1. Modellierung von UML State Machines für GeneSEZ

Diese Anleitung konzentriert sich auf Unterschiede zwischen dem UML- und dem GeneSEZ-Metamodell sowie für die Modellierung mit der GeneSEZ Generator Framework wichtige Punkte und versteht sich nicht als allgemeines Tutorial zum Thema GeneSEZ oder State Machines. Alle Angaben beziehen sich auf Behavioral State Machines, da Protocol State Machines aktuell nicht unterstützt werden. Da für viele Notationselemente der UML State Machines entweder keine oder mehrere deutsche Bezeichnungen existieren, wird in diesem Tutorial aus Gründen der Eindeutigkeit auf die englischen Begriffe zurückgegriffen. Wo gängige deutsche Bezeichnungen vorhanden sind, werden diese beim ersten Auftreten des englischen Begriffes in Klammern erwähnt. Das UML-Modell wird mit der Software "MagicDraw 16.0" erstellt.

1.1. Walkthrough am Beispiel

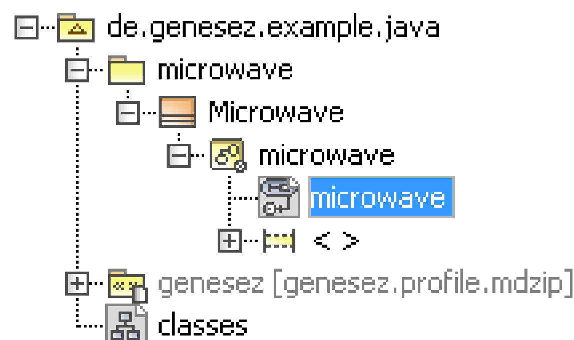
State Machines (in UML 1.x Statecharts genannt) oder Zustandsautomaten sind besonders sinnvoll, um das Verhalten beliebiger Classifier abzubilden. Im Folgenden wird ein Beispiel-Zustandsautomat eingeführt, der beschreibt, in welchen Zuständen sich ein Objekt der Klasse `Microwave` befinden kann und welche Ereignisse zu diesen Zuständen führen.

1.1.1. Aufbau von State Machine Projekten

Da zur Generierung mit der GeneSEZ-Software eine State Machine immer einem Classifier zugeordnet sein muss, wird zunächst ein Klassendiagramm angelegt, welches die Klasse `Microwave` enthält. Ein Classifier kann immer nur eine State Machine enthalten. Die Definition von Attributen oder Operationen ist nicht notwendig, wenn diese nicht für die weitere Arbeit benötigt werden. Danach wird für diese Klasse ein neues State Machine Diagramm kreiert. Dessen Name kann, muss aber nicht, mit dem Namen des Classifiers übereinstimmen; er ist für die spätere Generierung nicht von Bedeutung.

Nach diesen Arbeitsschritten sollte der Containment-Tree folgende Struktur aufweisen:

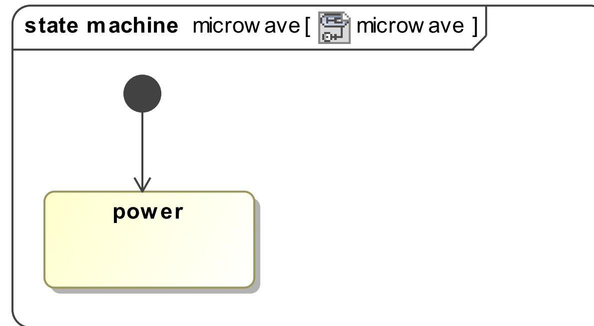
Figure 1. Der Containment-Tree nach Anlegen der benötigten Diagramme



Oberklassen dürfen keine State Machines enthalten, wenn die von ihnen abgeleiteten Klassen eigene Zustandsautomaten besitzen. Sollte unsere Klasse `Microwave` also von einer Oberklasse `Kitchenware` erben, so dürfte diese keine eigene State Machine erhalten. Umgekehrt gilt dies genauso für Unterklassen, an deren Elternklasse bereits eine Zustandsautomat existiert.

Jede State Machine benötigt zwingend genau einen Initial State (Startzustand), welcher die Erzeugung des beschriebenen Classifiers darstellt. Die Modellierung beginnt also mit dem Initialzustand und einem Übergang zu jenem Zustand, in dem sich der Classifier nach der Erzeugung befindet. Die Transition darf weder Trigger noch Guard enthalten. Im Falle des ausgewählten Beispiels geht die Mikrowelle in den Zustand `power` über, der anzeigt, dass das Gerät aktuell mit Strom versorgt wird - sonst wäre es nicht eingeschaltet.

Figure 2. Initial State einer State Machine

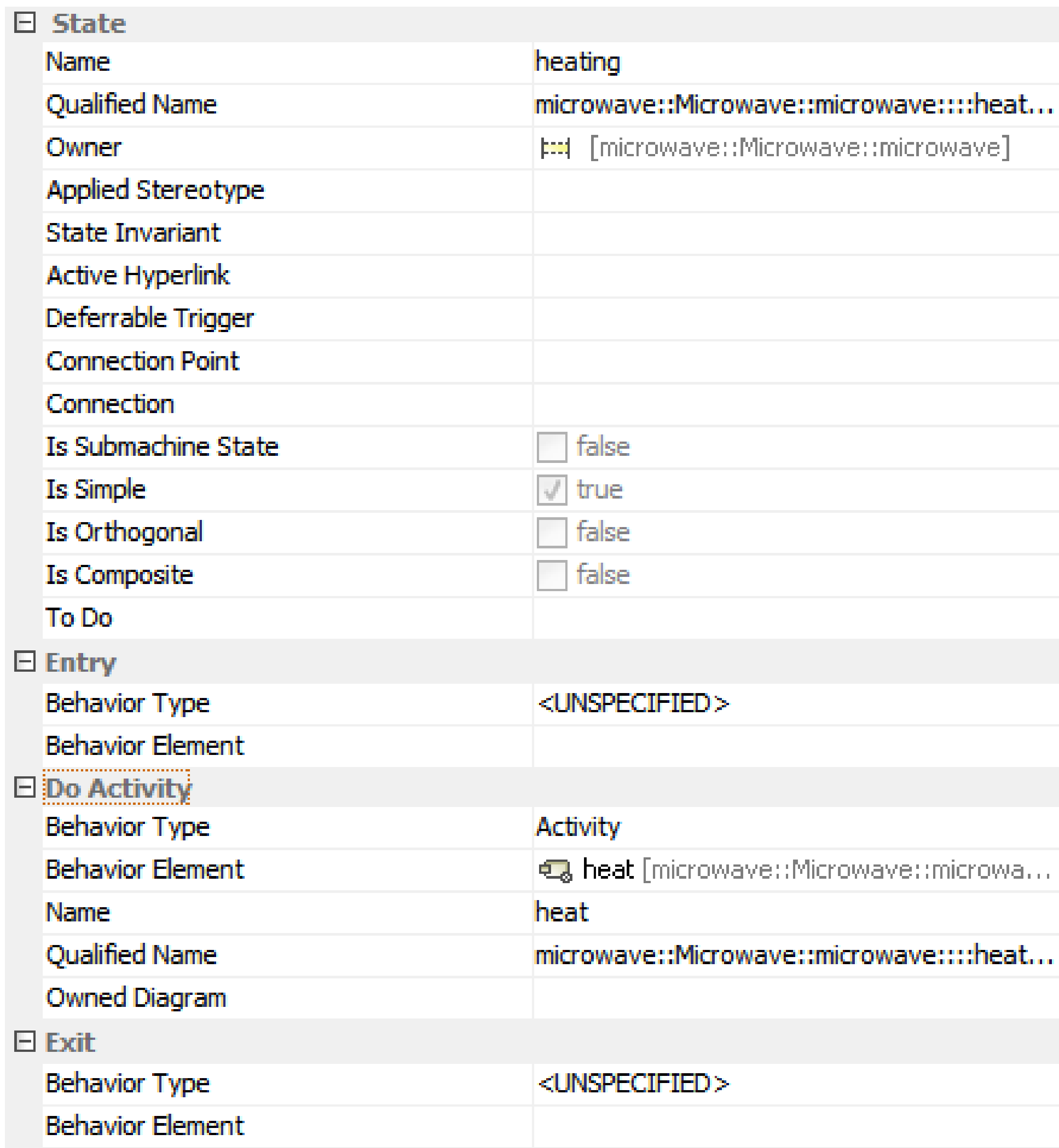


1.1.2. State

Ausgehend von diesem Zustand soll die Mikrowelle nun in Betrieb genommen werden, also in einen Zustand `heating` übergehen. Zunächst wird nur dieser modelliert, die Übergänge werden später betrachtet. Jeder Zustand kann eine Entry-, Do- und/oder Exit-Behavior (Verhalten) besitzen. Die UML unterscheidet an dieser Stelle zwischen verschiedenen Arten von Verhalten, vom GeneSEZ Generator Framework wird allerdings jede Art als Methode, bzw. im Falle eines Do-Behaviors als Thread, umgesetzt. Es empfiehlt sich also, die modellierten Verhalten auf den Typ `Activity` zu beschränken. Jede Acitivity benötigt einen Namen, welcher weder Leer- noch Sonderzeichen enthält.

Solange sich die Mikrowelle im Zustand `heating` aufhält, soll sie aufwärmen, also wird diesem Zustand eine Do-Activity namens `heat` zugeordnet.

Figure 3. Ein Zustand mit einer Do-Activity

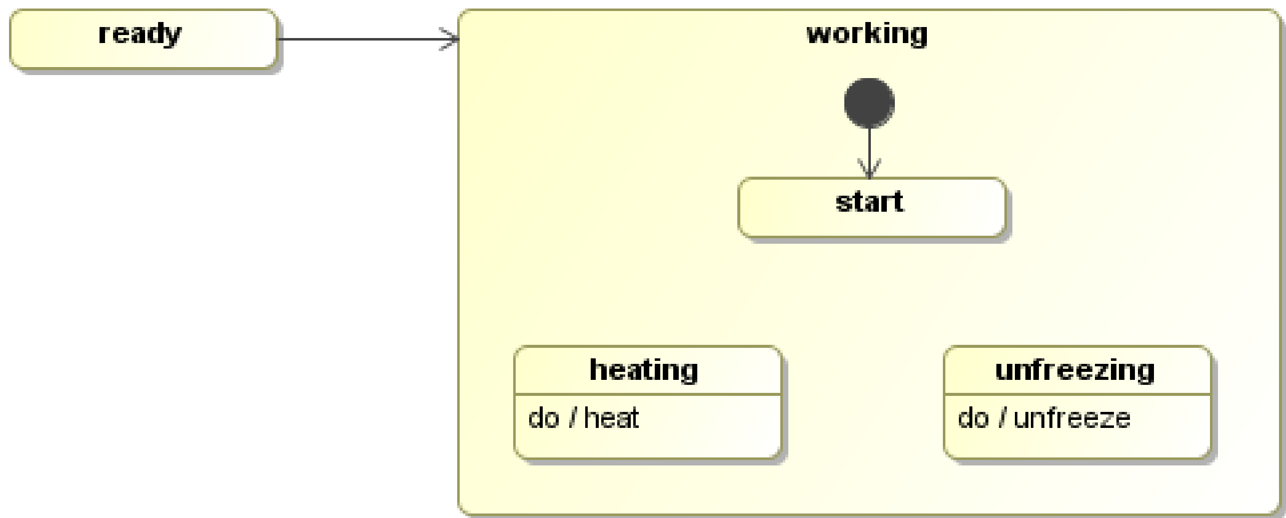


An dieser Stelle wird das Beispielmodell um einige States erweitert, da eine Mikrowelle mehr Funktionen als nur das Aufwärmen zur Verfügung stellt und auch nur im geschlossenen Zustand arbeiten sollte (und, um noch einige Notationselemente im Modell unterzubringen, auf die im weiteren Verlauf eingegangen werden soll). Es werden die Composite States (zusammengesetzte Zustände) `ready` und `working` hinzugefügt, welche die Zustände `opened`, `closed` und `evaluateDoorState` sowie `heating` und `unfreezing` enthalten.

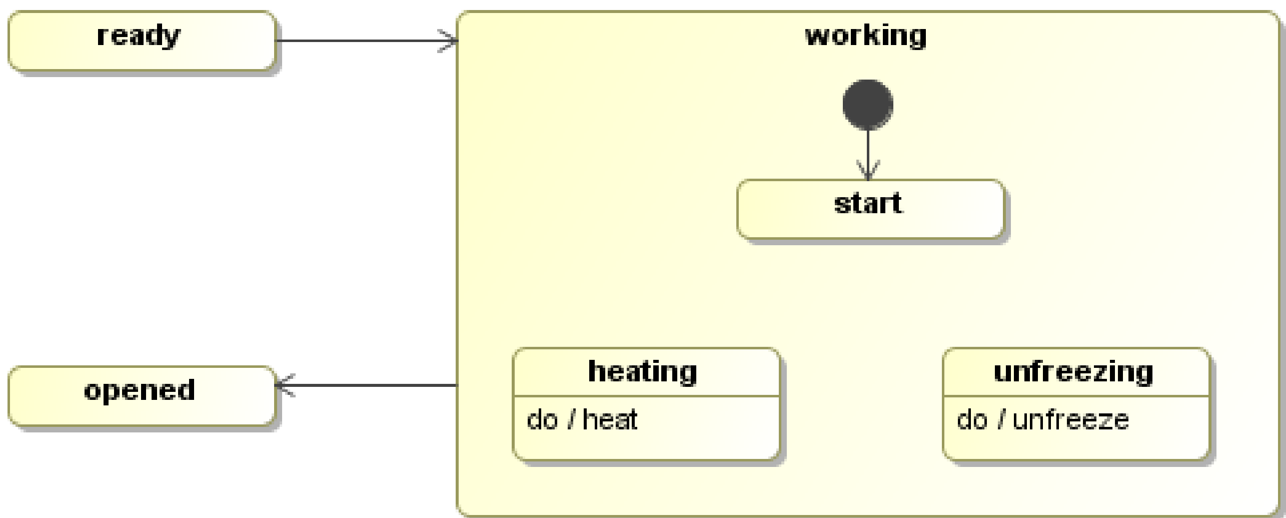
1.1.3. Composite State

Für zusammengesetzte Zustände oder Composite States gelten die selben Bestimmungen wie für States und auch sie können Behaviors enthalten. Zusätzlich benötigt jeder Composite State, anders als in der UML, einen internen Startzustand, auch, wenn nur Explicit Entries benutzt werden. In der UML ist es möglich, einen internen Startzustand mit einer Transition vom Rand des Composite State zum ersten aufgerufenen Zustand zu modellieren; dies wird allerdings von GeneSEZ nicht als Initialzustand erkannt. Die Modellierung von Explicit Entries oder Exits ist möglich, auch über mehrere Ebenen.

Eine Transition von einem beliebigen Zustand zum Rand des Composite State wird wie in der UML als Aufruf des internen Initialzustandes erkannt und von GeneSEZ als Transition zwischen diesen beiden States generiert.



Eine Transition vom Rand des Composite State zu einem beliebigen Zustand außerhalb wird von GeneSEZ bei der Generierung in mehrere Transitions umgesetzt - eine ausgehend von jedem Zustand innerhalb des Composite State.



Nach Veranschaulichung der States werden nun Transitions und die verschiedenen Arten von Triggern betrachtet.

1.1.4. Transitions

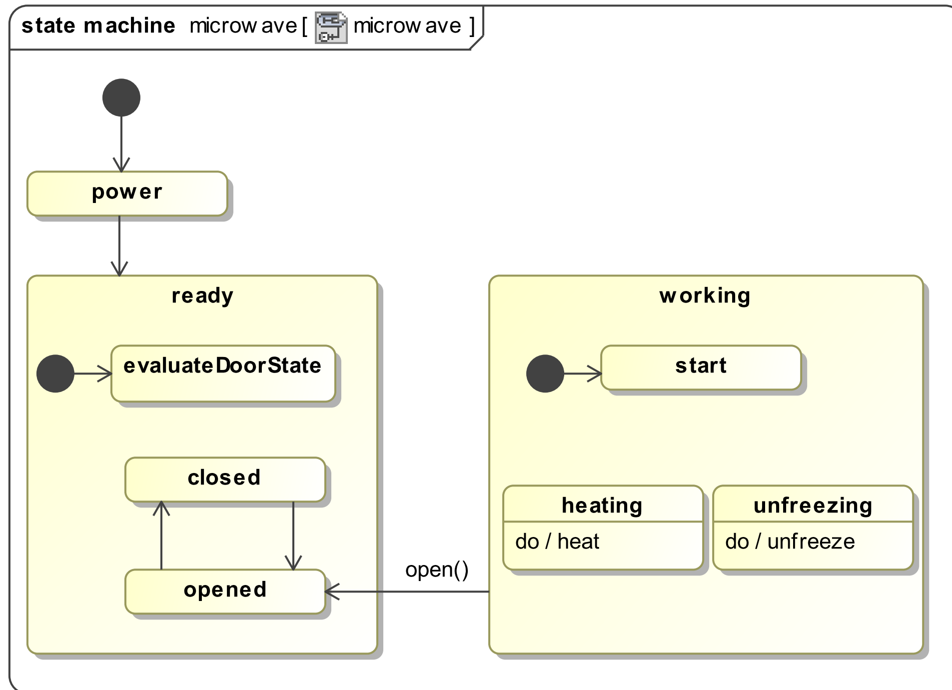
Der Name einer Transition wird beim Generieren durch GeneSEZ ignoriert, er dient also lediglich der besseren Übersicht während der Modellierung. Mehrere Trigger an einer Transition modelliert sind zwar laut UML möglich, im Gcore-Metamodell allerdings nicht enthalten.

CallTrigger

Die UML empfiehlt, CallTrigger nur in Protokollzustandsautomaten (Protocol State Machines) einzusetzen, ihre Modellierung in State Machines ist jedoch sowohl im UML- als auch im Gcore-Metamodell vorgesehen.

Im vorliegenden Beispiel kann es geschehen, dass die Mikrowelle während des laufenden Betriebs geöffnet wird. Die zugehörige Transition wird hier mit einem Trigger des EventTypes `CallEvent` modelliert. Jedem CallTrigger muss eine Operation zugewiesen werden, die an der State Machine oder deren Klasse definiert ist. Mehrere Transitions innerhalb einer State Machine können auf die selbe Operation zugreifen. Sie wird von GeneSEZ als Event realisiert.

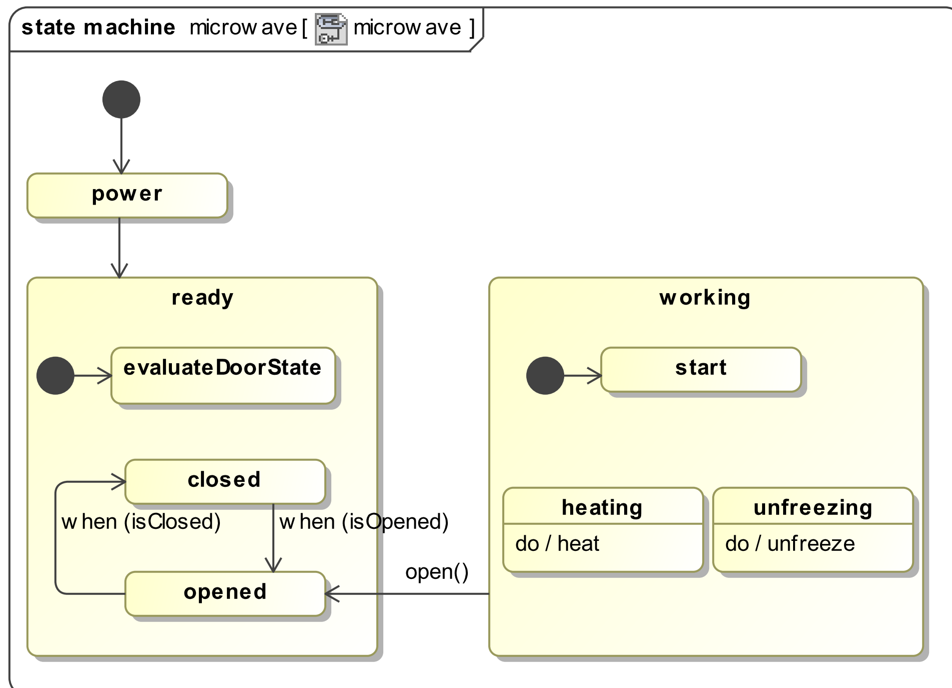
Figure 4. CallTrigger im Modell



ChangeTrigger

Die Verwendung von ChangeTriggern wird hier beispielhaft am Übergang des Objektes vom Zustand `closed` in den Zustand `opened` und anders herum dargestellt. Ein Trigger vom EventType `ChangeEvent` benötigt eine Change Expression. Die Expression wird von GeneSEZ, wie beim CallTrigger, als Event realisiert.

Figure 5. ChangeTrigger im Modell



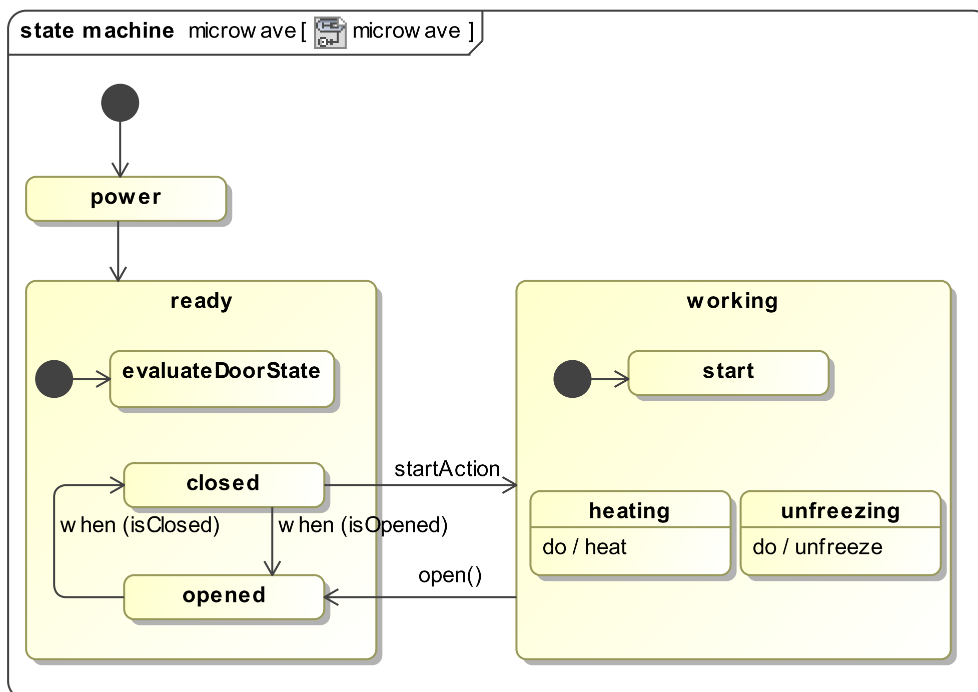
SignalTrigger

SignalTrigger sind die am häufigsten in State Machines verwendeten Arten von Triggern. Sie benötigen ein zugehöriges Signal, welches immer an der State Machine definiert wird, die sie verwendet.

Der Zugriff auf Signals anderer State Machines im selben Package ist möglich, allerdings wird dies bei der Generierung durch GeneSEZ nicht umgesetzt, da im generierten Quellcode jede State Machine ihre eigenen Signals enthält. Im Sinne eines übersichtlichen Modells sollte also jedes in der State Machine verwendete Signal auch an dieser definiert werden.

Im vorliegenden Beispiel wird die Verwendung von SignalTriggern am Übergang vom Zustand `closed` in den Zustand `working` demonstriert - weil eine Mikrowelle nur arbeiten sollte, wenn sie geschlossen ist.

Figure 6. Ein SignalTrigger im Modell



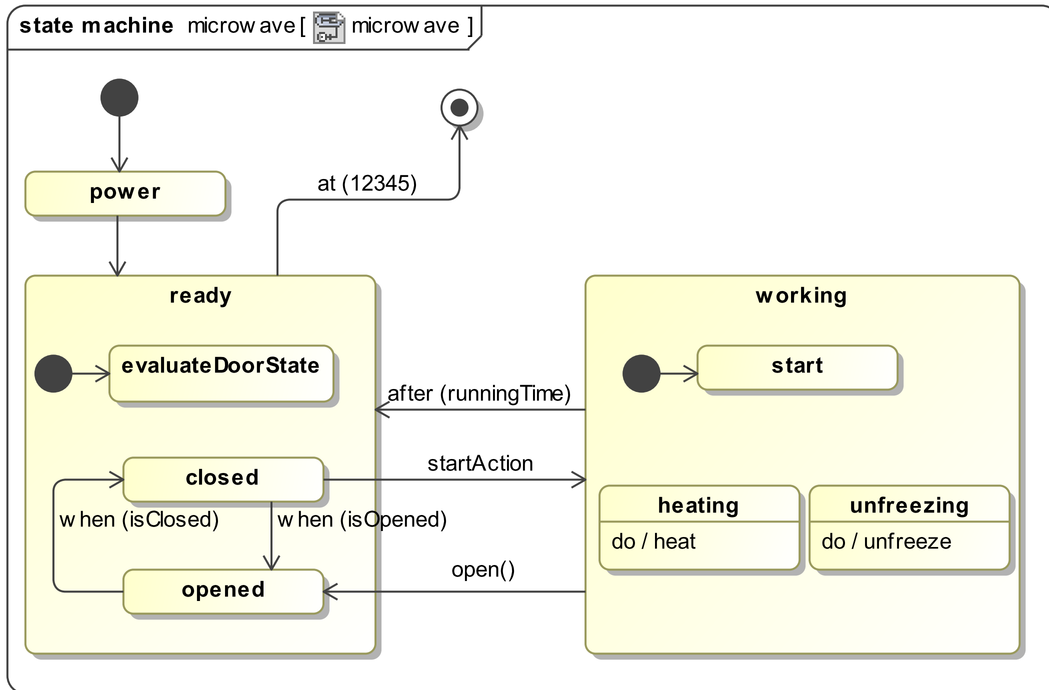
TimeTrigger

TimeTrigger sind die einzigen Trigger, welche zwingend einen Namen erhalten müssen, da dieser bei der Generierung als Name des Events verwendet wird. Aus diesem Grund muss der Name innerhalb des Modells eindeutig sein und den Namenskonventionen für Klassennamen genügen.

GeneSEZ unterstützt die Arbeit mit den TimeTrigger-Typen `non-relative/absolute` sowie `relative`. Die Zeitangabe erfolgt in Millisekunden; es ist jedoch auch möglich, Zeichenketten zu verwenden und diese später im generierten Quellcode als Variable zu deklarieren.

Am Beispiel der Mikrowelle soll das System zu einem fest definierten Zeitpunkt in den Endzustand übergehen; außerdem soll der Zustand "working" verlassen werden, wenn eine vom Nutzer festgelegte Zeitspanne verstrichen ist.

Figure 7. TimeTrigger im Modell



AnyTrigger

MagicDraw modelliert AnyTrigger als Trigger vom Typ `AnyReceiveEvent`, diese sind jedoch im GeneSEZ-Metamodell nicht enthalten. Hier wird jede Transition, die keinen Trigger enthält, als AnyTrigger erkannt und generiert.

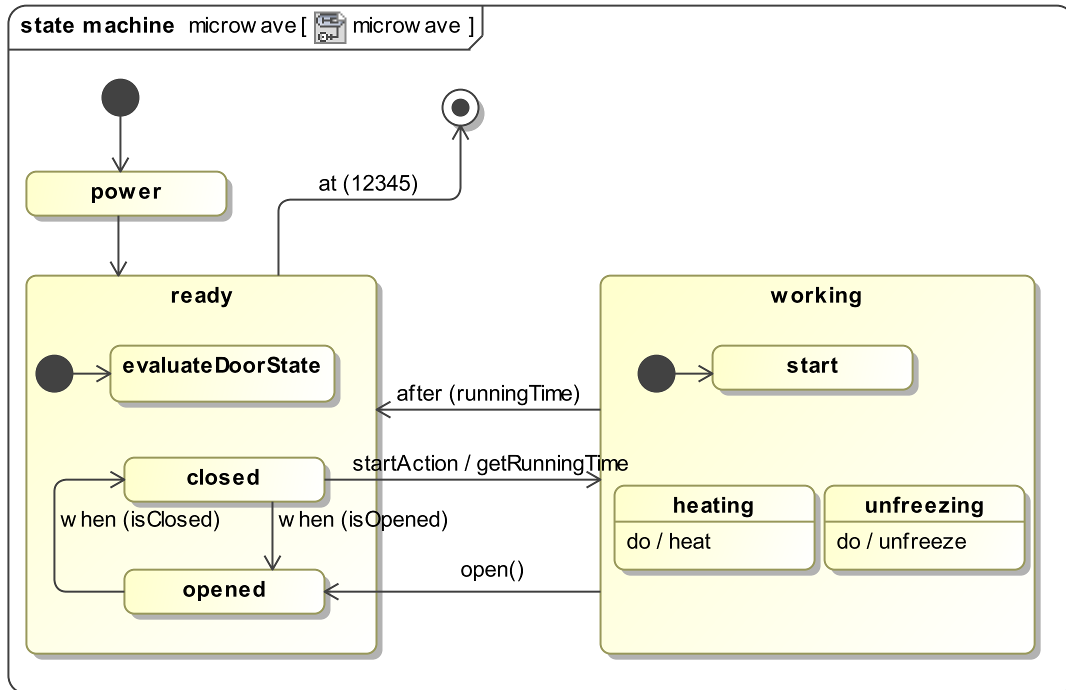
Der Übergang vom Zustand `power` in den Zustand `ready` stellt einen AnyTrigger dar.

Behavior

Jede Transition kann ein Behavior besitzen. Dessen Type sollte sich hier wieder auf `Activity` beschränken, da wie schon bei den Zuständen das Verhalten von GeneSEZ als Methode umgesetzt wird.

So erhält die Transition vom Zustand `closed` in den Zustand `working` ein Verhalten, welches die für den TimeTrigger benötigte Variable `runningTime` ausliest.

Figure 8. Transition mit Behavior



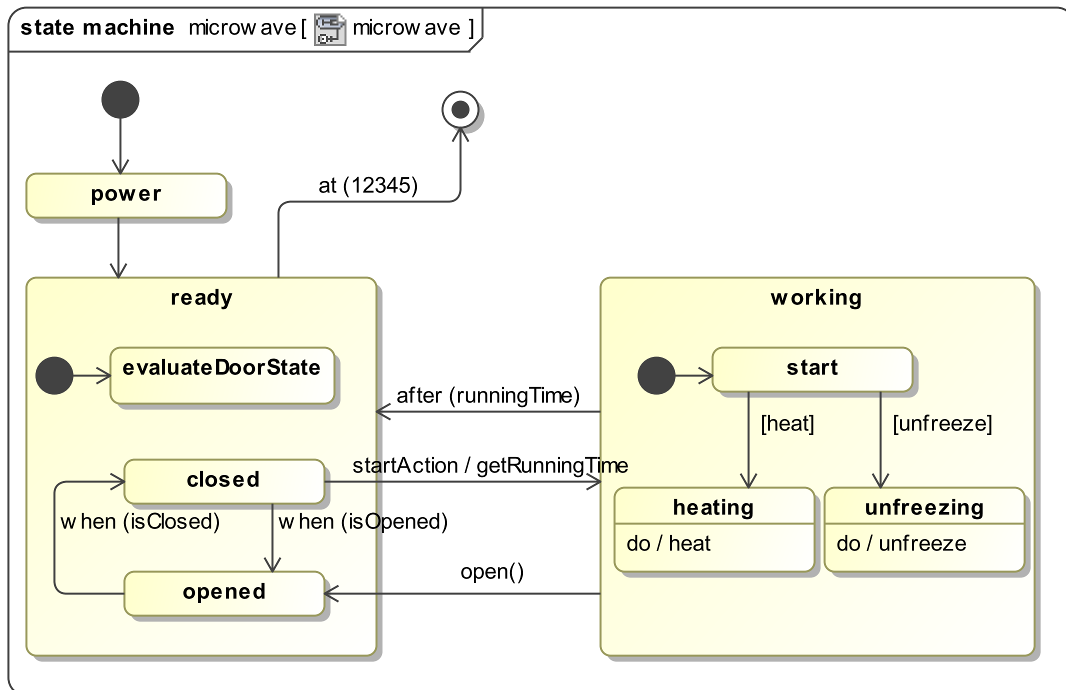
Guard

Das GeneSEZ-Metamodell kennt zwei Arten von Transitions: `Transition` und `GuardedTransition`. Jede Transition, die einen Guard besitzt, wird als `GuardedTransition` generiert. Hierbei wird allerdings Typ und Bezeichnung des Guard nur in den Kommentar der generierten Methode übernommen, die Elemente werden im Quellcode lediglich mit fortlaufenden Nummern, beginnend bei `guard0`, benannt.

Transitions mit gleichen Triggern und dem selben Quellzustand sind daher im Gcore-Metamodell nur vorgesehen, wenn alle oder alle bis auf eine Transition Guards besitzen. Hierbei können die Guards jedoch nicht auf Überschneidung überprüft werden.

Am Beispiel der Mikrowelle wird im zusammengesetzten Zustand `working` anhand von Guards entschieden, ob das Gerät aufwärmen oder auftauen soll.

Figure 9. Transition mit Guard



1.1.5. Pseudostates

Shallow und Deep History

Befindet sich die Mikrowelle im Zustand `working` und wird geöffnet, so wird die Arbeit unterbrochen. Sinnvoll für ein modernes Gerät wäre, die vorherige Aktion fortzusetzen, wenn die Tür wieder geschlossen wird. Dafür unterstützt GeneSEZ sowohl flache als auch tiefe Historien. Ein History State benötigt genau eine ausgehende Default Transition.

Final State

Final States (Endzustände) werden im GeneSEZ-Metamodell unterstützt, jedoch nicht zwingend in jeder State Machine benötigt.

1.1.6. Nicht unterstützte Elemente der UML und Alternativen

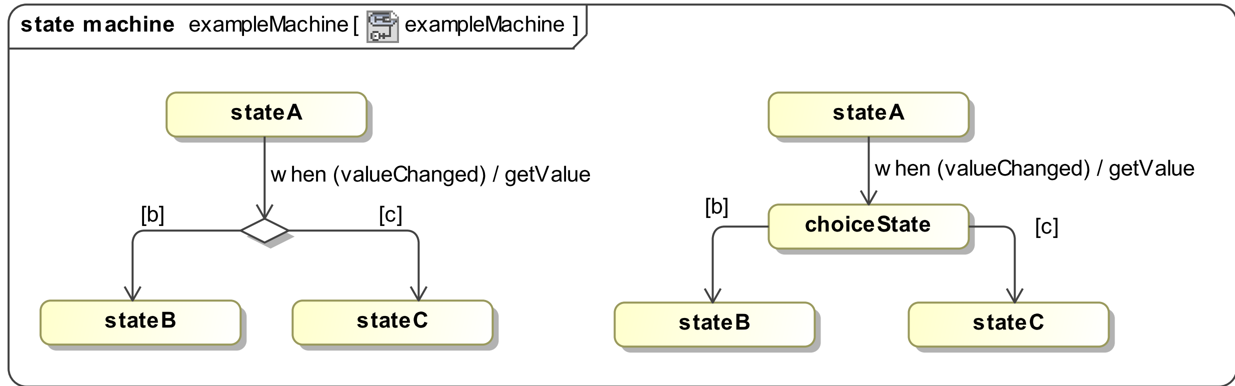
Entry/Exit Point, Fork/Join und Junction

Im Gcore-Metamodell nicht vorgesehen sind Entry/Exit Points (Ein-/Austrittspunkte), Forks und Joins (Gabelungen und Vereinigungen) sowie Junctions (Kreuzungen). Die zugehörigen Transitions sollten in jedem Fall einzeln modelliert werden.

Choice

Eine Alternative zu den nicht unterstützten Choices (Entscheidungen) ist, einen State zu benutzen, welcher den Choice ersetzt. Soll der Choice ein Verhalten beinhalten, so kann dieses entweder an der Transition zum ersetzenden Zustand oder an diesem selbst modelliert werden.

Figure 10. Ein Choice und seine Alternative im GeneSEZ-Metamodell



Terminator

Ein benötigter Terminator kann im GeneSEZ-Metamodell durch eine Transition zu einem Endzustand simuliert werden, welche eine Operation anstößt, die die Lebensdauer des beschriebenen Classifiers beendet.

1.2. Anhang: Übersichten

1.2.1. Alphabetische Übersicht über die Notationselemente

Table 1. Alphabetische Übersicht über die Notationselemente

Name	unterstützt	benötigt zwingend	Alternative
AnyTrigger	ja	Transition ohne Trigger	-
Behavior/Verhalten am State	ja	Name	-
Behavior/Verhalten am Composite State	ja	Name	-
Behavior/Verhalten an Transition	ja	Name	-
CallTrigger	ja	Operation	-
ChangeTrigger	ja	Change Expression	-
Choice/Entscheidung	nein	-	State anstelle des Choice modellieren
Composite State/ zusammengesetzter Zustand	ja	Name, interner Initial State	-
Deep History/Tiefe Historie	ja	ausgehende Default Transition	-
Entry Point/Eintrittspunkt	nein	-	Transitions einzeln modellieren
Exit Point/Austrittspunkt	nein	-	Transitions einzeln modellieren
Explicit Entry	ja	-	-
Explicit Exit	ja	-	-
Final State/Endzustand	ja	-	-
Fork/Gabelung	nein	-	Transitions einzeln modellieren

Name	unterstützt	benötigt zwingend	Alternative
Guard	ja	-	-
Initial State/Initialzustand	ja	-	-
Join/Vereinigung	nein	-	Transitions einzeln modellieren
Junction/Kreuzung	nein	-	Transitions einzeln modellieren
Shallow History/Flache Historie	ja	ausgehende Default Transition	-
SignalTrigger	ja	Signal, an der State Machine definiert	-
State/Zustand	ja	Name	-
Terminator	nein	-	Final State, Lebensdauer wird an eingehender Transition beendet
TimeTrigger (relative/non-relative)	ja	Name, Zeitangabe	-

1.2.2. Vollständiger Beispiel-Zustandsautomat

Um eine lauffähige State Machine zu erzeugen, wurde das Diagramm noch um einige Elemente erweitert. Der Zustand `power` wurde als Composite State modelliert, um zwei eingehende Transitions mit dem selben Trigger zum Final State zu vermeiden. Außerdem wurden in den Composite States `ready` und `working` jeweils Behavior realisiert, um die Werte abzufragen, welche später für die Guards verwendet werden. Beide Modellierungsarten realisieren hier die selbe Funktionalität.

Figure 11. Das vollständige Beispiel

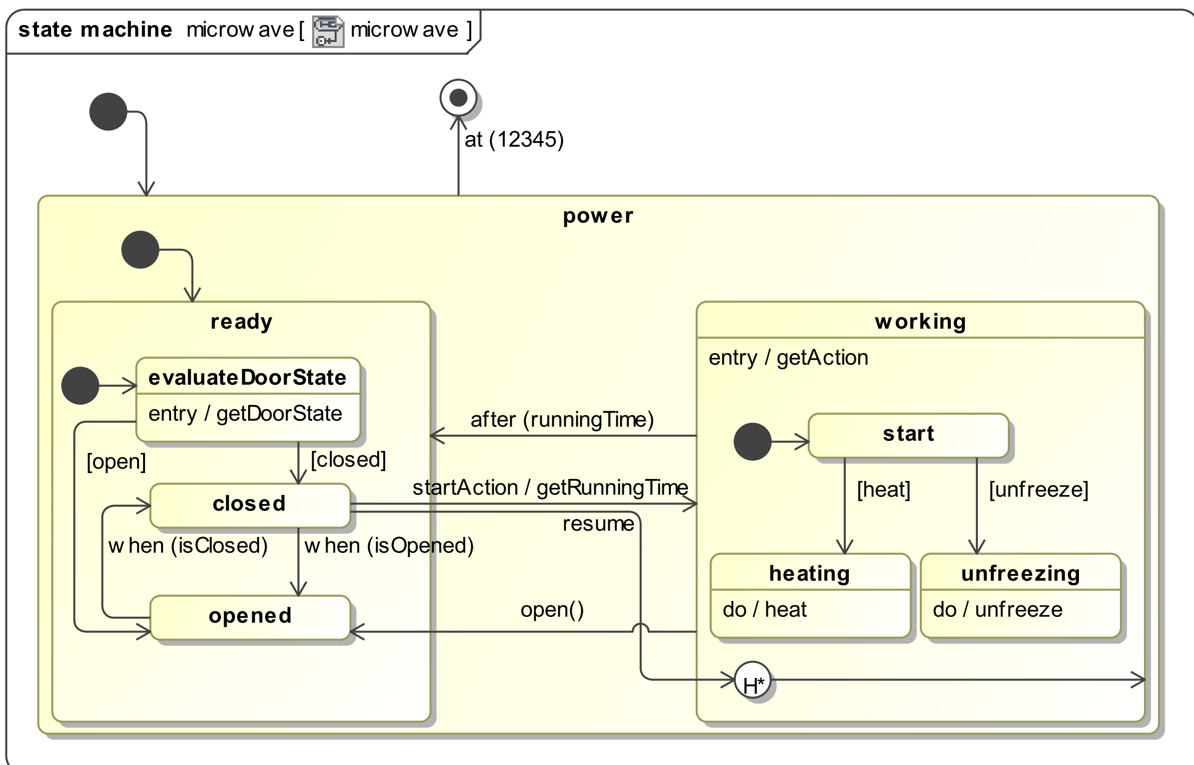


Table 2. Transitions des Beispiel-Zustandsautomaten

Quellzustand	Trigger	Guard	Behavior	Zielzustand
Initial	-	-	-	power
power	at(12345)	-	-	Final
working	open()	-	-	ready(opened)
ready(closed)	when(isOpened)	-	-	ready(opened)
ready(closed)	startAction	-	getRunningTime	working(Initial)
working(Initial)	-	-	-	working(start)
working(start)	-	action=heat	-	working(heating)
working(start)	-	action=unfreeze	-	working(unfreezing)
working	after(runningTime)	-	-	ready(Initial)
working	open()	-	-	ready(opened)
ready(opened)	when(isClosed)	-	-	ready(closed)
ready(closed)	resume	-	-	working(H*)
working(H*)	-	-	-	working(Initial)
ready(Initial)	-	-	-	ready(evaluateDoorState)
ready(evaluateDoorState)	-	doorState=open	ready(opened)	
ready(evaluateDoorState)	-	doorState=closed	-	ready(closed)