Modellierung mit CIRO

Eine UML-Erweiterung zur Modellierung reaktiver Systeme

Johannes Scheier, Zürcher Hochschule für angewandte Wissenschaften

Modellgetriebene Software-Entwicklung ist ein Ansatz mit enormem, nur teilweise ausgeschöpftem Potential. Je verständlicher und lesbarer die Modelle, desto grösser ist ihr Nutzen. UML (Unified Modelling Language) sieht die Möglichkeit vor, die Sprache durch eigene Abstraktionen zu erweitern. Nachfolgend wird CIRO (Communicating Interacting Reactive Objects), eine auf diesem Mechanismus basierende Erweiterung vorgestellt.

Modellgetriebene Software-Entwicklung ist eine Vorgehensweise zur Modellierung und anschliessenden Code-Generierung von Software-Systemen. Dabei wird ein formales, vollständiges und plattformunabhängiges Modell erstellt, und daraus der plattformspezifische Code generiert. Das Modell beschreibt die Struktur und das Verhalten des Systems.

CRIO ist eine Sammlung von Abstraktionen und Konzepten, die bei der Modellierung von Embedded Systemen hilfreich sind. CIRO definiert eine generische Software-Architektur, synchrone und asynchrone Kommunikationsmechanismen zwischen Objekten, sowie eine Systemstruktur als Komposition von wieder verwendbaren Komponenten. Eine fundamentale Rolle kommt dabei den qualifizierten Konnektoren zu, einem Konzept zur Definition der Interaktion von Komponenten innerhalb einer Komposition.

CIRO basiert auf CIP (Communicating Interacting Processes), einer Modellbasierten Entwicklungsmethodik für Embedded Systeme. CIP wurde 1999 an der ETH Zürich, von Prof. Dr. Hugo Fierz entwickelt [1].

Generische Software-Architektur

Als Ausgangspunkt für die Modellierung in CIRO dient die nachfolgend beschriebene generische Software-Architektur für Embedded Systeme (Abb. 1). Diese Architektur lässt sich auf Systeme anwenden, welche technische Prozesse, auf Grund von physikalischen Ereignissen steuern. Das Embedded System reagiert auf Ereignisse in den technischen Prozessen mit Aktionen, welche die technischen Prozesse beeinflussen. Sei ein technischer Prozess beispielsweise eine Heizung, dann reagiert das Embedded System auf das Ereignis *tempHigh* mit der Aktion *heatingOff*.

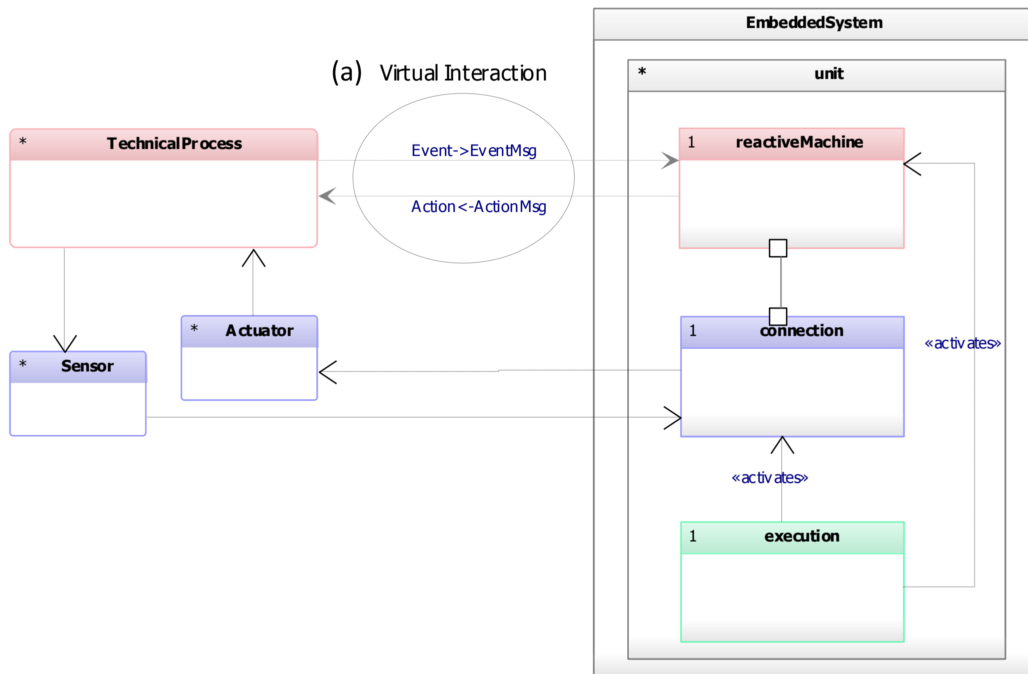

Abb. 1: Generische Software-Architektur

Wir stellen uns vor, dass die technischen Prozesse direkt mit der Reactive-Machine kommunizieren, und mit dieser, Event- und Action-Messages austauschen – diese vorgestellte Interaktion bezeichnen wir als virtuelle Interaktion (siehe (a) in Abb. 1). In Tat und Wahrheit wird diese Kommunikation in der Connection-Schicht des Embedded Systems implementiert, wobei der technische Prozess über Sensoren mit dem Embedded System verbunden ist. Aus den Signalen der Sensoren werden die Ereignisse detektiert und als Event-Messages über eine Message-Queue an die Reactive-Machine weitergereicht. Die Reactive-Machine, sowie die Connection sind passive Komponenten. Die Execution, quasi die main()-Funktion ist für den Aufruf dieser Komponenten verantwortlich.

Eine Unit ist eine ausführbare Einheit, welche parallel zu anderen Units ausgeführt werden kann. Eine Unit besteht aus Reactive-Machine, Connection und Execution. Mehrere Units können auf verschiedene Threads oder auch auf mehrere Prozessoren verteilt werden.

Die Reactive-Machine, reagiert auf Events in den technischen Prozessen und löst Actions an denselben aus.
Die Interaktion zwischen Reactive-Machine und technischen Prozessen erfolgt jedoch indirekt über die Connection, welche Ereignisse detektiert und als Event-Messages an die Reactive-Machine weiterleitet. Andererseits nimmt die Connection Action-Messages von der Reactive-Machine entgegen und leitet diese als elektrische Signale an die Aktuatoren weiter.

Die Reactive-Machine und die Connection sind passiv, und werden durch die Execution aktiviert.

Im Folgenden werden die Komponenten dieser Software-Architektur beschrieben. Als Beispiel dient ein Modell für die Steuerung einer Schiebetüre, welche im Rahmen eines Praktikums von den Studenten der ZHAW entwickelt wird (Abb. 2).

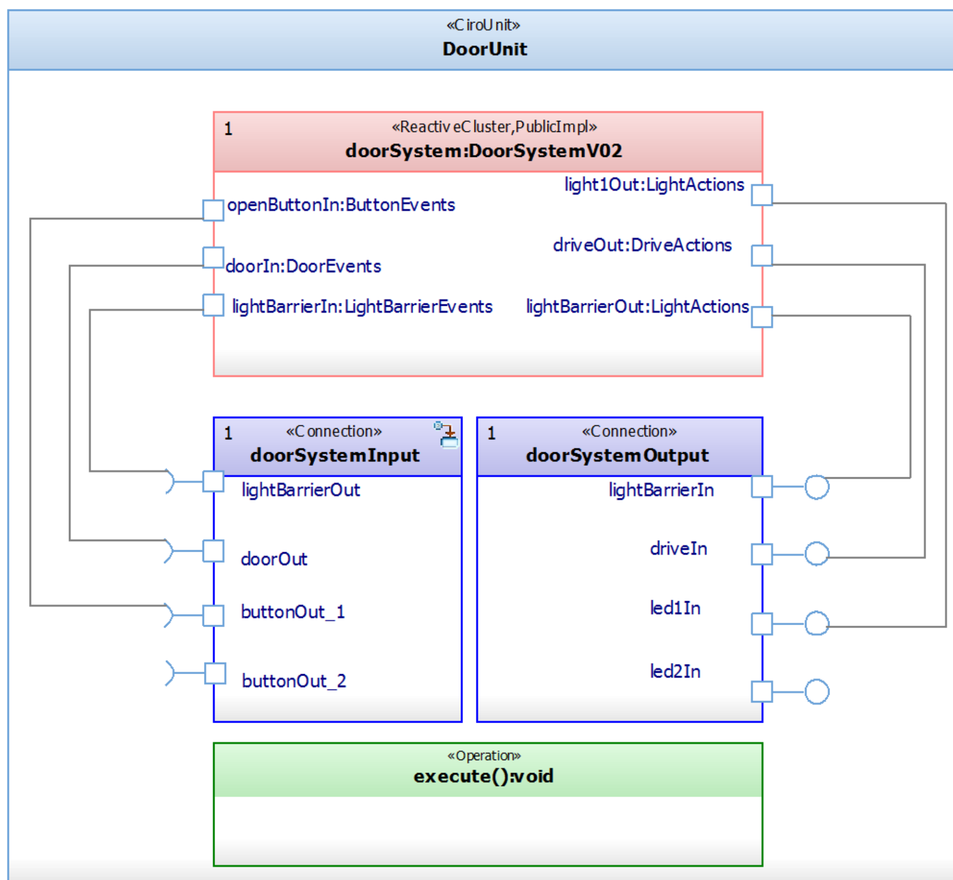

Abb. 2: Architektur Beispiel Schiebetürsteuerung

Reactive-Machine

Das Interface der Reactive-Machine lässt sich aus der virtuellen Interaktion ableiten. Es besteht aus Ports. Für jedes Port wird definiert, welche Messages empfangen oder gesendet werden können (Abb. 3).

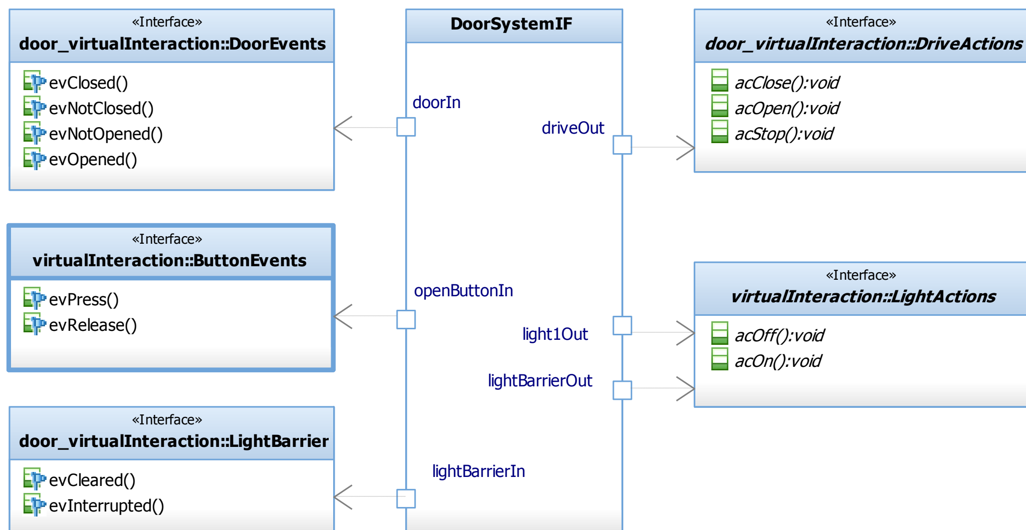

Abb. 3: Reactive-Machine Interface

Die Reactive-Machine selbst ist eine Komposition von **Reactive-Clusters**, welche ihrerseits aus Reactive-Objects bestehen. Über Reactive-Cluster-Grenzen kann nur asynchron über Event- und Action-Messages kommuniziert werden. Innerhalb der Cluster-Grenzen kann synchron über Pulse und State-Queries kommuniziert werden.

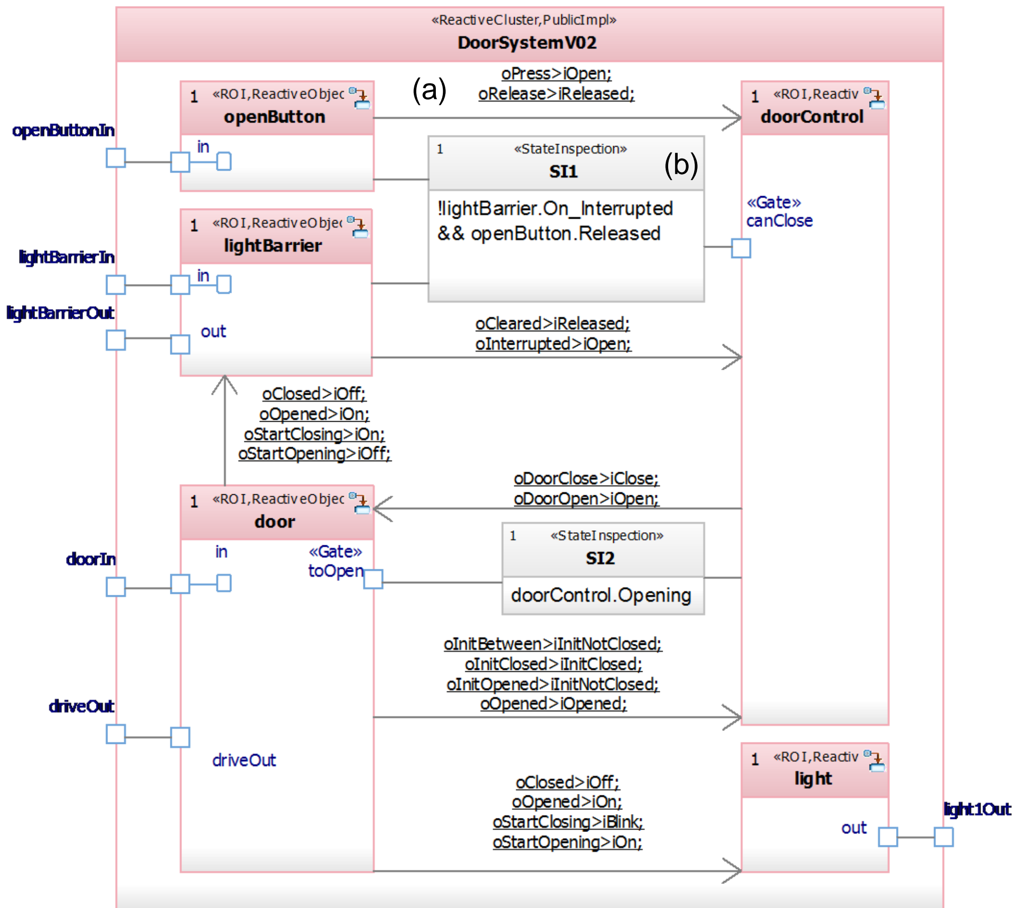

Abb. 4: Strukturdiagramm Reaktive-Machine

Reactive-Object (rote Elemente in Abb. 4)
Das Verhalten eines Reactive-Object ist in einer State-Machine definiert (Abb. 5). Jede State-Transistion hat einen Auslöser (Trigger) und kann einen Output (Action) erzeugen. Nebst dem Trigger, können Bedingungen (Guards) zu State-Transitions definiert werden. Als Trigger kommen In-Pulse und Event-Messages in Frage, als Actions, Out-Pulse und Action-Messages, und als Guard sogenannte Gates. Dabei gilt folgende Syntax: trigger[guard]/action (Abb. 5)

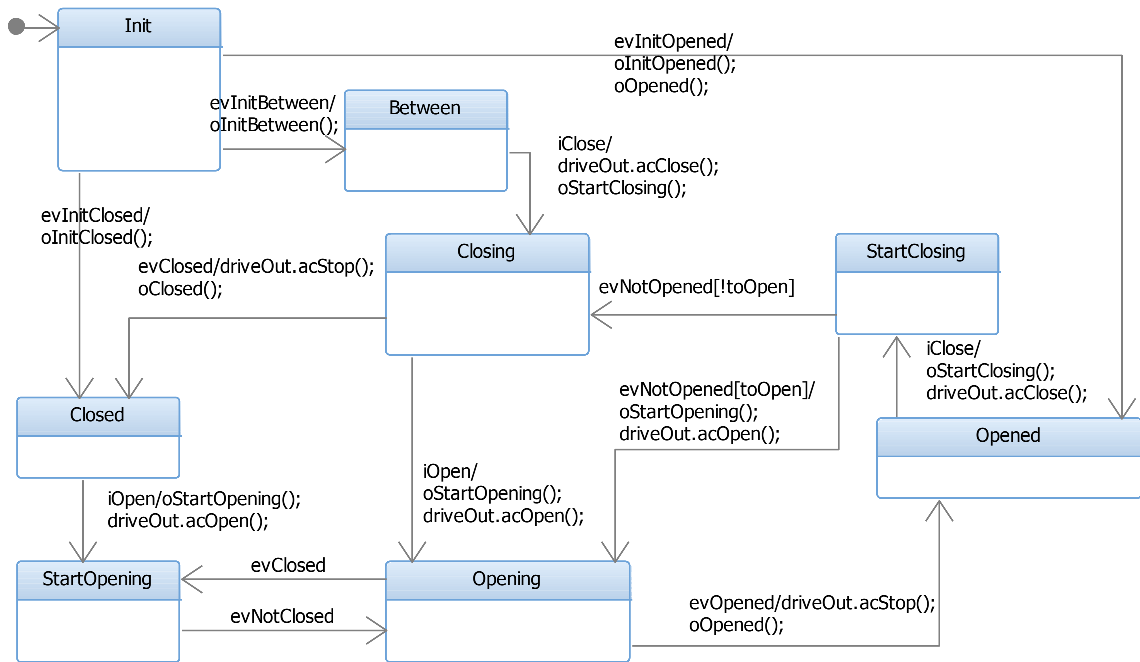

Abb. 5: State-Diagramm für Reactive-Object Door

Synchrone Interaktion von Reactive-Objects (graue Elemente in Abb. 4)
Innerhalb eines Reactive-Clusters können Reactive-Objects synchron miteinander interagieren. Die Mechanismen zur synchronen Interaktion sind Pulse-Cast und State-Inspection.

Ein **Pulse-Cast** ist eine Interaktion zwischen Reactive-Objects, wobei ein ausgesendeter Out-Pulse in In-Pulses von Empfängerobjekten übersetzt werden. So wird beispielsweise in ((a) Abb. 4) der Out-Pulse *oPress* von *button* in den In-Pulse *iOpen* von *doorControl* übersetzt.

State-Inspection ist eine Interaktion zwischen Reactive-Objects, bei der das inspizierende Objekt eine Gate-Operation aufruft, welche in eine logische Abfrage von Zuständen der inspizierten Objekte übersetzt wird. So ist beispielsweise der logische Ausdruck für *canClose: !lightBarrier.Interrupted && openButton.Released* ((b) in Abb. 4), d.h. die Türe kann nur geschlossen werden, wenn die Lichtschranke nicht unterbrochen ist, und der Öffnen-Knopf losgelassen ist.

Die Interaktionsdefinitionen sind den Verbindungen zwischen Objekten zugeordnet, den sogenannten **qualifizierten Konnektoren**. Diese sind, wie die Objekte selbst, Bestandteile der Komposition. Die interagierenden Objekte, bzw. deren Klassen, sind unabhängig voneinander. Dies führt zu einer grossen Flexibilität und Wiederverwendbarkeit.

Die **Connection** liest die Eingangssignale, detektiert Event-Messages und leitet diese an die Reaktive-Machine weiter. Das Modell der Connection ist wiederum eine hierarchische Struktur, deren Komponenten Instanzen wiederverwendbarer Klassen sind.

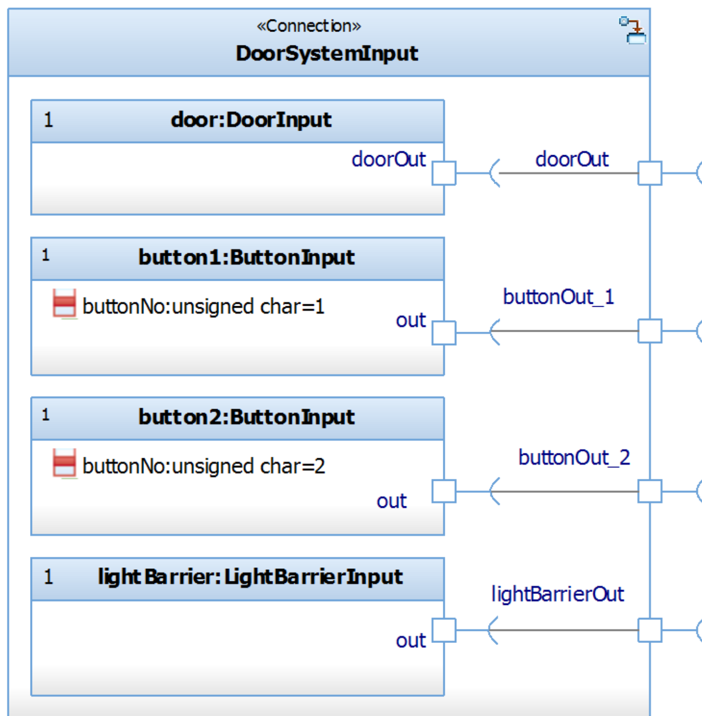

Abb. 6: Input-Connection für Schiebetüre

Execution ist für das Scheduling innerhalb der Unit verantwortlich. In der Regel ist das die main()- oder run()-Operation der Unit-Task, welche zyklisch die Connection und die Reactive-Machine aufruft (Abb. 7).

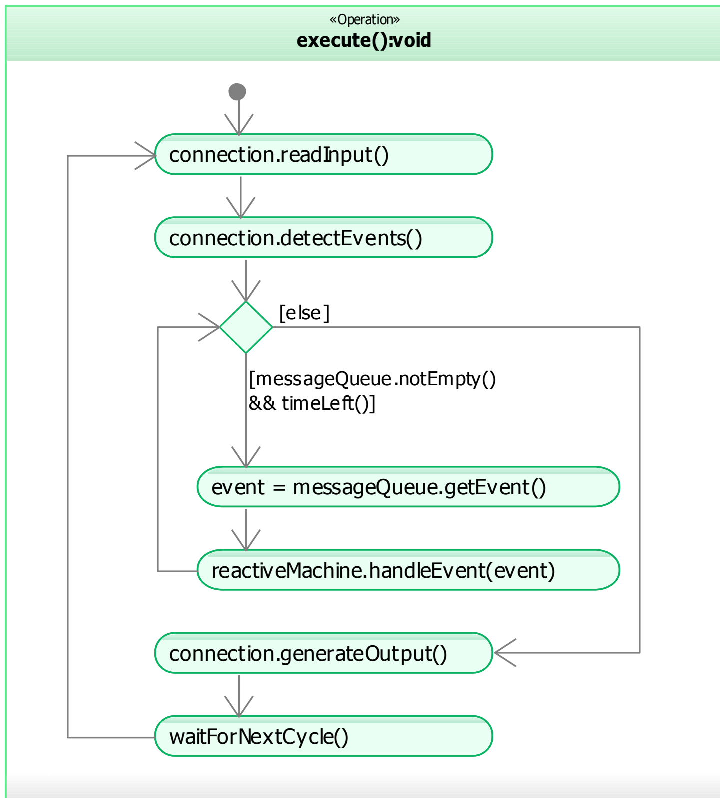

Abb. 7: Flow-Chart für Execution

Simulation und Testing

Das Modell der Reactive-Machine ist ausführbar, bzw. simulierbar, d.h. es können Event-Messages eingespeist und die entsprechenden Reaktionen verfolgt werden.

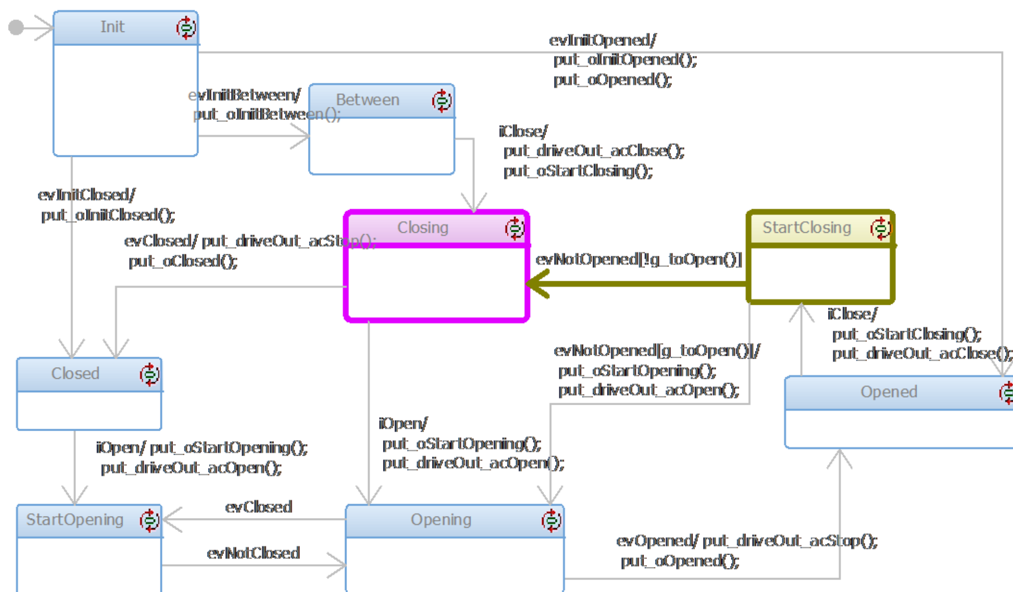

Abb. 8: Simulation des Modells

Sobald die funktionalen Anforderungen bekannt sind und das Interface zur Reactive-Machine definiert ist, können **Test-Cases** definiert werden, z.B. als Sequenz-Diagramme (Abb. 9). Aus Sicht der Test-Cases ist die Reactive-Machine eine Blackbox.

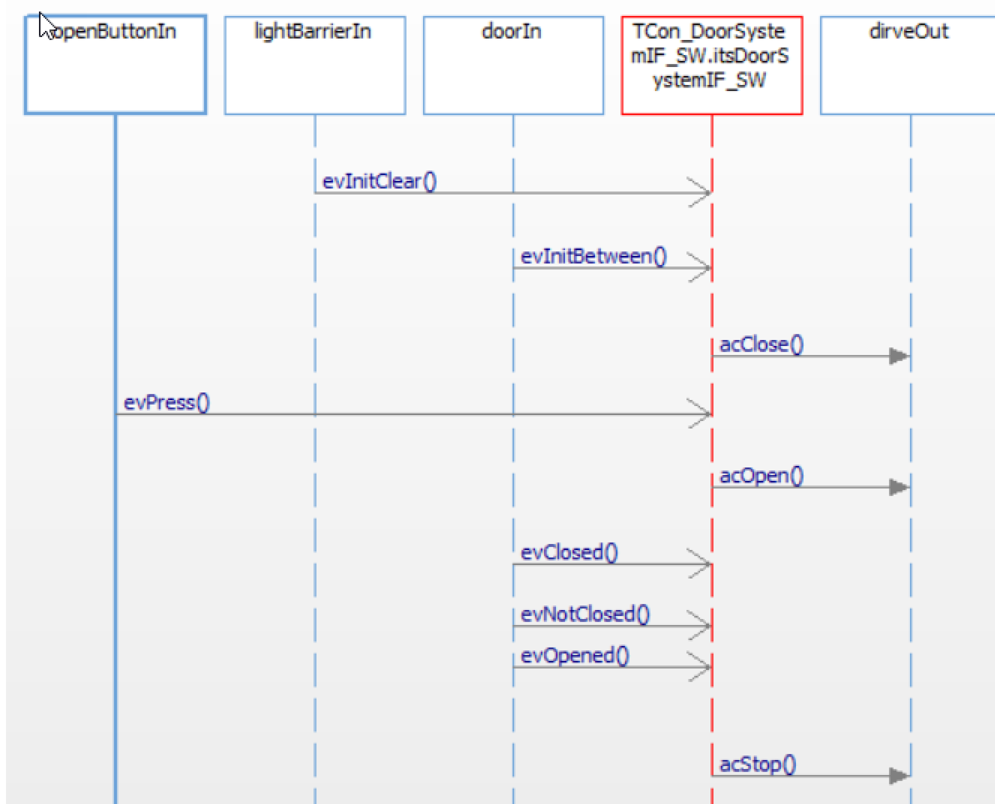

Abb. 9: Test-Case Definition als Sequenz-Diagramm

Code-Generierung
Der Code für auf dem Target ausführbaren die Units kann vollständig aus dem Modell generiert werden.

Qualifizierte Konnektoren
Die qualifizierten Konnektoren sind UML-Links, welche um zusätzliche Aussagen zur Verbindung erweitert werden, beispielsweise Pulse-Cast oder State-Inspection (siehe oben). Dieses Konzept lässt sich ganz allgemein auf jede Verbindung von Objekten innerhalb einer Komposition anwenden und erhöht die Flexibilität und Wiederverwendbarkeit enorm.

Zusammenfassung
CIRO ermöglicht mit wenigen, kleinen UML-Erweiterungen eine effektivere Anwendung von modellgetriebener Software-Entwicklung für Embedded Systeme. Diese Erweiterungen basieren zum grossen Teil auf dem allgemeineren Konzept der qualifizierten Konnektoren, welches auch in anderen Gebieten der Modellierung angewendet werden kann.
CIRO ist tool-unabhängig. Es existieren Realisierungen für ActifSource und IBM Rhapsody. Weitere sollten sich mit vertretbarem Aufwand implementieren lassen.

Literaturverzeichnis

[1] Fierz, Hugo. (1999). The CIP Method: Component- and Model-Based Construction of Embedded Systems. ACM Sigsoft Software Engineering Notes. 24. 375-392. 10.1007/3-540-48166-4_23.

Autor

Johannes Scheier, dipl. math. verfügt über mehr als 30-jährige Erfahrung in Software-Entwicklung, und mehr als 20-jährige Erfahrung in Entwicklung, Beratung und Schulung objektorientierter Modellierung und Modellgetriebener Softwareentwicklung. Seit 2011 doziert er an der Zürcher Hochschule für angewandte Wissenschaften (ZHAW) in Computertechnik und Embedded Software Engineering auf Bachelor- und Master-Level.