess ausführen zu können, müssen die funktionalen Anforderungen (*Functional Requirements)* und die wesentlichen Eigenschaften der zu steuernden Technischen Prozesse (*Technical Processes*) bekannt sein. Die wichtigste Voraussetzung

ist aber die Kenntnis der Ereignisse, auf die das reaktive Modell reagieren muss, und die Aktionen, die dabei bewirkt werden können (*Events&Actions Msg*).

4.2.2 Konstruktionsschritte bei der Entwicklung von CIP-Modellen

Die Festlegung der Ereignisse und Aktionen und die Spezifikation der virtuellen Interaktion mittels Ereignis- und Aktionskanälen sind Entwicklungsschritte des allgemeinen domänenorientierten Entwicklungsprozesses. Es sind diejenigen Entwicklungsschritte, welche die Aufteilung in eine Steuerproblem und ein Einbettungsproblem ermöglichen. Für den Entwickler des reaktiven Modells ist durch die Ereignisse und Aktionen die Abstraktionsebene für die Modellierung bereits klar vorgegeben.

Aufgrund von spezifizierten Ereignis- und Aktionskanälen können die CIP-Kanäle (*Source&Sink-Channels*) spezifiziert werden, welche die virtuelle Verbindung zu den Technischen Prozesse auf der Modellseite herstellen. Über *Source Channels* wird das reaktive Modell durch *Event Messages* getriggert, und über *Sink Channels* werden mittels *Action Messages* bei den Technischen Prozessen Aktionen bewirkt. Damit vervollständigen die externen CIP-Kanäle (Message Quellen und Senken) das Spezifikationsmodell für die virtuelle Interaktion zwischen Prozessen und reaktiver Maschine. Aufgrund der Semantik von Ereignis- und Aktionskanälen ist klar, dass Event Messages desselben Inputkanals zu sequentiell abhängigen Ereignissen gehören, und Action Messages desselben Output Kanals zu sequentiell abhängigen Aktionen. Die Messages verschiedener Kanäle werden gegenseitig parallel übertragen.

Oft wird die Definition der Ereignisse und Aktionen nicht als erste Entwicklungsphase des gesamten DESC-Entwicklungsprozesses betrachtet, sondern lediglich als erster Schritt bei der Erstellung eines CIP-Modells. Dabei wird noch nicht an die Rolle gedacht, die dieser Spezifikation bei der Einbettung der reaktiven Maschine auf dem Zielsystem zukommt. Der Grund dafür liegt darin, dass man CIP-Modelle sehr oft als Prototypen benützt (Rapid Prototyping), sei es um die funktionalen Anforderungen zu evaluieren oder um bei der Diskussion mit dem Auftraggeber bereits bestimmte Steuerfunktionen visuell simulieren zu können. Ereignisse und Aktionen werden so bereits als externe Stimuli für CIP-Modelle spezifiziert, bevor überhaupt entschieden worden ist, was vom geplanten System zu realisieren ist.

Entwicklungsschritte für ein CIP-Model

1. Ereignisse und Aktionen definieren
2. Entwicklung eines Kontextmodells
3. Konstruktion der Steuerfunktionen

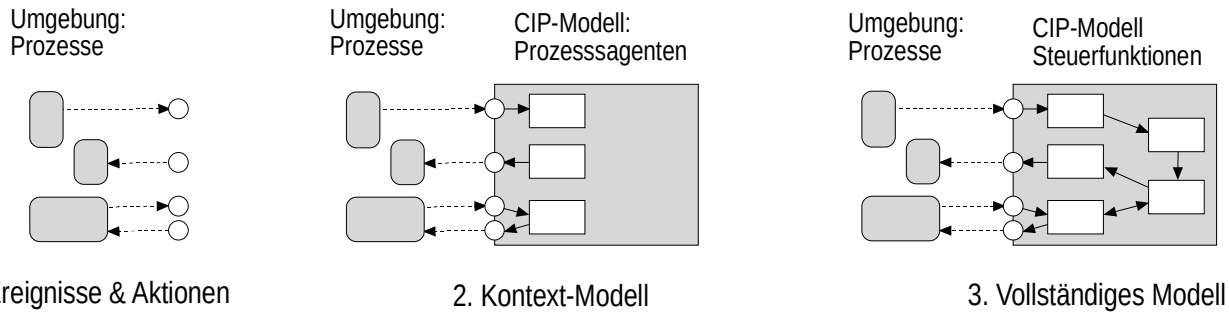


Abb. 43: Entwicklungsschritte für ein reaktives embedded Modell

Schritt 1 – Ereignisse und Aktionen

In einem ersten Schritt wird mittels Ereignissen und Aktionen die virtuelle Interaktion mit dem CIP-Modell spezifiziert. Ereignisse (*Events*) und Aktionen (*Actions*) sind reale Prozessphänomene, die

für das funktionale Verhaltens des Embedded Systems wesentlich sind. Ereignisse werden durch die Technischen Prozesse, Aktionen durch das eingebettete System bewirkt.

Entsprechende Ereignis- und Aktionsmeldungen definieren die Input- und Output-Schnittstelle auf der Modell-Seite der virtuellen Verbindung. Attribute von Ereignissen und Aktionen werden als Daten der entsprechenden Meldungen übertragen. Ein Ereignisattribut beschreibt einen Umstand, der beim Auftreten des Ereignisses zutrifft, wie zum Beispiel der erfasste Strich-Code eines Scanner-Ereignisses oder die Parameter eines Bediener-Befehls. Ein Aktionsattribut hingegen beschreibt einen Sachverhalt, für den beim Erzeugen der Aktion gesorgt werden muss, wie etwa die verlangte Durchflussrate eines geöffneten Ventils.

Ein Ereignis wird entweder als Prozess- oder Zeitereignis klassifiziert. Prozessereignisse werden aufgrund der Eigendynamik technischer Prozesse erzeugt und treten zu nicht vorausbestimmbaren Zeitpunkten auf. Solche Ereignisse bezeichnen meistens diskrete Zustandsübergänge technischer Prozesse. Diskrete Zustände stellen oft Abstraktionen dar, die eine ganze Menge von physikalischen Zuständen beinhalten, wie zum Beispiel die Niveaubereiche eines Kessels oder der Ready-Zustand eines technischen Gerätes.

Kontinuierliche physikalische Zustandsänderungen werden wie üblich durch Abtastung erfasst. Abtastereignisse sind periodische Zeitereignisse, deren Attribute die erfassten Prozesszustände beschreiben. Der Einfluss auf kontinuierliche Prozesse erfolgt entsprechend durch periodisch abgesetzte Aktionsmeldungen, welche die Stellgrössen übermitteln.

Das Erarbeiten der Ereignis- und Aktionslisten ist der grundlegende Entwicklungsschritt für das Embedded System, weil damit die Abstraktionsebene für die funktionale Problemlösung festgelegt wird. Die spezifizierten Ereignisse und Aktionen müssen einerseits erlauben, ein CIP-Modell zu konstruieren, welches den funktionalen Anforderungen genügt. Andererseits muss auch sichergestellt werden, dass alle Ereignisse erfasst und alle Aktionen erzeugt werden können.

Wir definieren hier die Ereignis- und Aktionslisten im Rahmen der CIP-Modellierung. Wird umfassend mit der DESC-Entwicklungsmethode gearbeitet, werden die Ereignis- und Aktionslisten bei der der Modellierung der Verbindungssoftware mit der DEC Methode (DEC Tool) definiert und können im CIP Tool als entsprechende Source und Sink Channels importiert werden (siehe Kapitel 5).

Sobald die Ereignisse und Aktionen festgelegt sind, können sowohl das funktionale wie auch das Einbettungsproblem parallel gelöst werden. Veränderungen an den Anforderungen betreffen nur dann beide Problemkreise, wenn die Spezifikation der Ereignisse oder Aktionen und damit die Spezifikation der virtuellen Verbindung geändert werden muss.

Schritt 2 – Kontextmodell: Prozessagenten

Das Kontextmodell ist ein initiales Teilmodell der zu entwickelnden Steuerung und besteht aus einer Menge von unabhängigen Zustandsmaschinen, die wie gesagt als Prozessagenten bezeichnet werden. Die Prozessagenten sind über Ereigniskanäle (source channels) und Aktionskanäle (sink channels) mit den Technischen Prozessen verbunden (virtuelle Verbindung). Die Transitionsstrukturen der Prozessagenten definieren die gültigen Folgen empfangener Ereignismeldungen und davon abhängig die zu erzeugenden Aktionsmeldungen. Dabei muss dafür gesorgt werden, dass ein Prozessagent und der verbundene Technische Prozess übereinstimmendes Verhalten haben, das heisst die produzierten Meldungssequenzen müssen auf der Gegenseite als gültige Einwirkungen erzeugen. Die Prozessagenten haben damit den Stellenwert sequentieller Kommunikationsprotokolle. Das Kontextmodell beschreibt gewissermassen das mögliche Verhalten der einzelnen Technischen Prozesse aus der Sicht des zu entwickelnden Systems. Die Prozessagenten bilden so die Grundlage für die Entwicklung robuster reaktiver Systemfunktionalität.

Für die Erstellung des Kontextmodells muss vom Verhalten der Technischen Prozesse ausgegangen werden, das heisst, wir müssen uns im Klaren sein, welche Ereignissequenzen auftreten können, und welche Aktionssequenzen zu erzeugen sind. Welche Aktion jeweils bewirkt werden muss, hängt sowohl von der angeforderten Funktionalität des Embedded Systems als auch vom Zustand des

Technischen Prozesses ab. Der aktuelle Zustand eines Technischen Prozesses sollte aufgrund der aufgetretenen Ereignisse bekannt sein. Zudem ist der Zweck einer Aktion ja das Verhalten eines Technischen Prozesses in bestimmter Art und Weise zu beeinflussen. Eine bewirkte Aktion bestimmt damit auch meistens, welche neuen Ereignisse in naher Zukunft zu erwarten sind. Solche Zusammenhänge sind oft durch die Kausalität physikalischer Interaktionen bestimmt. Wenn zum Beispiel bei einer geschlossenen Türe der Motor eingeschaltet wird (Aktion *MotOpen*), ist zu erwarten, dass sich die Türe nach einer gewissen Zeit ganz geöffnet hat (Ereignis *Opened*). Oder eine Heizung bewirkt das Ansteigen der Temperatur, oder ein geöffnetes Ventil ist der Grund, das sich ein Kessel füllt.

Damit dürfte klar sein, dass ein Prozessagent nur aufgrund eines Verhaltensmodells des entsprechenden Technischen Prozesses erstellt werden kann. Prozessmodelle wurden ja auch bereits bei der Diskussion des Übereinstimmungsprinzips verwendet. Auch wenn kein explizites Prozessmodell erstellt wird, existiert ein solches zumindest im Kopf des Entwicklers, oder es steckt implizit einer technischen Prozessbeschreibung. Bei komplexen Prozessen ist es jedoch kaum mehr möglich, ohne explizites Prozessmodell entsprechende Prozessagenten mit übereinstimmendem Verhalten zu konstruieren.

Schritt 3 – Steuerfunktionen: Kooperation von Prozessagenten und Funktionsprozessen

In weiteren Entwicklungsphasen wird das Gesamtverhalten des eingebetteten Systems modelliert. Dazu werden Koordinationsprozesse eingeführt und mittels Modell-Interaktion die notwendige Wechselwirkung mit den Prozessagenten erzeugt. In einem ersten Schritt entwickelt man die primäre Funktionalität des Systems, die sich auf das durch die Prozessagenten definierte Normalverhalten abstützt. Damit auch auf unerwartetes Verhalten der Umgebung und auf Übertragungsfehler reagiert werden kann, müssen meistens in weiteren Schritten die Strukturen der Prozessagenten erweitert und Fehlerprozesse eingeführt werden. Wie die Erfahrung zeigt, gehört die Spezifikation des Systemverhaltens für den Fehlerfall zu den schwierigsten Problemen bei der Entwicklung robuster und sicherer Systeme. Ohne explizites Modell des Normalverhaltens kann diese Aufgabe meist gar nicht befriedigend gelöst werden.

4.2.3 Einfaches Modellierungsbeispiel: Motorsteuerung

Das Beispiel ist vom Verhalten der technischen Prozesse her sehr einfach. Der Zweck des Beispiels ist zu zeigen, wie die minimal notwendigen Schritte im Modellierungsprozess durchgeführt werden. Am Schluss ist ein ausführbares Modell vorhanden, das die funktionalen Anforderungen erfüllt.

Anforderungen

Die Steuerung für einen Motor soll nach folgenden Vorgaben realisiert werden:

Der Motor wird mit zwei separaten Tasten manuell ein- und ausgeschaltet.

Ein diskreter Temperatursensor kann Überlast des Motors anzeigen. Daraufhin soll der Motor automatisch ausgeschaltet und die Störung mit einer Lampe signalisiert werden.

Mit einer Quittungstaste kann die Behebung der Störung quittiert werden.

Das Ausschalten der Störungslampe zeigt an, dass der Motor wieder gestartet werden kann.

Schritt 1 – Ereignisse und Aktionen

In einem ersten Entwicklungsschritt werden von jedem Technischen Prozess diejenigen Ereignisse und Aktionen aufgelistet, welche für die Konstruktion der Prozesssteuerung relevant sind. Zudem muss sichergestellt sein, dass die Ereignisse erfasst (Sensoren) und die Aktionen produziert (Aktoren) werden können.

Ereignisse

Ereignis-kanal	Technischer Prozess	erzeugtes Ereignis	Beschreibung
Befehle	Taste1	Start	Die Starttaste wird gedrückt.
Befehle	Taste2	Stop	Die Stoptaste wird gedrückt.
Befehle	Taste3	Quittung	Die Quittungstaste wird gedrückt.
MotorEvt	Motor	UeberT	Die Motortemperatur überschreitet den kritischen Wert T.
MotorEvt	Motor	UnterT	Die Motortemperatur unterschreitet den kritischen Wert T.

Aktionen

Aktions-kanal	Technischer Prozess	bewirkte Aktion	Beschreibung
MotorAct	Motor	MotEin	Der Motor wird eingeschaltet.
MotorAct	Motor	MotAus	Der Motor wird ausgeschaltet.
Stoerungen	Lampe	S_LmpEin	Die Störungslampe wird eingeschaltet.
Stoerungen	Lampe	S_LmpAus	Die Störungslampe wird ausgeschaltet.

Das Verhalten der Technischen Prozesse ist sehr einfach und kann z. B. mit BNF beschrieben werden:

Tasten-Ereignissequenzen = {Start | Stop | Quittung};

Lampen-Aktionssequenzen = {S_LmpEin, S_LmpAus};

Motor-Aktionssequenzen = {MotEin, MotAus};

Temperatur-Ereignissequenzen = {UeberT, UnterT};

Wir nehmen dabei an, dass die kritische Temperatur auch kurz nach dem Ausschalten des Motors noch detektiert werden kann, d. h. bei Normaltemperatur muss immer mit dem Ereignis *UeberT* gerechnet werden.

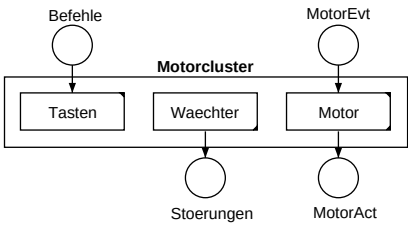
Schritt 2 – Kontextmodell: Inputkanäle, Outputkanäle und Prozessagenten

Im zweiten Entwicklungsschritt wird mittels Input- und Outputkanälen die virtuelle Verbindung zum Embedded System spezifiziert. Die Kanäle werden mit CIP-Zustandsmaschinen verbunden, die als Prozessagenten bezeichnet werden. Die Prozessagenten kommunizieren virtuell mit dem entsprechenden Technischen Prozess und definieren durch ihre Transitionsstrukturen sequentielle Kommunikationsprotokolle. Sie müssen sicher stellen, dass genau die gültigen Ereignisse erfasst und nur gültige Aktionen bewirkt werden.

Input- und Outputkanäle

CHANNEL Befehle
MESSAGES Quittung, Start, Stop
CHANNEL MotorEvt
MESSAGES UeberT, UnterT
CHANNEL MotorAct
MESSAGES MotAus, MotEin
CHANNEL Stoerungen
MESSAGES S_LmpAus, S_LmpEin

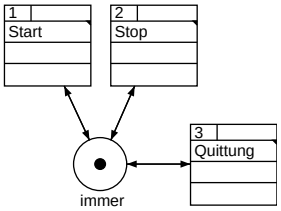
COMMUNICATION NET MotNet



Prozessagenten

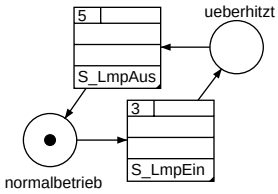
Die Prozessagenten bilden die ersten Komponenten des formal spezifizierten reaktiven Systems. Die Transitionsstruktur eines Prozessagenten ist durch das Verhalten des entsprechenden Technischen Prozesses bestimmt und stellt ein sequentielles Protokoll dar, welches die Kommunikation von System und Umgebung auf einer abstrakten funktionalen Ebene beschreiben. Zwischen den Prozessagenten bestehen in dieser Entwicklungsphase noch keine Abhängigkeiten.

PROCESS Tasten



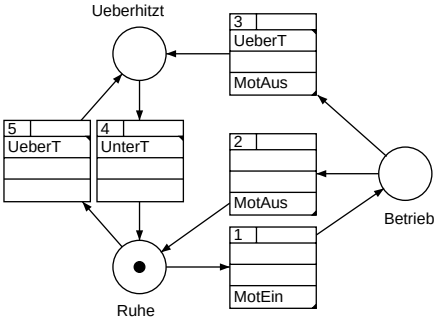
INPORT Cmdport
MESSAGES Quittung, Start, Stop

PROCESS Waechter



OUTPORT anzStoerung
MESSAGES S_LmpAus, S_LmpEin

PROCESS Motor



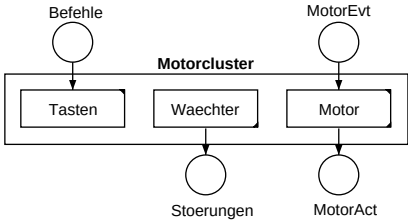
INPORT MotEvents
MESSAGES UeberT, UnterT
OUTPORT ActionPort
MESSAGES MotAus, MotEin

Schritt 3 – Koordinieren und Steuern

Die Transitionsstrukturen der Prozessagenten bilden die Basis für Konstruktion der Interaktion (Pulse Cast, State Inspection, Mode Control) zwischen den Prozessagenten. Dabei müssen die Transitionsstrukturen vervollständigt und z. T. auch erweitert werden. Um Strukturkonflikte und komplexe Strukturerweiterungen der Prozessagenten zu vermeiden, wird man spezifische Koordinationsprozesse einführen und die Interaktion zwischen den Prozessagenten über diese Funktionsprozesse ablaufen lassen.

Vollständiges CIP-Modell

COMMUNICATION NET MotNet



PULSE CAST NET

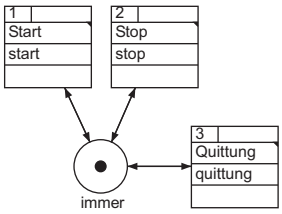
PULSE TRANSLATIONS

SENDER Motor
tempOk -> Waechter.tempOk
zuHeiss -> Waechter.zuHeiss

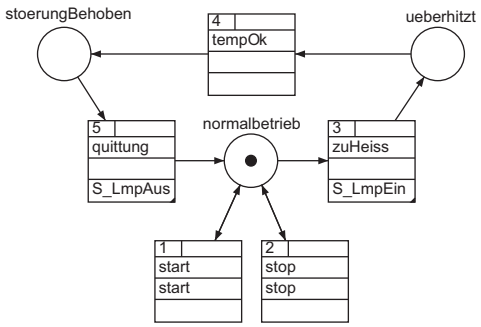
SENDER Tasten
quittung -> Waechter.quittung
start -> Waechter.start
stop -> Waechter.stop

SENDER Waechter
start -> Motor.start
stop -> Motor.stop

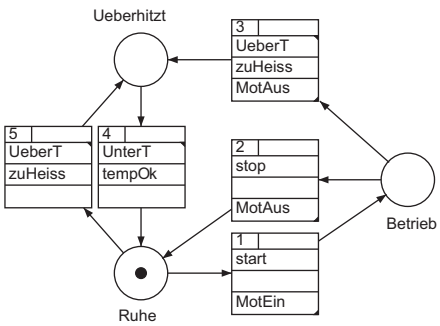
PROCESS Tasten



PROCESS Waechter



PROCESS Motor



In diesem einfachen Beispiel macht es keinen grossen Sinn, einen separaten Koordinationsprozess einzuführen. Wir verwenden direkt den Prozessagenten *Waechter*, indem wir seine Transitionsstruktur entsprechend erweitern. Der Prozess wirkt als Filter, der abhängig von seinem Zustand Tastenereignisse weiterleitet.

4.3 Modellierungsbeispiel: DoorControlSystem - DCS

Das Beispiel zeigt den Modellierungsprozess für eine einfache Türsteuerung. Bei der Erstellung des Prozessagenten für die Türe werden wir auf typische Modellierungsprobleme treffen, die Ihren Ursprung im Closed-Loop-Charakter der Steuerung und in der asynchronen Verbindung von Anlage und Computer haben.

Anlage

Schiebetüre mit Motor und Positionssensoren

Taste

Lampe

Eine Türe kann mit Hilfe eines Motors geöffnet und geschlossen werden. Der Motor wird mit digitalen Signalen gesteuert,

Wenn sich die Türe ganz geöffnet hat, wird ein digitaler Sensor aktiviert. Ein weiterer Sensor wird aktiviert, sobald sich die Türe geschlossen hat.

Es ist gibt eine Taste, die sofort in den Ausgangszustand zurückkehrt, wenn sie losgelassen wird.

Eine Kontrolleuchte kann hell oder halbhell leuchten, oder sie ist ausgeschaltet.

Funktionale Anforderungen

Wenn die Taste gedrückt wird, muss sich die Tür öffnen - bei gedrückter Taste bleibt die Türe offen. Die offene Türe muss bei losgelassener Taste noch 3 Sekunden offen bleiben, bevor sie sich zu schliessen beginnt.

Die Kontrolleuchte orientiert über den Zustand von Türe und Taste. Wenn die Taste gedrückt ist, muss die Kontrolleuchte hell leuchten, wenn sie nicht gedrückt ist und die Türe nicht geschlossen ist leuchtet sie halbhell. Wenn die Türe geschlossen ist, soll die Kontrolleuchte nicht leuchten.

Schritt 1 – Ereignisse und Aktionen

In einem ersten Entwicklungsschritt werden von jedem Technischen Prozess diejenigen Ereignisse und Aktionen aufgelistet, welche für die Konstruktion der Prozesssteuerung relevant sind. Zudem muss sichergestellt sein, dass die Ereignisse erfasst (Sensoren) und die Aktionen produziert (Aktoren) werden können.

Ereignisse

Ereignis-kanal	**Technischer Prozess**	**erzeugtes Ereignis**	**Beschreibung**
E_Button	Taste	Push	Die Taste wird gedrückt.
E_Button	Taste	Release	Die Taste wird losgelassen.
E_Door	Tuere	Opened	Die Türe erreicht die *Offen*-Position.
E_Door	Tuere	Closed	Die Türe erreicht die *Geschlossen*-Position.

Aktionen

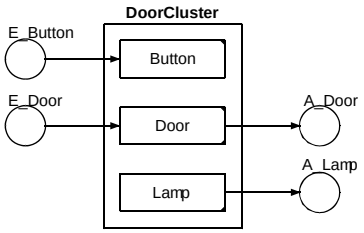
Aktions-kanal	**Technischer Prozess**	**bewirkte Aktion**	**Beschreibung**
A_Door	Tuere	MotOpen	Der Türmotor wird auf ‘Öffnen’ gestellt.
A_Door	Tuere	MotClose	Der Türmotor wird auf ‘Schliessen’ gestellt.
A_Door	Tuere	MotOff	Der Türmotor wird ausgeschaltet.
A_Lamp	Lampe	Bright	Die Lampe wird auf 'hell' gestellt.
A_Lamp	Lampe	Medium	Die Lampe wird auf 'halbhell' gestellt.
A_Lamp	Lampe	Dark	Die Lampe wird ausgeschaltet.

Schritt 2 – Kontextmodell: Inputkanäle, Outputkanäle und Prozessagenten

Input- und Outputkanäle

```
CHANNEL E_Button
        MESSAGE Push, Release
CHANNEL E_Door
        MESSAGE Opened, Closed
CHANNEL  A_Door
        MESSAGE MotOpen, MotClose, MotOff
CHANNEL A_Lamp
        MESSAGE Bright, Medium, Dark
```

COMMUNICATION NET EventActionChannels



Prozessmodell Button

Ereignissequenzen = {Push, Release};

Die Ereignisse *Push* und *Release* können nur abwechlungsweise auftreten, was einfach mit der BNF-Notation beschrieben werden kann:

Die bediente Taste wirkt als aktiver Prozess. Zustandsänderungen werden dem Prozessagenten *Button* mit den Ereignismeldungen *Push* und *Release* bekanntgegeben.

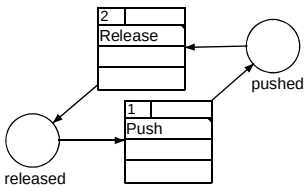
Prozessmodell Lamp

Aktionssequenzen = {Bright | Medium | Dark};

Die Aktionen dürfen in beliebiger Reihenfolge auftreten (BNF-Notation).

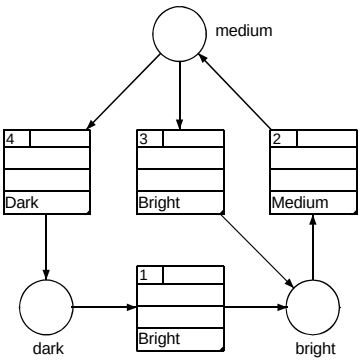
Die Lampe ist vom CIP-Modell her gesehen ein passiver Prozess, der mit den Aktionsmeldungen *Bright*, *Medium* und *Dark* über entsprechende Aktoren gesteuert wird. Durch den Prozessagenten *Lamp* werden die

Prozessagent Button




```
INPORT Evt
        MESSAGES Push, Release
```

Prozessagent Lamp




```
OUTPORT Act
        MESSAGES Bright, Medium, Dark
```

Tasten und Lampen sind einfache Input-, bzw. Outputgeräte, die entweder rein aktiven, bzw, passiven Charakter haben. Die Bildung der entsprechenden Prozessagenten ist deshalb fast trivial. Sie sind im wesentlichen Kopien der Prozessmodelle. Bei aktiven Geräten (*Button*) - vom Agenten her gesehen - sind die Prozessmodelle immer den entsprechenden Agenten zeitlich voraus, bei passiven Geräten (*Lamp*) sind sie hingegen immer zeitlich hintendrein.

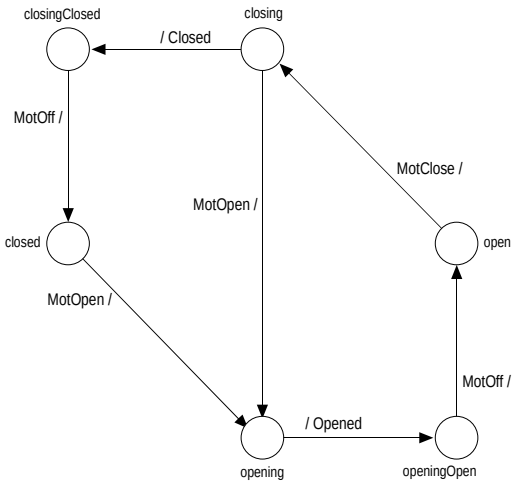
Bei der Türe ist das anders. Sie ist ein steuerbarer (responsiver) Prozess, der sowohl aktive als auch passive Komponenten hat. Die Position ist vom Agenten her gesehen eine aktive Zustandsgrösse, der Motorzustand eine passive. (Für die Türe wirkt der Motor mit seinem Drehmoment natürlich als aktive Komponente.)

Das Verhalten der Türe wird im folgenden als abstraktes Prozessmodell durch eine Zustandsmaschine beschrieben. Für das Prozessmodell verwenden wir zur besseren Unterscheidung vom Prozessagenten nicht die CIP-Notation. *Ereignisse* werden vom Modellprozess erzeugt (Output Stimuli), *Aktionen* treiben hingegen den Modellprozess (Input stimuli).

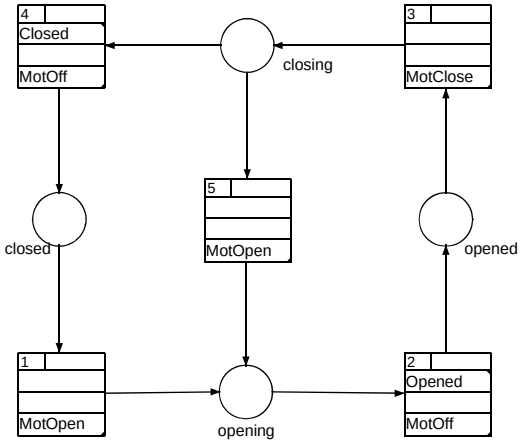
Es wird eine Folge von Folge von Paaren von Modellprozess und Prozessagent gezeigt, eine Folge, wie sie in einem realen Entwicklungsprozess auftreten könnte.

a) Zu einfach

Prozessmodell Door



Prozessagent Door



Events
Opened, Closed

Actions
MotOpen, MotClose, MotOff

INPORT Evt
MESSAGES Opened, Closed

OUTPORT Act
MESSAGES MotOpen, MotClose, MotOff

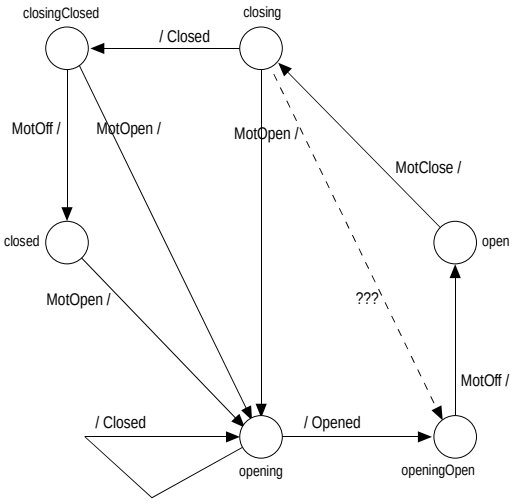
Problem: Beim Agenten kann unter Umständen im Zustand *opening* die Ereignismeldung *Closed* auftreten (context error). Dafür gibt es zwei Gründe.

Erstens muss aufgrund der Trägheit der Türe damit gerechnet werden, dass sich die fast geschlossenen Türe noch ganz schliesst, obwohl der Motor mit *MotOpen* auf Öffnen gestellt wurde.

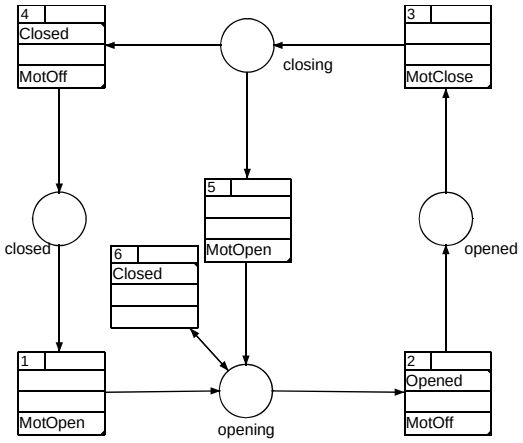
Zweitens kann das Ereignismeldung *Closed* auch bei einer Türe ohne Trägheit im Zustand *opening* auftreten. Der Grund ist die asynchrone Interaktion von Prozessmodel und Prozessagent. Während der Übertragung des *MotOpen* Stimulus kann sich die Türe ganz schliessen und damit den *Closed* Event erzeugen.

b) Weniger einfach, aber immer noch zu einfach

Prozessmodell Door



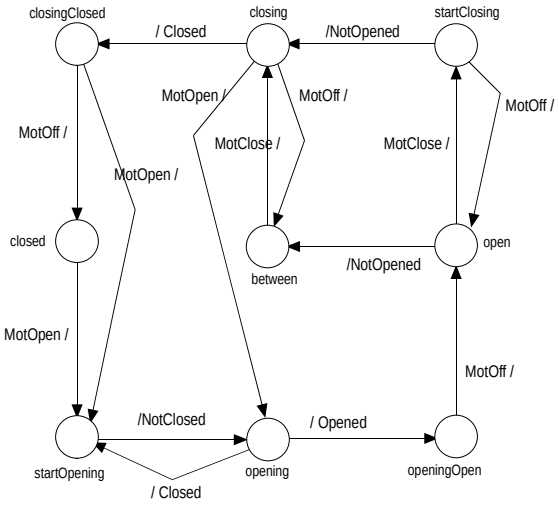
Prozessagent Door



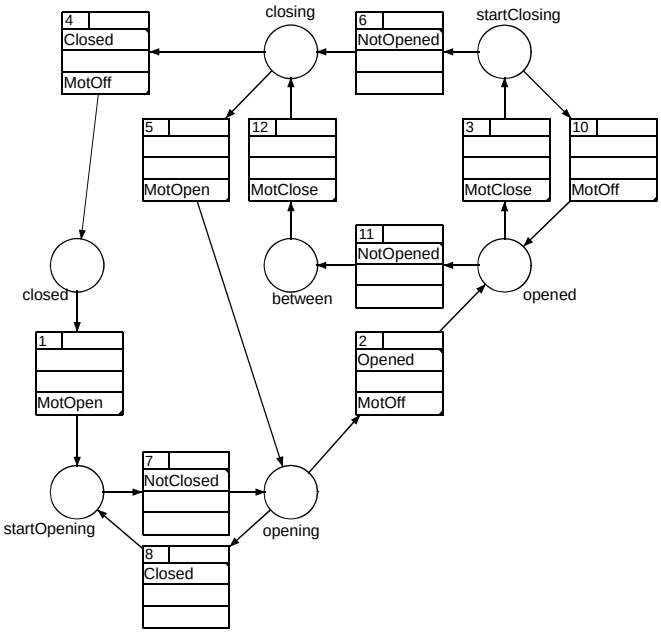
Problem: *MotOpen* wird u. U. erzeugt, obwohl die Türe den Offen-Sensor noch nicht verlassen hat. In der Folge wird der Agent im Zustand *opening* vergeblich auf die Ereignismeldung *Opened* warten, d. h. der Motor wird nie ausgeschaltet, ... Crash!

c) Korrekt: Erweiterung mit den Ereignissen NotClosed und NotOpened

Prozessmodell Door



Prozessagent Door



Events
Opened, Closed, NotOpened, NotClosed

Actionst
MotOpen, MotClose, MotOff

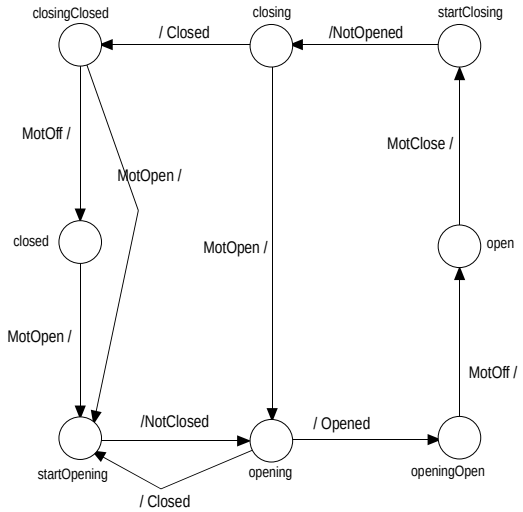
INPORT Evt
Opened, Closed, NotOpened, NotClosed

OUTPORT Act
MotOpen, MotClose, MotOff

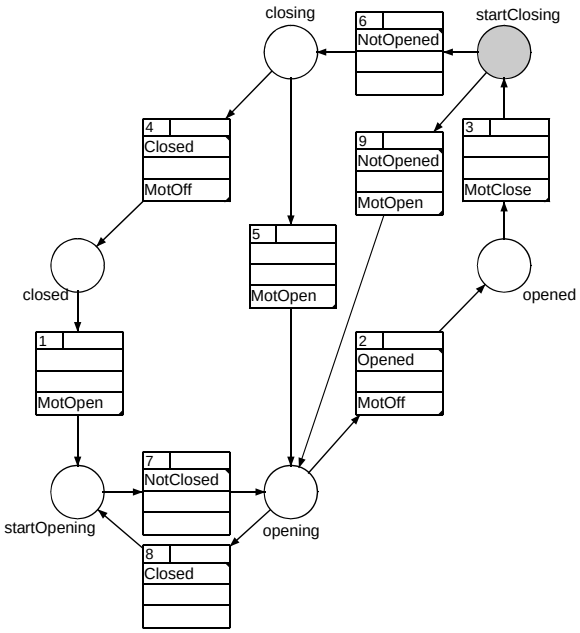
Die Menge der Türereignisse wurde um ***NotOpened*** (Offenbereich verlassen) und ***NotClosed*** (Geschlossenbereich verlassen) erweitert. Die externen Race-Condition, die bei der Erzeugung von *MotOpen* entstehen können, sind jetzt klar und deterministisch modellierbar.

d) Reduzierung der Race-Conditions mit synchroner Reaktion: NotOpened / MotOpen

Prozessmodell Door



Prozessagent Door



Die Race-Condition, die entsteht, wenn die sich zu schliessen beginnende Türe (*startClosing*) wieder geöffnet werden soll, kann verhindert werden. Der Agent muss einfach warten, bis die *NotOpened* Ereignismeldung angekommen ist und dann entscheiden, ob die *MotOpen* Aktion initiiert werden muss. Dabei wird der Agent im Zustand *startClosing* non-deterministisch, was aber problemlos bei der Vervollständigung des Modells mittels Zustandsinspektion determiniert werden kann.

Schritt 3 – Koordinieren und Steuern

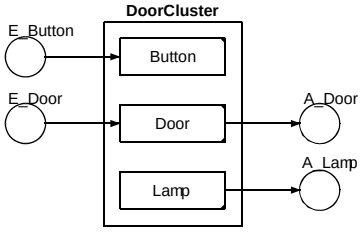
Die Transitionsstrukturen der Prozessagenten bilden die Basis für Konstruktion der Interaktion (Pulse Cast, State Inspection, Mode Control) zwischen den Prozessagenten. Dabei müssen die Transitionsstrukturen vervollständigt und z. T. auch erweitert werden. Um Strukturkonflikte und komplexe Strukturerweiterungen der Prozessagenten zu vermeiden, wird der Koordinationsprozess *Controller* einführt und die Interaktion zwischen den Prozessagenten über diesen Funktionsprozess geleitet.

Es folgt die vollständige Spezifikation des CIP-Modelles. Die Dokumentation entspricht einem automatisch generierten System-Report des CIP Tool.

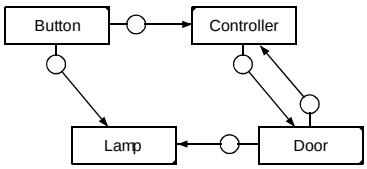
Vollständiges CIP-Modell

```
CHANNEL E_Button
        MESSAGE Push, Release
CHANNEL E_Door
        MESSAGE Opened, NotOpened, Closed, NotClosed
CHANNEL  A_Door
        MESSAGE MotOpen, MotClose, MotOff
CHANNEL A_Lamp
        MESSAGE Bright, Medium, Dark
```

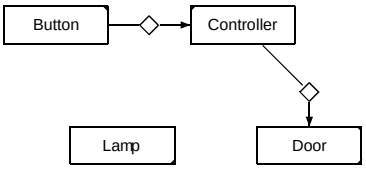
COMMUNICATION NET
EventActionChannels



PULSE CAST NET



INSPECTION NET




```
PULSE TRANSLATIONS
SENDER Button
        push -> Controller.push, Lamp.push
        release -> Controller.release , Lamp.release
SENDER Controller
        doClose -> Door.doClose
        doOpen -> Door.doOpen
SENDER Door
        isClosed -> Lamp.isClosed
        isOpened -> Controller.isOpened
```

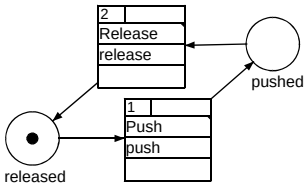
Die Namen der Inpulse eines Empfängers sind gleich gewählt worden wie die Namen der verknüpften Outpulse des entsprechenden Senders.

```
TRUTH TABLES
INSPECTOR Controller
        RESPONDER Button
        GATE ButtonIsReleased
        TRUE   <-  released
INSPECTOR Door
        RESPONDER Controller
        GATE toBeReopened
        TRUE   <- opening
```

In der Dokumentation der Truth Tables sind diejenigen Zustände des inspizierten Prozesses aufgelistet, die den Wert TRUE erzeugen. Die nicht aufgelisteten Zustände erzeugen FALSE.

PROCESS Button

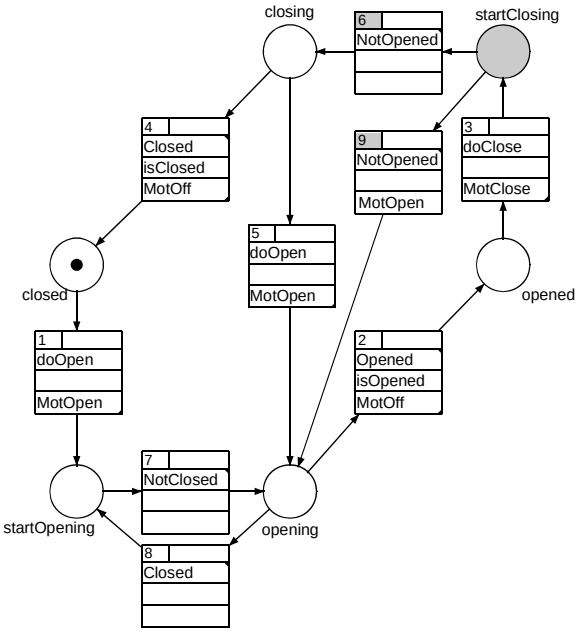
MODE normal



PROCESS Door

GATE toBeReopened

MODE normal



SWITCH

STATE startClosing MESSAGE NotOpened

TRANSITION 9 / toBeReopened

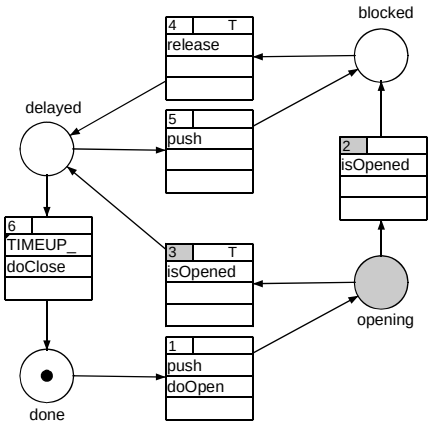
TRANSITION 7 / ELSE

PROCESS Controller

GATE ButtonIsReleased

DELAY openDelay 30

MODE normal



SWITCH

STATE opening INPULSE isOpened

TRANSITION 3 / GATE ButtonIsReleased

TRANSITION 2 / ELSE

TRANSITION ASSIGNEMENTS

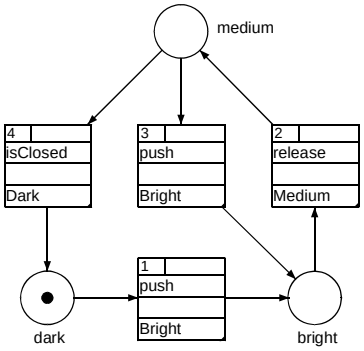
TRANSITION 3, 4

SET TIMER: openDelay

DELAY openDelay 30

PROCESS Lamp

MODE normal



Wie bereits bei der Erstellung der Agenten festgestellt wurde, ist der Prozess *Door* im Zustand *start-Closing* bezüglich der Message *NotOpened* nicht-deterministisch. Damit die Ausführung eindeutig wird, sind in einem SWITCH den entsprechenden Transitionen alternative Bedingungen zugeordnet worden (*GATE* toBeOpened und die komplementäre Standardbedingung ELSE). Der Wert des Gate ist durch Zustandsinspektion des *Controller* bestimmt (siehe Truth Table beim Inspection Net).

Desgleichen ist der Prozess *Controller* im Zustand *opening* bezüglich des Inpulses *isOpened* nicht-deterministisch. Der Non-Determinismus wird über das GATE ButtonIsReleased durch Inspektion des *Button* Prozesses aufgelöst.

In den Transitionen 3 und 4 setzt der Controller mit openDelay einen Timer.

4.4 Verhaltensmuster

Der Begriff *steuerbarer Prozess* stammt aus der Regelungstechnik. Solche Prozesse haben ein aktives Verhalten, das aber ständig von aussen beeinflusst werden kann. Die Werte der kinematisch aktiven Freiheitsgrade werden kontinuierlich oder quasi-kontinuierlich mittels Sensoren durch einem Regler überwacht. Aufgrund der Geschichte der erfassten Grössen können wiederholt neue Stellgrössen berechnet werden, um gezielt über Aktoren auf die überwachten Freitsgrade einzuwirken.

Diskrete steuerbare Prozesse sind ähnlich gebaut. Einer oder mehrere aktive Freiheitsgrade werden mittels diskreter Sensoren überwacht. So kann zum Beispiel erfasst werden, in welchem Bereich der Wert einer kontinuierlichen Grösse liegt. Bei unserer Türe war das die Position, die im Bereich *Offen*, *Geschlossen* oder *dazwischen* liegen kann. Das Wechseln eines Bereiches wird als diskretes Ereignis definiert. Um den Prozess zu beeinflussen, werden über diskrete Aktoren “Kraft”-Grössen verändert, die eine bestimmte dynamische Wirkung auf den Prozess haben. Typische Aktionen bei der Türe sind Veränderungen der Motoreinstellung.

Das Modell eines steuerbaren Prozesses lässt sich oft systematisch durch ein 2-dimensional-strukturiertes Zustandsdiagramm darstellen (bei n-dimensionalen Strukturen mit n > 2 können meistens Teilprozesse gebildet werden). Eine Dimension zeigt die möglichen Veränderungen des kinematisch aktiven Freiheitsgrades, die andere die Veränderungen der einwirkenden Grösse.

Beispiel: Kessel mit Schalter

Das Beispiel zeigt die typische 2-dimensionale Struktur eines steuerbaren Kessels.

Funktionale Anforderungen

Ein Kessel soll durch Steuern eines Zufluss- und eines Abflussventiles gefüllt und geleert werden. Der volle und der leere Zustand des Kessels kann durch zwei elektronisch überwachte Schwimmer erfasst werden.

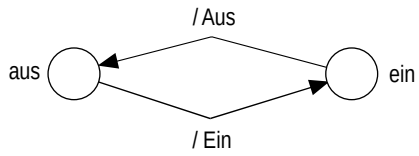
Das Füllen soll mit einem Schalter gestartet werden können. Wenn der Kessel voll ist, muss jeweils eine bestimmte Zeit gewartet werden, bis er wieder geleert werden kann. Wenn der Kessel gefüllt wird, soll der Vorgang jederzeit durch Ausschalten unterbrochen werden können. Wird der Kessel leer, so soll er sofort wieder gefüllt werden, ausser es sei beim Leeren ausgeschaltet worden.

Prozessmodelle und Prozessagenten

Für die Erstellung des Kontextmodells (Prozessagenten) muss die angeforderte Funktionalität des Embedded Systems bekannt sein, obwohl diese Funktionalität bei der Modellierung der Prozessagenten erst teilweise (lokale Reaktionen) realisiert wird. Die Prozessagenten schränken durch ihre Transitionsstrukturen die gültigen Sequenzen von Ereignissen und Aktionen ein.

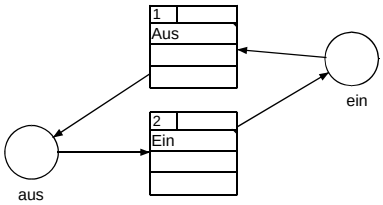
Kessel und Schalter sind vorerst unabhängige Anlagteile.

Prozessmodell Schalter



Events
Ein, Aus

Prozessagent Schalter



INPORT SchalterEvt
MESSAGES Ein, Aus

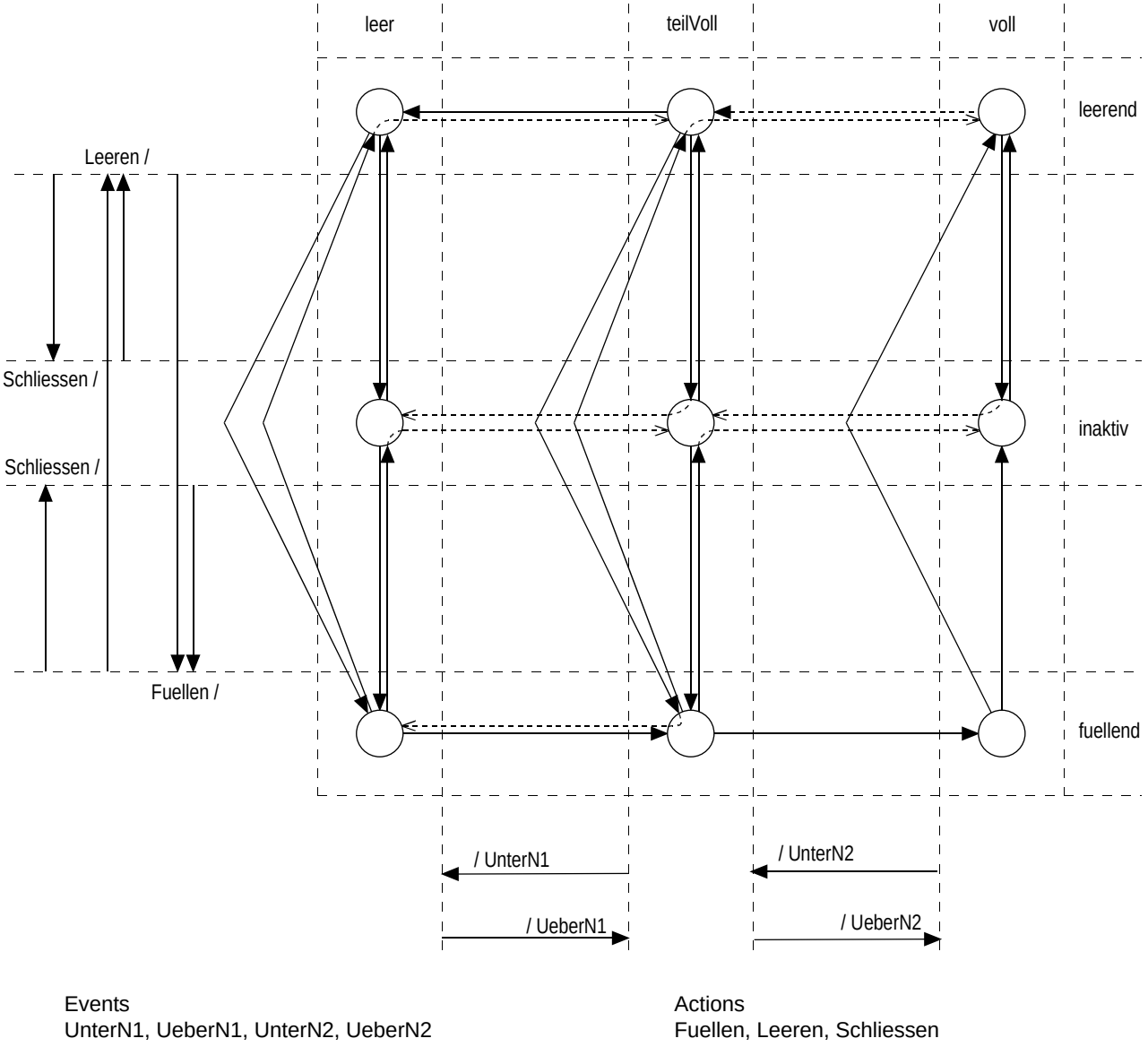
Bedienungselemente wie Schalter und Knöpfe haben ein einfaches Standardverhalten (sonst sind sie nicht brauchbar). Wir modellieren den realen Schalter auch als Zustandsmaschine. In der Praxis wird man sich mit der Spezifikation des Prozessagenten begnügen.

Der Kessel ist ein steuerbarer Anlageteil. Das über Sensorsignale erfassbare Flüssigkeitsniveau kann durch die Steuerung der Ventile beeinflusst werden.

Es wird angenommen, dass bei vollem Kessel die Ventile nie auf *Fuellen,* und bei leerem Kessel die ventile nie auf *Leeren* gestellt sein sollen.

Prozessmodell Kessel

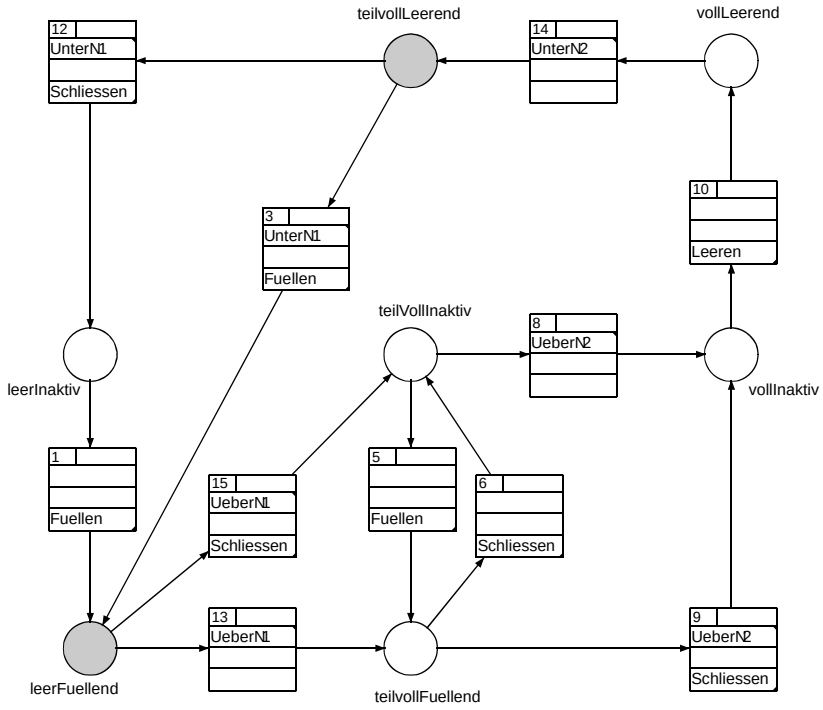
Die allgemeine Transitionsstruktur des Kesselmodells entsteht aus der Kombination einer aktiven Transitionsstruktur für die Kesselniveauzustände (*leer. teilVoll, voll*) und einer passiven für die Ventilzustände.(*leerend, inaktiv, füllend*). Die Struktur erinnert an ein Phasendiagram aus der Mechanik.



Weil die durch eine initiierte Aktion ausgelöste Wirkung eine kleine, endliche Zeit benötigt (Trägheit der Ventile), können im neuen Kesselzustand während einer kurzen Zeit noch Ereignisse auftreten, die eigentlich nur im alten Zustand erwartet werden. Solche Zustandsübergänge sind in der Grafik durch gestrichelte Pfeile dargestellt.

> *Beispiel*: Wenn der *teilvolle,* sich *leerende* Kessel (Zustand *teilvollLeerend*) auf *Fuellen* gestellt wird, kann während einer kurzen Zeit noch das Ereignis *UnterN1* auftreten.

Prozessagent Kessel



INPORT KessselEvt
MESSAGES UnterN1, UeberN1, UnterN2, UebeerN2

OUTPORT KesselActl
MESSAGES Fuellen, Leeren, Schliessen

Die aus dem Zustand *teilvollFuellend* erzeugte Aktion *Schliessen* wird als kritisch bezeichnet, weil anschliessend das Ereignis *UeberN2* nicht mehr zu erwarten ist, aber aufgrund der Ventilträgheit trotzdem noch auftreten kann.

Soll das *Füllen* bereits im Zustand *leerFuellend* unterbrochen werden, wird mit dem Schliessen der Ventile gewartet, bis das Ereignis *UeberN1* auftritt (synchrone Reaktion ist unkritisch).

Trägheit der virtuellen Interaktion. Die Problematik solcher „Race Conditions“ besteht auch, wenn keine physikalische Trägheit vorhanden ist, weil die Übertragungsdauer von Ereignis- und Aktionsmeldungen grösser als null ist (Propagationszeit der virtuellen Interaktion).

Zur Kritischen Initiierung von Aktionen

> Eine in einer bestimmten Transition erfolgte Aktionsinitiierung heisst *kritisch*, wenn im neuen Zustand noch mit einer Ereignismeldung gerechnet werden muss, die aufgrund der Wirkung der Aktion eigentlich nicht mehr zu erwarten wäre.

Das typische Szenario einer kritischen Aktionsinitiierung ist leicht zu erkennen:

> Wenn eine Aktionsinitiierung in einer Transition erfolgt, die nicht durch eine Ereignismeldung getriggert wird, ist diese Aktion möglicherweise kritisch.

Der Grund für das Problem liegt darin, dass eine solche Transition im vervollständigten Modell durch einen Puls getriggert wird, der aufgrund einer Ereignismeldung eines anderen Prozessagenten erzeugt worden ist. Damit wird die Aktion *asynchron* zu den eigenen Ereignismeldungen initiiert.

Mögliche Behandlungen kritischer Aktionsinitiierungen

- *Strukturerweiterung* des Prozessagenten
- *Warten* mit der Aktion, bis das erwartete Ereignis auftritt. Synchrone Initiierung ist unkritisch
- *Technische Tricks*, z.B. durchdachte Positionierung der Sensoren
- *Triviales Exception Handling*: Ignorieren des möglicherweise auftretenden, ungültigen Ereignisses

4.5 Automatische Implementierung von CIP-Modellen

Ein CIP-Modell spezifiziert das funktionale reaktive Verhalten einer oder mehrerer nebenläufiger *Embedded Units*. Eine Embedded Unit besteht aus einer funktionalen Steuerkomponente (*Reactive Machine*) und den Komponenten der *Embedded Interaktion*, welche die virtuelle Interaktion implementieren und so die Steuerkomponente in der realen Umgebung einbetten (Kapitel 5 bis 7).

Der Source-Code wird bei CIP-Modellen automatisch erzeugt (C, C++, Java). Die Codemodule für eine Steuerkomponente werden mit dem Begriff *CIP Unit* zusammengefasst. Eine *CIP Unit* besteht immer aus einer *CIP Shell* (Komponentenschnittstelle) und einer *CIP Machine* (Module des Komponentenkörpers).

4.5.1 CIP UNITS

Für die Bildung der *CIP Units* wird ein CIP-Modell in Clustergruppen aufgeteilt. Jede Clustergruppe modelliert als *CIP Unit* das reaktive Verhalten einer Steuerkomponente. Der ausführbare Code für *CIP Unit* implementiert eine reaktive Maschine und wird durch das CIP Tool automatisch erzeugt.

Ausgangspunkt für die Implementierung eines Modells ist die Festlegung Schnittstellen zu den Komponenten der *Embedded Interaktion*. Bei CIP-Modellen sind diese Schnittstellen durch eine Menge von Input- und Output-Kanälen bestimmt. Diese Kanalschnittstellen für eine *CIP Unit* bilden die *CIP Shell*. Mit dem Codegenerator des CIP Tool lässt sich aus einer spezifizierten *CIP Shell* bereits das Interface der reaktiven Maschine generieren (structs of function pointers), unabhängig von den Clustern, welche deren Verhalten modellieren.

Bei der Spezifikation der *CIP Shell* einer *CIP Unit* ist aufgrund der festgelegten externen Ereignisse und Aktionen bereits bestimmt, welche *Source* und *Sink Channels* zur *CIP Shell* gehören. Neben diesen Kanälen, welche die virtuelle Interaktion zu den Technischen Prozessen spezifizieren, sind fast immer weitere Kanäle notwendig, die Verbindungen zu andern CIP Units herstellen.

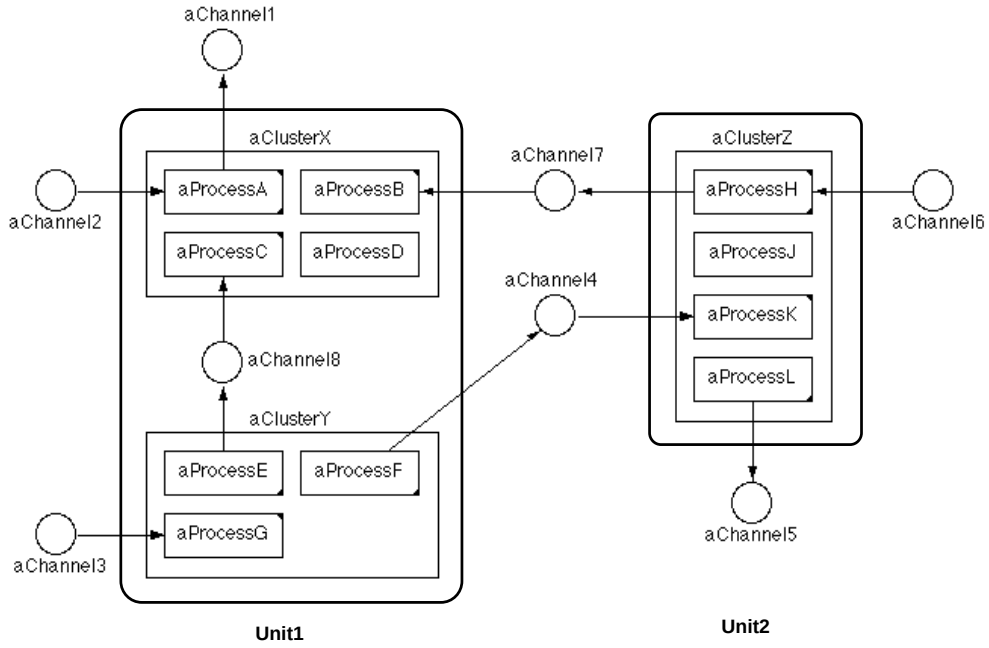
Für Spezifikation der *CIP Machine* einer *CIP Unit* muss lediglich die entsprechende *Clustergruppe* definiert werden, Es ist Aufgabe des Tools dafür zu sorgen, dass für eine *CIP Machine* nur dann Code generiert werden kann, wenn die *CIP Machine* zur entsprechenden *CIP Shell* passt. (Das Tool stellt zwar einen Dienst zur Verfügung (adapt CIP Shell), mit dem auf Wunsch eine CIP Shell an die aktuelle CIP Machine angepasst wird, was aber zu unerwünschten Nebeneffekten auf der andern Seite der implementierten Verbindung führen kann!

Es ist typisch für einen komponentenorientierten Softwareentwicklungsprozess, das die Festlegung der Schnittstellen Vorrang gegenüber der Konstruktion der Komponenten hat. Weil eine Komponentenverbindung immer mindestens zwei Komponenten betrifft, haben auch mindestens zwei Teilprozesse des Entwicklungsprozesses mit der Verbindung zu tun. Aus diesem Grund sorgt man dafür, dass die Spezifikation der Verbindungen im Entwicklungsprozess stabiler sind als die zu verbindenden Komponenten. Die Spezifikation einer *CIP Shell* kann im Tool gesperrt werden (lock shell), um so die erwähnte Stabilität im Entwicklungsprozess zu erhalten.

Spezifikationsbeispiel für die Implementation eines CIP-Modells

In der folgenden *Abbildung 44* ist dargestellt, wie ein CIP-Modell, das aus drei Clustern besteht, in die CIP Units *Unit1* und *Unit2* aufgeteilt worden ist.

Bemerkung:
sUnit1 und *mUnit1* sind Standardnamen für die CIP Shell und die CIP Machine der CIP Unit *Unit1* (wird vom Tool vergeben). Diese Namen werden für das generierte Schnittstellenmodul *sUnit1.c* und das Maschinen-Modul der CIP Machine *mUnit1.c* verwendet.




```
IMPLEMENTATION anImplementation
    CIP UNIT Unit1
        CIP SHELL sUnit1
            INPUT CHANNELS
                Channel2, Channel3, Channel7
            OUTPUT CHANNELS
                Channel1, aChannel4
        CIP MACHINE mUnit1
            CLUSTERS aClusterX, aClusterY
    CIP UNIT Unit2
        CIP SHELL sUnit2
            INPUT CHANNELS
                Channel4, Channel6,
            OUTPUT CHANNELS
                Channel5, Channel7
        CIP MACHINE mUnit2
            CLUSTERS aClusterZ
```

Abb. 44: Aufteilung von drei Clustern in zwei CIP Units

CIP Shell

Die *CIP Shell* einer CIP Unit modelliert deren Schnittstelle durch eine von Menge Kanälen. Die Meldungen der Kanäle definieren die Datenstruktur der Schnittstelle unabhängig von der Spezifikation der entsprechenden CIP Machine. Aus der Spezifikation der CIP Shell wird vom *CIP Tool* der Code für entsprechende Schnittstellenobjekte generiert. Bei der C-Code-Generierung sind das Verbunde (struct) von Funktionspointern, bei C++ und Java entsprechende Object Interfaces.

CIP Machine

Die CIP Machine einer CIP Unit besteht aus einem oder mehreren Clustern. Für eine Clustergruppe wird von *CIP Tool* C-Code für eine CIP Machine generiert. Jeder Cluster kann nur in einer CIP Unit einer Implementation enthalten sein.

Die Kanäle einer CIP Machine werden folgendermassen klassiert:

Interne Kanäle

Als *intern* werden diejenigen Kanäle einer CIP Unit bezeichnet, für welche alle Schreib- und Leseprozesse in der entsprechenden Clustergruppe enthalten sind. Interne Kanäle können als Teil der spezifizierten CIP Machine betrachtet werden. Für interne Kanäle wird der Code mit dem Code für die CIP Machine generiert. Im obigen Beispiel ist *aChannel8* ein interner Kanal.

Interne Kanäle können optional auch vom Anwender implementiert werden. Ein solcher Kanal wird bei der Codegenerierung wie ein externer Kanal behandelt, das heisst er wird in derselben CIP Shell sowohl als Input- als auch als Outputkanal auftreten.

Externe Kanäle

Alle nicht internen Kanäle werden als *extern* bezeichnet. Das sind alle Quellen und Senken, sowie die Kanäle, die eine Verbindung zu anderen CIP Units spezifizieren.

Schnittstellenkonsistenz einer CIP Machine

Alle externen Kanäle einer CIP Machine müssen in der entsprechenden CIP Shell enthalten sein. Eine CIP Shell kann aber auch Kanäle enthalten, die von der entsprechenden CIP Machine nicht benötigt werden, was in gewissen Entwicklungsphasen sinnvoll sein kann.

4.5.2 Erzeugter C-Code

Pro CIP Unit wird vom Codegenerator von *CIP Tool* in getrennten Generierungsschritten ANSI-C-Code für die CIP Shell und für die entsprechende CIP Machine erzeugt (*Abbildung 45*).

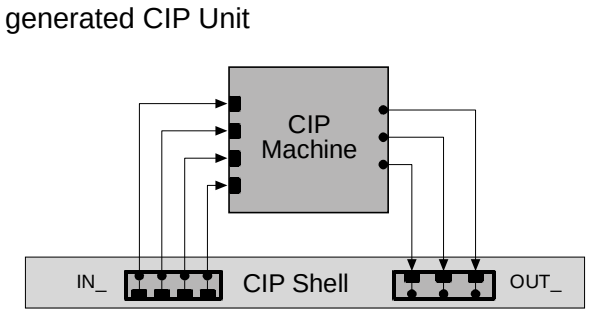


Abb. 45: CIP Shell und CIP Machine Code

Der generierte C-Code für eine CIP Shell besteht aus einer Kompilationseinheit, welche die Schnittstellenobjekte IN_ und OUT_ enthält. IN_ ist das *Provided Interface* (Ereignismeldungen) und OUT_ das *Required Interface* (Aktionsinitiierungen) der Komponente. Zusätzlich können Schnittstellenobjekte für das Triggern von Model Extensions, Trace-Funktionen und die Anbindung von Fehlerfunktionen generiert werden.

Die Code für die CIP Shell besteht aus Verbunden (struct) von Funktionspointern, die erst zur Laufzeit des Systems initialisiert werden. Bei der Codegenerierung kann ab CIP Tool 5 eine von zwei Varianten für die Form der Schnittstellenobjekte gewählt werden (Code Options):

Channel Interface - eine Schnittstellenfunktion mit Message-Parametern pro externem Kanal

Message Interface - eine Schnittstellenfunktion pro Message der externen Kanäle

Der Code für die CIP Machine besteht aus einer Reihe von Kompilationseinheiten. Neben dem Top-Modul für die CIP Machine wird für jeden Prozess ein separates C-Modul erzeugt (Prozessname.c). Der Code für die Fehler-Schnittstelle wird mit dem Code für die CIP Machine generiert.

Die detaillierte Beschreibung des erzeugten Codes ist im entsprechenden CIP Tool Handbuch zu finden. Zudem enthält der Anhang des *Embedded System 1* Kurses zwei Implementierungsbeispiele.

4.5.3 Der Betrieb einer CIP Machine (Embedded Interaction)

Eine CIP Unit muss mit den gesteuerten Technischen Prozessen verbunden werden, d. h. die mittels Kanälen spezifizierte virtuelle Interaktion muss implementiert werden. Diese Arbeit ist stark durch die an der Prozessperipherie angewendete Interface-Technologie bestimmt und stellt oft ein schwieriges Problem für sich dar. Die Aufteilung eines CIP-Modells in verschiedene CIP Units verlangt aber auch die externe Implementierung derjenigen Kanäle, die Prozesse verschiedener Units verbinden. Diese Aufgabe kann jedoch meistens mit standardisierten Kommunikationstechniken gelöst werden (Task-Kommunikation, Feldbusse, serielle Kommunikation, Sockets).

Aufgrund der in CIP-Modellen synchron modellierten Kooperation von Clustern und Prozessen entfällt die übliche Implementierung pseudo-paralleler funktionaler Ausführungseinheiten (Multitasking) vollständig. Oft wird pro Prozessor nur eine Unit implementiert. Task Scheduling und Interrupt Handling wird in diesem Fall auf Hardware-Schnittstellenfunktionen und mögliche Hintergrunddienste beschränkt.

Der domänenorientierte Entwicklungsprozess erlaubt wie bereits erwähnt die Schnittstellen der CIP-Modelle zu einem sehr frühen Zeitpunkt festzulegen und über längere Entwicklungsperioden zu sperren (locking). Der generierte CIP Shell Code bekommt damit eine Bedeutung als stabilisierbares Verbindungselement zwischen den Einbettungsmodulen. Das Konzept ist erfolgreich in industriellen Projekten angewendet worden. In unterschiedlichen Validierungsphasen wurden verschiedene Aufteilungen desselben CIP Modells mit Simulationsmodellen, Testumgebungen, Prüfständen oder der realen Prozessumgebung verbunden.

Ausführung einer CIP Machine

Der Code für eine CIP Machine ist ein passives reaktives Datenobjekt, das ständig von der aktiven Einbettungssoftware (Embedded Interaction) getriggert wird (*Abbildung 46*).

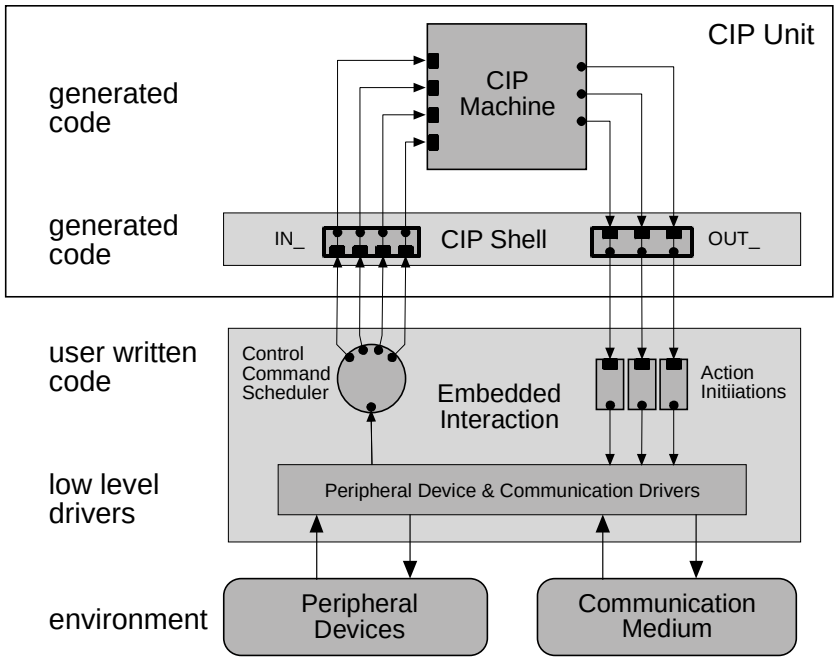


Abb. 46: CIP Unit und Einbettungssoftware

Die iterative Aktivierung der CIP Machine erfolgt typischerweise durch einen Scheduler (Event Monitor) für Ereignismeldungen der Embedded Unit. Die Aufrufe der entsprechenden Steuerfunktionen gehen immer über das Input-Interface der CIP Shell. Pro Aktivierung der CIP Machine wird maximal eine Clustertransition ausgeführt. Die durch die reagierende CIP Machine erzeugten Initiierungen von Aktionen werden über das Output-Interface der CIP Shell aktiviert. Die so aufgerufenen

Callbacks der Einbettungssoftware greifen entweder direkt auf die Hardwareschnittstellen zu, oder es werden Aufträge für die entsprechenden Output-Handler erzeugt.

Interne Standard Funktionen einer CIP Machine

Um die CIP Machine für interne Verarbeitungen (Model Extensions) zu triggern, werden über eine eigene Schnittstelle der CIP Shell Standard-Funktionen der CIP Machine aufgerufen:

Mit `CHAIN_()`, `TIMEUP_()` oder `AUTO_()` werden die Trigger für die entsprechende Model-Extensions aktiviert.

Mit dem `TICK_()`-Aufruf wird die Zeit der CIP Machine um einen Tick erhöht

Mit `READ_()` wird eine Message aus dem generierten Puffer für interne Kanäle abgeholt.

Fehlerfunktionen

Zur Laufzeit detektierte Kontextfehler (nicht erwartete aufgetretene Ereignisse, Protokollfehler) führen zum Aufruf der ContextError-Funktion, falls eine entsprechende Option für die Codegenerierung gesetzt wurde. Fehlerfunktionen werden in der Einbettungssoftware als Callback-Funktionen implementiert.

Der C-Code für die Fehler-Schnittstelle wird mit dem Code für die CIP Machine generiert. Die Files für die Fehler-Schnittstelle der CIP Unit *Unit1* bestehen aus der Kompilationseinheit *eUnit1.c*, in welcher das Schnittstellenobjekt definiert ist und einem Headerfile *eUnit.h* mit den Typendefinitionen:

Initialisierung

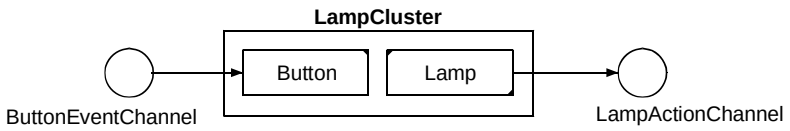
Das generierte ‚CIP Machine'- Modul *mUnit1.c* enthält die Initialisierungsfunktion `fINIT_()`, die von der Einbettungssoftware aufgerufen werden muss, um die CIP Machine zu „booten". Die Initialisierungsfunktion initialisiert die CIP Machine und die Input-Funktionspointer der CIP Shell.

Anschliessend werden die externen Initialisierungsfunktionen `iCHAN_()` bzw. `iMSG_()` und `iERROR_()` aufgerufen. Diese Initialisierungsfunktionen müssen die von der Einbettungssoftware zur Verfügung gestellten Kanal- und Fehler-Outputfunktionen den entsprechenden Funktionspointern der CIP Shell zuweisen. Die Initialisierungsfunktion überprüft anschliessend, ob alle Output- und Fehler-Funktionspointer initialisiert worden sind. Bei erfolgreicher Initialisierung gibt `fINIT_` 1 zurück, andernfalls 0. Bei ungültiger Initialisierung reagiert die CIP Machine bei jeder Aktivierung mit dem Input-Fehler `UNUSED_CHANNEL_`.

4.5.4 LampSys – Ein einfaches Beispiel einer Implementation

SYSTEM LampSys

COMMUNICATION NET EventActionChannels



IMPLEMENTATION LampSysImplementation

UNIT LmpUnit

CIP SHELL sLmpUnit

CHANNEL ButtonEventChannel

MESSAGES Push, Release

CHANNEL LampActionChannel

MESSAGES Bright, Dark

CIP MACHINE mLmpUnit

CLUSTERS LampCluster

CIP Shell

Folgende Sourcefiles werden generiert:

sLmpUnit.c, sLmpUnit.h	*Channel and Standard Trigger Interface*

CIP Machine

Folgende Sourcefiles werden generiert:

mLmpUnit.c, mLmpUnit.h	*CIP Machine Module*
Button.c	*Process Module*
Lamp.c	*Process Module*

Einbettung

Die CIP Machine wird direkt mit Tastatur und Bildschirm betrieben. Folgende Aufgaben sind im File LmpCon.c (main) gelöst worden:

Definition der Funktionen für die Aktionsinitiierungen:

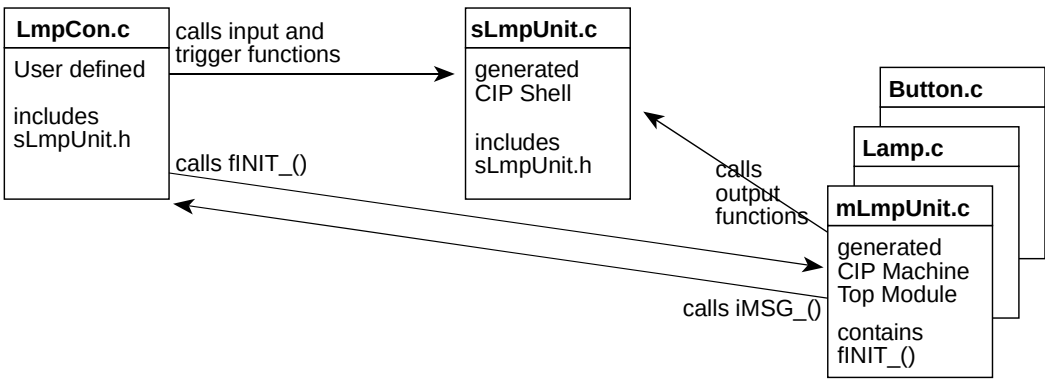
`printBright()` **und** `printDark()`

Definition der Initialisierungsfunktion der CIP Shell: `iMSG_()`

Initialisierung der CIP Machine: Aufruf von `fINIT_()`

Im Main Loop (EventMonitor) werden Eingabesymbole von der Tastatur gelesen und in entsprechende Aufrufe der CIP Machine umgesetzt.

Funktionsaufrufe zwischen Kompilationseinheiten



Implementation mit Message Interface (Code Option)

```
/*********************************************************************
   CIP SHELL sLmpUnit
   Filename: sLmpUnit.c
   generated by CIP Tool(R) Version 4.19.00
*********************************************************************/

/* Include Files */
#include "sLmpUnit.h"

/* Input Interface */
struct tIN_ IN_;

/* Output Interface */
struct tOUT_ OUT_;

/* Trigger Interface  */
struct tTRG_ TRG_;

/*********************************************************************
   Header File for CIP SHELL sLmpUnit
   Filename: sLmpUnit.h
   generated by CIP Tool(R) Version 4.19.00
*********************************************************************/

/* Input Interface Type */
struct tIN_{
   struct{
      void (*Push)(void);
      void (*Release)(void);
   }ButtonEventChannel;
};

/* Output Interface Type */
struct tOUT_{
   struct{
      void (*Bright)(void);
      void (*Dark)(void);
   }LampActionChannel;
};

/* Trigger Interface Type */
struct tTRG_{
   void (*TICK_) (void);/* increment CIP MACHINE time */
   void (*STEP_) (void);/* trigger a pending CIP MACHINE transition */
   void (*CHAIN_) (void);/* trigger pending chain CIP MACHINE transition */
   void (*TIMEUP_) (void);/* trigger pending timeup CIP MACHINE transition */
   void (*READ_) (void);/* trigger CIP MACHINE to read internal msg buffer */
};

/* Interface Declarations */
extern struct tIN_ IN_;          /* Input Channel Interface */
extern struct tOUT_ OUT_;        /* Output Channel Interface */
extern struct tTRG_ TRG_;        /* Trigger Interface */

/* Unit Initialization Function, called by User to initialize CIP Machine*/
int fINIT_(void);

/* User defined Initialization Function for CIP Shell Initialization */
void iMSG_(void);
```

```
/*************************************************************************
  Main Modul: Embedded Interaction for CIP Model LampSys - Filename: LmpDrive.c
*************************************************************************/

#include "sLmpUnit.h"
#include <stdio.h>

/* Local Function Prototypes */
void printBright (void);
void printDark (void);


/*  Output Message Functions (Action Initiations) */
void printBright (void){
    printf("   Lamp is on\n\n");
}

void printDark (void){
   printf("   Lamp is off\n\n");
}


/* Shell Initialization Function */
void iMSG_(void){
    OUT_.LampActionChannel.Bright = printBright;
    OUT_.LampActionChannel.Dark = printDark;
}



/* Interaction Loop */
int main(){
   int active;
   char InputSymbol;
   active = fINIT_();
   if (!active){
      printf("\nInitialization failed\n");
   }
   else{
      while (active){
         printf("Enter P (Push), R (Release), T (TICK), Q (Quit)\n" );
         scanf("%1s", &InputSymbol);
         if (InputSymbol == 'P'){
            IN_.ButtonEventChannel.Push();
         }
         else if (InputSymbol == 'R'){
            IN_.ButtonEventChannel.Release();
         }
         else if (InputSymbol == 'T'){
            TRG_.TICK_(); TRG_.TIMEUP_();
         }
         else if (InputSymbol == 'Q' ){
            active = 0;
         }
         else printf
          ("\n   Invalid Input Symbol\n\n");
      }
   }
   return 1;
}
```