Dr. Hartmut Schorrig, www.vishia.org

Hinweis: Es handelt sich um eine Vorab-Fassung zum Thema, Verwertungsmöglichkeiten müssen noch geklärt werden.
Der Artikel ist vollständig von mir erstellt, Quellen sind angegeben.

Design und Programmierung von Statemaschinen – eine allgemeine Betrachtung

Statemaschinen sind mindestens bekannt seit Alan Turing den Code der Enigma-Verschlüsselung geknackt hat. Einen wesentlichen Beitrag für Software-Statemaschinen hat David Harel mit der Definition von parallelen und geschachtelten Software-Statemaschinen geliefert, die einen Standard oder Quasi-Standard darstellen. Wesentliche Fragen sind auch Trigger vs. Event.

Dennoch werden Statemaschinen teils verschieden angewendet oder dargestellt. Dieser Artikel soll zu einer einheitlichen Sichtweise zwischen Hardware- und verschiedenen Software-Lösungen führen.

Inhaltsverzeichnis

1. Veranlassung, einheitliche Strategie der Notation von Statemaschinen....3
2. Entstehung von Standards....4
3. Einfache Sicht auf synchrone Hardware-Statemaschinen....5
4. Statemaschinen in Software....6
5. Verarbeitungsschritt bzw. Schaltzeitpunkt der Statemaschinen....7
6. Unterschied zwischen eventgetriebenen und conditional Software-Statemaschinen....8
7. Run to Completion....9
8. Eventverarbeitung prinzipiell....10
9. Geschachtelte States....11
10. Parallele States....11
11. During-Action....12
12. History-Pseudostate....13
13. Alle anderen Pseudostates....13
14. Abbildung von Entry-, Exit- und Transition-Actions auf in Hardware-Statemaschinen....14
15. Simulink-Statemaschinen-Realisierungen....17
  15.1 Conditional Statemachines – Abarbeitung im Berechnungszyklus....17
  15.2 Mealy- und Moore-Statemaschinen in Simulink....18
  15.3 Statemachines mit Events....22
  15.4 Time-gesteuerte Transitionen in Simulink-Statemaschinen....23
16. Literatur, Verweis auf die Standard-Dokumente....24

1. Veranlassung, einheitliche Strategie der Notation von Statemaschinen

Betrachtet man vorhandene Softwarewerkzeuge und Entwicklungsprozesse, dann kann in der Praxis folgendes festgestellt werden:

Mit den vorhandenen Softwarewerkzeugen soll ein Entwicklungsergebnis realisiert werden. Dabei muss das Entwicklungsergebnis den Kundenanforderungen genügen, und die Realisierung muss einem Qualitätsprozess entsprechend nachvollziehbar sein. Wie das Ergebnis ansonsten zustandekommt, insbesondere ob die Softwarewerkzeuge außerhalb der geforderten Qualitätssicherungsmerkmale irgendwelchen weiteren Standards entsprechen oder ob Entwicklungswerkzeuge austauschbar sein sollen, ist vollkommen egal.

Im Klartext, wenn ein Entwicklungswerkzeug irgendwie qualifiziert ist und diese Qualifikation vom Auftraggeber anerkannt ist, ist das ausreichend. Wie sich das Entwicklungwerkzeug im Vergleich zu anderen Entwicklungswerkzeugen verhält, oder ob es einem übergeordneten Standard entspricht, ist außerhalb von Betrachtungen.

Betrachtet man diese Situation auf einen längeren Zeitraum bezogen, dann ist folgendes zu Entwicklungswerkzeugen anzumerken:

* Für ein Entwicklungswerkzeug könnte eine Nutzungsdauer von 10 Jahren angesetzt werden. Das entspricht etwa auch den Zeiträumen für eine konstante Zusammensetzung von Entwicklungsabteilungen entsprechend der Laufzeit von Großprojekten oder Produktlinien (PLM, Prozess-Lifecycle-Management)
* Nach weniger als 10 Jahren wird ein Entwicklungswerkzeug kaum ausgesetzt werden, es sei denn schwerwiegende Gründe sprechen dafür. Man muss Schulungsaufwände, Entscheidungszeiten etc. beachten.
* Nach 10 Jahren jedoch werden sich auch beim Tool-Hersteller Bedingungen geändert haben, es gibt technische Neuerungen, junge Kollegen mit einem anderen Fokus etc. etc. so dass neue Entscheidungen durchaus zweckdienlich sein könnten.
* Für die „Altlastenpflege“ für Produkt-Verantwortungszeiträume z.Bsp. Von 30 Jahren gibt es diverse Strategien für aufgehobene Entwicklungswerkzeuge. Das ist aber bereits deshalb notwendig, weil eine Bearbeitung älterer Quellen mit neueren Versionen der gleichen Entwicklungswerkzeuge möglicherweise auch diverse Randprobleme hervorrufen könnte.

Die Softwarequellen sind aber häufig eigentlich für einen längeren Zeitraum als 10 Jahre relevant:

* Es geht die Mär um, das beispielsweise für große Simulationssysteme in der Forschnung immer noch alte FORTRAN-Bibliotheken verwendet werden, in teils angepasster Form.
* Es ist schlichtweg unnötig, Algorithmen neu zu schreiben, wenn sie mathematisch noch genauso gültig sind, nur weil es einen neuen Standard für C++ gibt.
* Programme, die in den 90-ger Jahren nach C89/C99-Standard geschrieben wurden und damals mit mehreren Compilern auch mit C++ getestet worden sind, laufen heute noch auf jedem Compiler. Wenn deren Algorithmen korrekt und getestet sind, werden sie weiter verwendet.

Die Toolhersteller haben meist kein sehr hohes Interesse, eine Kompatibilität mit anderen Tools herzustellen:

* Es beschränkt die eigenen Möglichkeiten, auch durch besondere Features hervorzutreten.
* Ein Import aus anderen Tool-Speicherformaten ist ja ganz interessant, um den Wechsel auf das eigene Tool zu erleichtern.

Die Toolhersteller können zum Einhalten eines Standards eigentlich nur auf zwei Wegen gezwungen werden:

* Wenn es der Kunde definitiv so verlangt, und es gibt entsprechend viele Kunden
* Wenn es eine übergeordnete Organisation für die Sache gibt und ein entsprechendes Zertifikat daran hängt, dass ggf. auch für den Softwarequalitätsnachweis eine Rolle spielt.

2. Entstehung von Standards

In der allgemeinen Industrie, Baugewerbe etc. sind Standards schlichtweg eine Pflicht und historisch schon länger verbreitet, z.Bsp. DIN = Deutsche Industrie-Norm.

In der Softwareentwicklung selbst sind Standards dagegen kaum vorgeschrieben. Sie entstehen als Quasi-Standards, aus einem marktbeherrschendem Tool oder einer Herangehensweise, letztlich dann auch als Qualitätssicherungsnachweis.

Der *American Standard Code of Information Interchanche* (ASCII) hat sich, 1963 erstmalig gebilligt, derart schnell weltweit durchgesetzt, dass dem Verfasser die gleiche Codierungsart jenseits des eisernen Vorhanges in den 1970-gern als eine TGL-Nummer bekannt war (TGL = „Technische Güte- und Liefervorschriften“ in der DDR) und auch im sowjetischen GOST-Standard verankert war. Heutige Textcodierungen wie ISO-8859-x und UTF-8 sind abwärtskompatibel zu ASCII, für grundlegende Texterkennung gilt immer noch ASCII. Die weltumspannende Einführung dieses Standards ist praktikabel.

Es gab 1969 bis 1973 eine Entwicklung bei Bell Laboratories unter Führung von Dennis Ritchie, deren dritter Wurf nach A und B „C“ genannt sich als Quasi-Standard so verbreitet hat, dass die meisten Programmiersprachen in fast gleicher oder ähnlicher Syntax definiert sind. Der erste C-Standard gab es als C89 20 Jahre nach entstehen der Sprache, und dennoch hat sich die Java-Entwicklung im Folgejahr 1990 an C bzw. C++ angelehnt. Die Sprache war als informeller De-facto-Standard mit dem Buch „The C Programming Language“ von Brian W. Kernighan und Dennis Ritchie aus 1978 beschrieben.

Digitale Automaten oder Statemaschinen werden nach Veröffentlichungen von George H. Mealy und Edward F. Moore in den 50-ger Jahren einheitlich mit runden Knoten, den States und Übergängen als Pfeil, den Transitionen beschrieben. Wann und ob dies standardisiert wurde ist mir nicht bekannt, aber angewendet wurde es.

Im Jahr 1987 hat David Harel eigene Gedanken zu Statemaschinen niedergelegt /Harel/, die dann über I-Logix, Statemate und Rhapsody in die UML-Welt eingegangen sind und mit der /omg-UML/ standardisiert worden sind. Bezüglich der *„Unified“* Modelling Language hat man für die Version 2.0 festgelegt, dass Tools dies nur dann ausweisen können, wenn sie mit /omg-UML/ wirklich kompatibel sind.

Die Harel-like Statemaschinen werden auch außerhalb der UML verwendet, zumindestens teilweise in der Syntaxschreibweise und in der Dokumentation so benannt, mehr oder weniger tatsächlich kompatible bzw. dem Standard entsprechend. Harel-Statemaschinen sind bis heute auch wegen der Präsenz der UML und deren Eignung für die Praxis bewährt, etwas grundlegend Neues zeichnet sich hier nicht ab.

Es gibt auch Standardisierungen von Programmiersprachen, die gleich zu Anfang erfolgen und deren breiter Einsatz noch zu erwarten ist. Hier kann die Arbeit am Standard zusammen mit Pilot- und wichtigen Einsatzerfahrungen eine gute Basis für erfolgreiche zukünftige Einsätze bilden.

3. Einfache Sicht auf synchrone Hardware-Statemaschinen

Hardware-Statemaschinen arbeiten im Kern mit getriggerten Flipflop (D-FF). Diese schalten mit einer Taktflanke die Belegung des D-Einganges auf Q. Das Grundelement aus der TTL-Schaltkreisserie ist in etwa der 7474. Dieses Grundelement ist in den Zellen eines FPGA in adäquater Form enthalten.

Mit diesen D-FF und der Verknüpfungslogik vor den D-Eingängen wird die folgende Beziehung verwirklicht:

```
Q := fn(Q, U)
```

`Q` ist der State, die Gesamtheit der Zustände aller Flipflops, als Vektor lesbar.

`U` sind Eingangsbelegungen

`fn(…)` ist die Verknüpfung der Eingänge mit den Zuständen.

`:=` Das ‘Ergibt’-Symbol oder die Zuweisung, bekannt aus Programmiersprachen außerhalb C/++ bezieht sich auf den Folgezustand. Der Folgezustand wird mit der Taktflanke ausgelöst.

In FPGA-Schaltkreisen spielt es eine wesentliche Rolle, dass die Laufzeit der Logikverknüpfungen, also das fn(…) im gesamten Temparaturbereich kleiner ist als die Taktperiode. Dann funktioniert die Statemaschine richtig so wie programmiert und gewollt. Ansonsten funktioniert sie nur zufällig, das ist nicht brauchbar. Die Routing-Tools der FPGA berücksichtigen bekannte Zeitverzögerungen entsprechend den verwendeten Elementen und sichern die Korrektheit. Der Routing-Prozess dauert längere Zeit und wird ggf. mit einem Fehler abgebrochen, wenn das time-richtige Routing nicht möglich ist.

Wenn die Ausgangsbelegung nur aus dem Zustand berechnet wird

```
y = fn(Q)
```

dann spricht man von einer Moore-Statemaschine. Wird die Ausgangsbelegung aus Inputs und Zustand berechnet

```
y = fn(Q, U)
```

dann ist dies eine Mealy-Statemaschine. Dieser Unterschied ist aber nicht wesentlich für die Statemaschine selbst, da man die Ausgangsbelegung nach der Statemaschine so realisiere kann wie man will. Wichtig ist das aber für Stabilität und Laufzeit. In der Moore-Form hängt die Ausgangsbelegung nur vom Zustand ab und ist in der Umschaltzeit nach der Taktflanke stabil.

Die synchone Statemaschine schaltet mit einem unabhängigen festen Takt. Die asynchrone Form bildet Umschaltsignale aus den Eingängen, was wesentlich schwerer zu durchschauen ist, wird daher heute fast in der Regel nicht mehr verwendet.

Das ist der Essenz dessen, was zu Hardware-Statemaschinen allgmein zu sagen ist.

4. Statemaschinen in Software

Nachdem bis in die 80-ger Jahre Statemaschinen in Software verschiedenlich gebaut worden sind, mit Bedingungsabfragen, im Programmablaufplan, teils als Petrinetz dargestellt, hat David Harel (https://de.wikipedia.org/wiki/David_Harel) mit einigen Schriften geschachtelte (nested) und parallele Statemachinen mit History-Connection vorgestellt, die unter seiner Leitung im Tool Statemate (https://de.wikipedia.org/wiki/Statemate) realisiert wurden. Diese Entwicklung ist von der Firma I-Logix dann über das UML-Tool Rhapsody (https://de.wikipedia.org/wiki/Rational_Rhapsody) in die UML-Welt eingegangen und folglich in den UML- Standards der Object Modelling Group (https://www.omg.org/) aufgenommen. In dieser im wesentlichen seit David Harels Arbeiten unveränderlicher Form sind Software-Statemaschinen mehr oder weniger dem Standard entsprechend in vielen Software-Tools wie beispielsweise auch in Simulink anzutreffen.

Der letzte Satz muss nochmals auseinandergenommen werden: David Harels Arbeiten gelten bis heute in unveränderter Form. Sie entsprechen dem in den omg-Standards niederlegten Dingen. Aber die Implementierungen weichen mehr oder weniger davon ab.

Es stellt sich nun die Frage, ob die wesentlich komplexeren Statemaschinen in Software mit den Statemaschinen in Hardware verglichen werden können?

Auch bei Software-Statemaschinen kann man schreiben:

```
Q := fn(Q, U)
```

Also, der Folgezustand hängt vom Vorzustand und den Eingangsbedingungen ab.

Anstelle der Taktflanke in der Hardware, die dem `:=` -Symbol entspricht, gilt der Moment der Abarbeitung der Statemaschine. Das ist eine Frage von Event oder Trigger, siehe Folgekapitel.

Die Frage, parallel und nested ist im `fn(…)` -Symbol verankert. Das `Q` ist die Gesamtheit aller Zustände einzelner Flipflop oder durch Variable in der Software repräsentiert.

Es gibt einen wesentlichen Unterschied:

In Software-Statemaschinen können Events oder Trigger für die eigene oder andere Statemaschinen in den Übergängen oder auch zugeordnet zu den Entry- und Exit-Actions der States (die bei den Übergängen, Transitionen aufgerufen werden) erzeugt werden.

Man kann dies allerdings auch auf die Hardware-Statemaschinen übertragen. Events oder Trigger schalten andere Statemaschinen. Das ist insoweit wichtig, dass heutzutage Lösungen in Software mit entsprechender Codegenerierung auch in Hardware ablaufen können sollten. Gibt es eine gut ausgearbeitete Beschreibung von Software-Statemaschinen, die etwa wegen kurzer Berechnungszeiten gleichermaßen in Hardware beispielsweise in einem FPGA ablaufen sollten, dann wäre es gut wenn dies unterstützt würde.

5. Verarbeitungsschritt bzw. Schaltzeitpunkt der Statemaschinen

Hardware-Statemaschinen schalten mit der Taktflanke. Es gibt, mindestens bei Xilinx, zusätzlich einen Freigabetakt CE. Dieser funktioniert eigentlich so, dass bei CE=0 der Ausgang Q direkt auf den Eingang D verknüpft ist, so dass es bei der Taktflanke keine Veränderung gibt.

Die Taktflanke ist bei Hardware-Statemaschinen synchron und zyklisch. Mit dem CE-Eingang kann man aber logikverknüpft entscheiden, ob geschaltet werden soll oder nicht. Ist CE=0, dann spielt die Eingangsbelegung der Logikverknüpfung der D-Eingänge keine Rolle. Das ist also so als würde das Schalten (Triggern) nicht aufgerufen werden. CE kann typischerweise zyklisch freigeben werden (CE=1) als Taktuntersetzung, oder auch von Bedingungen selbst abhängen.

Software-Statemaschinen schalten, wenn ihr Verarbeitungsalgorithmus aufgerufen wird. Das ist allgemeingültig formuliert. Die Formulierung *„wenn ein Event kommt“* ist dagegen schon implementierungsspezifisch und gilt nur, wenn es so programmiert ist.

Damit ist der Aufrufmoment des Verarbeitungsalgorithmus von Software-Statemaschinen vergleichbar mit dem Takt der Hardware-Statemaschinen. Beides kann als *„atomic“* - nicht unterteilbar – bezeichnet werden, wenn beim Hardwareschalten (benötigt auch Laufzeit) nicht unverzüglich die schaltenden Ausgänge per XOR verknüpft weitere Schaltvorgänge von diesen erzeugten Flanken auslösen würden (das könnte man machen, ist aber nicht zielführend). Bei der Softwarelösung gilt, dass die Abarbeitung nicht unterbrochen wird beziehungsweise Zwischenergebnisse während der Abarbeitung nicht schon benutzt werden. Letzteres lässt sich verhindern durch entsprechende Mutex-Lösungen (*„mutual exclusion“*, *„Wechselseitiger Ausschluss“* des Zugriffs auf die Daten). Also sind hier Prinzip-Ähnlichkeiten von Hard- und Softwarelösungen sichtbar.

Häufig wird dargestellt, dass Statemaschinen in Software *„eventgetrieben“* sind, also nur aufgrund von Events schalten. Events werden von einem Softwareteil zu einem anderen Softwareteil als Messages verschickt und können insbesondere in anderen Statemaschinen in den Übergängen (Transitionen) erzeugt werden.

Auf die einzelne Abarbeitung eines Statemaschinen-Verarbeitungsalgorithmus sieht das wie folgt aus:

Ein Event wird bei Empfang zumeist in einer Queue gespeichert, ein first-in-first-out-Container. Das ist wichtig für ein anderes Software-Statemaschinen-Prinzip *„Run to Completion“*. Diese Event-Speicherung entkoppelt auch Threads, Prozesse oder Geräte. Es gibt dann genau einen Thread, der für das Ausräumen dieser Queue zuständig ist und eine oder mehrere Statemaschinen versorgt. Ein Event kann eine bestimmte Statemaschine (in einer Instanz von Software-Daten, einer *class*) ansprechen, das Event bzw. die Message enthält die dazu nötige Zieladresse. Ein Event aus der Queue kann aber auch an alle Statemaschinen dieses Threads appliziert werden sollen.

Im Event-Queue-Thread wird dann der Verarbeitungsalgorithmus der Statemaschine aufgerufen, mit übergebenem Event.

Für den Verarbeitungsalgorithmus der Statemaschinen ist das Event (dessen Typ) eine boolsche Bedingung des Schaltens. Wenn es am entsprechenden State Transitionen gibt, die diesen Event als Trigger benennen, dann ergibt die Prüfung des empfangenen Events im Vergleich zum erwartenden Event *„true“* als erste Bedingung neben gegebenenfalls danach noch folgenden Guard-Bedingungen. Wenn es etwa eine Liste erwarteter Events gibt, diese wird getestet und ein nicht erwartetes Event

erst gar nicht an die Statemaschine appliziert, dann ist das das Gleiche. Die Prüfung wird nur zuvor vorgenommen und nicht erst an den konkreten Transitionen.

Folglich gibt es eine Übereinstimmung in der Abarbeitungsorganisation von Hard- und Software-Statemaschinen. Bei beiden gibt es die Atomic-Abarbeitung, die bei Hardware-Statemaschinen nur schneller und konsequent zyklischer gerufen wird, bei Software-Statemaschinen nur dann wenn es überhaupt ein Event in der Queue gibt. Man spart in der Software damit etwas Rechenzeit. Die Bedingungsbildung ist aber identisch, das Event wird reduziert auf eine der boolschen Bedingungen zur Bildung des Zielzustandes.

Man kann bei der Umsetzung der eventgetriebenen Software-Statemaschinen auf Hardware-Abarbeitung so vorgehen, dass der Takt zwar zyklisch kommt, das CE-Signal aber zusätzlich gesperrt wird wenn es keine Information über vorliegende Abarbeitungsnotwendigkeit gibt. Die Event-Queue kann als Speicher mit einer gezählten Adresse als Ringspeicher realisiert werden. Sie ist leer wenn Lese- und Schreibadresse identisch sind.

Ein Event enthält in der Regel auch Daten. Diese sind von einem bestimmten Typ, brauchen diesen entsprechenden Platz und sind über die Queue referenziert oder direkt in der Queue enthalten. Nicht zu komplexe Daten können somit auch als Hardware abgebildet werden. Mit den Daten können weitere Bedingungen neben den Events für Stateübergänge gebildet werden.

6. Unterschied zwischen eventgetriebenen und conditional Software-Statemaschinen

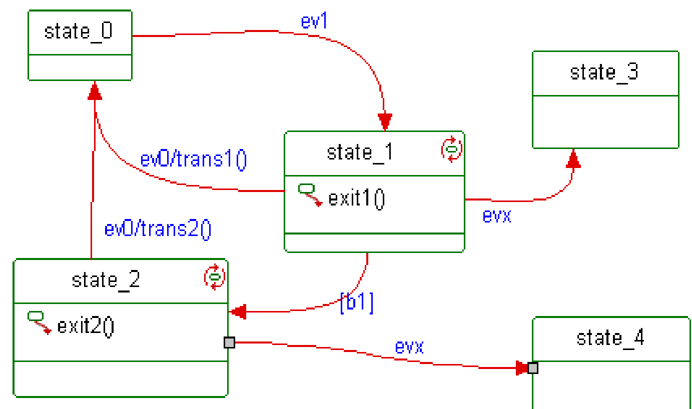
Entsprechend den Ausführungen des Vorkapitels verschwindet dieser Unterschied auf die Statemaschine-Ausführungsalgorithmus selbst bezogen. Der Unterschied liegt ‚nur‘ in dessen Aufrufbedingung:

* Der eventgetriebe Aufruf erfolgt nur wenn ein Event vorliegt, im Eventqueue-Ausräum-Thread.
* Der Aufruf einer conditional Statemaschine erfolgt zyklisch, ohne die zusätzliche Eventbedingung.
* Beide Aufrufe können auch in einem Hardware-Interrupt erfolgen. In einem Fall prüft der Hardwareinterrupt die Eventqueue und ruft den Statemaschinen-Algorithmus nur mit vorliegendem Event auf, im anderen Fall wird er immer aufgerufen, die Umschaltbedingungen werden in den Transitionen in beiden Fällen geprüft.

Damit ist aber die Mischung von eventgetriebenen und conditional Transitionen in ein und derselben Statemaschine nicht möglich. Denn: Es muss außen entschieden werden, ob der Aufruf nur bei Event oder immer erfolgen soll. Das ist wesentlich für die Software des Aufrufs. Siehe auch Folgekapitel *„Run to completion“*.

7. Run to Completion

Dieses Prinzip bedeutet, dass in einer eventgetriebenen Statemaschine nach Aufruf mit dem Event der Ausführungsalgorithmus wiederholt ohne Event aufgerufen wird, bis sich der Zustand nicht mehr ändert. Damit ist es möglich, nach einem eventgetriggerten State-Übergang noch weitere rein bedingungsgesteuerte Übergänge auszuführen. Damit liegt aber keine *„conditional Statemaschine“* vor. Denn diese würde unabhängig von Events aufgerufen. Die Eventgetriebene Statemaschine führt eine rein bedingungsgesteuerte Änderung nicht aus wenn vorher kein Event die Abarbeitung insgesamt ausgelöst hat. Um das klar zu verdeutlichen, eine kleine Beispiel-Statemaschine:



Wenn das ev1 kommt und die Bedingung b1 vorliegt, dann wird state_2 angenommen. Ist b1 == false, dann verbleibt die Statemaschine im Zustand state_1, auch wenn später die Bedingung b1 auf true wechselt. Es ist keine conditional Statemaschine, sondern die Condition ohne Event in der Transition von state_1 nach state_2 wird nur im *Run to Completion*-Zyklus geprüft.

Folglich wird später beim ev0 entweder trans1() aufgerufen, oder trans2(), je nachdem welcher Zustand zuvor nach ev0 angenommen wurde.

Bei einer conditional Statemaschine kann von *„Run to Completion“* gesprochen werden, wenn die Statemachine im selben Aufrufzyklus wiederholt gerufen wird, bis der Zustand nicht mehr wechselt. Bei Simulink beispielsweise ist dieses Verhalten auswählbar. Bei einer manuellen Codierung braucht es eine while-Schleife. Erfolgt dies nicht, dann wird aber im folgenden Aufrufzyklus weitergeschaltet.

Das *„Run to Completion“* hat noch eine weitere wesentliche Bedeutung: Das folgende Event wird erst aus der Queue gelesen, wenn der *„Run to Completion“*-Zyklus abgeschlossen ist. Das ist bedeutungsvoll auch im obigen Beispiel sichtbar: Liegt in der Queue nach dem ev1 das evx, dann wird abhängig von b1 eindeutig geregelt state_3 oder state_4 angenommen. Das b1 wird zuerst, im *„Run to completion“*-Zyklus, geprüft.

Was ist, wenn der *„Run to completion“*-Zyklus in einer Eigenschleife hängt, Bedingungen also zum Kreiseln zwischen States führen. Dann liegt faktisch ein Fehler im Statemaschinen-Entwurf vor. Das lässt sich debuggen, bzw. das Verhalten lässt sich kontrollieren, indem die Anzahl der Aufrufe bis *„completion“* auf maximal der Anzahl vorliegender States begrenzt wird. Im Simulink ist diese Sache so geregelt, dass im Simulationsmode wahlweise eine Fehlermeldung mit Abbruch der Simulation gerufen werden kann, in der codegenerierten Form bleibt aber die Abarbeitung nach der maximalen Anzahl von Aufrufen beim zufällig erreichten State stehen. Man könnte unter Ausnutzung von Exceptionhandling aber auch in der codegenerierten Form eine Exception erzeugen, und die Situation übergeordnet geeignet abfangen (try-throw-catch).

8. Eventverarbeitung prinzipiell

Events werden aus einer Queue entnommen, in der sie vom Sender eingespeichert werden. Die Events können auch über Prozess- und Gerätegrenzen per Kommunikation übertragen werden, als Messages. Auch der eigene Prozess, der Statemaschinenverarbeitungsalgorithmus, kann ein Folgeevent in die Queue einschreiben.

Die Queue ist first-in-first out organisiert. Folglich werden vom eigenen Verarbeitungsalgorithmus erzeugte Events erst wieder verarbeitet, wenn die noch vorhandenen Events verarbeitet sind. Das kann wesentliche Effekte auf die Statereihenfolge haben. Daher ist es wichtig, dass diese Regel bei verschiedenen Implementierungen eingehalten wird. Sie ist in der omg-Spezifikation festgehalten:

Die Ausnahme ist der Run to Completion-Zyklus, wenn er als Realisierung eines Eigen-Event aufgefasst wird. Dazu steht in /omg-UML/ auf S. 314 im Kapitel *14.2.3.9.1 The run-to-completion paradigm*:

NOTE. As explained above, completion events have priority and will be dispatched ahead of any pending Event occurrences in the event pool.

Allerdings kann „Run to completion“ auch ausgeführt werden in einer bedingten Schleife, solange ein Statewechsel vorliegt. Es braucht also kein *completion event*. Das ist eine Implementierungsfrage.

Ein Event wird aus der Queue geholt. Es wird an die entsprechenden Statemaschine appliziert. Unabhängig davon ob es dort zum schalten führt oder nicht, wird es danach gelöscht. Ein nicht appliziertes Event wird also keinesfalls etwa aufgehoben, wieder in die Queue eingereiht oder dergleichen. Das erscheint manchem intuitiv denkenden als ein „Verlust einer Information“. Jedoch: Ein Event, was jetzt nicht zu gebrauchen ist, kann später appliziert zu Irretationen führen.

Der Verlust eine Information, der auch in einer Informationsübertragungskette passieren kann, muss durch Timeout-Mechanismen in den Statemaschinen, die der Hand des Anwenders liegen, abgefangen werden. Ist beispielsweise bei einer Buchung der Zustand erreicht, in dem der Bediener eigentlich „OK“ oder „Abort“ betätigen müsste, es liegt aber keines der beiden Event vor, dann kann nach angemessener Zeit der Bediener, oder die entfernte Gegenstelle informiert werden, dass diese Information erforderlich ist. Wie das geschieht, ist Anwendungssache.

Konstruktionen wie ein *14.2.4.9.6 Deferred triggers*, so beschreiben auf S. 331 in /omg-UML/ sind nicht gut.

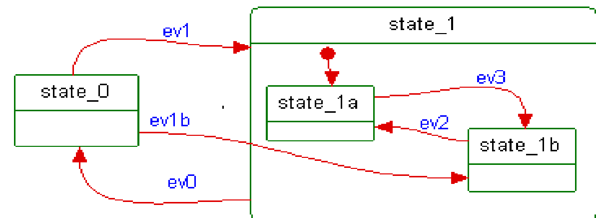
Ein Event wird nur je für einen Stateübergang appliziert. Danach ist es verbraucht. Folgen zwei Transitionen mit dem gleichen Event, dann werden dafür zwei Events benötigt. Das betrifft auch geschachtelte States, nicht aber parallele States. Ein Event wird in jedem Parallelzweig eigentständig appliziert.

Das ist implementierungstechnik so lösbar, dass mit einem Event der Statemaschine-Ausführungsalgorithmus abgearbeitet wird. Danach folgt eine Schleife für „Run to completion“ aber ohne das Event. Danach wird der Statemaschine-Ausführungsalgorithmus erst wieder mit dem nächsten Event aus der Queue aufgerufen. Parallele Statemaschinen-Teile werden aber mit dem selben Event nacheinander mit ihrem „Run to completion“ ausgeführt. Wenn in einem der Parallelzweige aber der State mit parallelen Statemaschinen verlassen wurde, dann gibt es diesen Parallelzweig nicht mehr. Sollte zuvor, also vor dem Verlassen, der Parallelzweig mit dem Event noch bearbeitet werden? Das wäre ***zu klären***.

9. Geschachtelte States

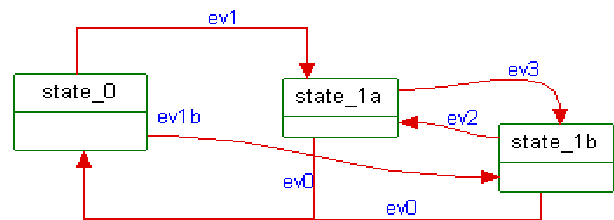
Die *„nested states“* in den Harel-Statemaschinen sind eigentlich nur eine zweckdienliche Zusammenfassung.

Wenn rechtsstehend ev1 kommt, wird im Substate state_1a angenommen, weil dies der default-State ist. Mit ev1b kann man dagegen direkt state_1b anwählen. Bei ev0 wird unabhängig vom Substate dieser verlassen, die passende exit-Action aufgerufen und jedenfalls state_0 angenommen.



Wichtig ist, dass state_1, also der umfassende State eine eigene exit-Action haben kann. Diese wird aufgerufen nach der exit-Action aus den inneren States.

Ohne die Möglichkeit der Nested-Darstellung ist dieses Beispiel auch darstellbar. Aber, die Nested-Darstellung ist übersichtlicher. Zusätzlich gibt es auch die Möglichkeit, die Nested-States in einem eigenem Statechart darzustellen. Das ist eine Frage der Modularität.



Jedenfalls lassen sich Nested States immer auflösen, was für deren Hardware-Realisierung bedeutungsvoll ist. Die exit-Action müssen dann zusammengefasst werden.

10. Parallele States

… wird hier nur verkürzt dargestellt (TODO), wichtige Elemente Fork & Join, zu Eventverarbeitung siehe 8 Eventverarbeitung prinzipiell Seite 8

11. During-Action

Für Simulink gibt es eine Mathworks-Doku:

https://in.mathworks.com/help/stateflow/ug/parallel-and-state-examples.html

In einem Beispiel wird gezeigt, wann welche Actions (in diesem Beispiel mit parallelen States) durchlaufen werden. Dabei ist die During-Action selbstverständlich mit dabei.

Aus dem Text ist erkennbar, das die During-Actions nur aufgerufen wird, wenn die Statemachine von einem Event animiert wird. Sie ist aber unabhängig (unconditional) und wird hier jeweils als erstes aufgerufen.

In zyklisch aufgerufenenen Statemaschinen wird die During-Action in jedem Abarbeitungszyklus aufgerufen. Das ist beispielsweise der Fall, wenn in Hardwareinterrupts in Embedded Control ein schneller Zyklus mit einer conditional Statemachine implementiert ist.

Man könnte nun für den letzten Fall auf die Idee kommen, stateabhängige Aktionen in die during-Action der Statemachine einzuordnen. Dabei könnte argumentiert werden, dass Rechenzeit eingespart wird, weil die Abfrage des States in der Machine ja sowieso erfolgt.

Es wird dabei allerdings die Möglichkeit verbaut, aus später sich ergebenden Gründen die Statemachine auf Event-getriggert umzubauen. Das ist auch bei Fast Embedded Control und auch im schnellen Hardware-Interrupt möglich: Eine minimalistische Event-Queue kann auf Vorliegen eines Event abgefragt werden, mit dieser einfachen Abfrage wird dann entschieden, ob die gesamte Statemachine überhaupt aufgerufen wird. Dies könnte ebenfalls rechenzeitsparend sein.

Folglich sind During-Actions kein gutes Entwurfsmuster. Es könnte einfacher sein, einen zyklischen Algorithmus in der üblichen Form mit if-Zweigen zu programmieren. Die if-Zweige werden von boolean-Variable gesteuert (schnelle Abfrage). Diese boolean-Variable werden passend in den States (im Entry) gesetzt. Das könnte sogar übersichtlicher sein.

12. History-Pseudostate

Der History-Konnektor wurde von David Harel im Orignal eingeführt, siehe /Harel/ Seite 238 (8 ab Anfang).

Implementierungstechnisch ist der History-Pseudostate sowohl als *deep*- als auch als *shallow*-History einfach zu realisieren: durch Speicherung des verlassenen States einer Sub-Statemachine. Für den History-Entry ist es notwendig nicht nur den State zu restaurieren, sondern auch die entsprechenden Entry-Actions zu durchlaufen, adäquat wie beim Verlassen eines Sub-States auch alle exit-Action durchlaufen worden sind. Pro Ebene und pro Parallelität mit dem gespeicherten State sollte das bei Software-Statemaschinen kein Problem sein. Man muss in jeder nested- und parallel-Ebene der gesamten Statemaschine jeweils eine eigene Statevariable führen.

13. Alle anderen Pseudostates

wie junction und choice sollten ebenfalls beachtet und umgesetzt werden, siehe /omg-UML/

14. Abbildung von Entry-, Exit- und Transition-Actions auf in Hardware-Statemaschinen

Wenn schon in den vorangegangenen Kapiteln mehrfach auf Parallelitäten von Hard- und Softwarestatemaschinen eingegangen wurde, dann muss auch dieses Thema beleuchtet werden. Es soll hier keine vollständige Lösung dargestellt werden, sondern es wird lediglich die mögliche Parallität und Realisierung beleuchtet.

Wenn in den Actions lediglich Variable belegt werden, dann gilt für Entry-Actions die einfache Zuordnung:

```
y := fn(Q)
```

Die Variablen `y` (als Gesamtheit) folgen direkt den Zuständen `Q`, da die Belegung nach Entry eindeutig ist. Bei den exit-Actions ist das so einfach nicht darstellbar.

Die Transistion-Actions entsprechen eigentlich den Entry-Actions, nur das pro Transition nochmals ein Unterschied gemacht werden kann. Es gilt dann:

```
y := fn(Q, u)
```

Dieses entspricht formell der dem Zusammenhang im Mealy-Automaten, allerdings mit einem Takt Verzögerung erst nach Einnahme des neuen States. Es gibt tatsächlich einige Literaturstellen, die genau die Zuordnung so darstellen. Das ist aber deshalb nicht korrekt, da die Zuordnung nur erfolgt im Moment des Schaltens der Transition. Der `y`-Zustand wird also gespeichert mit der Flanke des Transitionsübergangs, wobei beim Mealy-Automaten der y-Ausgang vom Zustand und den ständig ausgewerteten Eingängen abhängt, was sich mit

```
y = fn(Q, u)
```

darstellen lässt. Der Unterschied ist der `:` vor dem `=`. Folglich ist es immer ein Moore-Automat, bei dem die Ausgangsbelegung als eigener Zustand mit dem Schalttakt abgelegt wird.

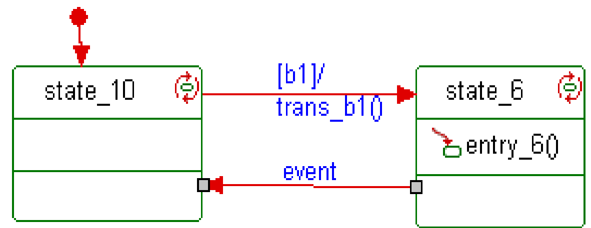
Nun zur Frage wie Events oder Funktionsaufrufe sich abbilden lassen:

Funktionsaufrufe sind in Hardware eine angestoßene Berechnungskette. Der Anstoß ist ein Impuls, das Berechnungsende kann ebenfalls von einem Impuls abgebildet werden, der synchron sofort im Folgetakt dieser Berechnung kommt (ggf. mit Taktuntersetzung, CE-Signal).

Events haben die Eigenschaften, dass sie gespeichert und übertragen werden können. Sie können aber ebenso als Impuls dargestellt werden, dessen Auftreten eben gespeichert und übertragen wird.

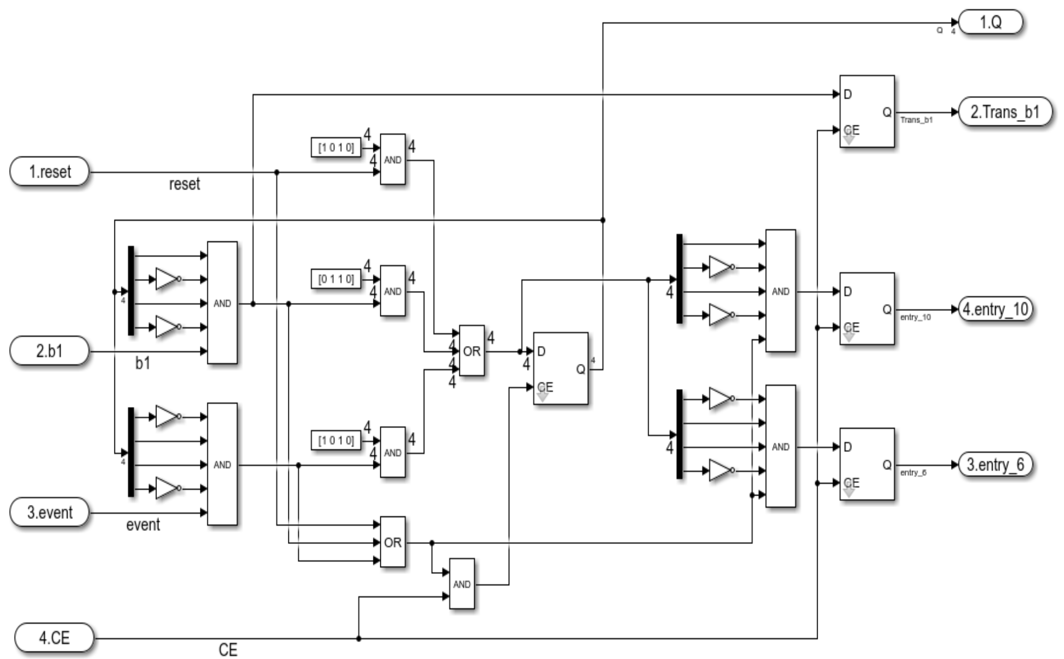
Der Impuls lässt sich ableiten bei eventgetriebener Hardware-Abarbeitung vom Vergleich des Zustandes AND-Verknüpft mit dem Freigabesignal des Taktes. Der Freigabetakt (CE) wird erzeugt wenn ein Event vorliegt dass

Zum internen Aufbau von FPGAs siehe beispielsweise
https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

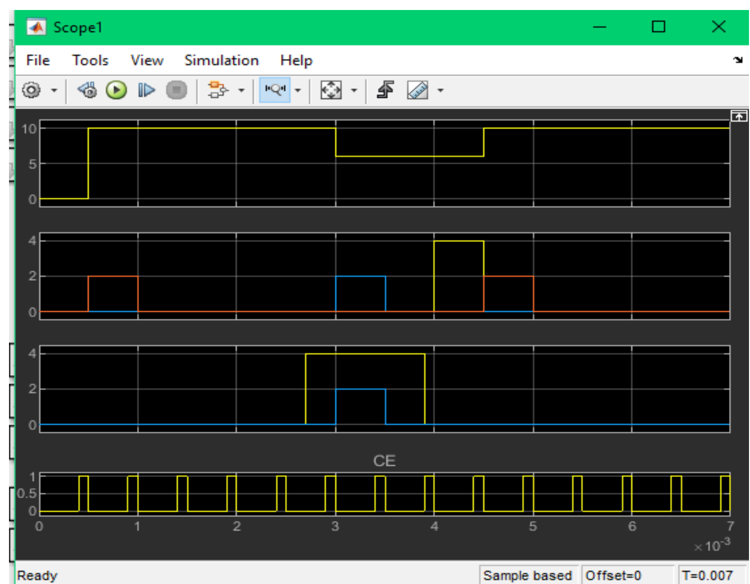


Die rechtsstehende Statemaschine ist in Hardware dargestellt:

Es ist ein wenig Aufwand notwendig, um die States zu detektieren. Das zentrale DFF in der Mitte enthält intern 4 DFF, erkennbar an der 4 an der Leitung, ebenso wie an einigen Logikgattern. Das vereinfacht die Darstellung, erstellt mit Simulink.



Die Darstellung soll in etwa den Gepflogenheiten in den Logic Cell Blocks in einem Xilinx-FPGA folgen. Dort ist allerdings die Logik selbst grundsätzlich in Lookup-Tables enthalten, das FPGA kennt keine AND und OR. Die Freigabesteuerung mit CE ist original wie in Xilinx. Der Takt ist hier nicht gezeichnet, es ist der Simulationstakt.



Das Simulationsergebnis für ein Beispiel ist rechts gezeigt. Die beiden Inputsignale b1 und b2 sind gelb 2. und 3. Spur. Im Beispiel ist die Event- und die conditional-Triggerung etwas vermischt, was eigentlich ausgeschlossen sein sollte, für dieses Beispiel jedoch verwendet. Es handelt sich hier nur um eine prinzipielle Demonstration. Der Unterschied liegt in der äußeren Bildung: Ein Event wird aus einer Queue (in Hardware) ausgelesen und ist folglich genau einen Arbeitstakt breit. Es ist das gelbe Signal im 2. Track. Das Event führt zum Schalten von 6 auf 10, danach (!). Die Bedingung ist dagegen beliebig breit, hier das gelbe Signal im 3. Track. Es führt zum Schalten, wenn es mit der folgenden Taktflanke erkannt wird, hier von 10 auf 6.

Der 2. Track zeigt in blau und rot die Entry-Action-Impulse, im 3. Track ist in blau der Transition-Impuls dargestellt.

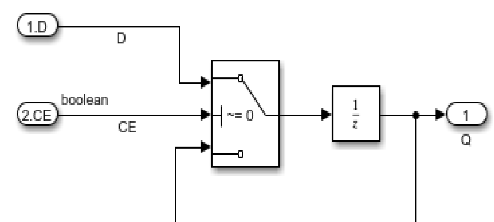
Sowohl die Trans-Action als auch die beiden entry-Actions werden als Impus ausgegeben, und zwar genau eine Taktbreite jeweils nach der Zustandsänderung, also begleitend zum Eintritt. Das gilt auch und insbesondere für die Trans-Action. Diese wird nicht etwa wie intuitiv oder teils fälschlich angenommen mit der Transition aktiviert, sondern mit dem Eintritt aufgrund der Folge der Transaktion.

In der Hardwareschaltung ist das entsprechend realisiert. Die Entry-Action-Impuls werden mit Ausdekodierung des Ziel-Zustandes selbst gewonnen, allerdings AND-verknüpft mit dem Schaltimpuls gebildet aus OR aller Transitions. Damit kommt der Impuls nicht etwa mit jedem Zustand statisch, sondern nur wenn der Zustand eingenommen wurde. Er kommt auch wenn vom selben Zustand auf ihn selbst gewechselt wurde mit einem Event oder einer Bedingung (hier nicht dargestellt). Das entspricht dem Verhalten in der Software nach David Harel.

Die Trans-Action wird gebildet, wenn die Transition aktiv schaltet. Das ist mit dem Vorzustand verbunden, diese Bedingung gehört dazu.

Die Actions werden kombinatorisch **vor dem Schalten** gebildet, was direkt zur Trans-Action passt. Danach folgt ein Flipflop, was den positiven Nebeneffekt der zeitlichen Entkopplung bringt. Die Laufzeiten im FPGA sind entscheidend, sie sind länger wenn die Kombinatorik komplexer wird. Damit sind die Impuse selbst zeitglich mit der Einnahme der Zustände und können im Folgezustand weitere Aktionen auslösen (Action erzeugt Event für andere Statemaschine usw.).

Der „*Run to completion*“-Zyklus kann in Hardware so nicht abgebildet werden, denn er muss im Ausführungsalgorithmus insgesamt ablaufen, was hier einem Takt entspricht. Das „*Run to completion*“ benötigt je einen Takt pro Stateübergang, anders geht es in Hardware nicht. Um es hier abzubilden, muss eine zusätzliche Freigabe der Bedingung nach einem Event-Schalten solange gebildet werden, bis die Freigabe insgesamt wieder auf 0 geht (Signal vor dem AND des CE für die Zentral-FF). Das ist in Hardware abbildbar, „*Run to completion*“ also auf mehrere Hardwaretakte verteilt, ansonsten wie in Software. Das Folgeevent darf erst dann aus der Queue geholt werden, wenn die „*Run to completion*“-Freigabe vorbei ist, lässt sich organisieren.



Die Innenschaltung des D-FF mit CE-Freigabetakt in Simulink entspricht der Realisierung in einem Xilinx-FPGA. Der Takt selbst ist hier der Berechnungsschritt und daher hier nicht dargestellt. Wenn die Freigabe CE = 0 ist, dann ist per Auswahllogik (Multiplexer oder Auswahl-Logik im Xilinx, hier als Umschalter dargestellt) der Ausgang Q auf den Eingang geschaltet, mit sehr kurzer Signallaufzeit. Die Taktflanke, beim Xilinx immer wirksam, hier immer wirksam mit dem Berechnungsschritt, übernimmt also den Ausgang selbst wieder, was zu keiner Änderung führt. Ist CE=1, dann wirkt der D-Eingang. Man kann also mit einer CE-Freigabe den Takt untersetzen, damit mehr Laufzeit in der FPGA-Logik zulässig ist als eine Systemtaktperiode. Es ist keinesfalls so, dass mit CE der Takt AND-verknüpft ist. Das funktioniert nicht. Der CE-Freigabetakt ist also jeweils für genau eine Systemtaktperiode auf Hi, wie es im Scope oben sichtbar ist.

15. Simulink-Statemaschinen-Realisierungen

In Mathworks-Simulink sind Statemaschinen mit dem entsprechenden Package „Stateflow“ für die Modellierung verfügbar.

Eine einfache Statemaschine ist conditional-orientiert, wird also in jeder Abtastzeit aufgerufen und verarbeitet Bedingungen an seinem Eingang, siehe *6 Unterschied zwischen eventgetriebenen und conditional Software-Statemaschinen,* Seite 8. Diese einfache Statemaschine kann als Mealy oder Moore *State Machine Type* deklariert werden. Sie kann aber auch unter der Bezeichnung *classic* wie eine Harel-Statemachine designed werden (syntaktisch).

Man kann die Statemaschine mit **Event-Inputs** versehen. Wenn das Event bzw. mehrere Events als ***Function call*** deklariert werden (nicht als edge, rising oder falling), dann wird der Statechart-Fblock adäquat wie ein ***„Function-call Subsystem“*** behandelt, also nur aufgerufen wenn ein Event vorliegt. Im Statechart kann man dann die Events und zusätzliche Bedingungen als guards in den Transitions abfragen. Wenn man eine Event-Queue geeignet außerhalb realisiert und den Abarbeitungstrigger mit einem Event setzt, zusammen mit den Daten, dann erhält man die *eventgetriebene Statemaschine* wie sie in /omg-UML/ beschrieben ist.

Allerdings gibt es einige Detailunterschiede. So werden bei Nested-Statemaschinen entgegen der /omg-UML/ -Norm Events zuerst außen appliziert, dann innen.

Statemaschinen können nun auch einen Message-Input haben, wobei in der Statemaschine eine Message-Queue realisiert ist (entspricht der Event-Queue). Es kann vermutet werden, dass der Anwender, der eine Event-Queue kennt, diese bei den Simulink-Statemaschinen zunächst vermisst hat, und daher als Lösung in einem Release die Message-Queue bekommen hat.

15.1 Conditional Statemachines – Abarbeitung im Berechnungszyklus

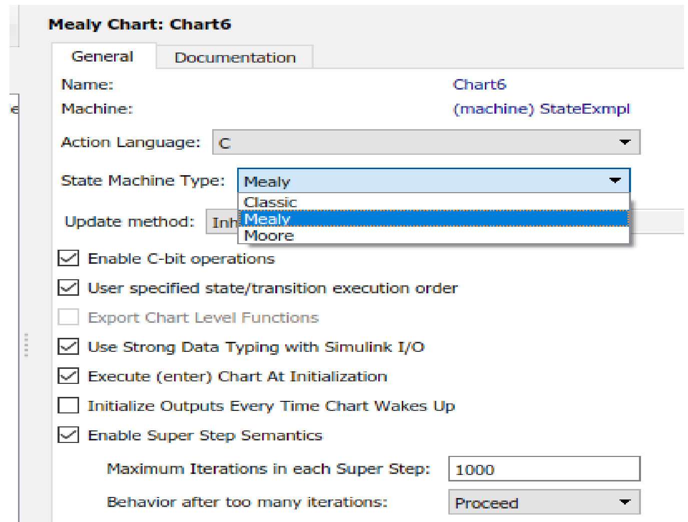
Conditional soll auf deutsch etwa *bedingungsgesteuert* heißen, allerdings ist der englische („denglische“) Begriffe schlichtweg kürzer und jeder versteht ihn.

In Simulink sind die klassischen *conditional Statemachines* die einfachste Art Statemachines zu modellieren, auch was die Codegenerierung betrifft. Man ist auch sehr nahe an Hardwarerealisierungen, was für Simulink-Modelle mit Ziel: FPGA-Codes wichtig ist. Die Abarbeitung erfolgt in der angegebenen Abtastzeit oder über *„inherit“* in der Abtastzeit der Eingangssignale. Als Ausgangssignale können boolean erzeugt werden, aber auch beliebige andere Werte sind setzbar. Die Statecharts in Simulink können interne Variable speichern, wodurch ein Statechart-Subsystem mit einer Instanziierung einer class verglichen werden kann: In UML kann jede class ein „behaviour“ haben, was mit dem Statechart zur class ausgedrückt ist. In Simulink kann ein Statechart lokale Variable definieren, was in der UML die class-Instanzvariablen sind. Außer dass in UML noch andere im Konkretfall ggf. konkret unnötige Dinge mit classes ausgeführt werden können (beliebige weitere Operations neben dem Statechart) läuft es irgendwie auf ein Gleiches hinaus.

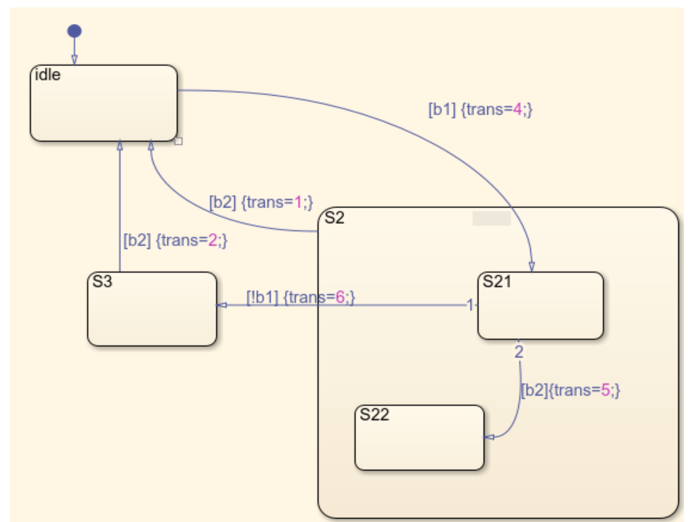
Bei den klassischen conditional Statemachines wird zwischen Mealy und Moore unterschieden.

Auch bei der conditional Statemachines kann eingestellt werden, ob pro Rechenzyklus nur ein Stateswitch bearbeitet wird oder ob die Bearbeitung im Rechenzyklus solange wiederholt wird, bis ein stabiler Zustand erreicht ist, entspricht dem *„Run to completion“*-Paradigma. Das ist das Setting *„Enable super step semantic“*. Wenn enabled ist, kann angegeben werden was passiert, wenn kein stabiler Zustand erreicht wird: Maximale Anzahl von Zyklen, Verhalten bei Fehlern.

15.2 Mealy- und Moore-Statemaschinen in Simulink



In den Settings zu einem Statechart-Fblock kann als „*State Machine Type*“ gewählt werden zwischen *Classic*, *Mealy* und *Moore*. *Classic* ist dabei die Notation wie bei einer Harel-Statemaschine, wobei in der Dokumentation auch explizit *Harel* erwähnt wird, nicht aber in den Settings. Die Bezeichnung Classic ist dafür eventuell etwas irreführend.



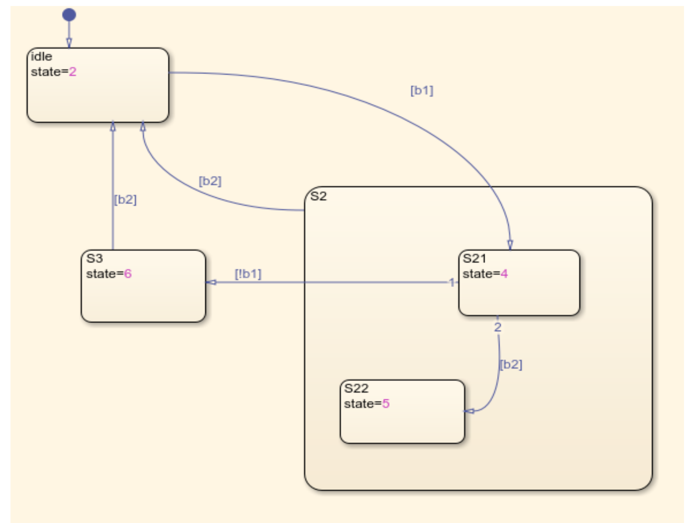
Die Mealy- und Moore-Einstellung ist dagegen echt classic, also so wie man es von Hardware-State-Darstellungen außerhalb des Harel-Ansatzes kennt. Die Syntax ist eingeschränkt. Man kann zwar mit genau einem Event arbeiten, dass als ***„Function-call Subsystem“*** wirkt, kann aber weder „Transition Action“ noch Entry- und Exit-Actions in den States ausführen. Syntaktisch wird zwischen einer *„Transition Action“* unterschieden, die bei classic (eigentlich Harel) mögich ist:

```
event [ condition ] / { assignment }
```

und einer *„Condition Action“* für Mealy-Statecharts, die wie folgt formuliert wird:

```
event [ condition ] { assignment }
```

Der Unterschied ist eigentlich nur das `/` als Trenner, ein syntaktischer Unterschied. Semantisch sind dies in Simulink aber zwei verschiedene Dinge. In Mealy-Statecharts sind nur die *„Condition Action“* zulässig. Die falsche Schreibweise wird mit einer entsprechenden Fehlermeldung angezeigt.



In Moore-Statemachines sind weder *Transition-* noch *Condition-Action* zulässsig, die Transition darf nur die Schaltbedingung enthalten. Statt dessen kann in einem State ein Assignment formuliert werden. Das ist also der klassische Mealy- und Moore-Ansatz:

* Bei Mealy werden die Ausgaben mit der Transition erzeugt.
* Bei Moore werden die Ausgaben mit dem Zustand (im State) erzeugt.

Dabei wird auch die zeitliche Bedingung eingehalten: Die Mealy-Ausgabe in der Transition wird in dem Rechenzyklus auf den Ausgang gelegt, der das Schalten veranlasst. Die Moore-Ausgabe wird dagegen erst im folgenden Rechenzyklus auf den Ausgang gelegt, der zum umgeschalteten State gehört. Das betrifft die Ausgabe in der Simulation oder in einer Software-Codegenerierung.

Der eigentliche Unterschied zwischen Mealy und Moore in einer Hardwareimplementierung (also auf Simulink bezogen für eine FPGA-Codegenerierung) liegt aber in folgendem:

* Sowohl Mealy als auch Moore schalten ihren Zustand abhängig vom aktuellem Zustand und den Eingängen:

```
q[t+1] = f_fn ( q[t], u[t] )
```

* Mealy erzeugt die Ausgabe kombinatorisch aus den vorhandenen States und den Inputs:

```
y[t] = g_fn ( q[t], u[t] )
```

Wechseln also die Inputs vor dem folgenden Schalt-Takt des Zustandes, dann wird unverzüglich (kombinatorisch) die neue Ausgabe erzeugt. In der obigen Gleichung steht dafür der selbe Zeitpunkt `[t]`. In einer Hardwarerealisierung entstehen dabei Laufzeiteffekte, da sich *„unverzüglich“* auf die Taktung des Systems bezieht. Gegen Laufzeiteffekte, die im Negativfall auch als *Hazard*, *Race condition* bekannt sind, kann man nichts machen, sie entstehen mit der Verknüpfungsabarbeitung. Will man diese Signale auswerten, dann ist stark anzuraten, diese einzusynchronisieren. Mit einer Taktflanke werden sie also wie bei einem Moore-Statechart damit erst zur folgenden Taktzeit nutzbar. Der Unterschied zu Moore bleibt aber: Sie hängen zusätzlich vom Input ab.

```
y1[t+1] = y[t]
```

oder

```
y1[t+1] = h_fn(y[t], u[t], q[t])  //uses additionally any other combinatoric
```

In Hardwareschaltungen ist das `y[t]` also nicht wirklich nutzbar.

* Moore erzeugt die Ausgabe nur aus dem Zustand, ohne Berücksichtigung der Inputsignal-Situation:

```
y[t] = g_fn ( q[t] )
```

beziehungsweise, da `q[t]` der alte Zustand ist, der möglicherweise auch identisch mit `q[t-n]` ist:

```
y[t+1] = g_fn ( q[t+1] )
```

als Reaktion auf die Eingabeänderung.

Es ist bei einer Hardwarerealisierung mit Moore-Automaten anzuraten, die Flipflop-Ausgänge direkt auf die genutzten Ausgänge zu legen, also

```
y[t] = q[t]
y[t+1] = q[t+1]
```

da sonst wieder aufgrund der Kombinatorik Race-Conditions entstehen können:

Bei

```
y1[t] = q1[t] || q2[t]     (oder-Verküpfung)
```

würde beim Schalten q1 von 1 auf 0 und q2 von 0 auf 1 möglicherweise y1 kurzzeitig auf 0 gehen, wenn q1 schneller als q2 verarbeitet wird. Ist ein solches Signal etwa an einen Leistungsschalter, ein Schütz oder ein Relais angeschlossen, dann würde das mechanische Schaltelement nicht zucken, da eine mechanische Trägheit besteht. Der möglicherweise sehr schnelle Ansteuerverstärker wird aber eine Störung (EMV) verursachen können, da er einen schnellen Impuls auf Leitungen ausgibt, oder er wird wärmer als nötig aufgrund der Schaltverluste. Das wäre für Hardwarerealisierungen zu beachten.

Die Hardwarerealisierung (für Mealy) unterscheidet sich nicht von der Software-Realsierung, wenn von dem Laufzeit/Hazard/Race-condition - Problem abgesehen wird. Die Software-Realisierung ist in Wirklichkeit eine zyklische Bearbeitung. Eine Ausgabe des Mealy-Statechart wird also im nachfolgenden Algorithmus verarbeitet (wie in der Hardware mit einer weiteren Verknüpfung), wird aber mit dem Speichern der berechneten Werte im Arbeitszyklus erst im folgenden Arbeitszyklus wirksam. Wenn eine Hardware-Ausgabe erfolgt, dann ist es eine Frage der Ausgabegestaltung, ob diese „so schnell wie möglich“ erfolgen soll, also direkt mit dem Ausgabebefehl in der Software, oder ob dabei eine Rasterung der Abarbeitungszeit berücksichtigt wird, was auf Softwareebene mit der noch folgenden ProzessBus-Übertragungen an Aktoren etc. sowieso erfolgt.

Der bedeutende Unterschied zwischen der Mealy-Realisierung in Simulink und der als Classic bezeichneten Harel-Ansatzes ist aber:

* Beim Harel-Ansatz wird auch eine Transition-Action erst mit dem Schalten des States wirksam, also immer im Folge-Takt. Dabei hängt die Ausgabe der Transition-Action vom Vor-Zustand und der Eingangsbedingung ab, also:

```
y[t+1] = g_fn ( q[t], u[t] ) )
```

Die Moore-Realisierung entspricht, außer der anderen Syntax, genau dem Harel-Ansatz: Die Ausgabe im State ist identisch mit einer entry-Action im State.

Schlussfolgernd wäre folgendes für die Anwendung von Moore- und Mealy-Statemaschinen in Simulink anzuraten:

* Da bei Mealy-Statemaschinen die synchronisierte Weiterverarbeitung der Ausgangssignale offen ist, damit die Mealy-Statemaschine nur einen Teilausschnitt einer gesamten Logik beschreibt, der für sich nicht in einem Hardware-FPGA existent sein kann (der FPGA-Router muss den vollständigen Pfad zwischen zwei Flipflops kennen um die Laufzeit zu berechnen), erscheint es angeraten, die Mealy-Statemaschinen nicht zu verwenden.
* Jede Mealy-Statemaschine mit einsynchronisierten Ausgängen (Flipflops an den Ausgängen) ist eine Moore-Statemaschine. Die Ausgangs-Flipflops gehören mit zur Gesamt-Statedefinition. Existieren ursprünglich mit Mealy zwei Transitionen mit unterschiedlichen Ausgaben zum selben State, dann führen diese zwei Transitionen unter Einbeziehung der Ausgangsflipfloos zu zwei verschiedenen States. Ein gemeinsamen umfassenden State (als Nested-Statemachine) repräsentiert dann den ursprünglichen Mealy-State, von dem auch die entsprechenden Abgänge der Transitionen designed werden können.

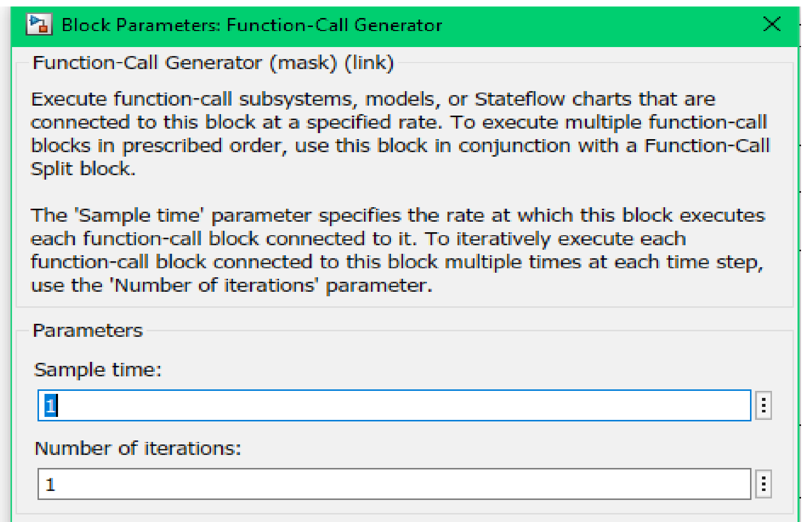
* Eine Moore-Statemaschine ist sematisch identisch bzw. eine Untermenge einer Harel-Statemaschine. Anstatt eine Moore-Statemachine im Simulink sollte einheitlich die als „Classic“ bezeichnete moderne Haral-like Variante verwendet werden.
* Es ist jedoch zu klären, ob die Codegenerierung für FPGA mit Harel (Classic)-Statecharts im Simulink gleiche Ergebnisse wie die Verwendung von Moore-Automaten bringt. Erfahrungen dazu könnten mittelfristig von mir eingeholt werden (FPGA-Coder ist zugänglich)

15.3 Statemachines mit Events

Statemachines können in Simulink ähnlich designed werden wie in UML mit Events und zugehörigen Conditions (guards), wobei *„Run-to-completion“* berücksichtigt wird.

In Simulink werden die Events adäquat dem *„Function call“* verstanden (die andere angebotene Einstellmöglichkeit ist: als Flanken von Boolean-Signalen). Hier soll nur die *Function call*-Variante interessieren.

Die *Function call*-Herangehensweise (*approach*) ist außerhalb von Statecharts mit dem *„Function Call Subsystem“* gegeben. Ein Function call kann mit besonderen Ausgängen aus Fblocks ausgelöst werden. Das ist insbesondere auch in S-Functions möglich (also aus der C-Realisierungsebene von Fblocks) oder beispielsweise mit einem rechtsstehenden Spezial-Fblock.

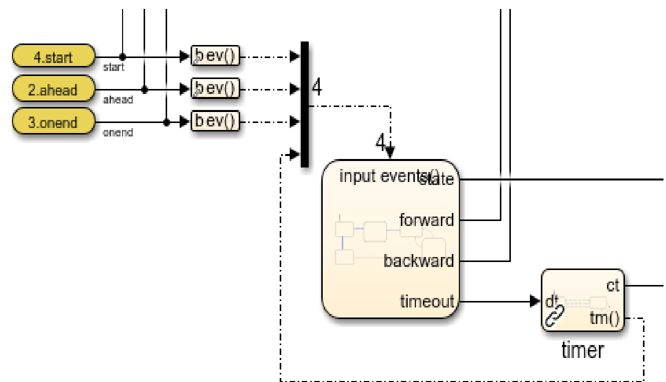


Das Function call bedeutet grundsätzlich, dass der damit verbundene Fblock nicht in einer festen Abtastzeit bearbeitet wird, sondern nur wenn das Function call gesetzt ist. Ob die Bearbeitung faktisch bedingt dennoch in einem festen Zyklus aufgerufen wird oder auch anderweitig einordenbar ist, ist eine Frage der Codegenerierung. Wenn das Function call in einer eignen Subroutine steht und die Aufruforganisation dieser Subroutine in einer anderen Subroutine liegt, die aber gar nicht aufgerufen zu werden braucht, dann kann man mit der manuellen Rahmen-Programmierung des generierten C-Routinen diese Function-call auch beliebig organisieren, beispielsweise in einem Thread der eine Event-Queue ausräumt. Dann ist man mit den Event-Statemachines gleichauf mit der Realisierungsstratgie, die auch bei UML oder auch manuellen event-Statemachines verwendet wird. Die Event-Queue ist bei den Event-gesteuerten Statemachines in Simulink selbst nicht dabei (anders als bei Message-controlled Statemachines, siehe Folgekapitel). Bei der Codegenerierung gibt es einige Einstellmöglichkeiten, die zu beachten wären.

15.4 Time-gesteuerte Transitionen in Simulink-Statemaschinen

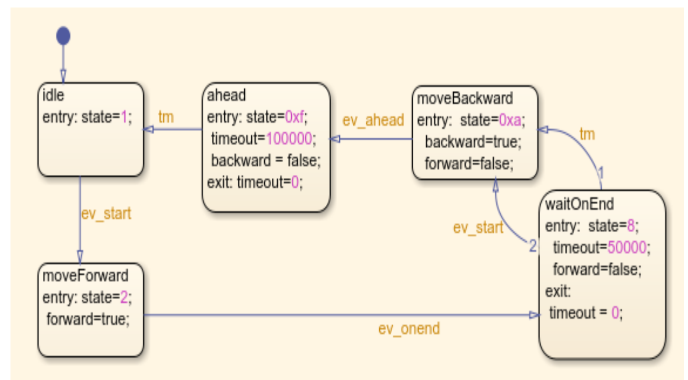
TODO. Es gibt ein Construct `after(10, sec)` etc. das aber nur für zyklisch gerufene Statecharts funktioniert.

Es sei verwiesen auf https://de.mathworks.com/help/stateflow/ug/using-temporal-logic-in-state-actions-and-transitions.html. In dieser Doku wird explizit erwähnt dass in einem Enabled Subsystem der after-temporal transition erst dann schaltet, wenn das Subsystem wieder enabled ist, also meist später als die angegebene Zeit. Ein Statechart mit Event-Inputs nach Error: Reference source not found Error: Reference source not found Seite Error: Reference source not found ist ebenfalls ein Enabled Subsystem, es wird aufgeweckt wenn das Event kommt. Das entspricht der Strategie der Eventverarbeitung in IEC-61499. Damit muss für das Timeout aber ein Time-Event kommen. Das Konstrukt mit `after` ist dafür nicht geeignet, es ist für zyklisch gerufene Statecharts geeignet.



Für das in UML gängige tm(time)-Event gibt es keine direkte Entsprechung. Man kann aber das tm-Event durch einen Timer außen erzeugen, wie dies bei IEC-61499 derzeit auch ist:

Der Timer ist ein Library-Statechart, dass selbst nicht eventgesteuert ist, folglich im Berechnungszyklus läuft, aber ein tm()-Event erzeugt. Abhängig von der Belegung dt = 0 ist der Timer gestoppt, oder er läuft bis zum vorgegebenen dt und erzeugt danach im Berechnungszyklus das Timerevent.



Die Statemachine die das Timeout braucht, hat den timeout-Ausgang und empfängt das tm-Event, ist also wie im obigen Bild erkennbar eventgetriggert. Das Bild rechts stellt die Statemachine innen dar. Das tm ist ein event für Timeout. Die Zeiten werden in entry: angegeben (ab dem Eintritt läuft das timeout), im exit wird das timeout auf 0 gesetzt, damit ein unerwartetes Timeout-Event verhindert wird, falls der State anderweitig verlassen wird. Dieser Zusatzaufwand ist notwendig, ansonsten verhält sich das Statechart wie bei einer Harel-Statemachine. Bei der Implementierung direkt mit Angabe `tm(time)` muss die Implementierung der Statemachine genau diese Aktionen intern ausführen, die hier explizit angegeben sind: Timer initialisieren bei Eintritt und Abschalten bei Austritt. Zu beachten ist weiterhin, dass es bei parallelen Statemachines mehrere Timer geben muss, einen für jeden Parallelzweig, wenn es parallele Timeouts gibt.

Diese Statemachine hat zwei tm-Übergänge, dafür ist nur ein Timer notwendig.

16. Literatur, Verweis auf die Standard-Dokumente

/omg-UML/: OMG Unified Modeling Language TM (OMG UML) Version 2.5,
OMG Document Number formal/2015-03-01
Normative Reference: http://www.omg.org/spec/UML/2.5

/Harel/ David Harel: *„Statecharts: A visual Formalism for complex systems*“*
in Science of Computing Programming 8 (1987) 231-274 North Holland
0167-6423/87/$3.50 (C) 1987, Elsevier Science Publishers B.V. (North Holland)

/61499/: 65B/845/FDIS FINAL DRAFT INTERNATIONAL STANDARD,
Project number IEC 61499-1/Ed.2, Supersedes document 65B/799/CDV, 65B/817/RVC
Title: IEC 61499-1/Ed.2: Function blocks - Part 1: Architecture