Universität Paderborn

Fachbereich 17 – Mathematik/Informatik

Arbeitsgruppe Softwaretechnik

Warburger Straße 100

D-33098 Paderborn

Generierung von Java Real-Time Code für zeitbehaftete UML Modelle

Diplomarbeit

für den integrierten Studiengang Ingenieurinformatik, Schwerpunkt Informatik
mit dem Zweitfach Elektrotechnik

im Rahmen des Hauptstudiums II

Sven Burmester

Kircherweg 9

33098 Paderborn

Matr.-Nr.: 3593909

vorgelegt bei

Dr. Holger Giese

und

Prof. Dr. Gregor Engels

Paderborn, im September 2002

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer, als der angegebenen, Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Ort, Datum

Unterschrift

Inhaltsverzeichnis

1. Einführung

Der Entwurfsprozess komplexer Softwaresysteme sieht eine Entwurfs- und eine Implementierungsphase vor. In der Entwurfsphase können Struktur, Abläufe und Verhalten eines Softwaresystems durch verschiedene Modellierungssprachen beschrieben werden. Die *Unified Modelling Language (UML)* [Obj01] stellt einen Standard zur Modellierung dieser unterschiedlichen Aspekte dar. In der anschließenden Implementierungsphase werden die Modelle in Quellcode einer Programmiersprache umgesetzt. Inzwischen existieren verschiedene Entwicklungsumgebungen, die diese Umsetzung „automatisch" anhand der Modellsemantik vornehmen. So ist z.B. in der Arbeitsgruppe Softwaretechnik an der Universität Paderborn das Case-Tool *Fujaba (From UML to Java and Back Again)* [Fuj] entstanden, das u.a. aus verschiedenen Diagrammtypen (Modellen) Java Code erzeugt. Bei der Modellierung wird zwischen Modellen unterschieden, die die Struktur der Software beschreiben und solchen, die das Verhalten einzelner Komponenten spezifizieren. Bei der Modellierung von Verhalten wird wiederum zwischen zeitbehaftetem und nicht-zeitbehaftetem Verhalten unterschieden. Bei dem zuletzt genannten ist neben dem Verhalten auch wichtig, in welcher Zeit reagiert wird. Systeme, für die ein bestimmtes zeitliches Verhalten gefordert wird, heißen zeitkritische Systeme. Typische Beispiele für zeitkritische Systeme sind Eingebettete oder Autonome Systeme (Roboter, Autos, Flugzeuge etc.), aber auch einfache Steuerungs- und Regelungsanlagen physikalischer Systeme. In einem Auto soll z.B. bei einem Unfall der Airback ausgelöst werden. Dabei ist es nicht nur wichtig, dass er geöffnet wird, sondern auch in welcher Zeit. Das folgende Teilkapitel zeigt ein, im Weiteren in dieser Arbeit genutztes, Beispiel für ein zeitkritisches System in Form einer Wasserbeckenregelung.

1.1. Beispiel eines zeitkritischen Systems

Abb. 1.1 zeigt ein Wasserbecken, dessen Wasserstand geregelt wird, um eine Wasserversorgung sicherzustellen. Über den Zufluss kann das Becken aufgefüllt werden, während es über den Abfluss geleert wird. Das Wasserbecken ist fest mit einer Regelungseinheit verbunden. Wird bei dieser über das Signal `WaterOn` Wasser angefordert, öffnet sie den Abfluss bis der Vorgang durch das Signal `WaterOff` beendet wird. Während der gesamten Zeit überprüft die Regelung den Wasserstand `h`. Damit sichergestellt ist, dass immer ausreichend Wasser vorhanden ist, ist festgelegt, dass der Zufluss geöffnet werden soll, sobald der Wasserstand ein bestimmtes Minimum (`MIN`) erreicht hat. Da das Becken auch nicht überlaufen soll, wird der Zufluss bei Erreichen eines Maximalwertes (`MAX`) geschlossen und der Abfluss geöffnet. Um zu berücksichtigen, dass das Überschreiten der Maximumsmarke in der Praxis mit einer gewissen Verzögerung registriert wird, und das Öffnen und Schließen der Ventiele Zeit benötigt, befindet sich die Maximumsmarke unterhalb des oberen Beckenrandes. Aus denselben Gründen liegt die Minimumsmarke oberhalb des Beckenbodens. Die Größe der Abstände gibt vor, wie schnell das System reagieren muss, denn vergeht vom Überschreiten der `MAX`-Marke bis zum Öffnen des Abflusses mehr Zeit, als benötigt wird, um das Becken um die Höhe des kritischen Bereiches aufzufüllen, dann kommt es zum Unfall in der Form, dass das Wasserbecken überläuft. Um die Versorgung auch bei der Anforderung größerer Wassermengen garantieren zu können, wird vorrausgesetzt, dass der Zufluss im geöffneten Zustand einen höheren Durchfluss hat als der Abfluss. Reale physikalische Systeme können Fehlfunktionen haben, wodurch es hier z.B. vorkommen kann, dass ein Ventil nicht dicht ist und –auch wenn sowohl Zufluss als auch Abfluss geschlossen sind– dennoch ein (geringer) Durchfluss in das Becken hinein- bzw. aus dem Becken herausfließt. Deshalb muss die Regelungseinheit den Wasserstand auch kontrollieren, wenn beide Ventile geschlossen sind.

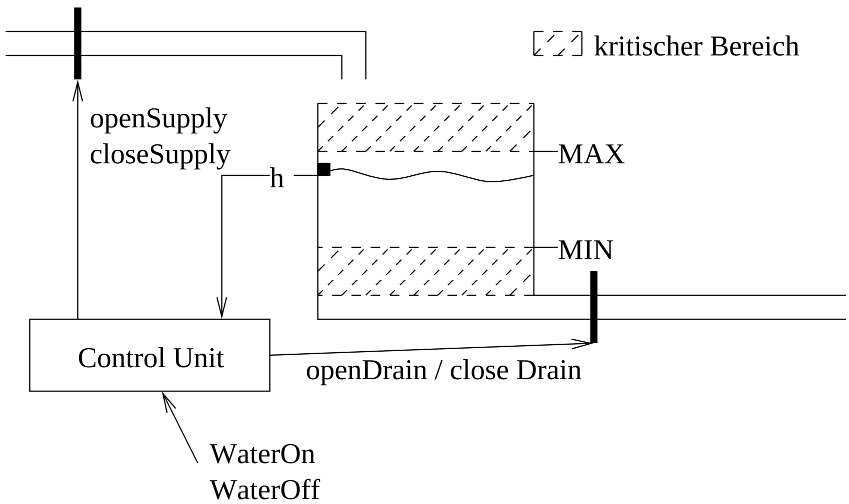

Abbildung 1.1.: Beispiel eines Echtzeitsystems

Um das Verhalten der Steuereinheit zu spezifizieren, muss ein Modell verwendet werden, in dem nicht nur dargestellt wird *wie*, sondern auch *in welcher Zeit* reagiert wird. Treten die Zeitschranken im Modell auf, können sie unter Berücksichtigung der Modell-Semantik auf Code abgebildet werden. UML bietet zwar verschiedene Möglichkeiten Verhalten zu spezifizieren, jedoch ist die Modellierung von zeitkritischem Verhalten nur begrenzt möglich, da viele Teilmodelle der UML vom realen Verhalten abstrahieren. So gibt es z.B. Modelle, die davon ausgehen, dass Aktionen (z.B. Berechnungen, Ein- oder Ausgaben) keine Zeit verbrauchen oder „schnell genug" sind, um angemessen zu reagieren. Diese Annahme genügt in einigen Bereichen, ist aber in zeitkritischen Anwendungen nicht ausreichend.

1.2. Ziele

Da, wie später gezeigt wird, UML keine ausreichenden Möglichkeiten bietet zeitkritisches Abläufe zu modellieren und somit auch kein Code, der Zeitanforderungen genügt, aus UML-Diagrammen erzeugt werden kann, werden in dieser Arbeit bestehende Modelle um geeignete Zeitannotationen erweitert, um ein solches *Echtzeitverhalten* spezifizieren zu können. Dabei wird der Abstraktionsgrad so gewählt, dass aus dem Modell echtzeitfähiger Code erzeugt werden kann. Werden in einem Modell Möglichkeiten angeboten Zeitverhalten zu spezifizieren, müssen temporale Konsistenzregeln eingehalten werden (Es muss berücksichtigt werden, dass das Öffnen und Schließen der Ventile aus Abb. 1.1 ein physikalischer Vorgang ist und nicht in „Null-Zeit" geschieht). Ein weiteres Ziel dieser Arbeit ist, diese Regeln aufzustellen und für deren Einhaltung zu sorgen. Die erarbeiteten erweiterten UML-Modelle, die Regeln zur Konsistenzsicherung und die Konzepte zur Codesynthese[1] werden daraufhin mit den bereits existierenden Modellen der Entwicklungsumgebung Fujaba kombiniert. Beispielhaft wird eine Umsetzung des Modells für *Realtime Java* (siehe Kapitel 2.2.2) als Zielsprache implementiert. Durch den Einsatz von Realtime Java wird sowohl Echtzeitverhalten ermöglicht, als auch Plattformunabhängigkeit gewährleistet.

1.3. Gliederung

Diese Arbeit gliedert sich in 7 Kapitel. Auf das aktuelle einführende Kapitel folgt Kapitel 2 zur Erläuterung der Grundlagen. Dieses Kapitel widmet sich der Beschreibung von Echtzeitsystemen und deren Eigenschaften sowie der, durch die Realtime Specification for Java gegebenen, Echtzeiterweiterung von Java. Des Weiteren werden verschiedene Modelle der UML und aus anderen Bereichen vorgestellt, und es wird die Notwendigkeit der oben angesprochenen Erweiterungen diskutiert. In Kapitel 3 werden existierende Ansätze und verwandte Arbeiten vorgestellt und diskutiert. Kapitel 4 widmet sich der Erweiterung der UML-Modelle, deren Syntax und Semantik sowie der Konsistenzprüfung. Daraufhin werden in

[1] Der Begriff "(Code-) Synthese" wird in dieser Arbeit dazu verwendet, um das Erzeugen / Generieren von Quelltext einer Programmiersprache zu bezeichnen.

Kapitel 5 Konzepte zur Umsetzung des modellierten Verhaltens erarbeitet. Wie diese Konzepte implementiert und in Fujaba integriert werden, beschreibt Kapitel 6. Kapitel 7 bietet eine Zusammenfassung der Arbeit und einen Ausblick auf mögliche weiterführende Arbeiten.

2. Grundlagen

Im Verlauf dieser Arbeit werden bestehende Modelle erweitert, um Echtzeitsysteme beschreiben zu können. Dieses Kapitel dient dazu, die Grundlagen der Echtzeitsysteme und verschiedenen Modellierungssprachen zu vermitteln. Dazu wird als Schwerpunkt im ersten Teilkapitel –nach einer Klärung der Grundbegriffe– das Taskmanagement in Echtzeitsystemen beschrieben. Im 2. Abschnitt diesen Kapitels wird der Bezug zur Programmiersprache Java her- und eine Echtzeiterweiterung vorgestellt. Im 3. und 4. Teilkapitel werden existierende Modelle zur Beschreibung von Verhalten und ihre Einsatzmöglichkeiten in Echtzeitsystemen beschrieben.

2.1. Echtzeitsysteme

Ein System, von dem nicht nur gefordert wird bestimmte Ergebnisse zu liefern, sondern von dem zusätzlich verlangt wird, diese Ergebnisse in einer vorgeschriebenen Zeit zu erbringen, heißt *Echtzeitsystem*. Dabei wird zwischen *harten* und *weichen* Echtzeitsystemen unterschieden: Von weichen Echtzeitsystemen wird verlangt, dass ein bestimmter Prozentsatz von Zeitanforderungen eingehalten werden muss. So wird z.B. bei einer Videoübertragung verlangt, dass die Einzelbilder innerhalb einer vorgeschriebenen Zeit übertragen werden, um ein "ruckelfreies" Anzeigen des Videos zu ermöglichen. Werden die Zeitanforderungen einzelner Bilder nicht eingehalten, ist der Film immer noch "sehenswert", solange nicht zu viele Bilder verzögert werden.

Bei dem in Kapitel 1.1 beschriebenen Beispiel oder bei einem Auto, das im Fall eines Unfalls die Aktion "öffne Airback" ausführen soll, hätte die einmalige Verfehlung einer Zeitanforderung katastrophale Folgen. Systeme, von denen gefordert wird 100 % aller Zeitanforderung einzuhalten, sind harte Echtzeitsysteme. In dieser Arbeit werden ausschließlich harte Echtzeitsysteme betrachtet.

Wie in Abb. 2.1 dargestellt, besteht ein Echtzeitsystem aus 3 Schichten: Der Hardware, dem *Echtzeitbetriebssystem* (engl. Realtime Operation System, *RTOS*) und der Applikation. Die Hardware setzt sich aus Mikroprozessoren (häufig auch als

Microcontroller oder CPUs bezeichnet) und Ressourcen zusammen. Die Ressourcen bestehen in der Regel zumindest aus Speicher sowie Sensoren und Aktoren, die die Ein- und Ausgaben der Mikroprozessoren behandeln. Das RTOS dient zum einen als Schnittstelle zwischen Applikation und Hardware und bietet Routinen an, um auf die Hardware zugereifen zu können, zum anderen –und das macht den Hauptunterschied zu einem nicht-echtzeitfähigen Betriebssystem aus– stellt es einen so genannten *Scheduler* zur Verfügung. Die Applikation besteht aus einem oder mehreren Threads (Tasks, Prozesse). Jeder einzelne Thread benötigt zur Ausführung die Rechenleistung einer CPU. In der Regel werden die Prozessoren, die jeweils immer nur einen einzigen Thread behandeln können, zur selben Zeit von mehreren Threads beansprucht. Für die Zuordnung welcher Thread, wann und wie lange auf welchen Prozessor –unter Berücksichtigung der Zeitanforderungen– zugreifen darf, ist der Scheduler zuständig. Die einzelnen Threads greifen während der Ausführung auf die Ressourcen zu. Da ein gleichzeitiger Zugriff auf dieselben Bereiche zu Konflikten führen kann, ist es eine weitere Aufgabe des Schedulers, solchen Zugriff zu steuern. In Kapitel 2.1.2 werden verschiedene Schedulingverfahren beschrieben. In den darauf folgenden Kapiteln wird ein ausgewähltes Verfahren ohne Berücksichtigung und anschließend unter Berücksichtigung von Ressourcen analysiert. Die in dieser Arbeit berücksichtigten Systeme stimmen mit der Architektur aus Abb. 2.1, bis auf die Einschränkung, dass sie genau einen Mikroprozessor besitzten, überein.

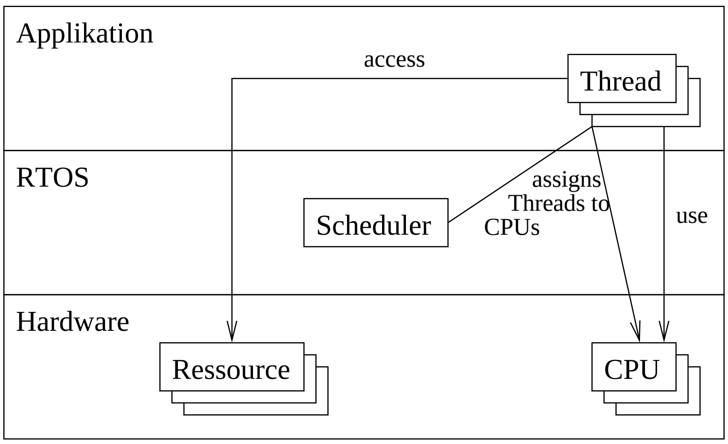

Abbildung 2.1.: Aufbau eines Echtzeitsystems

Als Alternative zur oben beschriebenen Architektur, kann die gesamte Logik der Applikation von einem einzigen Thread ausgeführt werden. In dem Fall kann dann auf den Einsatz von Betriebssystem und Scheduler verzichtet werden. Der Vorteil dieser Architektur ist, dass durch den Verzicht auf das Betriebssystems keine Performanz verloren geht. Nachteilig ist, dass es unter Umständen eine komplizierte Softwarearchitektur und eine geringere Fehlertoleranz zur Folge hat. Liegt bei einer Multi-Thread Applikation in einem Thread ein Fehler vor, kann ein anderer Thread den Fehler beheben oder den Thread neu starten. Dies ist bei einer Single-Thread Applikation nicht möglich.

Als dritte Möglichkeit bietet es sich an eine Applikation so zu implementieren, dass sie das Scheduling selbst regelt. In dem Fall werden zwar mehrere Threads gestartet, die Unterbrechungen und das Beenden der Unterbrechungen werden aber von den Threads selbst (und nicht vom Scheduler) ausgelöst. Diese Architektur bietet sich an, wenn ein Betriebssystem eingesetzt wird, dass zwar Nebenläufigkeit unterstützt, aber keinen Scheduler besitzt.

2.1.1. Grundbegriffe

Werden auf einem System mehrere Threads nebenläufig gestartet und beanspruchen Rechenleistung des Prozessors, muss die Zuteilung der Rechenzeit von einem Taskmanager geregelt werden. Wie in Kapitel 2.1 gesehen, wird bei Echtzeitsystemen ein Scheduler als Taskmanager eingesetzt. Jeder Thread besitzt eine Menge von Parametern, die vom Scheduler ausgewertet werden und anhand derer, er die Prozessorzuteilung vornimmt. Bevor die Parameter im Einzelnen erläutert werden, wird auf die 3 verschiedenen Arten von Threads eingegangen. In Grafiken, Formeln u.ä. werden Threads mit dem Buchstaben τ abgekürzt. Um mehrere Threads zu unterscheiden wird der Bezeichner indiziert (z.B. τ_1, τ_2, ... oder τ_i, $i = 1, 2, \ldots$). Einem Thread τ sind verschiedene Attribute zugeordnet (z.B. die Periode p, siehe unten). Um auszudrücken zu welchem Thread ein Attribut gehört, werden dem Thread und dem Attribut, dieselben Indizes zugeordnet (z.B. p_1 ist die Periode von τ_1).[1]

Periodischer Thread Ein periodischer Thread wird regelmäßig in äquidistanten Zeitabständen gestartet (wann er ausgeführt wird, hängt vom Scheduler ab). Die Abstände, in denen er gestartet wird, heißen Periode. Ein periodischer Thread muss beendet worden sein, bevor er erneut gestartet wird.

Aperiodischer Thread Ein Thread wird als 'aperiodisch' bezeichnet, wenn er in unregelmäßigen und unbekannten Zeitabständen gestartet wird.

Sporadischer Thread Ein sporadischer Thread ist ein Sonderfall eines aperiodischen Threads. Er wird ebenfalls in unregelmäßigen, unbekannten Abständen gestartet, es ist allerdings die Mindestzeit, die zwischen 2 Aufrufen liegt, bekannt. Im Worst Case verhält er sich wie ein periodischer Thread, dessen Periode dieser Minimum Interarrival Time entspricht.

[1] Im Anhang A ist eine Übersicht über alle in dieser Arbeit verwendeten Abkürzungen aufgelistet.

Da im Beispiel der Wasserbeckenregelung aus Kapitel 1.1 die Regeleinheit regelmäßig den Wasserstand misst und auswertet, würde sie durch einen periodischen Thread realisiert werden. Die Operationen `open/close Supply/Drain` werden aperiodisch bzw. sporadisch ausgeführt.

Thread-Parameter Wie bereits erwähnt, hat jeder Thread eine Menge von Parametern. Einige Parameter kommen bei jedem Thread vor, einige nur bei einzelnen. Im Folgenden werden die Parameter beschrieben, die das temporale Verhalten eindeutig spezifizieren. Sie sind in Abb. 2.2, die beispielhaft den Schedule eines periodischen und eines sporadischen Threads zeigt, dargestellt.

Worst Case Execution Time (Wcet) Die Wcet eines Threads gibt an, wie viel Zeit maximal vom Beginn bis zum Ende vergehen kann, wenn der Thread nicht durch den Scheduler unterbrochen wird. Die Wcet ist systemabhängig, d.h. sie muss neu bestimmt werden, wenn die Plattform gewechselt wird.

In dieser Arbeit wird die Wcet in Formeln und mathematischen Ausdrücken mit dem Buchstaben w bezeichnet. So bezeichnet z.B. w_a die Wcet der Anweisung `a`.

Startzeit Die Startzeit bezeichnet den Zeitpunkt, an dem der Thread gestartet werden soll. Wann er schließlich gestartet wird, ist vom Scheduler abhängig. Häufig wird die Startzeit relativ zur aktuellen Systemzeit angegeben. Möchte man erreichen, dass eine Aktion "so schnell wie möglich" erfolgt, wird sie auf 0 gesetzt. Abgekürzt wird sie mit dem Buchstaben s. Bei periodischen Threads wird in der Regel nur die Startzeit der ersten Ausführung angegeben. Die Startzeiten der folgenden Durchläufe ergeben sich aus der ersten und aus der Periode: $s^0 = s$; $s^1 = s + p$; Allgemein: $s^i = s + ip \quad , i \geq 0$.

Periode Wie der Name vermuten lässt, wird eine Periode nur bei periodischen Threads angegeben. Sie gibt an, in welchen Intervallen der Thread gestartet wird. In Formeln wird sie mit p abgekürzt.

Minimum Interarrival Time (MIT) Die MIT wird nur in sporadischen Threads spezifiziert und gibt an, wie viel Zeit mindestens zwischen 2 Starts dieser Threads liegt. Die MIT wird mit den Buchstaben m bezeichnet.

Deadline Eine Deadline ist ebenfalls eine Zeitkonstante (Bezeichnung d) und gibt an, wann ein Thread beendet sein muss. Dies kann eine Zeit relativ zum Start des Threads sein oder eine absolute Zeit. Wird die Deadline nicht getroffen, muss das System angemessen reagieren (z.B. in einen sicheren Zustand wechseln).

Es existieren verschiedene Algorithmen (so genannte *Schedulabilityalgorithmen*), die alle Parameter aller Threads auswerten und so vor dem Start des Systems bestimmen, ob alle Deadlines getroffen werden oder nicht (siehe 2.1.2).

Priorität Wird als Scheduler ein *Priority Scheduler* eingesetzt (was der am weitesten verbreitete Fall ist, siehe 2.1.2), wird die Zuteilung des Prozessors ausschließlich anhand der Priorität vorgenommen. Diese ist ein ganzzahliger, dimensionsloser Wert und wird in dieser Arbeit mit δ bezeichnet. Der Einsatz eines Priority Schedulers macht die anderen Parameter jedoch nicht überflüssig, da sie zum einen für eine Schedulinganalyse benötigt werden und es zum anderen sinnvoll ist, die Prioritäten auf Basis der Perioden oder Deadlines zu vergeben (z.B. kürzeste Deadline $\equiv$ höchste Priorität).

Antwortzeit Die Antwortzeit eines Threads ist die Zeit, die maximal zwischen Start und Ende des Threads liegt. Sie ist somit als Summe aus Worst Case Execution Time und Dauer aller Unterbrechungen bestimmbar. Die Antwortzeit erhält das Formelzeichen r.

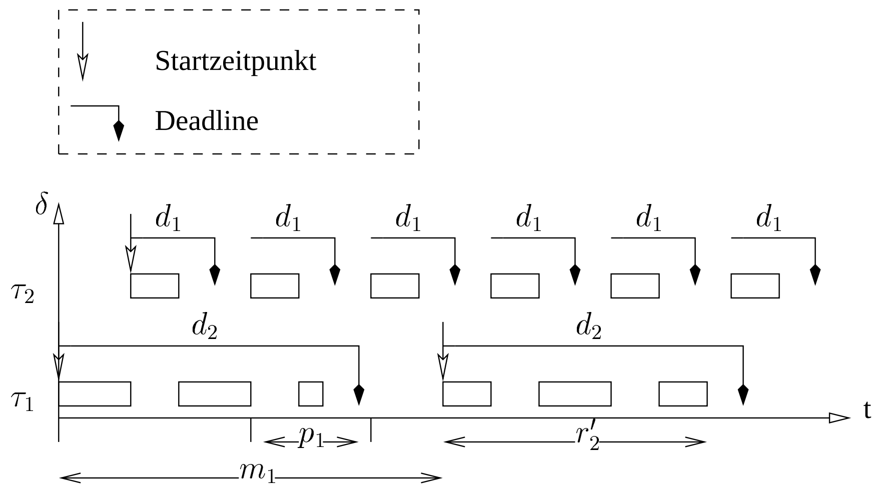

Abbildung 2.2.: Schedule eines periodischen und eines sporadischen Threads

Beispiele In Abb. 2.2 sind ein sporadischer Thread niedrigerer Priorität und ein periodischer höherer Priorität dargestellt. Die Prioritäten steigen entlang der y-Achse an. In der Grafik sind Periode, MIT und Deadlines eingezeichnet. Die benötigte Ausführungszeit ergibt sich aus der Summe der Länge der einzelnen Ausführungsabschnitte (dargestellt als Rechtecke) zwischen Startzeitpunkt und dem ersten Ende der Deadline und ist durch die Wcet nach oben beschränkt. Für die zweite Ausführung des sporadischen Threads ist der Wert r'_2 eingezeichnet.

Er zeigt, wie viel Zeit zwischen Beginn und Ende des Durchlaufs liegt. Er ist nicht zwingend identisch mit der Antwortzeit r, da diese durch das theoretische Maximum dieser Zeit bestimmt ist.

Im Beispiel der Wasserbeckenregelung vergeht sowohl beim Öffnen als auch beim Schließen des Zu- bzw. Abflusses Zeit. Für eine realistische Beschreibung des Systems dürfen diese Zeiten nicht vernachlässigt werden und werden deshalb als Worst Case Execution Times für die entsprechenden Aktionen berücksichtigt. Die Operation `openDrain` wird –wie bereits erwähnt– sporadisch ausgeführt. Die MIT ergibt sich daraus, wie häufig Wasser angefordert wird. Soll spezifiziert werden, dass der Abfluss „so schnell wie möglich“ geöffnet wird, würde als (relative) Startzeit für die Operation 0 angegeben werden. Existieren jedoch Threads, die zu der Zeit mit höherer Priorität laufen, verzögert sich die Ausführung. Um festzulegen, dass der Abfluss spätestens zu einem bestimmten Zeitpunkt offen ist, muss eine Deadline spezifiziert werden.

2.1.2. Schedulingverfahren

Schedulingverfahren können *unterbrechbar* und *nicht-unterbrechbar* sein. Wird ein Priority Scheduler eingesetzt (was der Regelfall ist), muss zusätzlich zwischen *statischem* und *dynamischem* Scheduling unterschieden werden. Verlangt ein Thread bei unterbrechbarem Scheduling Rechenleistung zwecks Ausführung und hat eine höhere Priorität als der zur Zeit aktive Thread, so unterbricht er ihn und wird an seiner Stelle ausgeführt. Bei nicht-unterbrechbarem Scheduling ist dies –wie der Name schon andeutet– nicht möglich. Der Wechsel von einem Thread zu einem anderen wird *Kontextswitch* genannt. Da der unterbrochene Thread zu einem späteren Zeitpunkt an derselben Stelle fortgesetzt werden soll, muss gegebenenfalls eine “Sicherheitskopie” der Register des Mikroprozessors angelegt werden. Wird er dann schließlich wieder aktiv, müssen die Werte aus dem Speicher wieder zurück in die Register kopiert werden. Da das Kopieren der Daten Zeit verbraucht, ist es erstrebenswert, beim Scheduling möglichst wenig Kontexswitche durchzuführen.

Jeder Thread erhält vor der Anmeldung beim Scheduler eine Priorität, die sich beim statischen Scheduling auch nicht mehr ändert. Beim dynamischen Scheduling ist es möglich, die Priorität zur Laufzeit zu ändern. Für die Bestimmung welcher Thread welche Priorität erhält, gibt es verschiedene Ansätze. Intuitiv könnte der Gedanke aufkommen, dem Thread, den man für den “Wichtigsten” hält, die höchste Priorität zu geben. Das Ziel des Schedulings ist aber nicht den *“wichtigsten”* Thread am schnellsten abzuschließen, sondern von *allen* Threads die Deadlines einzuhalten. Somit werden die Parameter aus Kapitel 2.1.1 analysiert und auf dessen Basis die Prioritäten bestimmt. Das *Rate Monotonic Assignment (RMA)* Scheduling z.B. vergibt die Prioritäten nach der Regel “Je kürzer die Periode, desto höher die Priorität.” [BR99], während beim *Deadline Monotonic*

Priority Assignment (DMPA) eine kürzere Deadline zu einer höheren Priorität führt [BW01]. Das Rate Monotonic Scheduling ist ein optimales, statisches, unterbrechbares Verfahren, d.h., dass eine Menge von schedulbaren Threads auch auf jeden Fall geschedult wird. Aus diesem Grund wird in dieser Arbeit dieses Verfahren verwendet. Um auch nicht-periodische Threads berücksichtigen zu können, wird die Periode bei sporadischen Threads durch die MIT und bei den übrigen aperiodischen Threads durch die Deadline ersetzt. Ist die Menge von Threads bekannt und sind die Prioritäten vergeben, soll in der Regel *vor* Start der Threads bestimmt werden, ob sie geschedult werden können, also ob alle Deadlines getroffen werden. Dies kann durch eine *Schedulinganalyse* (auch Feasibilityanalyse genannt) bestimmt werden.

2.1.3. Schedulinganalyse ohne Ressourcen

Für ein auf RMA basierendes Scheduling sind die Theoreme zur Schedulinganalyse u.a. in [BR99] beschrieben:

Theorem 2.1 (Liu, Layland) *Eine Menge von n Threads $\tau_1, \ldots, \tau_n$ ist nach RMA schedulbar, wenn gilt:*

$$\sum_{i=1}^{n} \frac{w_i}{p_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \tag{2.1}$$

Ist die Ungleichung nicht erfüllt, so kann keine Aussage über eine Schedulability getroffen werden und es sollte Theorem 2.2 angewendet werden. Bemerkenswert ist, dass $\lim_{n\to\infty} n(2^{\frac{1}{n}} - 1) = \ln 2 \approx 0.693$ gilt. Somit ist jede Menge von Threads, bei denen die Summe der Quotienten aus Wcet und Periode nicht 0.693 übersteigt, schedulbar.

Theorem 2.2 (Lehoczky, Sha, Ding) *Eine Menge von n Threads $\tau_1, \ldots, \tau_n$ mit $i < j \Rightarrow \delta_i > \delta_j$ ist nach RMA schedulbar dann und nur dann, wenn gilt:*

$$\begin{aligned} &\forall i,\ 1 \leq i \leq n :\ \min \textstyle\sum_{j=1}^{i} w_j \left\lceil \frac{lp_k}{p_j} \right\rceil \leq lT_k : \\ &(k,\, l) \in \left\{ (k,\, l)\, | 1 \leq k \leq i,\ l = 1, \ldots, \left\lfloor \frac{T_i}{T_k} \right\rfloor \right\} \end{aligned} \tag{2.2}$$

Theorem 2.2 sollte immer dann angewendet werden, wenn Theorem 2.1 keine Aussage bezüglich der Schedulability liefert (also wenn die Ungleichung aus Formel 2.1 nicht erfüllt ist).

Theorem 2.3 (Antwortzeitanalyse) *Eine Menge von n Threads $\tau_1, \ldots, \tau_n$ ist schedulbar dann und nur dann, wenn gilt:*

$$\forall i,\ 1 \leq i \leq n :\ r_i \leq d_i \,\mathrm{mit}\, r_i = w_i + \sum_{j \in \mathsf{hp}(i)} w_j \left\lceil \frac{r_i}{p_j} \right\rceil ;\ \mathsf{hp}\,(i) = \{ j |\, \delta_j > \delta_i \} \tag{2.3}$$

Eine Antwortzeitanalyse stellt eine Alternative zu Lehoczky, Sha und Dings Theorem dar. Sie ist u.a. in [Jos96] erläutert. Hier wird zunächst die Antwortzeit r_i eines Threads bestimmt. Ist die für jeden Thread kleiner oder gleich seiner Deadline, ist die Menge der Threads schedulbar. Dabei entspricht $\mathsf{hp}(i)$ der Menge der Indizies aller Threads, die eine höhere Priorität haben als der Thread i. Der Vorteil des Theorems 2.3 ist, dass es nicht wie die Theoreme 2.1 und 2.2 auf RMA beschränkt ist, sondern eine Analyse für jede beliebige Form der Prioritätszuweisung gilt. Werden die Prioritäten nach RMA vergeben, liefert sie dasselbe Ergebnis wie Theorem 2.2. Die Berechnung der Antwortzeit r_i ist allerdings kein triviales Problem, da r_i in der Formel mehrfach und u.a. innerhalb der oberen Gauß-Klammern vorkommt. Eine Lösung kann durch folgende Rekurrenzgleichung bestimmt werden:

$$r_i^0 = w_i; \qquad r_i^{k+1} = w_i + \sum_{j \in \mathsf{hp}(i)} w_j \left\lceil \frac{r_i^k}{p_j} \right\rceil ; \; k \geq 0 \tag{2.4}$$

Für $r_i^{k+1} = r_i^k$ ist eine Lösung gefunden und es gilt $r_i = r_i^k$. Da die Rekurrenzgleichung für k monoton steigend ist, kann die Berechnung abgebrochen werden, sobald $r_i^k > d_i$ gilt. In dem Fall ist die Menge von Threads nicht schedulbar.

2.1.4. Schedulinganalyse mit Ressourcen

In der Praxis tritt häufig der Fall auf, dass verschiedene Threads auf die gleichen Ressourcen zugreifen. Der Zugriff auf diese so genannten *shared Resources* muss dann synchronisiert werden, denn angenommen ein Thread liest aus z.B. einem Daten-Stack, während ein anderer schreibend auf denselben zugreift, kommt es zu offensichtlichen Problemen [Jos96]. In der Regel werden so genannte *Monitore* zur Synchronisation eingesetzt. Alle kritischen Abschnitte (Lesen / Schreiben) der Applikation sind Bestandteil des Monitors. Führt ein Thread eine Monitoroperation aus, sperrt er den Monitor für andere. Alle übrigen Threads, die Monitoroperationen ausführen wollen, werden blockiert, bis der Monitor wieder freigegeben wird.[2] Dies kann zu so genannten Deadlocks führen. Angenommen der Thread τ_1 hat Monitor A gesperrt und will –ohne die Monitoroperation vorher zu beenden (und somit die Sperrung aufzuheben)– auf Monitor B zugreifen, während Thread τ_2 Monitor B gesperrt hat und –ebenfalls ohne die Operation zu beenden– auf Freigabe von Monitor A wartet. Dann blockieren sich beide Threads gegenseitig, was als Deadlock bezeichnet wird [BR99]. Außerdem kann es vorkommen, dass ein Thread niedrigerer Priorität einen Monitor gesperrt hat und dadurch einen

[2]Hier ist eine Trennung der Begriffe bedeutend: Ein Thread wird von Threads höherer Priorität *unterbrochen* und aufgrund von parallelem Ressourcenzugriff von Threads niedrigerer Priorität *blockiert*.

Thread höherer Priorität blockiert. In ungünstigen Fällen kann dies bewirken, dass der höchstpriorisierte Thread sich faktisch so verhält, als habe er die niedrigste Priorität. Dieser Effekt wird als *Priority Inversion* (Prioritätsumkehrung) bezeichnet und ist u.a. in [Jos96, BR99] beschrieben. Mit der *Priority Inheritance (PI)* und der *Priority Ceiling Emulation (PCE)* existieren 2 Protokolle, die dieses Problem behandeln [BR99]. Für weitere Informationen und ausführliche Beispiele sei auf oben genannte Fachliteratur verwiesen.

Der Einsatz von Protokollen zur Vermeidung der Prioritätsinversion muss selbstverständlich bei einer Schedulinganalyse berücksichtigt werden. Dabei tritt ein weiterer Parameter auf: Die Blockierungszeit b. Sie gibt an, wie lange ein Thread maximal von einem Thread *niedrigerer* Priorität blockiert werden kann (höher priorisierte Threads blockieren nicht, sondern unterbrechen). Die Blockierungszeit berechnet sich wie folgt (vgl. [Jos96, BR99]):

Theorem 2.4 (Blockierungszeit) *In einem System existieren n Threads $\tau_1, \ldots, \tau_n$ und m Monitore $M_1, \ldots, M_m$. Thread τ_i hat w_i als Wcet, wovon er im Worst Case $w_{i,k}$ den Monitor M_k sperrt. Mit $\mathsf{lp}(i) = \{j \,|\, \delta_j < \delta_i\}$ ist die Menge der Indizes, deren Threads eine niedrigere Priorität als τ_i haben, und mit $|\mathsf{lp}(i)|$ die Anzahl jener Threads bezeichnet. Für die obere Grenze der Blockierungszeit b_i von Thread τ_i gilt:*

$$b_i = \begin{cases} \sum_{j \in \mathsf{lp}(i)}^{m} w_j & \text{, ohne Protokoll} \\ \min\left(\left|\mathsf{lp}\left(i\right)\right|, m\right) \max_{j \in \mathsf{lp}(i)} \left(\max_{k=1}^{m} \left(w_{j,k}\right)\right) & \text{, mit PI} \\ \max_{j \in \mathsf{lp}(i)} \left(\max_{k=1}^{m} \left(w_{j,k}\right)\right) & \text{, mit PCE} \end{cases} \tag{2.5}$$

Bei Einsatz der Priority Ceiling Emulation ist die Blockierungszeit geringer als bei Einsatz des Priority Inheritance Protokolls. Da Letzteres außerdem keine Deadlocks vermeidet, wird der Einsatz des Priority Ceiling Emulation Protokolls empfohlen.

Die Theoreme 2.1, 2.2 und 2.3 aus Kapitel 2.1.3 änderen sich bei Berücksichtigung der Blockierungszeit. Für $b_i = 0$ ergeben sich die Formeln aus Kapitel 2.1.3. Allgemein gilt:

Theorem 2.5 (Liu, Layland um Blockierungszeit erweitert) *Eine Menge von n Threads $\tau_1, \ldots, \tau_n$ ist nach RMA schedulbar wenn gilt:*

$$\max_{i}^{n} \left(\frac{b_i}{p_i}\right) + \sum_{i=1}^{n} \frac{w_i}{p_i} \leq n \left(2^{\frac{1}{n}} - 1\right) \tag{2.6}$$

Auch hier gilt, dass keine Aussage bezüglich der Schedulability getroffen werden kann, wenn die Ungleichung nicht erfüllt ist.

Theorem 2.6 (Lehoczky, Sha, Ding um Blockierungszeit erweitert) *Eine Menge von n Threads $\tau_1, \ldots, \tau_n$ mit $i < j \Rightarrow \delta_i > \delta_j$ ist nach RMA schedulbar dann und nur dann, wenn gilt:*

$$\begin{array}{l} \forall i,\, 1 \leq i \leq n:\; w_i + b_i + \min \sum_{j=1}^{i} w_j \left\lceil \frac{lp_k}{p_j} \right\rceil \leq lT_k : \\ (k,\, l) \in \left\{ (k,\, l)\,|1 \leq k \leq i,\, l = 1, \ldots, \left\lfloor \frac{T_i}{T_k} \right\rfloor \right\} \end{array} \tag{2.7}$$

Dieses Theorem sollte dann angewendet werden, wenn Formel 2.6 nicht erfüllt ist.

Theorem 2.7 (Antwortzeitanalyse um Blockierungszeit erweitert) *Eine Menge von n Threads $\tau_1, \ldots, \tau_n$ ist schedulbar dann und nur dann, wenn gilt:*

$$\forall i,\, 1 \leq i \leq n:\; r_i \leq d_i \,\mathrm{mit}\, r_i = w_i + b_i + \sum_{j \in \mathsf{hp}(i)} w_j \left\lceil \frac{r_i}{p_j} \right\rceil ;\; \mathsf{hp}\,(i) = \{ j |\, \delta_j > \delta_i \} \tag{2.8}$$

Die Gleichung lässt sich dann durch die Rekurrenzgleichung 2.9 lösen. Dabei kann –wie bei der Rekurrenzbeziehung 2.4– abgebrochen werden, wenn $r_i^{k+1} = r_i^k$ oder wenn $r_i^k > d_i$.

$$r_i^0 = w_i; \qquad r_i^{k+1} = w_i + b_i + \sum_{j \in \mathsf{hp}(i)} w_j \left\lceil \frac{r_i^k}{p_j} \right\rceil ;\; k \geq 0 \tag{2.9}$$

2.1.5. Reaktionszeit periodischer Threads

In Kapitel 2.1.1 wurde beschrieben, dass periodische Threads in äquidistanten Zeitabständen gestartet werden. Das impliziert, dass es Intervalle gibt, in denen er inaktiv ist. Tritt in diesen Intervallen ein Ereignis auf, kann der Thread frühstens bei der nächsten Aktivierung darauf reagieren, wodurch es zu einer Verzögerung kommt. Bei Echtzeitsystemen ist es wichtig zu bestimmen, wie groß diese Verzögerung maximal ist. Dabei werden 2 Fälle unterschieden:

- Der Thread hat die höchste Priorität im System und greift nicht auf (von Monitoren kontrollierte) Ressourcen zu.
- Der Thread hat nicht die höchste Priorität im System oder greift auf Ressourcen zu.

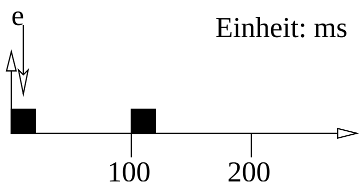

Abbildung 2.3.: Verzögerung beim höchstpriorisierten Thread

Im ersten Fall ist sichergestellt, dass der Thread zu Beginn seiner Periode immer ausgeführt wird.

Abb. 2.3 zeigt den Fall, in dem der Thread zum Zeitpunkt $t = 0ms$ mit der Periode $p = 100ms$ gestartet wird. In jeder Periode überprüft er, ob ein Ereignis aufgetreten ist. Hier trifft unmittelbar nach Ende dieser Überprüfung ein Ereignis e ein (Das Empfangen eines Ereignisses wird durch einen nach unten gerichteten Pfeil mit weißer Spitze dargestellt. Um das Senden darzustellen wird der Pfeil nach oben gerichtet). Das Ereignis wird erst erkannt, wenn der Thread erneut ausgeführt wird (bei $t = 100ms$). In diesem Worst Case ist die Verzögerung zwischen Ereignisauftritt und Ereigniserkennung maximal 1 Periode.

Hat der Thread nicht die höchste Priorität, dann hängt es vom verwendeten Scheduler ab, wann ihm Rechenzeit für die Ausführung zugeteilt wird. Eine Analyse wäre zum einen sehr aufwendig, zum anderen ergäbe sie unter Umständen für jede Periode verschiedene Ausführungszeiten. Aufgrund dessen wird für jede Periode der Worst Case betrachtet, der in Abb. 2.4 dargestellt ist. Der Thread wird gleich zu Beginn seiner ersten Periode und gegen Ende seiner zweiten Periode ausgeführt. Hier beträgt die maximale Verzögerung ca. $2p$ ($200ms$ im Beispiel). Hierbei ist es unerheblich, ob der Thread Zugriff auf shared Resources hat und ob Protokolle zur Vermeidung von Prioritätsumkehrung eingesetzt werden. Auch wenn er von einem niedriger priorisierten Thread geblockt wird, sorgt der Scheduler dafür, dass er spätestens vor dem Ende seiner Deadline (Periode) ausgeführt wird. Somit ist der maximale Abstand zwischen 2 Ausführungen der in Abb. 2.4 dargestellte.

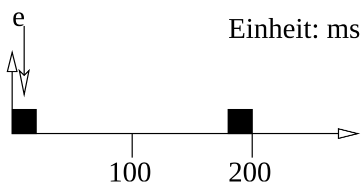

Abbildung 2.4.: Verzögerung bei niederpriorisierten Threads

Kommunizieren 2 Threads miteinander (z.B. indem sie Nachrichten oder Ereignisse in eine zum Thread gehörige Queue schreiben), gibt es ebenfalls Verzögerungen, wie in Abb. 2.5 dargestellt. Dort sendet τ_1 das Ereignis e an τ_2, welcher in

Form von Ereignis `f` antwortet. Wenn in beiden Fällen das Ereignis unmittelbar nach Ende der Queue-Überprüfung eintrifft, vergeht im ungünstigsten Fall vom Stellen der Anfrage bis zum Erhalt der Antwort $2(p_1 + p_2)$ ($280ms$).

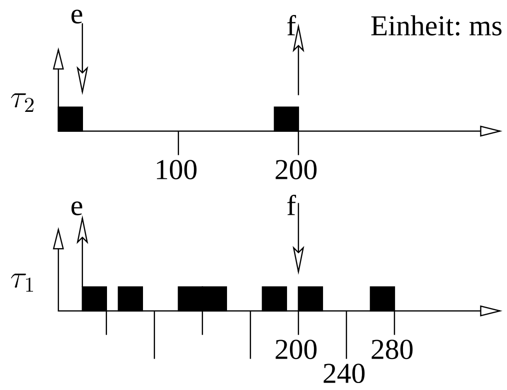

Abbildung 2.5.: Verzögerung bei der Kommunikation

In dieser Arbeit werden Threads betrachtet, die nicht mit höchster Priorität im System laufen, da es im Regelfall immer Prozesse des Betriebssystems gibt, die höher priorisiert sind. Wichtig ist, dass dann die intuitive Annahme, dass bei einer Periode von p die maximale Verzögerung nicht größer als p ist, falsch ist. Stattdessen können Verzögerungen bis zu $2p$ auftreten. Sind Threads voneinander abhängig (wie in Abb. 2.5), kann sich die Verzögerung aufsummieren.

2.1.6. Aperiodische Threads

Häufig werden aperiodische Threads von so genannten *Priority Servern* gesteuert. Das ist dann sinnvoll, wenn eine Menge von periodischen Threads mit harten Echtzeitanforderungen und eine Menge von aperiodischen Threads mit weichen Echtzeitanforderungen zu schedulen sind. Die Priority Server stellen sicher, dass die aperiodischen Threads die harten periodischen nicht beeinflussen. Ein Priority Server wird im Normalfall selbst als periodischer Thread implementiert, der pro Periode eine bestimmte Menge an Rechenzeit (auch *Kapazität* genannt) erhält, die er wiederum den aperiodischen Threads zur Verfügung stellt, sofern welche auf Ausführung warten. Ist die Kapazität erschöpft, unterbricht er den Thread und setzt ihn erst in der nächsten Periode des Servers fort (wenn die Kapazität wieder aufgefüllt ist). Der Unterschied zum Scheduler ist, dass die Unterbrechung nicht anhand von Prioritäten, sondern aufgrund der begrenzten Kapazität vorgenommen wird. Beispiele für Priority Server sind in [But97, BR99] gegeben (z.B. Deferred Server, Sporadic Server).

Der Vorteil dieses Verfahrens liegt darin, dass die Server bei einer Schedulinganalyse wie "normale" periodische Threads behandelt werden können. Allerdings kommt es vor, dass ein Deferred Server, zwar von allen zur Ausführung bereiten Threads die höchste Priorität hat, aber seine Ausführung verzögert (wenn kein aperiodischer Thread zur Ausführung bereit ist; daher auch der Name) und somit nicht mehr durch eine Analyse, wie in Kapitel 2.1.3 vorgestellt, betrachtet werden kann. Dieser unangenehme Effekt tritt zwar nicht beim Sporadic Server auf, dafür ist es aber sehr aufwendig und nur begrenzt möglich eine Vorhersage über die Schedulability von aperiodischen Threads mit harten Deadlines zu treffen (Eine ausführliche Beschreibung und Analyse von Priority Servern findet in [But97] statt).

Es gibt 2 mögliche Implementierungen eines Sporadic Servers: Auf Betriebssystemebene oder auf Anwendungsebene. Letztere ist allerdings ineffizient und soweit beschränkt, dass ein aperiodischer Thread nicht unterbrochen werden kann, wenn die Kapazität erschöpft ist [BR99]. Daher wird die Entscheidung für oder gegen ein bestimmtes Betriebssystem häufig von der Frage bestimmt, ob ein Priority Server implementiert ist.

2.2. Echtzeitplattformen

Ein Vorteil der Anwendungsentwicklung in der Programmiersprache Java ist die Plattformunabhängigkeit. Eine in Java implementierte Applikation kann auf jedem System ausgeführt werden, das eine Java Virtuelle Maschine (JVM) zur Verfügung stellt. Dies können verschiedene Betriebssysteme oder spezielle Javaprozessoren sein, deren Befehlssatz die Java Byte Code (JBC) -Instruktionen enthält. Bei der Entwicklung von Echtzeitapplikationen verhält es sich ähnlich: Wird Realtime Java –wie es in Kapitel 2.2.2 vorgestellt wird– eingesetzt, kann die Anwendung auf jeder Plattform eingesetzt werden, die eine „Realtime-JVM“ zur Verfügung stellt. Diese JVM kann entweder auf einem Javaprozessor laufen, der neben dem Standard-JBC auch den Byte Code der Echtzeiterweiterung verarbeiten kann oder für ein Echtzeitbetriebssystem implementiert sein. Da eine Vielzahl von Echtzeitbetriebssystemen mit unterschiedlichen Eigenschaften existieren, wurden Standards für unterschiedliche Anforderungen geschaffen. Diese Standards werden in Kapitel 2.2.1 vorgestellt.

2.2.1. POSIX-Echtzeitbetriebssysteme

Betriebssysteme, die in Eingebetteten Systemen eingesetzt werden, unterscheiden sich häufig nicht nur durch den Scheduler von den üblicherweise im Büro- oder Heimbereich eingesetzten Betriebssystemen. So existieren z.B. Echtzeitbetriebssysteme, die kein Dateisystem besitzen oder keinen parallelen Zugriff auf Ressourcen erlauben. Diese und andere Eigenschaften von Echtzeitsbetriebssystemen wurden in so genannten POSIX Profilen zusammengefasst. Dadurch existiert ein Standard, der es ermöglicht, die Vorraussetzungen, die für Software erforderlich sind, durch Angabe des Profils angeben zu können. Für Echtzeitbetriebssysteme existieren 4 Profile: *Minimal real-time system profile (PSE50)*, *Real-time controller system profile (PSE51)*, *Dedicated real-time system profile (PSE52)* und *Multi-purpose real-time system profile (PSE53)*. Die Profile PSE51 bis PSE53 sind Erweiterungen des Basis Profils PSE50. Schon um dem Basis Profil zu genügen, muss ein Echtzeitbetriebssystem die Möglichkeit bieten, mehrere Threads innerhalb einer Applikation starten zu können, einen Priority Scheduler zur Verfügung stellen und den parallelen Zugriff auf Ressourcen ermöglichen [BW01]. Obwohl es kein Dateisystem verlangt, reichen in dieser Arbeit Betriebssysteme aus, die dem POSIX Profil PSE50 genügen.

2.2.2. Realtime Java

Am 12.11.2001 wurde die endgültige Fassung der Version 1.0 der *Real-Time Specification for Java (RTSJ)* [BBF+00] von der *Real-Time for Java Expert Group* [RTJ] verabschiedet. Diese API (Application Program Interface) stellt verschiedene Klassen zur Verfügung, die u.a. Scheduling und Speichermanagement ermöglichen. Im Folgenden werden die in dieser Arbeit benutzten Bereiche von Realtime Java[3] vorgestellt:

Scheduling Abb. 2.6 zeigt den für das Scheduling interessanten Ausschnitt des Klassendiagramms der Real-Time Specification.

Von der Spezifikation wird gefordert einen Priority Scheduler zu implementieren. Durch den Einsatz der abstrakten Oberklasse `Scheduler` ist auch die Verwendung anderer Scheduler möglich. Unabhängig davon was für ein Scheduler schließlich verwendet wird, hält er Referenzen zu jedem Thread, der von ihm geschedult werden muss. Jene Threads sind dann Instanzen der Klasse `RealtimeThread` und eine Erweiterung von `java.lang.Thread`. Über die Assoziation zu `Scheduling-` und `ReleaseParameters` können die Parameter (Priorität, Deadline, Wcet, Startzeit und MIT) spezifiziert werden.

Für die Behandlung der Prioritätsumkehrung (siehe Kapitel 2.1.4) werden die Klassen `MonitorControl`, `PriorityInheritance` und `PriorityCeilingEmulation` (nicht im Klassendiagramm dargestellt) zur Verfügung gestellt. Dabei werden in der API zwar alle drei Klassen definiert, eine Implementierung der Klasse `PriorityCeilingEmulation` wird von der RTSJ jedoch nicht gefordert. Über den Aufruf `MonitorControl.setMonitorControl(monitor, protocol)` kann zu jedem so genannten Monitor-Objekt eines dieser Protokolle gesetzt werden. Taucht

[3] Mit *Realtime Java* sei Java unter Verwendung des Packages javax.realtime, also der Implementierung der RTSJ, bezeichnet.

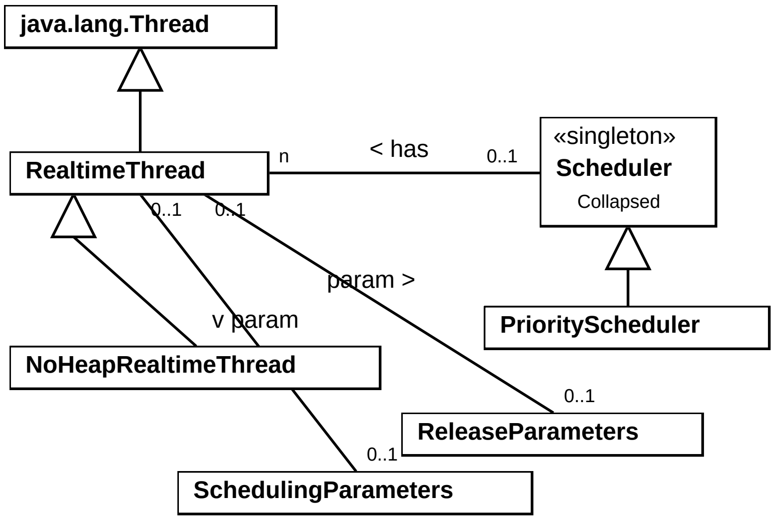

Abbildung 2.6.: Ausschnitt des Klassendiagramms der RTSJ

innerhalb dieses Objektes das Schlüsselwort `synchronized` auf, werden die betroffenen Bereiche nach jenem Protokoll synchronisiert. Auch hier ist wiederum Vorraussetzung, dass diese Protokolle vom unterliegenden Betriebssystem implementiert sind. Eine Methode, die feststellt welche Protokolle zur Verfügung gestellt werden, ist nicht in der RTSJ spezifiziert. Außerdem gibt es keine Klassen, die es ermöglichen einen evtl. bereitgestellten Priority Server anzusprechen.

Speicherverwaltung Eines der besonderen Merkmale von Java ist bekanntlich der Garbage Collector (GC). So nützlich er in vielen Fällen ist, um so nachteiliger wirkt er sich aus, wenn es darum geht sein nicht-deterministisches Auftrittsverhalten bei Anwendungen mit harten Echtzeitanforderungen zu berücksichtigen. Angenommen, ein Thread habe eine Worst Case Execution Time von $w = 50ms$ und eine Schedulinganalyse stellt fest, dass er zusammen mit weiteren Threads schedulbar sei. Die gesamte Analyse ist wertlos, wenn es vorkommen kann, dass der Garbage Collector diesen Thread für z.B. $200ms$ unterbrechen kann. Um das zu verhindern, können so genannte `NoHeapRealtimeThreads` (NHRT Threads) erzeugt werden. Ein NHRT Thread hat immer eine höhere Priorität als der Garbage Collector. Das bedeutet, dass ein NHRT Thread niemals vom Garbage Collector unterbrochen wird, und dass er selbst den GC unterbricht, falls dieser gerade aktiv ist, wenn der NHRT Thread gestartet wird.

Da durch den Einsatz von NHRT Threads der Garbage Collector außer Kraft gesetzt wird, ist ein explizites Speichermanagement notwendig. Dazu teilt die Real-Time Specification den Speicherbereich in verschiedene Bereiche auf. Neben dem "traditionellen" *Heap* Speicherbereich existieren u.a. noch der *Scoped* und der *Immortal* Speicherbereich. Objekte im Immortal Bereich werden nie vom Garbage Collector betrachtet und erst bei der Terminierung der Applikation gelöscht. Somit eignet sich dieser Bereich nur für statische Strukturen, die während der gesamten Laufzeit benötigt werden. Dynamische Datenstrukturen sollten von einem eigenen Manager verwaltet werden (z.B. durch Einsatz des *Factory Patterns* [GHJV94]). Definiert man einen Bereich des Speichers als Scope und erzeugt dort Objekte, werden diese so lange vom GC ignoriert, wie es mindestens eine Referenz auf *irgendein* Objekt in diesem Scope gibt. Um das zu verwalten, existiert zu jedem Scoped Bereich ein *Reference Count*, der die Anzahl der Referenzen auf Objekte innerhalb des Scopes wiedergibt. Das sei anhand eines Beispiels in Abb. 2.7 verdeutlicht:

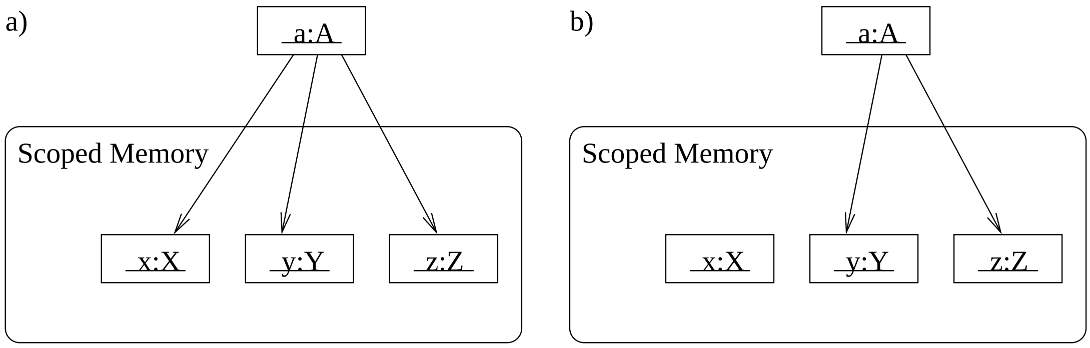

Abbildung 2.7.: Scoped Memory

In Abb. 2.7 a) werden die 3 Objekte `x`, `y` und `z` im Scoped Speicherbereich vom Objekt `a` referenziert.[4] Somit ergibt sich für den Reference Count ein Wert von 3. In Abb. 2.7 b) wurde das Object `x` dereferenziert. Wäre es im Heap Speicherbereich allokiert, würde es für den Garbage Collector freigegeben werden. Da es aber im Scoped Bereich liegt, wird lediglich der Reference Count um 1 auf 2 erniedrigt. Erst wenn die anderen beiden Objekte ebenfalls dereferenziert werden und der Reference Count somit auf 0 fällt, werden die Finalizer der 3 Objekte aufgerufen und der Speicher freigegeben. Die Freigabe des Speichers findet unmittelbar, nachdem der Reference Count auf 0 gefallen ist, statt, was zu einem deterministischen Verhalten des Garbage Collectors führt und es erlaubt eine Wcet für diesen Vorgang anzugeben.

Referenz Implementierung Die Firma *TimeSys* [Tim] hat im Auftrag der Real-Time for Java Expert Group eine vollständige Implementierung der RTSJ und einer Virtuellen Maschine für Linux-Betriebssysteme zur Verfügung gestellt.

[4] Um "hängende" Referenzen zu verhindern, dürfen Objekte im Scoped Speicherbereich nicht von Objekten vom Heap Speicher referenziert werden. Siehe hierzu auch [Dib02, BBF+00].

Außerdem liefert TimeSys einen Linux-Kernel, der u.a. einen Scheduler enthält und somit das unterliegende Betriebssystem echtzeitfähig macht. Ein Zugriff der Java-Klassen auf die Routinen des Betriebssystems erfolgt über das Java Native Interface (JNI) [Sun]. Darüber hinaus hat *aJile Systems* [aJi] für das 4. Quartal 2002 die Fertigstellung eines Javaprozessors angekündigt, der die RTSJ implementiert. *OTI* [OTI] hat nach Veröffentlichung einer Beta-Version die Arbeit an einer Referenz Implementierung vermutlich eingestellt. Weitere Hinweise zu den Implementierungen sind in Anhang C zusammengefasst.

2.2.3. Zusammenfassung

Die Real-Time Specification definiert Schnittstellen, um Scheduling und eine deterministische Speicherverwaltung in Java zu ermöglichen und somit plattformunabhängige Echtzeitanwendungen zu implementieren. Durch Verwendung des, von TimeSys zur Verfügung gestellten, POSIX-konformen Echtzeitbetriebssystems und der JVM kann die RTSJ in dieser Arbeit eingesetzt werden.

2.3. UML-Modelle

Mit der Unified Modelling Language (UML) [Obj01] existiert ein Standard für die objektorientierte Modellierung, die von Modellierern und Entwicklern eine breite Akzeptanz fand. Die einzelnen Modellsichten (die durch Diagramme repräsentiert werden) können in 2 verschiedene Kategorien eingeteilt werden. So gibt es welche, die die Struktur beschreiben und solche, mit denen Verhalten spezifiziert wird. Strukturbeschreibende Diagramme sind z.B. Klassen- und Objektdiagramme, während mit Statecharts, Aktivitäts- und Storydiagrammen das Verhalten spezifiziert wird. Da diese Arbeit Echtzeitverhalten behandelt, beschränkt sich dieses Kapitel auf die Verhaltensmodelle. Dabei wird mit Statechart-Diagrammen reaktives Verhalten beschrieben, während mit Aktivitätsdiagrammen ein Ablauf modelliert wird. Aus den Aktivitätsdiagrammen sind die Storydiagramme, im Rahmen des Fujaba Case Tools, als eine Sonderform entstanden.

2.3.1. Statechart-Diagramm

Ein Statechart-Diagramm [Har87, Obj01] ist eine Erweiterung eines Endlichen Automatens [Ram], bei der High-Level-Konstrukte eingeführt wurden, die versuchen komplexes reaktives Verhalten mit verhältnismäßig wenig Aufwand zu modellieren. Statecharts bieten im Einzelnen folgende Möglichkeiten, die in Abb. 2.8 dargestellt sind:

- Zu jedem Zustand können so genannte `entry()`, `exit()` und `do()`-Methoden spezifiziert werden. Diese werden bei Ein- bzw. Austritt aus dem Zustand, bzw. während des Aufenthaltes wiederholend ausgeführt (siehe Abb. 2.8 Zustand S_7).[5]

- Zustände können hierarchisch in mehreren Zuständen geschachtelt sein (S_7).

- Transitionen, deren Quellzustand hierarchisch tiefer geschachtelt ist, schalten (nach [Obj01]) mit höherer Priorität (Das Schalten von S_3 nach S_4 hat Vorrang gegenüber dem Schalten der Transition von S_2 nach S_7).

- Wurde innerhalb eines hierarchischen Zustands ein so genannter *History-Operator* angegeben (S_7), wird, bei Wechsel in den Zustand, nicht in den Startzustand gewechselt, sondern in den zuletzt aktuellen inneren Zustand, bevor der hierarchische Zustand verlassen wurde. Bei Ersteintritt wird in den Startzustand gewechselt. Es wird zwischen *Tiefer (Deep)* und *Flacher (Shallow)* History unterschieden. Der Unterschied ist, dass sich bei Deep History die History in alle hierarchischen Unterzustände fortsetzt. Deep History wird durch einen Asterix am History-Operator gekennzeichnet ("H*").

- In parallelen Zuständen (S_2) können mehrere Zustandswechsel parallel zueinander durchgeführt werden.

- Bei einem Zustandswechsel (Schalten einer Transition) können Seiteneffekte z.B. in Form von Operationen (`op()` beim Schalten von S_3 nach S_4) ausgeführt werden und neue Ereignisse erzeugt werden (`f` beim Übergang von S_2 nach S_7).

- In einem Statechart können Variablen verwendet werden. Das Schalten einer Transition kann dann zusätzlich von einer beliebigen Bedingung über den Variablenkontext abhängig gemacht werden, indem mit der entsprechenden Transition ein *Guard* assoziiert wird (`[x > 2]` bei der Transition von S_5 nach S_6).

- Ereignissen können Parameter übergeben werden, die dann bei der Ausführung des Seiteneffektes zur Verfügung stehen (siehe Transition von S_8 nach S_9).

Um Nicht-Determinismus, der Auftritt, wenn im aktuellen Zustand mehrere Transitionen gleichzeitig aktiviert sind, zu vermeiden, werden in [Zün01] zu den Transitionen Prioritäten spezifiziert, die diesen Nicht-Determinismus auflösen.

[5] In [Zün01] ist eine Variante definiert, in der eine `do()`-Methode genau einmal nach Eintritt in den jeweiligen Zustand ausgeführt wird.

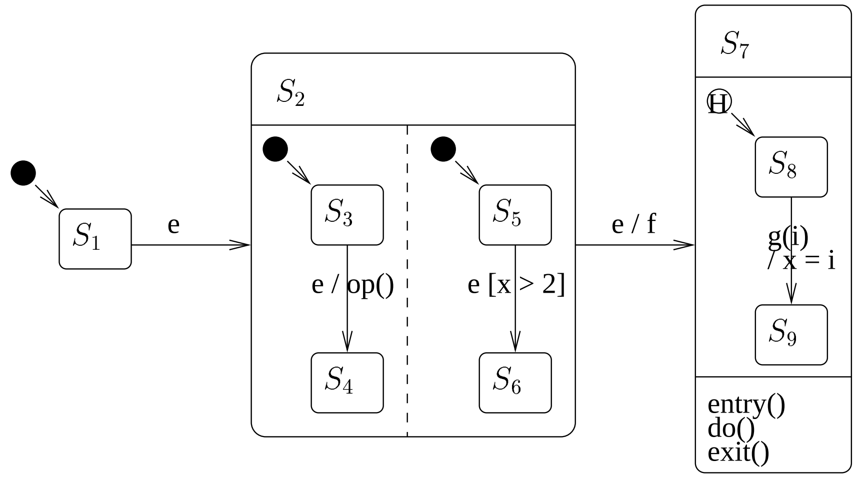

Abbildung 2.8.: Statechart-Diagramm

Die einzigen Möglichkeiten Zeitverhalten zu spezifizieren, sind das *After*- und das *When*-Konstrukt. Beim After-Konstrukt wird eine Transition mit `after(x)` bezeichnet, wobei `x` eine Zeitdauer angibt (z.B. `after(10ms)`). Die Transition wird dann nach dieser Zeit nach Eintritt in den Quellzustand aktiviert. Durch das When-Konstrukt können absolute Zeitpunkte spezifiziert werden. Durch `when(15:00)` kann z.B. spezifiziert werden, dass die Transition um 15 Uhr schalten soll.

Statecharts sind ereignisgesteuert, d.h., dass Transitionen nur auf Aktivierung überprüft werden, wenn ein Ereignis eingetroffen ist. Stünde in Abb. 2.8 der Guard an der Transition von S_5 nach S_6 ohne das Ereignis e dar, und würde der Wert des Guards von `false` auf `true` wechseln, wird das nicht sofort, sondern erst bei der nächsten Aktivierungsüberprüfung (also wenn das nächste Ereignis eintrifft) behandelt. Das When-Konstrukt stellt die einzige Ausnahme dar. Sobald der dort spezifizierte Ausdruck wahr wird, schaltet die entsprechende Transition.

Für weiterführende Betrachtungen über Statecharts und eine vollständige Definition und Spezifikation wird auf [Har87] und [Zün01] verwiesen.

2.3.2. Aktivitätsdiagramm

Methoden, die u.a. als Operationen in Statecharts verwendet werden, können durch Aktivitätsdiagramme spezifiziert werden. Ein Aktivitätsdiagramm besteht aus einem gerichteten Graphen, dessen Knoten als Aktivitäten und dessen Kanten als Transitionen bezeichnet werden. Jeder Aktivität sind Operationen zugeordnet. Bei Eintritt in eine Aktivität werden diese Operationen ausgeführt. Nach Beendigung der Operationen wird die Aktivität über eine Transition verlassen und die Folgetransition betreten. Existieren an einer Aktivität mehrere ausgehende Transitionen, müssen sie mit Bedingungen –gleichartig zu den Guards bei den Transitionen der Statecharts (siehe Kapitel 2.3.1)– verknüpft sein.

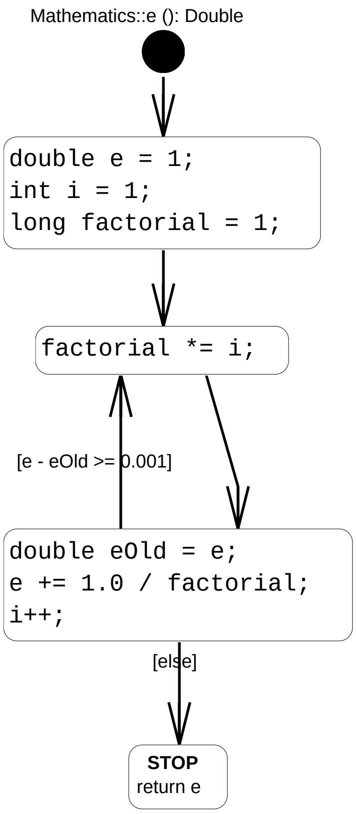

Abbildung 2.9.: Aktivitätsdiagramm zur Berechnung der Eulerschen Zahl

Abb. 2.9 zeigt z.B. das Aktivitätsdiagramm eines numerischen Verfahrens zur Bestimmung der Eulerschen Zahl e (mit einer Genauigkeit von 3 Nachkommastellen). Es besteht aus 4 Aktivitäten, in denen unterschiedliche Aktionen ausgeführt werden, die hier in Form von Java-Anweisungen angegeben sind. Nach der 3. Aktivität wird anhand einer Fallunterscheidung bestimmt, welche Transition verfolgt wird.

2.3.3. Story Diagramm

Ein Story Diagramm ist eine Spezialform eines Aktivitätsdiagramms und dient ebenfalls zur Spezifikation von Methoden. In einem Story Diagramm sind die Aktionen der Aktivitäten in Form so genannter *Story Pattern* spezifiziert. Ein Story Pattern zeigt einen möglichen Ausschnitt der Instanzstruktur zur Laufzeit und verschiedene Regeln diese Struktur zu modifizieren. Bei Ausführung der

Aktivitäten wird die Instanzstruktur gesucht und nach einer erfolgreichen Suche modifiziert (Eine ausführliche Definition und Spezifikation von Story Pattern und Story Diagrammen befindet sich in [Zün01]).

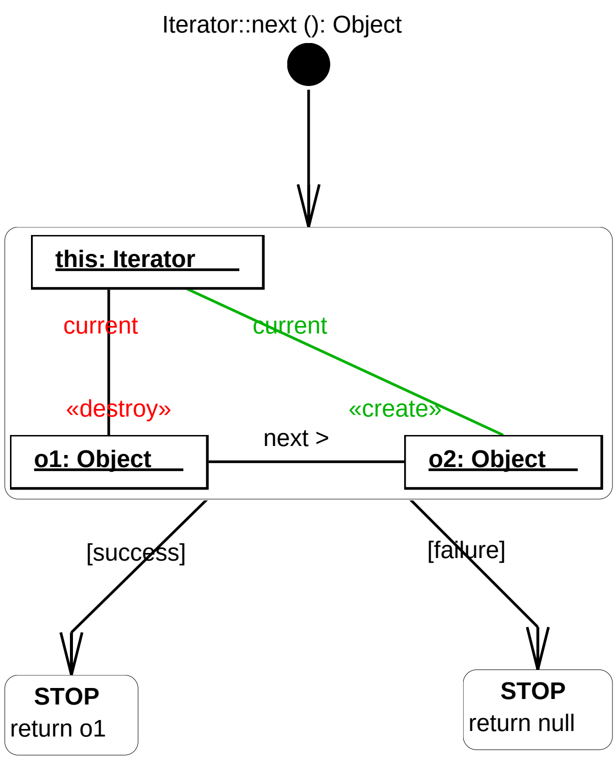

Abbildung 2.10.: Story Diagramm

In Abb. 2.10 ist ein Story Diagramm der Methode `next()` einer Klasse `Iterator` zu sehen. Dabei wird in der Laufzeitdatenstruktur nach einer Liste von Objekten gesucht. Die `Iterator`-Instanz muss dabei eine Referenz namens `current` auf eines der Listenobjekte haben. Wird diese Struktur gefunden, wird die `current`-Referenz auf das folgende Objekt gesetzt und das bis dahin aktuelle Element zurückgeliefert. Andernfalls liefert ein Methodenaufruf `null`.

2.3.4. Echtzeitmodellierung

Um den Kontroller der Wasserbeckenregelung des Beispiels aus Kapitel 1.1 zu modellieren, bietet sich das Statechart-Modell an. Abb. 2.11 zeigt das Diagramm mit den Zuständen, die die 4 Fälle Zufluss geöffnet/geschlossen und Abfluss geöffnet/geschlossen repräsentieren. Da es 2 Möglichkeiten gibt, weshalb der Abfluss geöffnet sein kann –entweder wenn Wasser angefordert wurde oder um ein Überlaufen zu verhindern– existieren 2 Zustände mit dem Namen `SupClosed-`

DrainOpen – einer in dem Superzustand NoRequestGateOpen und einer in dem Superzustand RequestGateOpen.[6]

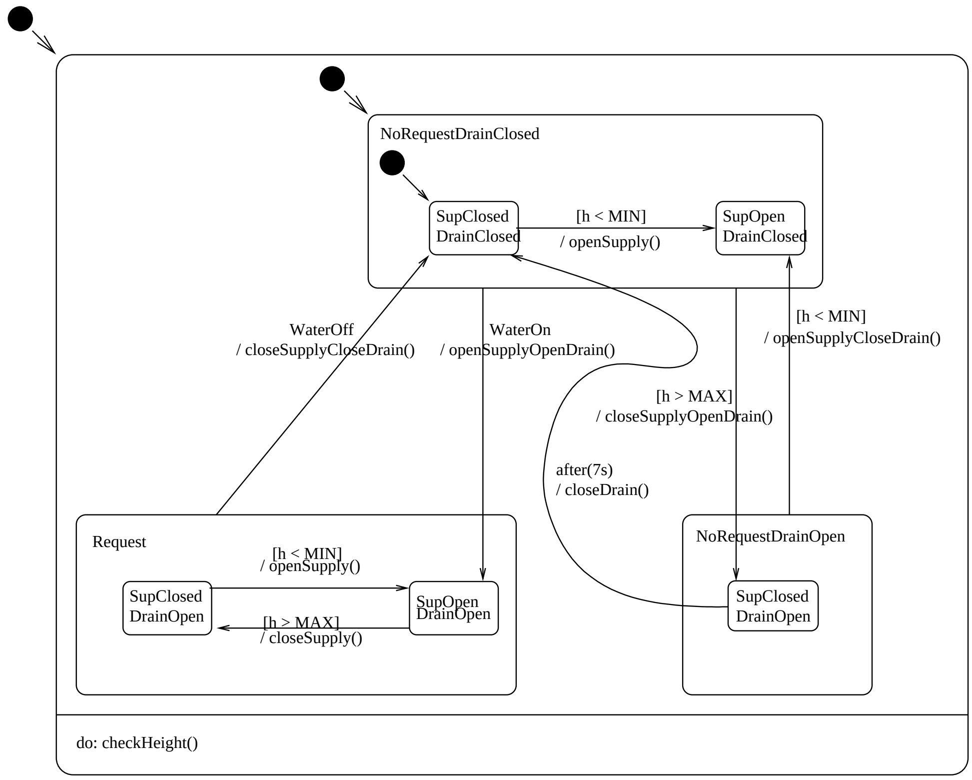

Abbildung 2.11.: Statechart der Wasserbeckenregelung

Das Statechart startet im Zustand SupClosedDrainClosed. Wird das Ereignis WaterOn konsumiert, schaltet die Transition zum Zustand SupOpenDrainOpen und öffnet als Seiteneffekt beide Ventile. Da vorrausgesetzt ist, dass der Zufluss einen höheren Durchfluss hat als der Abfluss (siehe Kapitel 1.1), ist es möglich, dass nach einiger Zeit die Maximumsmarke überschritten wird. Daraufhin wird der Zufluss geschlossen und das Statechart wechselt in den Zustand SupClosed-DrainOpen. Um die Höhe h des Wasserstandes zu bestimmen, wird periodisch die Operation checkHeight() ausgeführt. Durch das Ereignis WaterOff wird zurück in SupClosedDrainClosed gewechselt. Wurde aufgrund eines überhöhten Wasserstandes der Abfluss geöffnet, ohne dass eine Wasseranforderung vorlag (Zustand SupClosedDrainOpen im Superzustand NoRequestDrainOpen) soll das Wasserbecken nicht bis zur Minimumsmarke leerlaufen, sondern nur eine begrenzte Zeit Wasser ablassen, was durch das After-Konstrukt beschrieben wird.

[6] Es liegt die Vermutung nahe, dass im Superzustand NoRequestDrainOpen ebenfalls ein zweiter Zustand namens SupOpenDrainOpen benötigt wird. Da es allerdings keinen sinnvollen Übergang in diesen Zustand gibt, kann darauf verzichtet werden.

Durch das Statechart aus Abb. 2.11 ist eindeutig das Verhalten der Regelungseinheit beschrieben, jedoch nicht das Echtzeitverhalten: Es wird nicht berücksichtigt wie viel Zeit beim Öffnen und Schließen der Ventile -was ein physikalischer Vorgang ist, der nicht in „Null-Zeit“ ausgeführt wird– vergeht. Ist die Zeit, die benötigt wird um die Höhe des kritischen Abschnitts mit Wasser aufzufüllen, kleiner als die für das Schließen des Zuflusses, wird das Wasserbecken trotz Einsatz dieses Controllers überlaufen. In der Semantik der Statecharts ist die Zeit, die Seiteneffekte zur Ausführung benötigen, nicht berücksichtigt. Stattdessen wird angenommen, dass das Statechart-Modell „hinreichend schnell“ auf die Umgebung reagiert. Diese Annahme ist für den Einsatz in harten Echtzeitsystemen nicht ausreichend. Die Operation `checkHight()` wird periodisch ausgeführt. Die Verzögerungen, die bei periodischen Ausführungen entstehen können (siehe Kapitel 2.1.5), müssen bei der Modellierung ebenfalls berücksichtigt werden. Aber auch diese Zeit kann nicht spezifiziert werden.

Wünschenswert wäre es für Operationen Deadlines und Wcets spezifizieren zu können, die angeben wann bzw. in welcher Zeit die Operationen beendet sein müssen. Für periodische Prozesse (wie die Ausführung von `do()`-Operationen) wäre die Angabe einer Periode sinnvoll. Da eine Spezifikation solcher Zeiten nicht möglich ist und die Probleme nicht durch Verwendung der After- oder When-Konstrukte gelöst werden können, sind Statecharts nicht für die Modellierung von Echtzeitverhalten geeignet.

2.3.5. Eignung

Die in diesem Teilkapitel vorgestellten Modelle eignen sich zur Verhaltensspezifikation, wenn es sich dabei nicht um Echtzeitverhalten handelt. Da es in keinem der Modelle die Möglichkeit gibt, Zeitanforderungen (mit Ausnahme der After- und When-Konstrukte bei Statecharts) zu spezifizieren, eignen sie sich in der hier vorgestellten Form nicht, um Echzeitverhalten zu beschreiben. Die Möglichkeiten, die bei Statecharts durch das After- und das When-Konstrukt gegeben sind, lässt es zwar zu einfaches Zeitverhalten zu modellieren, ist für die Anforderungen in dieser Arbeit jedoch nicht ausreichend (siehe Kapitel 2.3.4). Aktivitäts- und Story-Diagramme eignen sich zur Spezifikation operationalem Verhaltens. Da aber in dieser Arbeit reaktives Verhalten im Vordergrund steht, werden diese Modelle zur Spezifikation von Seiteneffekten und anderen Operationen verwendet. Wird für solch eine Operation die Angabe einer Wcet gefordert, wird für den gesamten (durch das Diagramm beschriebenen) Ablauf eine Wcet angegeben.

2.4. Formale Verhaltensmodelle mit Zeit

Neben den Sprachelementen der UML existieren auch formale Modelle zur Verhaltensbeschreibung mit Zeit, wie z.B. das Modell der Timed Automata. Obwohl diese Modelle nicht zur UML gehören, sind sie mit dem Statechart-Modell verwandt, so dass sie in dieser Arbeit als Ausgangspunkt für eine Erweiterung der Statecharts für die Modellierung von Echtzeitsystemen betrachtet werden.

2.4.1. Timed Automata

Vor der Vorstellung der Timed Automata wird ein Überblick über den Endlichen Automaten gegeben, da er Basis der Statecharts und Timed Automata ist. Hier wird er in der Form des Mealy-Automaten[7], wie sie u.a. in [Ram] definiert ist, vorgestellt. Er besteht zum Teil aus einem gerichteten Graphen, dessen Knoten Zustände repräsentieren und dessen Kanten Transitionen, also Zustandsübergänge, darstellen.

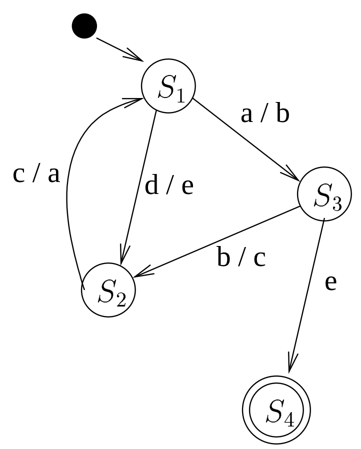

Abbildung 2.12.: Endlicher Automat

Abb. 2.12 zeigt beispielsweise einen Endlichen Automaten mit den Zuständen $S_1, \ldots, S_4$, wobei S_1 als Startzustand, also als derjenige, in dem sich der Automat nach der Initialisierung befindet, und S_4 als Endzustand gekennzeichnet ist. Die Transitionen sind mit Ereignissen (a, ..., e in Abb. 2.12) verknüpft. Ein Ereignis kann außerhalb des Endlichen Automaten ausgelöst werden (z.B. bei einem Fahrkartenautomaten durch das Einwerfen von Geld oder das Drücken einer Taste) oder selbst vom Automaten erzeugt werden. Die hier dargestellte Transition

[7] Bei Endlichen Automaten wird zwischen Mealy- und Moore-Automaten unterschieden. Hier wird der Mealy-Automat vorgestellt. Da jeder der beiden Automaten in den anderen umwandelbar ist, wird der Moore-Automaten hier nicht weiter betrachtet. Für eine Definition und Spezifikation beider Automaten siehe [Ram].

von S_1 nach S_3 schaltet, wenn sich der Endliche Automat im Zustand S_1 befindet und das Ereignis `a` auftritt. Während des Zustandsüberganges wird ein weiteres Ereignis (`b`) als Seiteneffekt erzeugt. Der Zielzustand S_3 kann über 2 verschiedene Transitionen verlassen werden: Eine, die bei Auftritt des Ereignisses `b` zu Zustand S_2 führt und eine zu S_4, die von `e` ausgelöst wird. Letztere erzeugt beim Schalten kein neues Ereignis.

Wie in [HNSY92, LPY97] beschrieben, haben Timed Automata gewisse Ähnlichkeiten mit Endlichen Automaten. Timed Automata werden durch 7-Tupel $(S, S_0, T, E, C, C_c, Inv)$ beschrieben mit

- S ist eine endliche, nicht leere Menge von Locations
- $S_0 \in S$ ist der Startzustand
- E ist eine endliche Menge von Ereignissen
- C ist eine endliche Menge von Uhren
- C_c ist eine Menge von boolschen Aussagen in Abhängigkeit von den Uhren aus C (Zeitbedingungen).
- $Inv : S \rightarrow C_c$ ist eine Funktion, die jeder Location eine Zeitbedingung (Zeitinvariante) zuweist.
- $T \subseteq S \times E \times C_c \times 2^C \times \{\text{true}, \text{false}\} \times S$ ist eine Menge von Transitionen.

Timed Automata bestehen, wie Endlicher Automaten, aus einer endlichen Menge von Zuständen (diese werden jedoch als "Locations" bezeichnet) und einer endlichen Menge von Transitionen. Zusätzlich existieren in Timed Automata Uhren, die es ermöglichen Zeitbedingungen anzugeben. Eine Zeitbedingung ist eine boolsche Kombination von Ausdrücken der Form $t_0 \sim n$ mit $\sim \in \{<, >, =, \leq, \geq\}$, $t_0 \in C$ und $n \in N$. Eine Transition ist neben Start- und Ziellocation mit einem Ereignis, einer Menge von Uhren, einer Zeitbedingung (Zeitguard) und einem so genannten *urgent-Flag* assoziiert. Die Transition kann schalten, wenn der Zeitguard wahr ist und das Ereignis anliegt. Wurde das urgent-Flag auf `true` gesetzt, schaltet die Transition sobald sie aktiviert ist. Ist das urgent-Flag `false`, kann die Transition das Schalten verzögern. Sie schaltet dann zu irgendeinem Zeitpunkt, während das Ereignis anliegt und der Zeitguard wahr ist. Beim Schalten einer Transition werden die mit der Transition assoziierten Uhren auf 0 zurückgesetzt. Somit messen die Uhren die Zeit, die seit Schalten der Transition verstrichen ist. Der Timed Automat besitzt 2 verschiedene Möglichkeiten zu schalten: Verbleibt der Timed Automat in der aktuellen Location, vergeht Zeit. Wird hingegen eine Transition geschaltet, geschieht dies in „Null-Zeit“ und es vergeht keine Zeit. Zeitbedingungen sind nicht nur an Transitionen, sondern auch an Locations geknüpft (in dem Fall werden sie als *Zeitinvarianten* bezeichnet). Während ein Zeitguard angibt, wann eine Transition schalten *kann*, gibt eine Invariante an, wann eine Location spätestens verlassen werden *muss*.

Abb. 2.13 zeigt einen Timed Automat mit den Locations S_1, S_2 und S_3. Der Transition von S_1 nach S_2 ist der Zeitguard $2 \leq t_1$ und das Ereignis e_1 zugeordnet. Schaltet die Transition, wird die Uhr t_0 zurückgesetz. S_1 besitzt die Zeitinvariante $t_1 \leq 2$, S_2 hat $t_0 \leq 10$ und S_3 hat $(t_0 \leq 15) \wedge (t_1 \leq 9)$ als Invariante. Die Transition von S_1 nach S_2 schaltet urgent, während jene von S_2 nach S_3 durch den gestrichelten Pfeil als nicht-urgent gekennzeichnet ist.

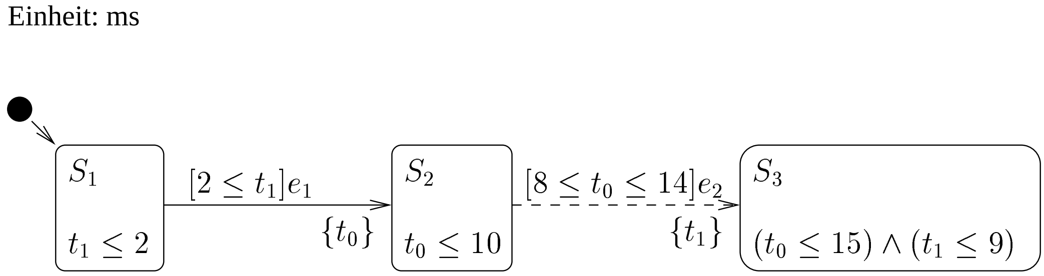

Abbildung 2.13.: Timed Automata

2.4.2. Grundproblem der temporalen Inkonsistenzen

Durch Verwendung von Zeitbedingungen als Oberschranken für den Aufenthalt in Locations und die Assoziation von Transitionen mit Zeitguards kann es innerhalb eines Systems zu Zeitinkonsistenzen kommen. Ein Problem tritt auf, wenn sich das Timed Automata Modell in einer Location aufhält, deren Invariante abläuft und keine aktivierte Transition existiert, über die diese Location verlassen werden kann. Das System ist in dem Moment inkonsistent, da ein so genannter *Time Deadlock* vorliegt.

Ein weiteres Problem in zeitbehafteten Modellen ist das so genannte *Zeno*-Problem: Das Timed Automata Modell aus Abb. 2.14 wechselt während des Intervalls [0; 1] zwischen den beiden Zuständen S_1 und S_2. Beide Transitionen sind an keinerlei Bedingungnen oder Ereignis geknüpft und schalten urgent. Das führt dazu, dass dieser Automat, der zu keinem Time Deadlock führt, innerhalb einer endlichen Zeitspanne unendlich oft schaltet. In diesem Modell führt das zu keinem Problem, da die Annahme getroffen wurde, dass beim Schalten einer Transition keine Zeit vergeht. Das Problem ist aber, dass reale physikalische Systeme somit nur bedingt durch das Modell beschreibbar sind.

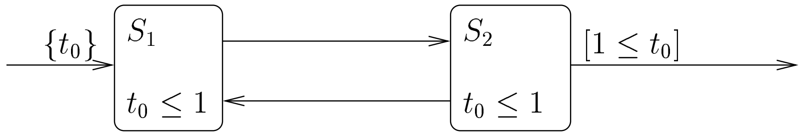

Abbildung 2.14.: Zeno-Problem

2.4.3. Eignung

Mit dem Modell der Timed Automata existiert ein Ansatz, um Echtzeitverhalten zu spezifizieren. Da es bei den Zustandswechseln –mit Ausnahme des Zurücksetzens von Uhren– keine Seiteneffekte gibt und die Timed Automata keine Variablen besitzen, tritt zwar das Zeno-Problem nicht auf, das Modell eignet sich dadurch aber ebenfalls nicht, um komplexes Echtzeitverhalten zu modellieren.

2.5. Zusammenfassung

Software für Echtzeitsysteme kann in verschiedenen Sprachen implementiert werden. Durch die Realtime Specification for Java existiert nun auch ein Ansatz für eine plattformunabhängige Implementierung. Um die Software unabhängig von der verwendeten Programmiersprache zu beschreiben, fehlen jedoch geeignete Diagramme. Durch Statecharts kann komplexes Verhalten ausgedrückt werden, jedoch sind die Möglichkeiten Zeitbeschränkungen zu spezifizieren begrenzt. Mit dem Modell der Timed Automata ist es möglich Zeitbeschränkungen zu modellieren, jedoch fehlt hier die Möglichkeit Aktionen (Seiteneffekte) auszulösen. Um diese Probleme zu beheben, wird im weiteren Verlauf dieser Arbeit ein Modell entstehen, das die Vorteile beider Diagrammarten kombiniert. Neben dem Ziel, Code für ein Echtzeitsystem anhand einer Digrammspezifikation zu generieren, beschäftigt sich ein weiterer Schwerpunkt mit der Vermeidung von Time Deadlocks und Zeno-Verhalten.

3. Verwandte Arbeiten

Im Rahmen verschiedener Arbeiten sind eine Reihe von Statecharterweiterungen und Modelle entstanden, die sich mit der Modellierung temporalen Schaltverhaltens beschäftigen. In Kapitel 3.1 wird eine Auswahl der Erweiterungen und Modelle vorgestellt. Kapitel 3.2 setzt sich mit der Abbildung formaler Modelle auf Echtzeitsysteme auseinander. Für diese Abbildung werden immer die Wcets der eingesetzten Operationen benötigt, für deren Bestimmung Verfahren in Kapitel 3.3 vorgestellt werden.

3.1. Verhaltensmodelle

Statechart-Erweiterungen Harel stellt in [Har98] u.a. 2 Erweiterungen für Statecharts vor, die –zusätzlich zu After- und When-Konstrukt– Zeit in Statecharts einführen: Beim Timeout-Konstrukt spezifiziert der Modellierer ein Ereignis e und eine Zeit t. Die Transition, die mit dem Timeout assoziiert ist, wird nach der Zeit t nach dem letzten Auftreten des Ereignisses e aktiviert. Neben der Verwendung von Timeouts können Seiteneffekte als so genannte *Scheduled Actions* deklariert werden. In dem Fall wird zum Seiteneffekt eine Zeit spezifiziert, die eine Verzögerung der Ausführung angibt. Das hat zur Folge, dass eine Transition schaltet und das Statechart den Zustand wechselt, der Seiteneffekt allerdings erst später ausgeführt wird.

Mit Scheduled Actions und Timeouts existieren weitere Möglichkeiten Zustandswechsel bzw. Aktionsausführungen zeitlich auszulösen. Allerdings wird auch hier lediglich spezifiziert wann der Schaltvorgang beginnt, nicht aber wann er beendet sein soll. Eine Angabe solcher Deadlines ist für eine Modellierung von Echtzeitverhalten unbedingt notwendig.

In [KP92] gibt es die Möglichkeit an Transitionen in so genannten Timed Statecharts ein Zeitintervall $[t_{low}, t_{up}]$ anzugeben. Die Transition schaltet dann frühstens t_{low} nachdem sie durch Anliegen eines Ereignisses und Wahrwerden des Guards aktiviert wurde und spätestens t_{up} nach diesem Aktivierungszeitpunkt. Ein ähnlicher Ansatz wird in [PAS94] verwendet. Dort wird eine Transition ebenfalls mit solch einem Zeitintervall verknüpft, allerdings bestimmt es das Schaltverhalten in der Form, dass der Quellzustand frühstens t_{low} und spätestens t_{up} nach Eintritt wieder verlassen werden kann. Zusätzlich wird jedem Ereignis bei Erzeugung eine Dauer verliehen, die angibt, wie lange es (nach Auslösung) konsumiert werden kann. Guards sind nicht mit Transitionen assoziiert.

Auch diese Konstrukte sind sinnvoll, um den Schaltzeitpunkt anzugeben, liefern aber nicht die für ein Echtzeitsystem notwendigen Informationen, wie z.B. Deadline und Worst Case Execution Time. Dieser Mangel wird durch keine der Statecharterweiterungen behoben. Im Folgenden werden andere Modelle vorgestellt, die sich mit der Modellierung von Echtzeitsystemen beschäftigen.

ROOM ROOM (Realtime Object Orientated Modeling) ist eine Sprache, die auf den Konzepten objektorientierter Modellierung beruht [RSRS99]. Ein durch ROOM beschriebenes System besteht aus 2 verschiedenen Diagrammen: Aktorendiagramm und ROOM Chart. Ein Aktorendiagramm definiert Struktur und (Kommunikations-) Schnittstellen von Aktoren, die mit Klassen in UML-Klassendigrammen vergleichbar sind. Aktoren besitzen Ports, von denen jeder einzelne über einen Konnektor mit einem Port eines anderen Aktors verbunden werden kann. Vorraussetzung ist dabei, dass beide Ports an dasselbe Protokoll geknüpft sind und dass einer der beiden Ports zum Empfang und der Andere zum Senden von Signalen deklariert wurde. Somit wird durch den Einsatz von Konnektoren spezifiziert, welche Aktoren miteinander kommunizieren und sich gegenseitig beeinflussen. Durch die Möglichkeit Aktoren ineinander zu schachteln, können hierarchische Strukturen modelliert werden. Jeder Aktor kann mit einem ROOM Chart assoziiert werden. ROOM Charts sind mit UML Statecharts vergleichbar. Sie sind gegenüber den Statecharts eingeschränkt, da sie keine Hierarchie und Parallelität kennen und keine konzeptionellen Erweiterungen bieten. Daher können ROOM Charts problemlos in Statecharts umgewandelt werden, lösen aber auch nicht die Probleme, die durch die Verwendung von Statecharts auftreten.

UML-RT UML bietet die Möglichkeit neue Stereotypes zu definieren. Dies wird genutzt, um die Konzepte von ROOM mit UML zu beschreiben. Diese Erweiterung von UML wird als UML-RT bezeichnet [SR98]. Dazu werden u.a. die Stereotypes „protocol“, „protocolRole“, „port“, und „capsule“ definiert. Die Aktoren aus ROOM werden in UML-RT als capsules bezeichnet. Port, protocol und protocolRole dienen der Modellierung der Kommunikation. Durch UML-RT lassen sich die ROOM-Konzepte durch die verbreitete UML-Notation beschreiben. Grundlegende konzeptionelle Neuerungen oder Erweiterungen finden sich nicht gegenüber ROOM wieder.

Sowohl ROOM als auch UML-RT liefern neue Ansätze bei der Strukturmodellierung, jedoch keine neuen Konzepte für die Verhaltensmodellierung.

HUppaal In [LPY97] wurde Uppaal als eine Variante der Timed Automata beschrieben, auf die in Kapitel 2.4.1 Bezug genommen wurde. In [DM01] wird die hierarchische Erweiterung HUppaal vorgestellt und gezeigt, wie ein hierarchisches Timed Automata Modell HUppaal auf ein flaches Uppaal Modell abgebildet werden kann. Die hierarchische Erweiterung bietet von Statecharts her bekannte Konstrukte wie parallele AND-States und shallow History. Durch HUppaal können hierarchische Strukturen von Timed Automata einfach modelliert werden –das Fehlen der Möglichkeit komplexe Seiteneffekte zu spezifizieren bleibt jedoch bestehen. Im Extended Hierarchical Timed Automata (ExHTA), der in [GB] definiert wurde und in Kapitel 4.2.1 vorgestellt wird, wurden Konzepte von Uppaal und HUppaal übernommen.

3.2. Design objektorientierter Echtzeitsysteme

Automatische Generierung eines Umsetzungsmodells aus objektorientierten Echtzeitsystemen In [SKW00] ist beschrieben, wie das Modell eines Automaten auf ein so genanntes Umsetzungsmodell abgebildet werden kann. Dabei bezeichnet ein Umsetzungsmodell eine Menge von Threads mitsamt Wcets, Deadlines und zugeordneter Aktionen. Das Automatenmodell ist ausschließlich ereignisgesteuert, wobei die Ereignisse sowohl synchrone als auch asynchrone Kommunikation zulassen. Ein Ereignis löst im Automaten eine Folge von Aktionen aus. Diese Aktionsfolgen werden auf Threads abgebildet. Dabei wird, neben der Schedulability des resultierenden Umsetzungsmodells, angestrebt, die Anzahl der Threads möglichst klein zu halten. Während dieses Abbildungsprozesses werden die Prioritäten und ein *Threshold* für die Threads bestimmt. Der Threshold-Wert wird für das *Preemption Threshold Scheduling (PTS)* –einer Erweiterung des Priority Schedulings– benötigt (siehe auch Abschnitt *PTS für objektorientierte Echtzeitsysteme*). Dies setzt vorraus, dass PTS vom unterliegenden System unterstützt wird. In der Arbeit wird lediglich das Umsetzungsmodell generiert, eine Implementierung der Threads findet nicht statt. Durch die Möglichkeit Deadlines und Wcets zu spezifizieren, kann das Modell auf Threads eines Echtzeitsystems abgebildet werden. Die Vorraussetzung eines Preemption Threshold Schedulers beschränkt das Verfahren auf Systeme, die diesen zur Verfügung stellen. Das Automatenmodell bietet synchrone und asynchrone Kommunikation, es gibt aber keine High-Level Konstrukte wie Parallelität und Hierarchie. Außerdem kann das Schalten von Transitionen nicht von Zeit oder Variablenwerten abhängig gemacht werden.

PTS für objektorientierte Echtzeitsysteme Im vorherigen Abschnitt wurde ein Beispiel gegeben, in dem das objektorientierte Design eines Echtzeitsystems auf eine Menge von Threads abgebildet wird. In der Praxis entsteht dabei häufig eine Vielzahl von Threads, die bei Einsatz eines Priority Schedulers zu vielen Kontextwechseln und somit einem hohen Overhead führt. In dem Fall wird häufig das Preemption Threshold Scheduling (PTS) [KHK02] eingesetzt, das auf dem Priority Scheduling basiert und die Anzahl der Unterbrechungen reduziert. Dazu erhält jeder Thread neben der Priorität einen so genannten Threshold-Wert. Dieser Wert sollte größer als die Priorität sein und regelt den „Grad der Unterbrechbarkeit“ eines Threads, denn wurde er gestartet, kann er nur von Threads unterbrochen werden, deren Priorität größer als der Threshold des aktiven Threads ist. Durch die Ausführung wird also zeitweise die Priorität auf den Wert des Thresholds erhöht. PTS führt zwar aufgrund von weniger Kontexwechseln zu effizienterem Scheduling als reines Priority Scheduling, weist aber bei der Synchronisation bei parallelem Ressourcenzugriff Mängel auf. Um diese Mängel zu beheben, werden in [KHK02] die Protokolle zur Vermeidung von Prioritätsinversion (u.a. Priority Inheritance und Priority Ceiling Emulation) für PTS erweitert und analysiert. Unterstützt das verwendete Echtzeitsystem PTS und jene Protokolle zur Vermeidung der Prioritätsinversion, ist diese Art des Schedulings empfehlenswert.

3.3. Wcet-Analyse

Für die Durchführung einer Schedulinganalyse ist –unabhängig vom eingesetzten Schedulingverfahren– die Kenntnis der Wcets der einzelnen Threads notwendig. Dieses Problem der Wcet-Analyse ist Gegenstand verschiedener Arbeiten, von denen hier einige vorgestellt werden.

Portierbare Wcet-Analyse von Java Byte Code In [BBW00] beschäftigen sich Bernat, Burns und Wellings mit der Wcet-Bestimmung anhand von Java Byte Code (JBC). JBC wird nicht nur aus Java-Source Code erzeugt. Es existieren inzwischen auch Ada- und C-Compiler, die JBC liefern. Da die Analyse erst auf Byte Code-Ebene durchgeführt wird, sind die Ergebnisse unabhängig von der verwendeten Programmiersprache. Dabei wird ausgenutzt, dass es verschiedene Einschränkungen, die ursprünglich aus Sicherheitsgründen eingeführt wurden, bezüglich des Aufbaus und der Struktur von JBC gibt, die eine Analyse nicht nur vereinfachen, sondern erst ermöglichen. So wird der Kontroll- und der Datenfluss bestimmt und anschließend unter der Verwendung von Standard Wcet-Techniken verarbeitet. Dabei muss zusätzlich die Zeit bekannt sein, die jede einzelne JBC-Instruktion im Worst Case benötigt. Diese Informationen können selbstverständlich nicht aus dem Byte Code bestimmt werden und werden deshalb durch ein separates Zeitmodell der Virtuellen Maschine beschrieben. Für jede Virtuelle Maschine und jedes unterliegende System muss ein eigenes Zeitmodell aufgestellt werden. Zusätzlich existieren weitere, für die Wcet-Bestimmungen notwendige, Informationen, die nicht zwangsläufig aus dem JBC extrahiert werden können und auch nicht Bestandteil des Zeitmodells sind. So gibt es Schleifen, von denen die maximale Anzahl an Ausführungen nicht automatisch bestimmt werden kann. In solchen Fällen ist es wünschenswert, Annotationen im JBC zu vermerken. Dies wird dadurch realisiert, dass im Quellcode statische Methoden einer speziellen Klasse aufgerufen werden, deren Rumpf leer ist. Dies ändert nicht die Semantik des Quellcodes, führt aber zu einem größeren Byte Code.

Der Schwerpunkt des Berichts [BBW00] liegt in der Einführung der Technik, um Anmerkungen im JBC unterzubringen. Die Art und Weise das Zeitmodell von Virtuellen Maschinen zu bestimmen, ist momentaner Gegenstand der Forschung.

Wcet-Analyse von Statechart Modellen In [Erp00] wurde ein Tool entwickelt, das aus Statecharts C-Code generiert und anschließend analysiert. Auch hier wird für die Wcet-Bestimmung der Kontrollfluss analysiert, wodurch der Längste auszuführende Pfad im C-Quelltext bestimmt wird. Um die Wcet zu erhalten wird die Assembler-Übersetzung jeder einzelnen C-Anweisung betrachtet und entweder die Anzahl der zur Ausführung benötigten Takte "zusammengezählt", oder es werden direkt die Ausführungszeiten der Assembler-Befehle berücksichtigt. Diese Zeiten können in einigen Fällen vom Hersteller des verwendeten Mikroprozessors bezogen werden.

Die Verfahren zur Bestimmung der Wcets haben gemeinsam, dass die Wcets für *Low-Lewel*-Anweisungen des verwendeten unterliegenden Systems auf einem bestimmten Level bekannt sein müssen. Die Wcets des analysierten Codes entsprechen dann der Summe der Wcets dieser "elementaren" Anweisungen. Während in [Erp00] die Analyse auf Basis der Wcets von elementaren Assembler-Anweisungen stattfindet, basiert das Verfahren aus [BBW00] auf jenen der JBC-Instruktionen.

3.4. Zusammenfassung

Es existieren verschiedene Erweiterungen für Statecharts und auf Statecharts basierende Modellierungssprachen. Diese bieten Möglichkeiten, Zustandswechsel und Aktionsausführungen an Zeitpunkte zu knüpfen, verlangen aber keine Spezifikation der für das Scheduling notwendigen Zeitannotationen. Ohne diese Informationen ist es nicht möglich Echtzeitverhalten zu spezifizieren. Andere Ansätze zeigen, dass ein objektorientiertes Modell auf Threads abgebildet werden kann, wenn diese Informationen vorliegen. Hier wäre ein komplexeres Modell wünschenswert, das neben den Informationen für Echtzeitsysteme auch erweiterte Modellierungsmöglichkeiten bietet. Für die Bestimmung der Wcets existieren verschiedene Ansätze und Verfahren. Diese Arbeit setzt eine Lösung zur Wcet-Bestimmung vorraus und wird eine entsprechende Schnittstelle zum Import der Wcet-Informationen von Blöcken bereitstellen (siehe Kapitel 6.8).

4. Modellierung von Echtzeitverhalten

In den vorrangegangenen Kapiteln wurden verschiedene Diagrammarten und einige Erweiterungen dieser Diagramme vorgestellt und diskutiert. Diese Diagramme eignen sich zwar, um Verhalten zu beschreiben, wenn es jedoch notwendig ist beim modellierten Verhalten Zeitschranken einzuhalten, treten Grenzen der Modellierungsmöglichkeiten auf. In diesem Kapitel wird eine Modellierungstechnik entwickelt, die die Vorteile der bisher diskutierten Modelle kombiniert, um Echtzeitverhalten zu modellieren. Nach einer informellen Beschreibung der Semantik wird das Modell der Extended Hierarchical Timed Automata (ExHTA) vorgestellt. Es wird gezeigt, dass das hier neu entwickelte Realtime Statechart auf den ExHTA abgebildet werden kann und somit seine Semantik durch die Abbildungsregeln und die Semantik der ExHTAs definiert wird. Nach der Festlegung der Semantik wird gezeigt, wie ein Modell auf Zeitinkonsistenzen untersucht werden kann und wie diese Inkonsistenzen behoben werden können.

4.1. Modellierung

In diesem Teilkapitel wird das im Rahmen dieser Arbeit entstandene Modell der Realtime Statecharts vorgestellt. Nach einer informellen Beschreibung der Syntax und Semantik wird es dazu verwendet, das Beipiel der Wasserbeckenregelung aus Kapitel 1.1 zu modellieren.

4.1.1. Realtime Statecharts

Um Echtzeitverhalten modellieren zu können, wird in diesem Kapitel das Statechart-Modell erweitert, indem sinnvolle Zeitanforderungen eingeführt werden. Da dabei Flexibilität im Vordergrund steht, kann das neue Modell wiederum unter ergonomischen Gesichtspunkten erweitert werden, indem einzelne Annotationen zu Highlevel-Konstrukten zusammengefasst und kompakt visualisiert werden. Abb. 4.1 zeigt solch ein erweitertes Statechart, das im Folgenden als *Realtime Statechart* bezeichnet wird.

Ein Realtime Statechart besitzt –mit Ausnahme des After- und des When-Konstruktes– alle Eigenschaften der Statecharts, wie sie in Kapitel 2.3.1 vorgestellt wurden. Zusätzlich existieren in dem Modell *Uhren*, die sich sowohl in den Zuständen als auch an den Transitionen wiederfinden. Die Zustände werden um folgende Attribute erweitert:[1]

- Zeitinvarianten
- Uhren-Resets, die mit den `entry()`- und `exit()`-Methoden assoziiert werden
- Wcets zu den `entry()`-, `do()`- und `exit()`-Methoden
- Periodenintervall für die `do()`-Methode.

Transitionen werden wie folgt erweitert:

- Zeitguards
- Uhren-Resets
- Prioritäten
- Deadlines
- Worst Case Execution Times
- Synchronisationskanal and Synchronisationsart.

In dieser Arbeit sei die Menge von Uhren mit C und die einzelnen Uhren als $t_0, t_1, \ldots \in C$ bezeichnet. Ähnlich wie beim Modell der Timed Automata gibt eine Zeitinvariante an, wann ein Zustand spätestens verlassen werden muss. Das Statechart-Modell darf sich niemals in einem Zustand mit abgelaufener Invariante aufhalten. Um einfachere und somit weniger rechenzeitintensivere Formeln und Algorithmen für eine Analyse (siehe Kapitel 4.3) der Zeitinvarianten zu ermöglichen, wird die Form $\prod_{t_i \in C}(t_i \leq T_i)$, $T_i \in N \cup \{\infty\}$ vorgeschrieben.[2][3] Sind

[1] Die Syntax der Realtime Statecharts ergibt dich zudem auch aus dem in Kapitel 6.2 beschriebenen Metamodell.

[2] In der boolschen Algebra entspricht eine additive Verknüpfung einem logischen 'ODER', eine multiplikative einem logischen 'UND' (siehe hierzu auch Anhang A).

[3] Anstelle der Menge N der natürlichen Zahlen könnten die Konstanten auch aus der Menge R^+ der positiven reellen Zahlen gewählt werden. Da aber jedes System eine minimale Uhrenauflösung besitzt und Zeiten immer als positive ganzzahlige Vielfache dieser Auflösung repräsentiert werden, bietet es sich für die Konstanten an ebenfalls nur positive ganze Zahlen zuzulassen.

Zustände ineinander geschachtelt, muss jeder Substate seine eigene Invariante sowie die aller übergeordneten States einhalten. Wurde eine Zeitinvariante nicht spezifiziert, bzw. nicht für die Uhr i, dann wird sie als $t_i \leq \infty$ interpretiert. Zu jeder `entry()`- und `exit()`-Operation kann eine Menge von Uhren angegeben werden, die bei Ein- bzw. Austritt in bzw. aus dem Zustand auf 0 zurückgesetzt werden. Da die Ausführung der `entry()`- und `exit()`-Operation im Modell implizit, also ohne explizite Deadline geschieht, wird für diese Aktionen keine extra Deadline spezifiziert, sondern ihre Ausführung in die Transitionen integriert. Allerdings ist die Angabe der Worst Case Execution Time notwendig, um dennoch eine realistische Vorhersage bezüglich der Schedulability treffen zu können. Bei der `do()`-Operation muss eine Periode angegeben werden, die beschreibt, wie häufig diese Aktion ausgeführt werden soll. Da es für die Praxis nützlich ist, wird hier eine untere und eine obere Grenze festgelegt ($[p_{low}, p_{up}]$). So kann z.B. angegeben werden, dass ein Datenbus nicht häufiger als alle 50 ms, aber mindestens alle 200 ms abgefragt werden soll.

Eine Transition erhält zusätzlich zu dem Guard, der einen boolschen Ausdrück über Variablen darstellt, noch einen Zeitguard der Form $\prod_{t_i \in C}(a_i \leq t_i \leq b_i)$, $a_i \in N$, $b_i \in N \cup \{\infty\}$, $a_i \leq b_i$. Auch hier wird ein nicht spezifizierter Zeitguard als $0 \leq t_i \leq \infty$ angenommen. Eine Transition ist nur aktiviert, wenn sowohl Guard und Zeitguard wahr sind als auch das mit der Transition assoziierte Ereignis anliegt. Zu einer Transition kann ebenfalls –wie zu `entry()`- und `exit()`-Operationen– eine Menge von Uhren angegeben werden. Jene Uhren werden zurückgesetzt, wenn die Transition schaltet und zwar zu dem Zeitpunkt, zu dem die Transition aktiviert wurde. Sobald eine Transition aktiviert ist, wird sie auch geschaltet. Dies wird als *urgent*-Schaltverhalten bezeichnet. Eine Alternative wäre ein *nicht-urgent* Verhalten, bei dem eine Transition nicht zwingend zum Zeitpunkt der Aktivierung schalten muss. Erst wenn die Invariante des aktuellen Zustands abläuft, muss die Transition schalten. Bei Realtime Statecharts schaltet jede Transition urgent, da dies zum einen das Verhalten wiederspiegelt, das in der Regel intuitiv angenommen wird und zum anderen einen möglichst frühen Start von Operationen verursacht, der häufig in Echtzeitsystemen angestrebt wird. Außerdem ist zu beachten, dass nicht-urgent Transitionen Nicht-Determinismus zur Folge hätten.

Werden mehrere Transitionen zum selben Zeitpunkt aktiviert, soll Nicht-Determinismus in der Form, dass zufällig eine der aktiven Transitionen ausgewählt wird, vermieden werden. Um mehr Determinismus zu erreichen, werden zusätzlich Prioritäten eingeführt. Man könnte die Prioritäten an die Ereignisse koppeln und somit ein Ereignis "wichtiger" als ein anderes machen, jedoch wird dadurch das Problem lediglich verringert, nicht aber behoben, denn es können dann noch immer 2 Transitionen zum selben Zeitpunkt durch dasselbe Ereignis aktiviert werden. Daher werden die Prioritäten nicht für die Ereignisse, sondern für die Transitionen vergeben. An dieser Stelle sei ausdrücklich darauf hingewiesen: *Es schaltet immer die Transition, die zuerst aktiviert war. Nur wenn mehrere Transitionen zu diesem Zeitpunkt aktiviert waren, schaltet diejenige dieser Transitionen, die die höchste Priorität hat.* Wird keine Priorität spezifiziert, wird der Standardwert 1 angenommen.

Des Weiteren wird zu einer Transition die Worst Case Execution Time des Seiteneffektes angegeben und eine Deadline, die spezifiziert, wann der Schaltvorgang beendet sein muss. Die Deadline wird in Form eines Zeitintervalls $[d_{low}; d_{up}]$ angegeben, wodurch modelliert wird, dass der Seiteneffekt spätestens d_{up} nach Transitionsaktivierung beendet sein muss, nicht aber vor d_{low} beendet sein darf. Alternativ zu dieser relativen Deadline kann ein Intervall mit Bezug auf die Uhren angegeben werden. Solch eine absolute Deadline hat dann die Form $\prod_i (t_i \in [d^i_{low}; d^i_{up}])$. Als dritte Möglichkeit können die relative und die absolute Deadline kombiniert werden. Der Zeitpunkt, zu dem der Seiteneffekt beendet sein muss, ergibt sich dann aus dem frühsten der, durch die relative und durch die absolute Deadline spezifizierten, oberen Grenzen. Ebenso wird der Zeitpunkt, zu dem der Seiteneffekt frühstens beendet sein darf, durch die späteste der unteren Grenzen festgelegt (siehe auch Absatz „Dynamische Zeitdifferenzen").

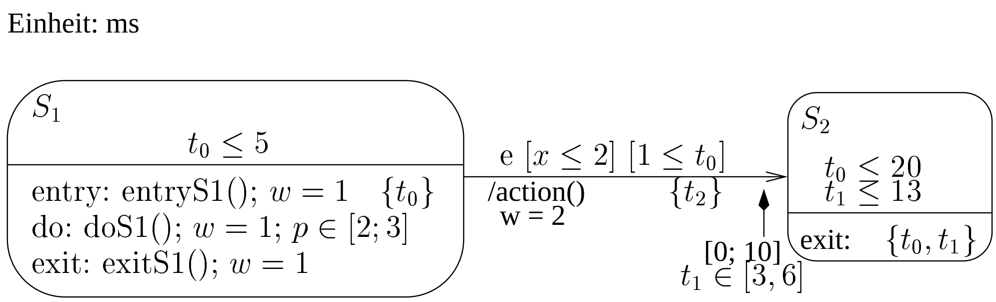

Abbildung 4.1.: Realtime Statechart

Beispiel Abb. 4.1 zeigt ein Realtime Statechart, dessen Zustand S_1 die Zeitinvariante $t_0 \leq 5ms$ besitzt. Bei Eintritt in den Zustand wird die Uhr t_0 auf 0 zurückgesetzt und die Operation `entryS1()` mit der Wcet $w = 1$ ausgeführt. Während des Aufenthaltes in S_1 wird die Methode `doS1()` mit einer Wcet von $w = 1$ und einer Periode, die zwischen $2ms$ und $3ms$ liegt, ausgeführt. Die dargestellte Transition schaltet, wenn das Ereignis `e` anliegt, der Guard $x \leq 2$ und der Zeitguard $1ms \leq t_0$ wahr sind. Beim Schaltvorgang wird als Seiteneffekt `action()` ausgeführt, der eine Wcet von $w = 2ms$ besitzt. Er muss spätestens $10ms$ nach Transitionsaktivierung, aber auch zum Zeitpunkt $t_1 = 6ms$ beendet sein. Der Seiteneffekt darf frühstens bei $t_1 = 3ms$ beendet werden. Zum Aktivierungszeitpunkt der Transition wird die Uhr t_2 zurückgesetzt. Die Invariante des Zustands S_2 lautet $t_0 \leq 20ms \wedge t_1 \leq 13ms$. Bei Verlassen des Zustands werden die Uhren t_0 und t_1 zurückgesetzt. Um eine bessere Übersicht zu erhalten, werden Prioritäten nur dargestellt, wenn sie ungleich dem Standardwert 1 sind.

Angenommen, das Realtime Statechart befinde sich in Zustand S_1 und es gelte seit Eintritt in den Zustand $x = 0$, was den Guard wahr macht. Zum Zeitpunkt $t_0 = 2ms$ trete das Ereignis `e` auf; der Zeitguard ist seit $t_0 = 1ms$ wahr. Dann ist die Transition zu dem Zeitpunkt $t_0 = 2ms$ aktiviert. Aufgrund des urgent-Schaltverhaltens startet sie auch zu dem Zeitpunkt den Seiteneffekt, dessen Ausführung abhängig von der zur Verfügung stehenden Rechenzeit geschedult wird. Bei welchem Wert die Deadline liegt, ist abhängig vom Wert der Uhr t_1, wie in Abb. 4.2 dargestellt. Da t_1 so läuft, dass $t_1 = 6ms$ erreicht ist bevor $t_0 = 2ms + 10ms = 12ms$ gilt, liegt die Deadline bei $t_0 = 7ms$, was $t_1 = 6ms$ entspricht.

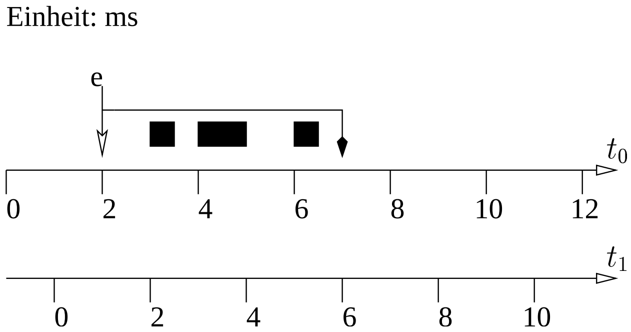

Abbildung 4.2.: Deadline der Transition aus Abb. 4.1

Die Form des Zeitguards wurde auf $\prod_{t_i \in C}(a_i \leq t_i \leq b_i)$ beschränkt. Will der Modellierer einen Zeitguard der Form $\sum_j \prod_{t_i \in C}(a_{ij} \leq t_i \leq b_{ij})$ darstellen, kann er dies expliziet erreichen, indem er für jede „ODER“-Verknüpfte Komponente (also für alle j) eine eigene Transition spezifiziert, diese mit dem Zeitguard $\prod_{t_i \in C}(a_{ij} \leq t_i \leq b_{ij})$ verknüpft und alle diese Transitionen parallel vom Quell- zum Zielzustand führen lässt (siehe Abb. 4.3).

Auslastungsfaktor Wenn das Realtime Statechart schließlich auf einem System ausgeführt wird, beansprucht es Rechenzeit für die Ausführung von Aktionen und Seiteneffekten, sowie für die Verwaltung der Zustände. Durch den Auslastungsfaktor $\nu \in (0; 1]$ kann der Anwender spezifizieren, wie hoch die Prozessorauslastung maximal sein darf.

Dynamische Zeitdifferenzen In Realtime Statecharts existiert nicht nur eine, sondern es existieren mehrere Uhren. Die Uhren können zur Laufzeit zurückgesetzt werden, was bewirkt, dass sie zueinander versetzt laufen können.

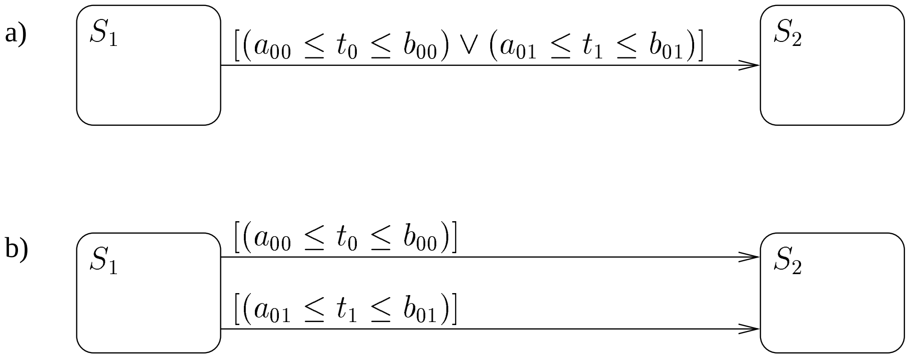

Abbildung 4.3.: a) „ODER“-Verknüpfung im Zeitguard b) Abbildung auf parallele Transitionen

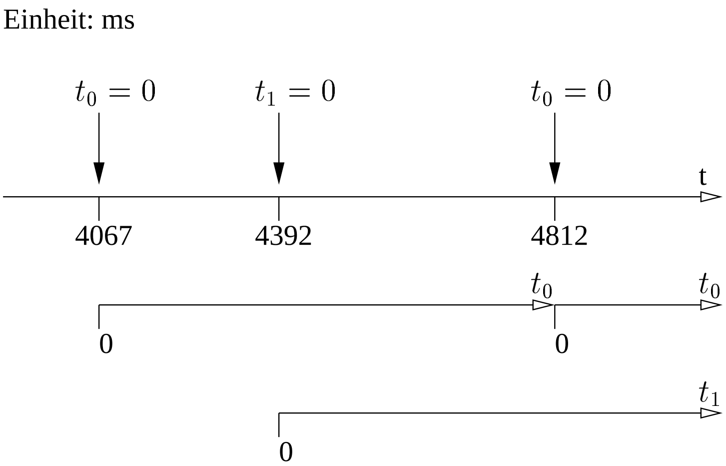

Abbildung 4.4.: Absolute Zeit und Uhren

Abb. 4.4 zeigt die beiden Uhren t_0 und t_1. t bezeichne die Systemzeit und repräsentiere *absolute* Zeitpunkte. Bei $t = 4067ms$ und $t = 4812ms$ wird t_0 zurückgesetzt, bei $t = 4392ms$ wird t_1 auf 0 gesetzt. Im Zeitintervall $[4392ms, 4812ms]$ beträgt die Zeitverschiebung zwischen den Uhren $4392ms - 4067ms = 325ms$. Ab $t = 4812ms$ ist diese $4392ms - 4812ms = -420ms$.

Wichtig ist an dieser Stelle zu bemerken, dass sich die Zeitverschiebung während der Laufzeit ändert. Mal kann eine Uhr in Bezug auf eine andere vorgehen, zu einem späteren Zeitpunkt kann sie dann nachgehen. Um vor der Laufzeit eine Aussage darüber zu treffen, wann eine Transition schaltet und somit eine Uhr zurücksetzt, müsste eine vollständige Zustandsraumanalyse durchgeführt werden. Der Zustandsraum setzt sich nicht nur aus der endlichen Menge der Zustände, sondern auch aus der nicht endlichen Wertemenge der Variablen und Uhren zusammen. Es ist zwar möglich eine Zustandsraumanalyse durchzuführen, dies wäre allerdings sehr aufwändig. Stattdessen wird in Kapitel 4.3 ein Verfahren beschrieben, dass Analysen des temporalen Verhaltens durchführt soweit dies ohne Kenntniss der Zeitdifferenzen möglich ist. Diese Algorithmen sind weniger rechenzeitintensiv als eine vollständige Zustandsraumanalyse und liefern dennoch eine für die Codesynthese hinreichende Aussage bezüglich der temporalen Konsistenz. Somit bleibt die Zeitdifferenz zwischen den Uhren vor der Laufzeit unbekannt. Dies führt zu folgendem Problem:

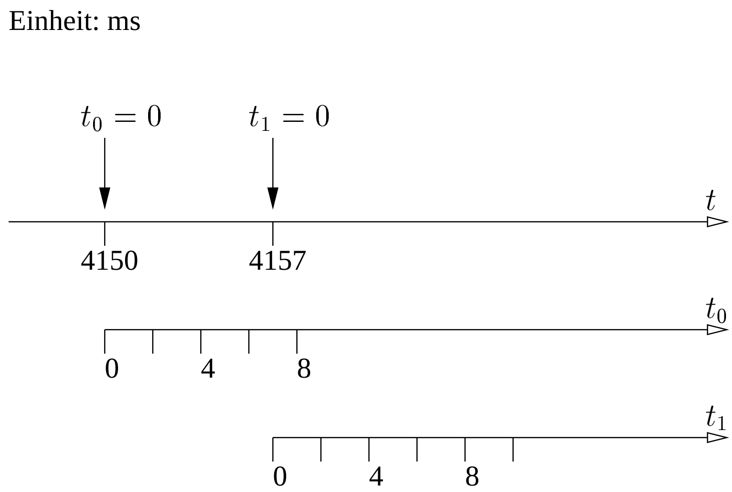

Abbildung 4.5.: Differenz zwischen Uhren

Seien die Uhren t_0 und t_1 wie in Abb. 4.5 zueinander verschoben und sei irgendein Zeitguard $(4ms \leq t_0 \leq 8ms) \wedge (0ms \leq t_1 \leq 10ms)$. Ohne zu wissen, wie die beiden Uhren zueinander stehen, ist zwar sicher, dass der Zeitguard nie länger als 4 Zeiteinheiten wahr ist, es kann aber auch vorkommen, dass der Zeitguard für weniger Zeit (wie im Beispiel nur für $1ms$) oder gar nicht wahr ist. Diese Ungewissheit, die immer auftritt, wenn mehrere Uhren miteinander verknüpft sind, wird erst zur Laufzeit aufgehoben.

High-Level Konstrukte Im Realtime Statechart kann ein `after(x)`-Verhalten modelliert werden, indem bei Eintritt in den Quellzustand eine Uhr (t_0) zurückgesetzt wird, die Zeitinvariante um $t_0 \leq x$ erweitert wird und der Zeitguard der Transition auf $[t_0 \geq x]$ gesetzt wird (siehe Abb. 4.6) .

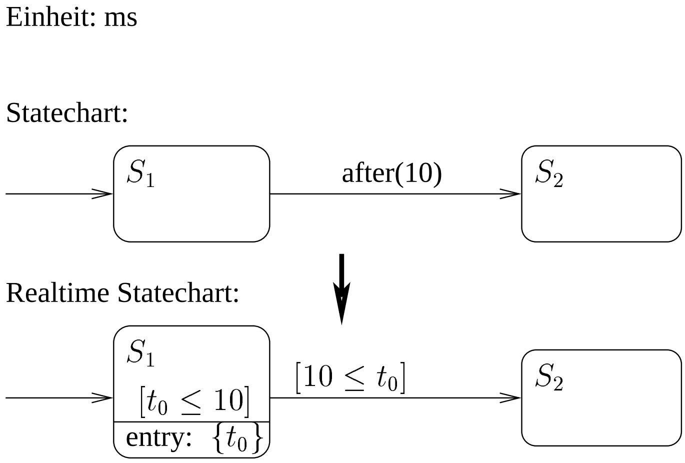

Abbildung 4.6.: High-Level-Konstrukt `after(10)`

Um das When-Konstrukt mit Realtime Statecharts nachzubilden, muss eine Uhr existieren, die mit der lokalen Zeit synchron läuft und alle 24 Stunden zurückgesetzt wird. Sei t_{24} solch eine Uhr, dann wird ein Ausdruck, wie `when(x)` auf einen Zeitguard der Form $x \leq t_{24} \leq x$ abgebildet.

Asynchrone Ereignisse Eine Eigenschaft, die von Statecharts übernommen wurde, ist die Möglichkeit, beim Schalten von Transitionen asynchrone Ereignisse zu versenden und die Aktivierung von Transitionen selbst vom Vorhandensein asynchroner Ereignisse abhängig zu machen. Dabei wird zwischen internen und externen Ereignissen unterschieden. Ein Ereignis heißt *intern*, wenn es von demselben (Realtime) Statechart erzeugt wurde, an das es auch gesendet wird. Wurde es von „außerhalb“ erzeugt –z.B. durch ein weiteres (Realtime) Statechart oder durch eine Benutzereingabe– dann heißt es *extern*. In [Zün01] werden interne Ereignisse bevorzugt gegenüber externen behandelt. Solch eine Vergabe von Prioritäten findet in Realtime Statecharts nicht statt. Stattdessen richtet sich die Abarbeitungsreihenfolge nach den Auftrittszeitpunkten der Ereignisse. Da das Erzeugen asynchroner Ereignisse beim Schalten von Transitionen geschieht, stellen sie somit einen weiteren Seiteneffekt dar. Um hier keine Unterscheidungen machen zu müssen, findet das Erzeugen innerhalb des Seiteneffektes statt. Der Auftrittszeitpunkt entspricht dem Zeitpunkt zu dem der Seiteneffekt, der das Ereignis erzeugt, beendet wird. Zusätzlich gibt es die Möglichkeit, asynchronen Ereignissen Parameter zu übergeben. Diese Parameter können dann bei der

Verarbeitung der Ereignisse wiederum in den Seiteneffekten verwendet werden. Somit eignen sich asynchrone Ereignisse zum Datenaustausch.

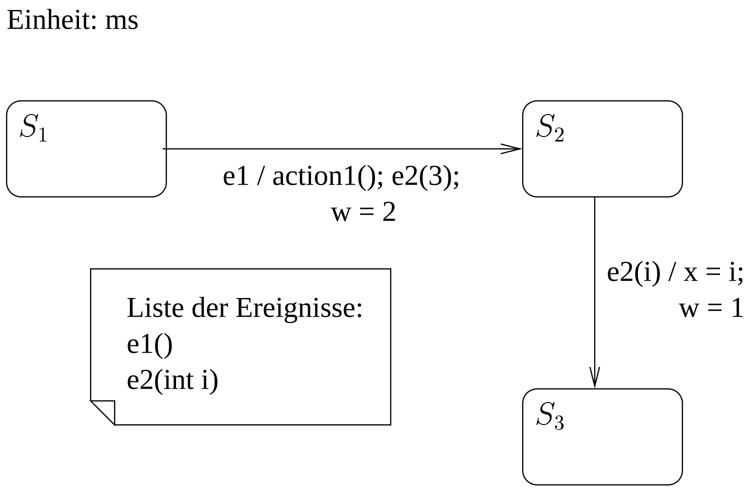

Abbildung 4.7.: Ereignisse mit Parametern

Abb. 4.7 zeigt ein Beispiel, in dem 2 Ereignisse vorkommen: `e1` ohne Parameter und `e2` mit dem `int`-Parameter `i`. Beim Übergang von S_1 nach S_2 wird im Seiteneffekt `e2` mit dem Parameter 3 erzeugt, was beim Übergang von S_2 nach S_3 konsumiert wird. Beim Seiteneffekt wird der Wert des Parameters (3) der internen Variablen x zugewiesen. Die Worst Case Execution Time beträgt für die Transitionen $2ms$ bzw. $1ms$.

Einige Erweiterungen von Statecharts erlauben auch Ereignisse zu spezifizieren, die einen Rückgabewert liefern [Zün01]. In dem Fall könnte der Seiteneffekt der Transition von S_1 nach S_2 zu `action1(); y=e2(3);` erweitert werden. Dann müsste zum einen garantiert werden, dass der Seiteneffekt, der `e2` verarbeitet ein Return-Statement enthält, zum anderen wären die Echtzeitspezifikationen im Konflikt. Die Transition von S_1 nach S_2 wäre erst beendet, wenn die Zuweisung `y=e2(3)` ausgeführt wurde. Dies kann nicht geschehen, bevor `e2` verarbeitet wurde. Also kann das Schalten der ersten Transition erst beendet werden, wenn die zweite schaltet, was ein klarer Wiederspruch zum spezifizierten Schaltverhalten ist.

Synchrone Ereignisse Larsen, Pettersson und Yi definieren in [LPY97] einen Synchronisationsmechanismus für ein (erweitertes) Modell der Timed Automata. Dieser Mechanismus wird für die hier vorgestellten Realtime Statecharts übernommen. Dabei stehen jedem Automaten eine Menge von Kommunikationskanälen zur Verfügung. Befindet sich das Realtime Statechart in einem parallelen Zustand, können sich die einzelnen Prozesse synchronisieren, indem einer über einen bestimmten Kanal sendet, während der Andere vom selben Kanal empfängt. Synchronen Ereignissen können keine Parameter übergeben werden. Sie dienen ausschließlich der Synchronisation und nicht dem Datenaustausch.

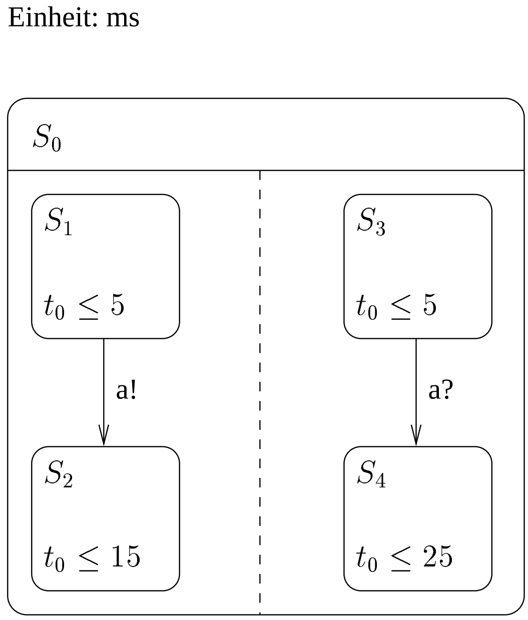

Abbildung 4.8.: Synchronisation

Abb. 4.8 zeigt den parallelen Zustand S_0 eines Realtime Statecharts, in dem beim Übergang von S_1 nach S_2 und S_3 nach S_4 über den Kommunikationskanal `a` synchronisiert wird. Dabei bezeichnen '!' und '?' die komplementären Arten "Senden" und "Empfangen". Eine Transition, die beim Schalten über einen Kanal sendet, kann nur dann schalten, wenn *gleichzeitig* eine weitere Transition schaltet, die von demselben Kanal empfängt. Ebenso ist ein Empfangen nie ohne ein gleichzeitiges Senden möglich.

4.1.2. Modellierungsbeispiel

Betrachtet werde nun erneut das in Kapitel 1.1 beschriebene Beispiel der Wasserbeckenregelung, die nun als Realtime Statechart modelliert wird (siehe Abb. 4.9). Dabei wird der Wasserstand im Becken mit einer Periode von 3 bis 5 Sekunden in der Methode `checkHeight()` überprüft und in der globalen Variablen h gespeichert. Mithilfe dieses Wertes wird auf eine Über- bzw. Unterschreitung der Min- und Maxwerte entsprechend reagiert. Wie beschrieben, gibt es 2 Möglichkeiten, weshalb der Abfluss geöffnet sein kann: Entweder, weil Wasser angefordert wurde oder um ein Überlaufen zu verhindern. Ebenso, wie im Statechart aus Kapitel 2.3.4 existieren 2 Zustände mit dem Namen `SupClosedDrainOpen`, um diese beiden Fälle unterscheiden zu können.

Einheit: s

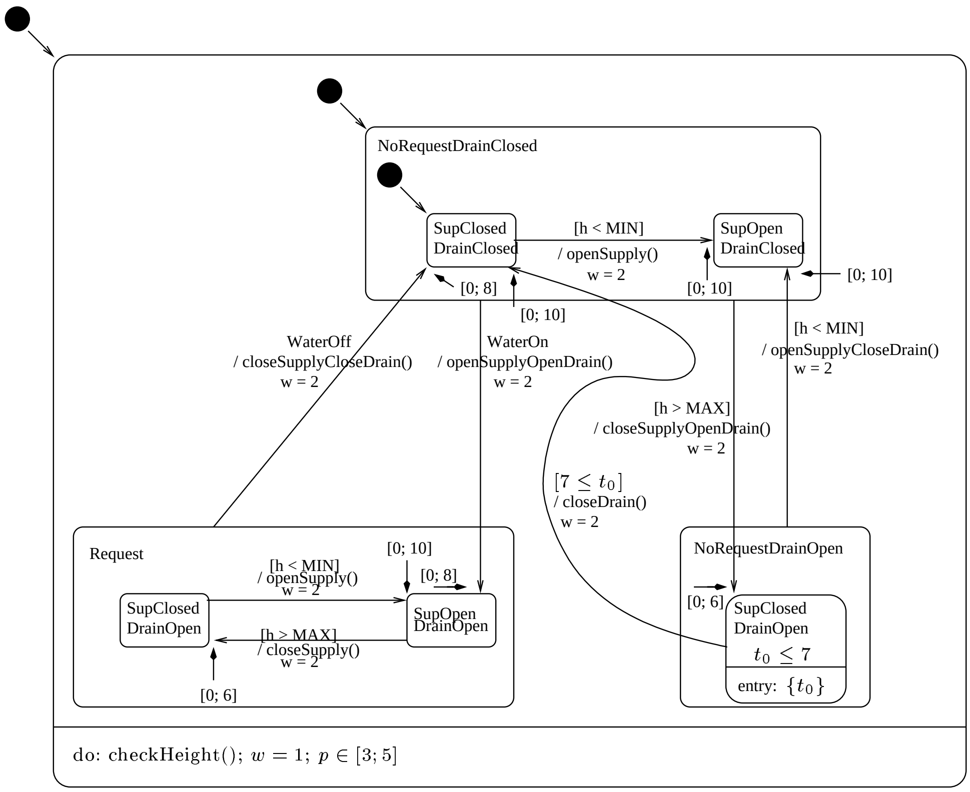

Abbildung 4.9.: Realtime Statechart der Wasserbeckenregelung

Der Beginn der Anforderung von Wasser wird durch das Ereignis `WaterOn` modelliert, während das Ende durch `WaterOff` beschrieben wird. Dabei liegt der Minimalwert des Wasserstandes so, dass bei einer Unterschreitung garantiert noch ausreichend Wasser für die nächsten 10 Sekunden vorhanden ist. In diesem Fall würde der Guard $[h < MIN]$ wahr werden. Daraufhin wird die Aktion `openSupply()` ausgeführt, die nun innerhalb von 10 Sekunden abgeschlossen sein muss, woraus sich die Deadline für die entsprechenden Transitionen ergibt. Abhängig davon, ob gerade Wasser angefordert wird oder nicht, wird zusätzlich `closeDrain()` ausgeführt. Da beide Seiteneffekte parallel ausgeführt werden können, wird als Wcet das Maximum der Wcets beider Aktionen angenommen. Die Worst Case Execution Time aller 4 Aktionen in diesem Beispiel sei $w = 2s$. Um auch bei der Anforderung von sehr viel Wasser die Versorgung garantieren zu können, wird vorrausgesetzt, dass der Zufluss im geöffneten Zustand einen höheren Durchfluss hat als der Abfluss.

Die Maximumsmarke ist so kalkuliert, dass auch im Worst Case (geöffneter Zufluss, geschlossener Abfluss und unter Berücksichtigung einer gewissen Druckschwankung) erst 6 Sekunden später das Becken überläuft. Somit ergibt sich für die Transitionen mit dem Guard $[h > MAX]$ eine Deadline von 6 Sekunden.

Wird das Ereignis `WaterOn` abgehandelt, wird nicht nur der Ab-, sonder um auch gleichzeitig das Becken wieder aufzufüllen, der Zufluss geöffnet. Erst nach Eintritt in den kritischen Bereich wird der Zufluss geschlossen. `WaterOff` schließt wiederrum Zu- und Abfluss. Die Reaktionen auf diese beiden Ereignisse sind für die Sicherheit des Systems nicht zeitkritisch, sollen aber trotzdem innerhalb einer gewissen Zeit (8 Sekunden) ausgeführt werden.

Wird der Maximalwert überschritten und der Abfluss geöffnet, soll das Wasserbecken nicht bis zum Minimalwert, sondern nur so weit geleert werden, bis sich das System wieder in einem sicheren Zustand befindet. Deshalb wird der Zustand, in dem das Becken geleert wird, bereits nach 7 Sekunden wieder verlassen. Beim Wechsel in den Zustand `NoRequestDrainOpen` wird die Uhr t_0 zurückgesetzt. Der Zustand soll auf jeden Fall nach 7 Sekunden, also spätestens zum Zeitpunkt $t_0 = 7$ wieder verlassen werden, was durch die Invariante $t_0 \leq 7$ beschrieben wird. Das der Zustand nach dieser Zeit auch wieder verlassen werden kann, wird durch den Zeitguard $[t_0 \geq 7]$ der Transition zum Zustand `SupClosedDrainClosed` sichergestellt.

4.2. Semantik der Realtime Statecharts

Um die Semantik der Realtime Statecharts zu definieren, wird zunächst das Modell der Extended Hierarchical Timed Automata (ExHTA), wie es in [GB] definiert wurde, vorgestellt. Anschließend werden Regeln angegeben, wie jedes Realtime Statechart auf einen ExHTA abgebildet werden kann. Da die Semantik der

ExHTAs in [GB] definiert wurde, implizieren die Abbildungsregeln die Semantik der Realtime Statecharts.

4.2.1. Extended Hierarchical Timed Automata (ExHTA)

Der ExHTA hat sowohl mit dem Realtime Statechart als auch mit dem Modell der Timed Automata Ähnlichkeiten. Im ExHTA existieren ebenfalls Uhren, die an den Annotationen der Locations und Transitionen auftauchen. Eine Location besitzt folgende Eigenschaften:

- Zeitinvariante
- Hierarchie
- Parallelität.

Transitionen bestehen aus folgenden Attributen:

- Ereignis
- Guard
- Zeitguard
- Uhren-Resets
- Datenzuweisungen
- Priorität
- Synchronisationskanal and Synchronisationsart.

Eine Zeitinvariante gibt –wie beim Realtime Statechart– an, bis zu welchem Zeitpunkt in dem aktuellen Zustand verweilt werden darf. Zustände können –wie in Statecharts und Realtime Statecharts– ineinander geschachtelt sein und auch parallel zueinander gültig sein. Transitionen sind mit Ereignis, Guard und Zeitguard assoziiert, die den Zeitpunkt der Aktivierung bestimmen. Dabei gilt eine Transition als aktiviert, wenn das Ereignis anliegt und sowohl Guard als auch Zeitguard wahr sind. Bei gleichzeitiger Aktivierung mehrerer Transitionen schaltet diejenige, mit der höheren Priorität. Auch hier gilt: *Es schaltet immer die Transition, die zuerst aktiviert war. Nur wenn mehrere Transitionen zu diesem Zeitpunkt aktiviert waren, schaltet diejenige dieser Transitionen, die die höchste Priorität hat.* Eine Priorität mit dem Wert 0 ist gleichbedeutend mit einer nicht-urgent schaltenden Transition. Besitzt eine Transition urgent Schaltverhalten (Priorität ungleich 0), schaltet sie sofort zu dem Zeitpunkt, zu dem sie aktiviert wurde (vorausgesetzt, sie steht nicht mit einer höherpriorisierten Transition im Konflikt, die zum selben Zeitpunkt aktiviert wurde). Eine Transition mit nicht-urgent Verhalten kann ebenfalls zum Aktivierungszeitpunkt schalten, kann dies aber auch verzögern. Auch hier wird der Standardwert 1 als Priorität angenommen, falls diese nicht spezifiziert wurde. Schaltet eine Transition, werden die mit ihr assoziierten Uhren zurückgesetzt und die Datenzuweisungen vorgenommen. Dabei kann eine Datenzuweisung sowohl einfache Variablenmanipulationen als auch die Ausführung komplexer Operationen beinhalten. Wird eine Transition an einen Synchronisationskanal gebunden, kann sie nur schalten, wenn –wie bei der Synchronisation beim Realtime Statechart– gleichzeitig eine weitere an denselben Kanal gebundene Transition schaltet. Dabei muss eine Transition über den Kanal senden, während die Andere über den Kanal empfängt.

Beim Schaltverhalten des ExHTA werden ebenfalls die 2 Schritte der Timed Automata unterschieden: Verbleibt er in der aktuellen Location, vergeht Zeit. Wird hingegen eine Transition geschaltet, geschieht dies in „Null-Zeit“ und es vergeht keine Zeit.

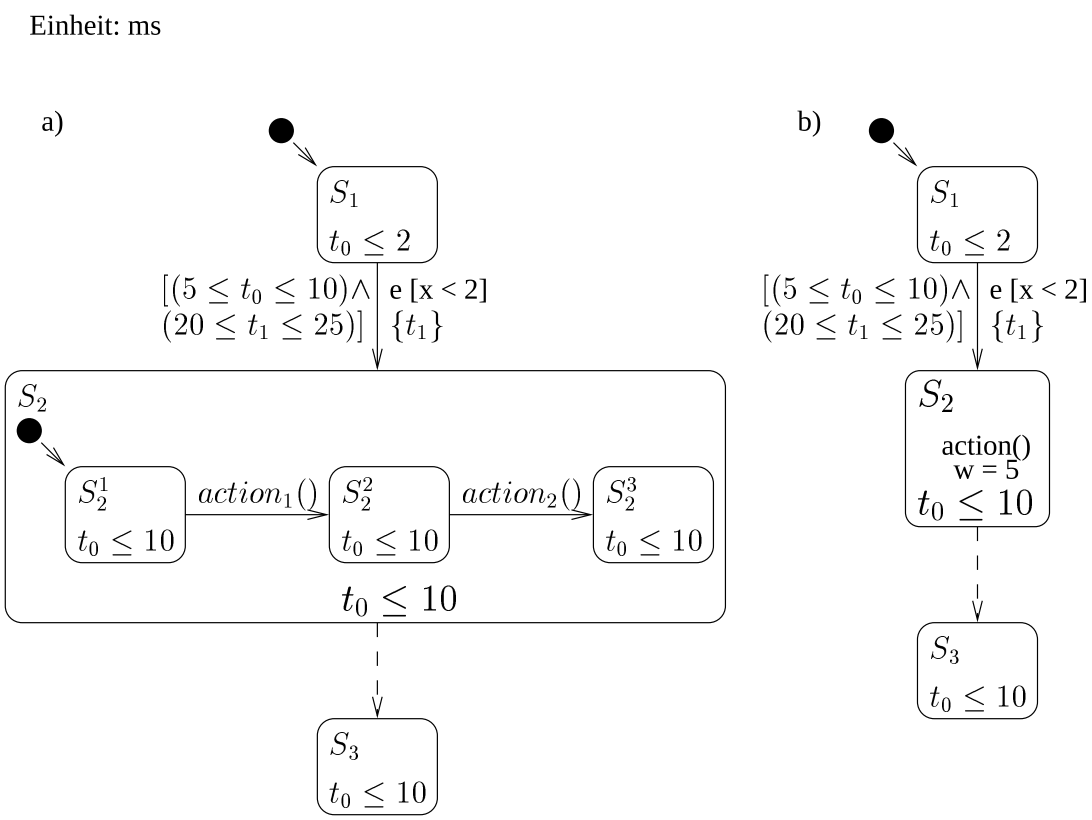

Abbildung 4.10.: Extended Hierarchical Timed Automata

Abb. 4.10 a) zeigt einen ExHTA. Die Notation wurde von den Realtime Statecharts weitgehend übernommen. $action_1$ und $action_2$ sind Datenzuweisungen. Eine hierarchische Location, wie S_2, deren Invariante mit der der Sublocations übereinstimmt und deren innere Transitionen und Locations eine sequentielle Abfolge von Datenzuweisungen darstellen, werden im weiteren Verlauf, wie in Abb. 4.10 b) dargestellt, abgekürzt. Solch ein Zustand wird frühstens nach Beendigung der einzelnen *atomaren* Aktionen $action_1$, $action_2$, ..., $action_n$ verlassen, was nach oben durch die Summe der Wcets der einzelnen atomaren Aktionen beschränkt ist und im Diagramm durch w dargestellt wird. Nach aussen verhält sich der hierarchische Zustand S_2 wie ein Zustand, in dem eine Aktion ausgeführt wird. Die Transition von S_2 nach S_3 wurde als nicht-urgent spezifiziert (Priorität ist 0), was durch einen gestrichelten Pfeil gekennzeichnet ist. Auch hier werden die übrigen Prioritäten nur dargestellt, wenn sie ungleich dem Standardwert 1 sind, um eine bessere Übersicht zu erhalten.

4.2.2. Abbildung von Realtime Statecharts auf ExHTA

Zur semantischen Fundierung wird ein Realtime Statechart auf einen Extended Hierarchical Timed Automata abgebildet. Außerdem wird das zum Realtime Statechart korrespondierende Modell in Kapitel 4.3 dazu genutzt, Inkonsistenzen zu erkennen und zu beheben. Für die Abbildung der einzelnen Komponenten der Realtime Statecharts gelten folgende Regeln:

Nicht-hierarchische Zustände

Diese Zustände werden nahezu “1:1” in Locations umgewandelt (siehe Abb. 4.11). Dabei wird sowohl der Name als auch die Invariante beibehalten. Die `do()`-Operation wird in Form einer Datenzuweisung an eine Transition gebunden, deren Quell- und Zielzustand identisch sind. Die periodische Ausführung der Transition wird durch Zeitguard und Uhren-Reset modelliert. Die `entry()`- und `exit()`-Operationen sowie die Menge von Uhren-Resets (CR_{entry} bzw. CR_{exit}) werden aus den Zuständen ausgelagert (siehe Abschnitt „Transitionen“). Eine Aktion wird nicht mit der Location assoziiert. Aufgrund dessen und weil sie einen Zustand repräsentiert, werden diese Locations in dieser Arbeit auch als *Statelocations* bezeichnet.

Transitionen

Beim Schalten einer Realtime Statechart Transition werden die `exit()`-Methode des Quellzustands, der Seiteneffekt der Transition und die `entry()`-Methode des Zielzustands sequentiell ausgeführt. Um dies im ExHTA darzustellen, werden die

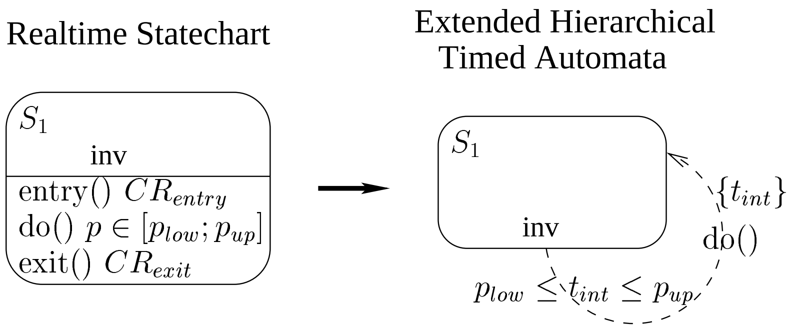

Abbildung 4.11.: Abbildung nicht-hierarchischer Zustände

Operationen in atomare Aktionen zerlegt und (wie im Beispiel aus Abb. 4.10 demonstriert) in einer eigenen hierarchischen Location zusammengefasst. Da innerhalb dieser hierarchischen Location Aktionen ausgeführt werden, werden solche Locations auch als *Actionlocations* bezeichnet. Somit wird eine Transition aus einem Realtime Statechart auf eine Location und 2 Transitionen im ExHTA abgebildet. Die Wcet dieser 3 sequentiell auszuführenden Operationen ergibt sich aus der Summe der einzelenen Wcets.

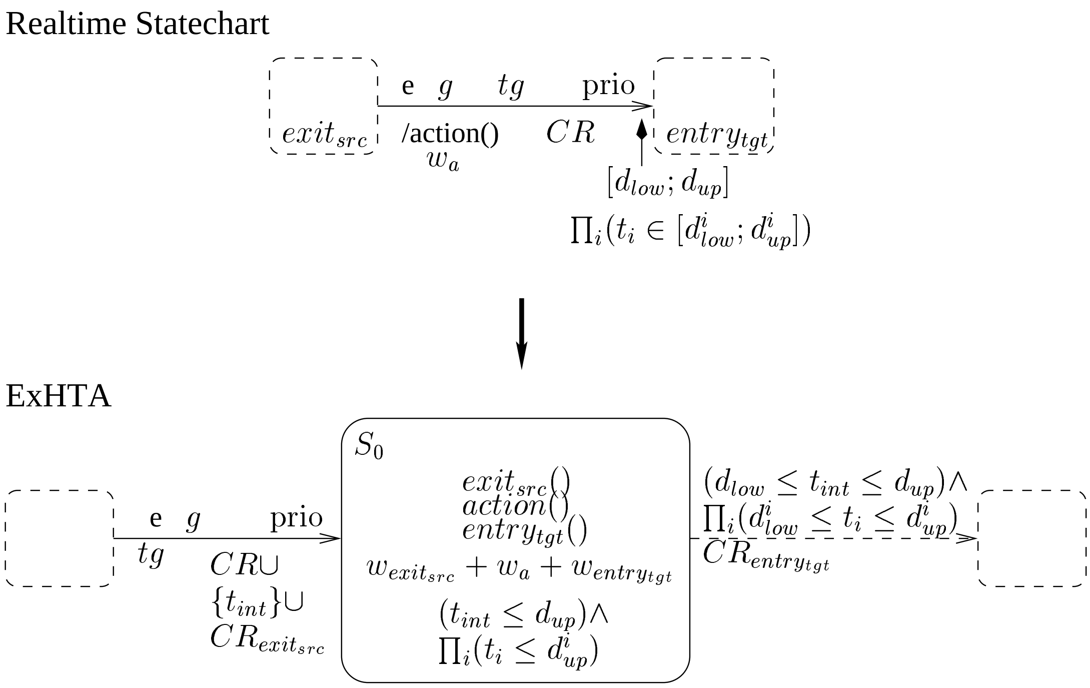

Abbildung 4.12.: Abbildung von Transitionen

In Abb. 4.12 ist gezeigt, dass sich die Attribute aus dem Statechart auf die 3 Komponenten verteilen: Guard g, Zeitguard tg und Ereignis e sollen nach wie vor die auszuführende Aktion auslösen und tauchen daher unverändert an der ersten

Transition (jene, die zur Actionlocation hinführt) auf. Die zurückzusetzenden Uhren CR werden ebenfalls mit dieser Transition assoziiert. Quell- und Zielzustand sind nur angedeutet und liefern die Methoden $exit_{src}$ und $entry_{tgt}$.

Die Deadline findet sich an mehreren Stellen wieder: Zum einen wird die Zeitinvariante der Actionlocation so erweitert, dass sie spätestens zum Ende der Deadline verlassen sein muss, zum anderen soll die zweite Transition (also jene, die zum Verlassen der Actionlocation führt) nur schalten können, nachdem der Seiteneffekt frühstens beendet sein durfte. Da die Actionlocation bis zum Ablauf ihrer Invarianten, also bis zum Ende der Deadline des Seiteneffektes, verlassen sein muss, wird das Ende der Deadline (die minimal die Wcet der Actionlocation zur Verfügung stellen muss) als obere Grenze des Zeitguards gesetzt. Diese Transition schaltet nicht urgent, wodurch bei einer Aktivierung die Operationen innerhalb der Actionlocation nicht unterbrochen werden müssen. Um den Teil der Deadline, der eine relative Deadline spezifiziert, im Extended Hierarchical Timed Automata umzusetzten, wird eine Uhr benötigt, die zum Schaltzeitpunkt zurückgesetzt wurde. Hierfür wird die interne Uhr t_{int} verwendet. Die Uhren $CR_{exit_{src}}$ und $CR_{entry_{tgt}}$, die an die `entry()`- und `exit()`-Operationen des Quell- und Zielzustands gebunden sind, werden mit den beiden Transitionen assoziiert.

Unterscheidet man Action- und Statelocations und bedenkt man, dass 2 Zustände im Realtime Statechart immer über eine Transition verbunden sind, entsteht bei der Abbildung ein bipartieter Graph.

Beispiel Abb. 4.13 zeigt einen Ausschnitt aus Abb. 4.9 und Abb. 4.14 den dazugehörigen Extended Hierarchical Timed Automata, der nach obigen Abbildungsregeln entstanden ist.

Hierarchie, Parallelität und History

Da sowohl Realtime Statechart als auch ExHTA Hierarchie, Parallelität und History kennen, können diese Konstrukte „1:1“ übernommen werden.

Besonderheiten

Eine Anwendung der oben beschriebenen Abbildungsregeln führt zu einem ExHTA mit spezieller Form. Folgende Eigenschaften gelten für ExHTA, die aus einem Realtime Statechart entstanden sind:

- Auf eine State- folgt eine Actionlocation und auf eine Action- eine Statelocation. Deshalb kann man den ExHTA als (gerichteten) bipartieten Graphen auffassen.

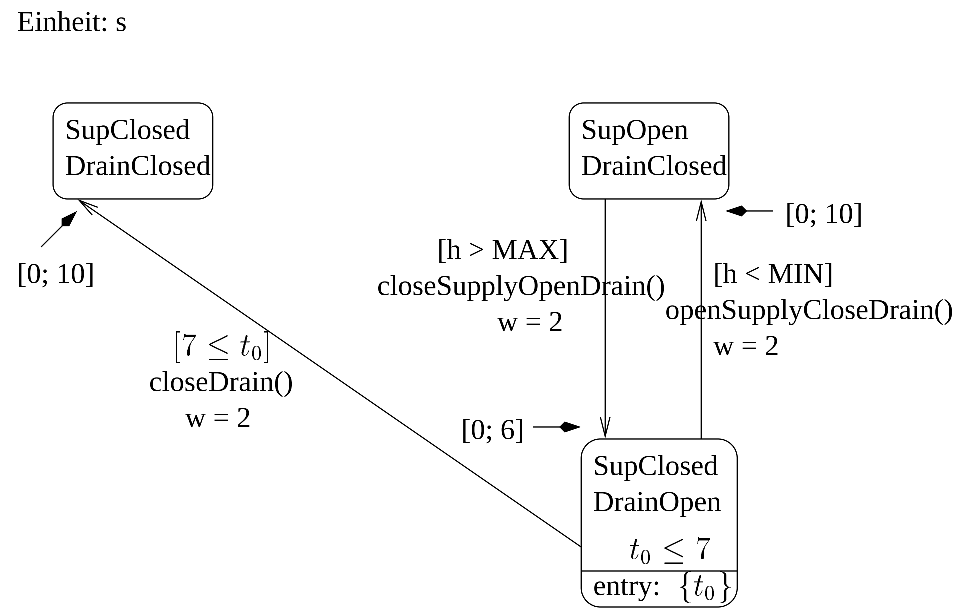

Abbildung 4.13.: Ausschnitt des Realtime Statecharts aus Abb.4.9

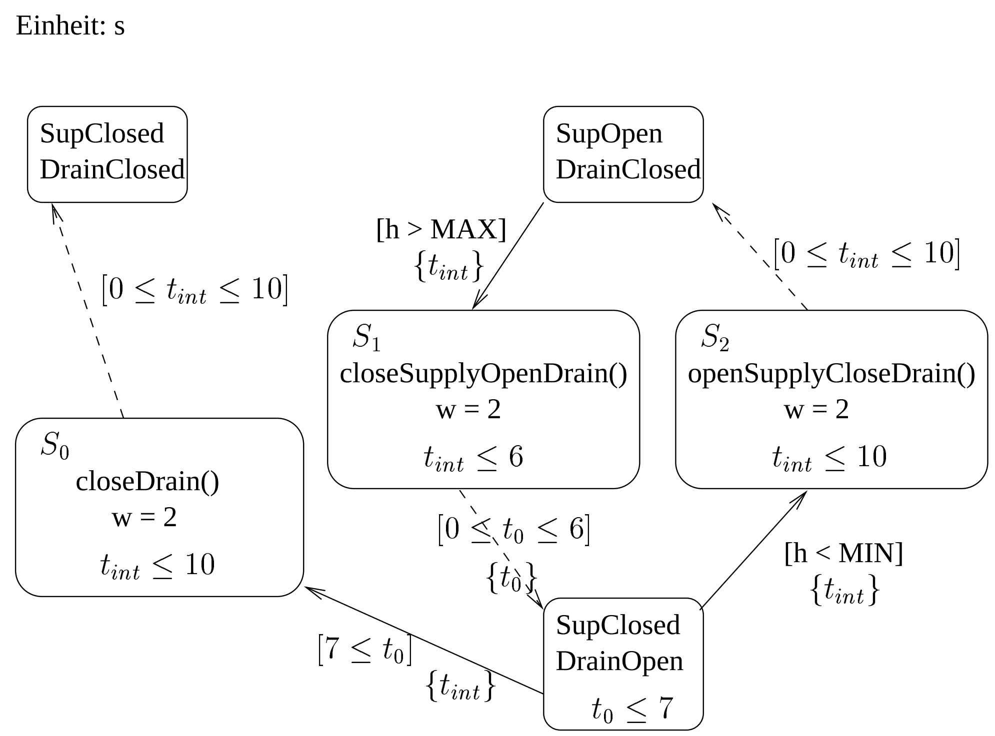

Abbildung 4.14.: ExHTA zu Abb. 4.13

- Von einer Actionlocation führt genau eine Transition weg, an die ein Zeitguard, jedoch kein Guard und kein Ereignis, gekoppelt ist. Der Zeitguard ist so beschaffen, dass kein Time Deadlock (siehe Kapitel 4.3.1) auftreten kann, d.h., es ist sichergestellt, dass die Transition noch vor Ablauf der Zeitinvarianten aktiviert wird.
- Zu einer Actionlocation führt genau eine Transition hin. Hier können sowohl Zeitguard als auch Guard und Ereignis auftreten. Diese entsprechen denen, die vom Anwender im Realtime Statechart spezifiziert wurden.
- Da die Zeitguards „1:1" aus den Realtime Statecharts übernommen werden, haben sie im ExHTA ebenfalls die Form $\prod_{t_i \in C}(a_i \leq t_i \leq b_i)$, $a_i \in N$, $b_i \in N \cup \{\infty\}$, $a_i \leq b_i$.
- Die Zeitinvarianten der Zustände aus den Realtime Statecharts werden ebenfalls im ExHTA übernommen, aber gegebenenfalls aufgrund der Deadline erweitert. Diese Erweiterung ändert jedoch nicht die Form $\prod_{t_i \in C}(t_i \leq T_i)$, $T_i \in N \cup \{\infty\}$.

4.3. Temporale Inkonsistenzen

Im Grundlagenkapitel zu den Timed Automata (2.4) wurde der Time Deadlock und das Zeno-Verhalten beschrieben. Ein Automat zeigt Zeno-Verhalten, wenn er innerhalb einer endlichen Zeitspanne unendlich oft schaltet. Ein Time Deadlock liegt vor, wenn eine Location verlassen werden muss, aber nicht verlassen werden kann.

Prinzipiell ist in den Realtime Statecharts Zeno-Verhalten möglich, da in einer Deadline der Form $[0, t_{up}]$ der Fall enthalten ist, dass die Transition in „Null-Zeit" schaltet. Durch Zyklen kann dann Verhalten spezifiziert werden in dem Transitionen in einem endlichen Zeitintervall unendlich oft schalten (siehe Kapitel 2.4.2). Dieser Fall wäre ausgeschlossen, wenn die Deadline auf die Form $(0, t_{up}]$ beschränkt wäre. Da eine Transition jedoch niemals in „Null-Zeit" schaltet und immer eine minimale Ausführungszeit min_{et} benötigt, verhält sich eine Deadline $[0, t_{up}]$ faktisch, wie $[min_{et}, t_{up}]$, $min_{et} > 0$.[4] Unter Berücksichtigung der minimalen Schaltzeit min_{et} kann kein Zeno-Verhalten auftreten.

Time Deadlocks finden sich sowohl im Realtime Statechart als auch im ExHTA wieder. In diesem Teilkapitel werden Algorithmen zur Erkennung und Behebung von Time Deadlocks mit Hilfe einer Analyse auf Basis der Syntax des entsprechenden ExHTA-Modells vorgestellt.

[4] Aus theoretischen Gründen müsste in Grafiken und Formeln die Intervallgrenze 0 ausgeschlossen werden. Aus praktischen Gründen bleibt sie erhalten, um mit der Normalform $\prod_{t_i \in C}(a_i \leq t_i \leq b_i)$ konform zu sein. In der Praxis ändert sich das Verhalten dadurch nicht.

4.3.1. Erkennung von Inkonsistenzen

Mit Hilfe der Zeitinvarianten kann spezifiziert werden, dass eine Location spätestens bis zu einem bestimmten Zeitpunkt verlassen sein muss. Locations werden über Transitionen verlassen, welche nur schalten können, wenn sie aktiviert sind. Durch die 3 Faktoren

- Guard
- Ereignis
- Zeitguard

wird spezifiziert wann die entsprechende Transition aktiviert ist. Die Konsistenzprüfungen werden dadurch vereinfacht, dass die Zeitguards in der Form $\prod_{t_i \in C}(a_i \leq t_i \leq b_i)$ und Zeitinvarianten in der Form $\prod_{t_i \in C}(t_i \leq T_i)$ vorliegen (vgl. Kapitel 4.2.2).

Neben den Zeitkonstanten ist der Auslastungsfaktor ν für die Konsistenzprüfung wichtig. w bezeichnet die Worst Case Execution Time einer Aktion, also die Zeit, die die Aktion maximal verbraucht, wenn sie nicht durch den Scheduler unterbrochen wird. Da für den Automaten aber in der Regel nicht 100% Rechenleistung zur Verfügung stehen, wird die Aktion unterbrochen und endet dann im Worst Case erst nach w/ν.

Obwohl vor der Laufzeit keine vollständige Zustandsraumanalyse durchgeführt wird, wird im Folgenden gezeigt werden, dass jedes Realtime Statechart und somit auch jeder ExHTA, der aus einer Abbildung eines Realtime Statecharts entstanden ist, in eine von drei Sicherheitskategorien bezüglich seines Zeitverhaltens eingestuft werden kann. Ein ExHTA ist

- *zeitkonform*, wenn alle Zeitrestriktionen garantiert eingehalten werden
- *unsicher*, wenn es vom Laufzeitverhalten abhängt, ob alle Zeitrestriktionen eingehalten werden
- *zeitinkonsistent*, wenn mindestens eine Zeitschranke nicht eingehalten werden kann.

Ein zeitinkonsistenter ExHTA muss modifiziert werden, bis er zumindest als unsicher eingestuft werden kann. Folgende Zeitinkonsistenzen können auftreten und werden in den folgenden Kapiteln beschrieben:

1. Schwache Zeitguards

2. Schwache Invarianten

3. Tote Transitionen

4. Time Deadlocks

Dabei führen häufig die 2 zuerst genannten Konflikte zu toten Transitionen und Time Deadlocks, so dass ein ExHTA zuerst auf die Konflikte 1 und 2 überprüft wird und anschließend auf 3 und 4.

Schwache Zeitguards

Beschreibung Der erste Konflikt, der hier vorgestellt wird, ist der *Schwache Zeitguard* Konflikt. Er ist der einzige Konflikt, der von der Wcet einer Operation abhängig ist. Wechselt der Automat in eine Actionlocation, soll er sich dort eine bestimmte Zeit aufhalten und eine Aktion ausführen. Das impliziert jedoch, dass der Automat nicht mehr in die Location wechseln darf, wenn er vor Ablauf der Zeitinvarianten nicht mehr ausreichend Zeit hat, um die Aktion auszuführen. Ist der Zeitguard der zur Actionlocation hinführenden Transition so gefasst, dass es möglich ist so spät zu schalten, dass für die Aktionsausführung nicht mehr genug Zeit ist, dann liegt ein schwacher Zeitguard vor.

Beispiel: Abb. 4.15 zeigt den Zustand S_0, der spätestens bei $t_0 = 10$ verlassen werden muss.[5] Für die dort ausgeführte Aktion `a()` gilt $w = 3$. Sei $\nu = 0.75$, dann gilt $w/\nu = 4$. Nun ist es aber möglich, dass die Transition von S_1 erst zum Zeitpunkt $t_0 = 7$ schaltet und `a()` somit erst dann gestartet wird. Wenn das passiert, kann die Invariante im Worst Case nicht eingehalten werden. Die Betrachtung in welcher Relation Zeitguard und Invariante zueinander stehen, muss für die Uhr t_{int} nicht durchgeführt werden, da sie erst beim Wechsel nach S_0 zurückgesetzt wird.

Erkennung Gibt es im ExHTA mit dem Auslastungsfaktor ν eine Location l, die mit der Zeitinvarianten $\prod_{t_i \in C}(t_i \leq T_i)$ assoziiert ist und deren Aktion eine Wcet w hat, sowie eine zu l hinführende Transition mit dem Zeitguard $\prod_{t_i \in C}(a_i < t_i < b_i)$ und der Clock-Reset-Menge CR, dann liegt ein schwacher Zeitguard vor, mit $\exists t_i \notin CR | b_i > T_i - w/\nu$. Der Konflikt liegt also vor, wenn die Ungleichung für eine der Uhren, die nicht beim Schalten der Transition zurückgesetzt werden, erfüllt ist.

[5] Da die Analyse von Zeitinkonsistenzen von der Einheit unabhängig ist, wird in den Beispielen dieses Kapitels auf die Darstellung der Einheit verzichtet.

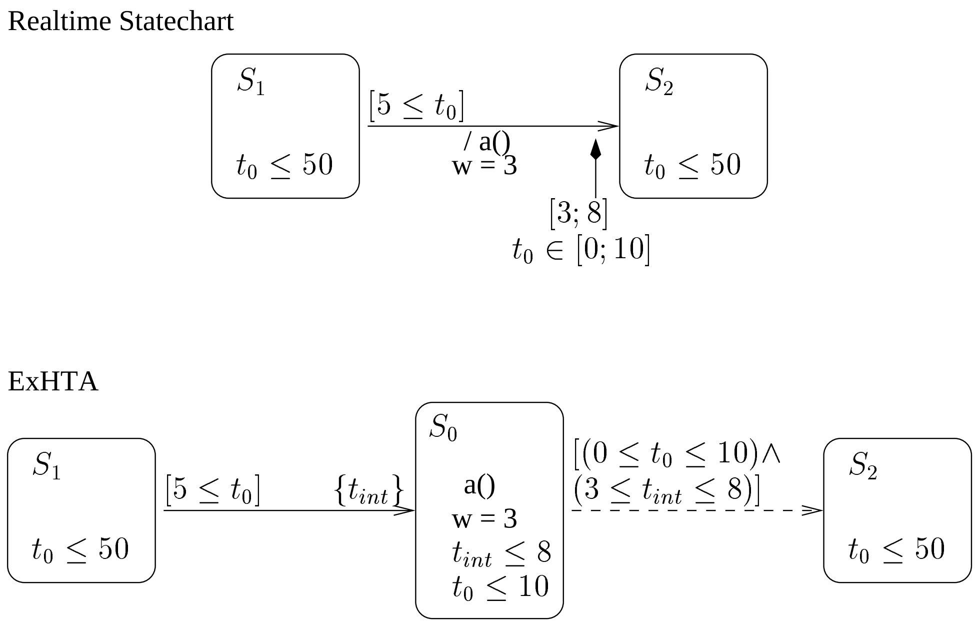

Abbildung 4.15.: Schwacher Zeitguard

Schwache Invarianten

Beschreibung Eine *schwache Invariante* liegt vor, wenn härtere Zeitinvarianten existieren, die das Schaltverhalten nicht ändern. Das ist der Fall, wenn es für den Automaten möglich ist, länger in einer Location zu verweilen als wegführende Transitionen aktiviert sind.

Beispiel In Abb. 4.16 sind die Zeitguards so beschaffen, dass die Location S_1 spätestens nach $t_0 = 9$ oder $t_1 = 10$ nicht mehr verlassen werden kann. Während des Intervalls $9 < t_0 \leq 12$ liegt zwar noch kein Time Deadlock vor, er kann aber auch nicht mehr verhindert werden.

Erkennung Gibt es im ExHTA eine Location l, die mit der Zeitinvarianten $\prod_{t_i \in C}(t_i \leq T_i)$ assoziiert ist, und seien $tg_k = \prod_{t_i \in C}(a_{ik} < t_i < b_{ik})$, $k = 1, 2, \ldots$ die Zeitguards aller von l wegführenden Transitionen , dann sei $b_i^{max} := \max_k (b_{ik})$. Eine schwache Invariante liegt vor, mit $\exists i | T_i > b_i^{max}$.

Tote Transitionen

Beschreibung Ähnlich, wie jeder ExHTA in eine von 3 Sicherheitskategorien eingestuft werden kann, können vor dem Start zu jeder Transition eine von drei Aussagen bezüglich der Aktivierung getroffen werden:

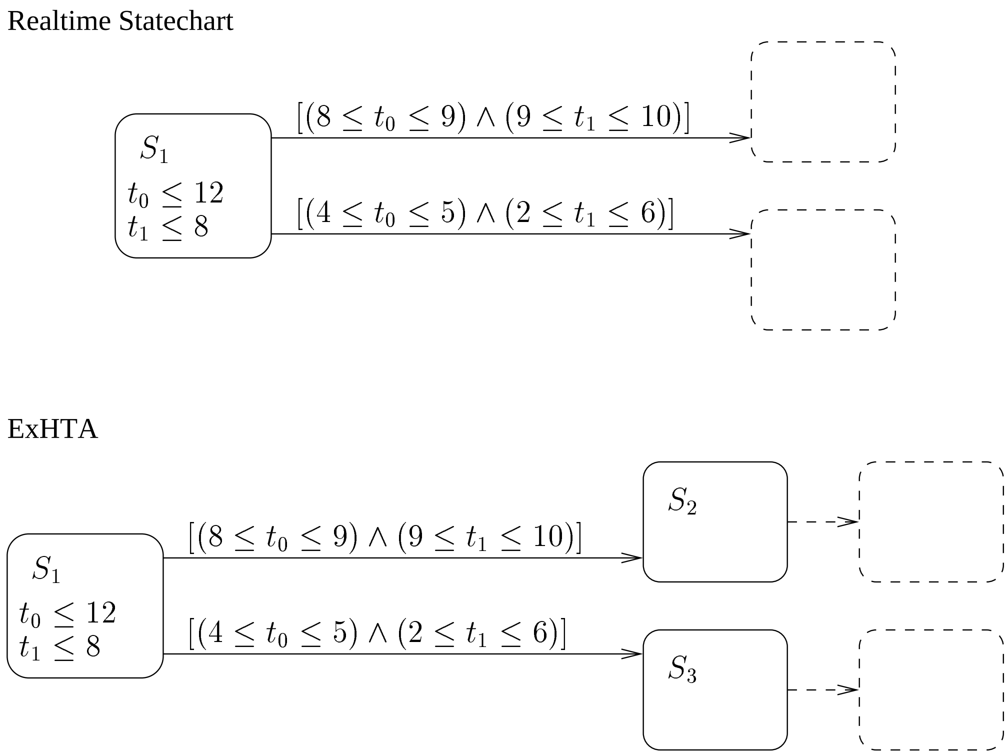

Abbildung 4.16.: Schwache Invariante

1. Die Transition wird zu gewissen Zeitpunkten definitiv aktiviert sein.

2. Die Transition wird definitiv nie aktiviert sein.

3. Unter Umständen wird die Transition aktiviert werden. Dies entscheidet sich aber erst zur Laufzeit.

Gilt für eine Transition Aussage 2, dann liegt eine *tote Transition* vor. Gilt Aussage 1, ist sie *lebendig* und bei Aussage 3 wird sie *unsicher* genannt.

Beispiel Hat eine Location $t_0 \leq 3$ als Invariante und sei der Zeitguard der wegschaltenden Transition $4 \leq t_0$, dann ist diese Transition tot.

Erkennung Gibt es im ExHTA eine Location l mit der Zeitinvarianten $\prod_{t_i \in C}(t_i \leq T_i)$ und bezeichne t_{sc} (sc: Spezial Case) eine von l wegführende Transition, deren Zeitguard $a_i < t_i < b_i$ den Spezialfall darstellt aus nur einer Uhr zu bestehen und deren Aktivierung nicht von einem Ereignis oder Guard abhängt, dann ist t_{sc} lebendig, mit $a_i \leq T_i$. Die Bedingung besagt, dass die Transition nicht erst nachdem die Invariante der Quelllocation abgelaufen ist, aktiv werden soll. Gibt es eine von l wegführende Transition t mit allgemeinem Zeitguard $\prod_{t_i \in C}(a_i < t_i < b_i)$, dann ist t tot, mit $\exists i | a_i > T_i$.

Für alle Transitionen, die nicht tot sind und deren Aktivierung von einem Ereignis oder von einem Guard (an dieser Stelle ist nicht der Zeitguard gemeint), der nicht durch den konstanten Wert `true` beschrieben wird (z.B.: $x > 2$), abhängig ist, kann lediglich Aussage 3 getroffen werden, da es hier nicht möglich ist, vor der Laufzeit vorherzusagen, wann ein Ereignis auftritt oder wann sich ein Variablenwert in welcher Form ändert. Um solch eine Vorhersage zu treffen, müsste eine vollständige Zustandsraumanalyse durchgeführt werden, die jedoch -wie in Kapitel 4.1.1 erläutert- nicht betrachtet wird. Außerdem sind alle Transitionen, deren Zeitguard mehrere Uhren beinhaltet und bei denen $\forall i : a_i \leq T_i$ gilt (also die nicht tot sind), unsicher. Denn dort hängt das Schaltverhalten davon ab, mit welcher Zeitdifferenz die einzelnen Uhren zueinander laufen. Da die Uhren erst zur Laufzeit gesetzt werden, ergibt sich hier dasselbe Problem.

Time Deadlock

Beschreibung Ein Time Deadlock liegt vor, wenn –wie schon mehrfach erwähnt– eine Zeitinvariante abläuft.

Beispiel Eine Location besitzt eine Invariante $t_0 \leq 10$ lediglich eine wegführende Transition, die an ein Ereignis gebunden ist. Wenn dieses Ereignis erst zu einem Zeitpunkt $t_0 > 10$ auftritt, kommt es zum Time Deadlock.

Erkennung Wenn jede Location mindestens eine lebendige ausgehende Transition besitzt (siehe Kapitel 4.3.1), existiert kein Time Deadlock und der Extended Hierarchical Timed Automata ist zeitkonform. Existiert für mindestens eine Location nicht solch eine Transition, sondern lediglich tote Transitionen, liegt ein Time Deadlock vor, da sie nicht verlassen werden kann, bevor die Invariante abläuft. In dem Fall ist der ExHTA zeitinkonsistent. Besitzt eine Location keine lebendige, aber unsichere Transitionen, kann er als unsicher bezeichnet werden: Es kann sein, dass es zum Time Deadlock kommt (wie im Bsp.), es ist aber nicht zwingend. Um einen Time Deadlock zu finden, müssen lediglich die Transitionen in die Kategorien “tot”, “lebendig” und “unsicher” eingestuft und analysiert werden.

4.3.2. Auflösung von Zeitinkonsistenzen

Bevor aus einem Realtime Statechart Code synthetisiert wird, wird es auf Zeitinkonsistenzen untersucht. Es sind sowohl Szenarien denkbar, bei denen der Anwender Zeitkonformität garantiert haben möchte, als auch Systeme, bei denen eine Unsicherheit toleriert wird. Insbesondere, wenn der Modellierer "Meta-Wissen" (z.B. bezüglich Auftrittsreihenfolge oder -wahrscheinlichkeit von Ereignissen) besitzt, liegt häufig letzterer Fall vor. Aufgrund dessen wird dem Modellierer die Möglichkeit gegeben, einen Sicherheitsgrad für das Realtime Statechart festzulegen. Er kann also auswählen, ob die Auflösung der Inkonsistenzen zu einem unsicheren oder zu einem zeitkonformen System führen soll.

Wurde eine Zeitinkonsistenz festgestellt, muss sie -je nach Sicherheitsgrad- behoben werden. Dies kann automatisch oder manuell geschehen. Bei der manuellen Variante wird der Anwender auf eine Inkonsistenz mitsamt der Ursache hingewiesen und aufgefordert, die Zeitannotationen zu ändern. Sollen diese automatisch behoben werden, werden die Zeitannotationen, wie in diesem Kapitel weiter unten beschrieben, angepasst. Der Anwender muss sich lediglich um die Konflikte kümmern, die aufgrund ihrer Natur nicht automatisch behoben werden können. Damit bei diesem Vorgehen nicht das grundsätzlich spezifizierte Verhalten geändert wird, können Zeitannotationen als *final* oder als *variable* deklariert werden. Könnte eine Inkonsistenz durch Modifikation einer als final bezeichneten Annotation behoben werden, wird sie nicht vorgenommen. In diesem Fall muss der Anwender den Konflikt manuell beheben.

Auflösung der schwachen Zeitguards Dieser Konflikt wird erkannt, wenn $\exists t_i \notin CR | b_i > T_i - w/\nu$ wahr ist.[6] Um ihn zu beheben, wird die konfliktverursachende Grenze des Zeitguards angepasst: $\forall i | t_i \notin CR \wedge b_i > T_i - w/\nu : b_i = T_i - w/\nu$. Da b_i, laut Definition in Kapitel 4.2.1, nicht negativ sein darf, wird diese Zuweisung nur vorgenommen, wenn $T_i \geq w/\nu$. Gilt dies nicht, kann die Inkonsistenz nicht automatisch behoben werden.

Im Beispiel aus Abb. 4.15 würde der Zeitguard der Transition nach S_0 auf $5 \leq t_0 \leq 6$ geändert werden.

Auflösung der schwachen Invariante Auch hier wird der Wert, der den Konflikt verursacht, angepasst: $\forall i | T_i > b_i^{max} : T_i = b_i^{max}$

Die Invariante im Beispiel aus Abb. 4.16 würde also auf $(t_0 \leq 9)\,(t_1 \leq 8)$ geändert werden.[7]

[6] Für die Bedeutung der Konstaten, siehe Kapitel 4.3.1.

[7] Die Änderung der Zeitinvarianten garantiert noch nicht, dass die Location über eine Transition verlassen werden kann (das hängt von der Zeitdifferenz ab, mit der die einzelnen Uhren laufen). Sie schließt aber die Zeitpunkte aus, zu denen die Location auf keinen Fall verlassen werden kann.

Auflösung toter Transitionen Eine tote Transition kann zwar erkannt, aber nicht durch eine Formel oder einen Algorithmus behoben werden. Somit ist es Aufgabe des Anwenders, die Zeitannotationen anzupassen.

Time Deadlock Auflösung Da ein Time Deadlock mit toten Transitionen zusammenhängt, wird er dadurch aufgelöst, dass die toten Transitionen behoben werden.

Propagieren von Zeitannotationen

Die automatische Auflösung der Zeitinkonsistenzen kann bewirken, dass sich Änderungen von Zeitannotationen durch das gesamte Netz des ExHTA fortpflanzen. So kann z.B. eine behobene schwache Invariante einen schwachen Zeitguard hervorrufen und umgekehrt.

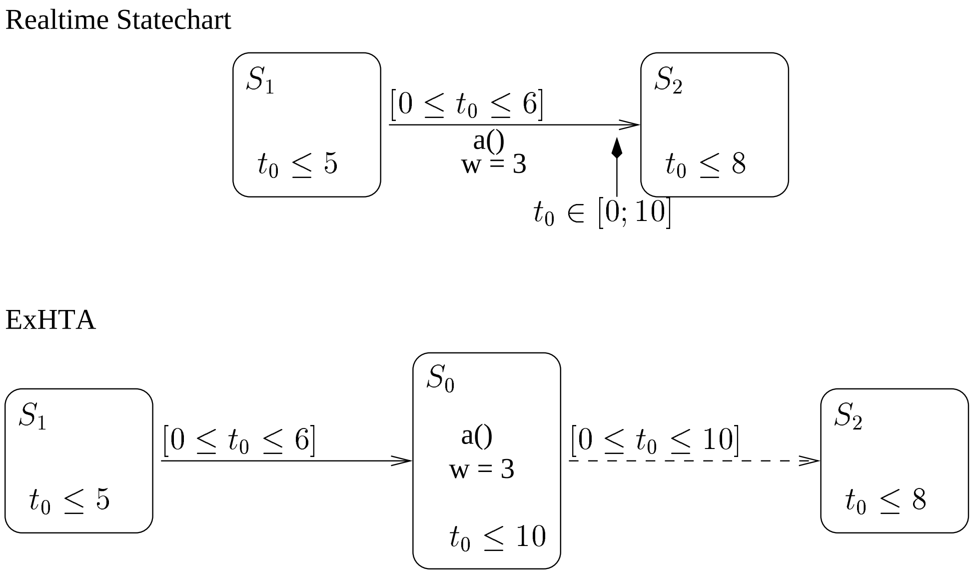

Abbildung 4.17.: Propagieren von Zeitannotationen

Sei im ExHTA aus Abb. 4.17 $\nu = 0.75$ und somit $w/\nu = 4$. Dann liegt in S_2 ein schwacher Zeitguard vor, dessen automatische Auflösung den Zeitguard $0 \leq t_0 \leq 10$ zu $0 \leq t_0 \leq 8$ ändert.[8] Dies verursacht eine schwache Invariante in S_0, wodurch die Invariante auf $t_0 \leq 8$ verringert wird. Da für die Ausführung der Aktion `a()` im Worst Case 4 Zeiteinheiten benötigt werden, verursacht die neue Invariante einen neuen schwachen Zeitguard, der den Zeitguard $0 \leq t_0 \leq 6$ zu $0 \leq t_0 \leq 4$ ändert. Auch diese Änderung ruft eine neue Inkonsistenz hervor: Die Invariante in S_1 wird aufgrund der schwachen Invariante zu $t_0 \leq 4$ geändert.

[8] In diesem Beispiel seien alle Annotationen variable.

Man kann sich vorstellen, dass das Gesamtverhalten des ExHTAs, bei längeren solcher Abhängigkeitsketten, nicht im Sinne des Modellierers geändert wird. So wurden in diesem Beispiel von der automatischen Inkonsistenzauflösung 4 Zeitannotationen geändert. Angenommen, der Modellierer hat sich lediglich bei der Invarianten in S_2 vertan, dann hätte es gereicht, diesen einen Wert von 8 auf 10 zu ändern. Dieser Lösungsvorschlag wird von der automatischen Auflösung jedoch nicht gemacht, da dadurch ein Verhalten spezifiziert wird, das nicht im ursprünglichen Verhalten "enthalten" ist. Nach den automatischen Änderungen im Beispiel muss S_1 spätestens zum Zeitpunkt $t_0 = 4$ verlassen werden. Dieses Verhalten entspricht der Forderung $t_0 \leq 5$ des "unbehandelten" Automaten. Auch der Zeitguard $0 \leq t_0 \leq 4$ ist ein Sonderfall von $0 \leq t_0 \leq 6$. Hätte die Inkonsistenzauflösung jedoch die Invariante von S_2 auf $t_0 \leq 10$ geändert, wäre der ExHTA zwar zeitkonform, würde es aber zulassen, länger in S_2 zu verweilen als ursprünglich spezifiziert. Dieses Vorgehen entspricht außerdem der Strategie, die häufig in der Praxis eingesetzt wird: Zeitrestriktionen werden zunächst größtmögliche Freiheitsgrade gegeben, bevor sie dann so lange eingeschränkt werden, bis das System konsistent ist.

Um bei der automatischen Auflösung der Inkonsistenzen eine ungewollte Kettenreaktion zu vermeiden, kann für die Auflösung eine Tiefe n angegeben werden, die bestimmt, dass nach n Modifikationen auf manuelle Auflösung umgeschaltet werden soll.

Bewertung der Konfliktsuche und Konfliktauflösung

In Kapitel 4.3.1 wurden Algorithmen vorgestellt, die Zeitinkonsistenzen erkennen und oben in diesem Kapitel wurde gezeigt, wie gegebenenfalls Zeitannotationen angepasst werden können, um diese Inkonsistenzen aufzulösen. Im Folgenden werden diese Algorithmen bezüglich der Eigenschaften

- Vollständigkeit
- Maximalität und
- Terminierung

bewertet. Ist ein Algorithmus vollständig, findet er garantiert eine Lösung, sofern eine Lösung existiert. Ein Algorithmus ist maximal, wenn er die Lösung mit den wenigsten Einschränkungen ermittelt.

Die Algorithmen zur Suche der Inkonsistenzen sind vollständig, d.h.: Liegt eine Inkonsistenz vor, wird sie garantiert erkannt.

Da die Auflösung der toten Transitionen und Time Deadlocks vom Anwender verlangt wird, ist eine Bewertung der Inkonsistenzauflösungssalgorithmen lediglich für die schwachen Invarianten sowie für die schwachen Zeitguards sinnvoll.

Wird ein Konflikt durch Modifikation der Zeitkonstanten behoben, gibt es eine Menge verschiedener Lösungen (diese Menge kann leer, aber auch nicht endlich sein). So könnte zum Beispiel bei der Inkonsistenz aus Abb. 4.15 der zu modifizierende Zeitguard auch auf $5 \leq t_0 \leq 5.5$ geändert werden. Um Maximalität zu erreichen, wird jedoch die Lösung gewählt, die das Systemverhalten am wenigsten einschränkt ($5 \leq t_0 \leq 6$). Ein Zeitguard wie z.B. $5 \leq t_0 \leq 6.5$ schränkt zwar das Systemverhalten noch weniger ein, löst aber die Inkonsistenz nicht. Es wird also angestrebt, in jedem Zeitguard $\prod_j (a_j \leq t_j \leq b_j)$ und in jeder Invarianten $\prod_j (t_j \leq T_j)$ alle a_j möglichst klein und die b_j und T_j möglichst groß zu halten und gleichzeitig die Inkonsistenzen zu beseitigen. Da diese Forderungen von den Algorithmen zur Auflösung der schwachen Invarianten und schwachen Zeitguards erfüllt werden, sind sie maximal.

Wird für die automatische Auflösung keine Tiefe angegeben, die bestimmt nach wie vielen Modifikationen die Algorithmen abbrechen sollen, tritt die Frage nach der Terminierung/Konvergenz auf. Bei der automatischen Auflösung werden Annotationen verringert, niemals vergrößert. Der Wert, der subtrahiert wird, setzt sich aus den spezifizierten Annotationen zusammen und ist somit aus der Menge der natürlichen Zahlen (da jedes System eine minimale Uhrenauflösung hat). Dadurch ist sichergestellt, dass bei einer Kettenreaktion der Wert irgendwann negativ wäre. In dem Fall wird der Algorithmus zur automatischen Inkonsistenzauflösung beendet, was schließlich die Terminierung sicherstellt.

4.4. Zusammenfassung

Mit Realtime Statecharts existiert nun ein Modell, mit dem Echtzeitsysteme beschrieben werden können. Eine Abbildung von Realtime Statecharts auf das Modell der Extended Hierarchical Timed Automata gibt eine Definition der Semantik und erleichtert eine Analyse auf Zeitinkonsistenzen. Die Annotationen der Realtime Statecharts sind zum einen so flexibel gehalten, dass sie die Einführung von High-Level Konstrukten ermöglichen und beschreiben zum anderen das Echtzeitverhalten hinreichend genau, dass Code einer Programmiersprache synthetisiert werden kann (vorrausgesetzt die Programmiersprache unterstützt Echtzeitverhalten). Im nächsten Kapitel werden die Konzepte erarbeitet, nach denen das Modell auf Code abgebildet wird. Dabei werden die Konzepte allgemein gehalten, so dass sie in verschiedenen Programmiersprachen implementiert werden können.

5. Konzepte für die Codesynthese

Im bisherigen Verlauf dieser Arbeit sind die Realtime Statecharts entstanden, die es ermöglichen das benötigte Echtzeitverhalten zu spezifizieren. Nun wird gezeigt, wie sie auf ein System abgebildet werden können, so dass es allen Zeitanforderungen genügt. Typischerweise entstehen bei der Abbildung Threads, die von einem Echtzeitbetriebssystem verwaltet werden müssen. Dieses Betriebssystem muss dem POSIX PSE50 Profil oder einer seiner Erweiterungen entsprechen (siehe Kapitel 2.2). Die Programmiersprache, in der die Threads implementiert werden, kann ebenfalls frei gewählt werden, sofern sie Nebenläufigkeit bietet. Die hier erarbeiteten Konzepte werden in Kapitel 6 für eine an Realtime Java angepasste Codesynthese verwendet.

5.1. Syntheseablauf

Wurde ein Realtime Statechart vom Anwender spezifiziert, soll schließlich Code erzeugt werden. Hier sei ein Überblick über die dafür notwendigen Schritte gegeben:

1. Spezifikation. Der Anwender modelliert das Realtime Statechart.
2. Wcet-Bestimmung. Die Worst Case Execution Times, die nicht vom Anwender angegeben wurden, werden aus einem externen Dokument eingelesen. Dieses Dokument könnte beispielsweise von einem Wcet-Analyse-Tool erzeugt werden.
3. Analyse und Behebung von Zeitinkonsistenzen. Treten nicht behebbare Zeitinkonsistenzen auf, muss das Modell geändert werden.
4. Partitionierung. Auf Basis der Wcets und der übrigen Zeitspezifikationen wird das Realtime Statechart auf verschiedene periodische und aperiodische Threads abgebildet.
5. Codesynthese. Es wird Code für das Realtime Statechart erzeugt.

6. Schedulinganalyse. Soll der Code mehrerer Realtime Statechart auf demselben System ausgeführt werden, muss eine Schedulinganalyse durchgeführt werden, die die Threads aller Realtime Statecharts berücksichtigt. Bei negativer Analyse müssen ggf. einzelne Modelle geändert werden.

Wird Code für eine Programmiersprache erzeugt, die auf verschiedenen Echtzeitsystemen einsetzbar ist (z.B. Realtime Java), muss bei einer Portierung des synthetisierten Programmes beachtet werden, dass sich die Wcets ändern können. Verringern sich die Wcets, beeinträchtigt das nicht die Schedulability der einzelnen Realtime Statecharts. Werden die Wcets jedoch größer, muss erneut eine Überprüfung auf Zeitinkonsistenzen und die Partitionierung durchgeführt werden.

5.2. Umsetzung des Zeitverhaltens

Für jedes Realtime Statechart wird ein Hauptthread erzeugt, der die Zustände verwaltet und periodisch die Transitionen auf Aktivierung überprüft und ggf. Schaltvorgänge auslöst. Die Periode des Hauptthreads wird so kurz gewählt, dass er trotz der in Kapitel 2.1.5 beschriebenen Verzögerungen eine aktivierte Transition noch so rechtzeitzig erkennt, dass er den Seiteneffekt ausführen kann ohne die Deadline zu verfehlen (Die Bestimmung der Periode wird ausführlich in Kapitel 5.4.2 behandelt). Ein erster Ansatz wäre, den Thread bei jeder Ausführung überprüfen zu lassen, ob *gerade* eine vom aktuellen Zustand ausgehende Transition aktiviert ist. Das würde dazu führen, dass die Periode nicht nur von den Deadlines und Zeitinvarianten abhängig ist, sondern zusätzlich so kurz sein muss, dass der Hauptthread garantiert mindestens einmal in jedem Zeitfenster eines jeden Zeitguards ausgeführt wird. Das hätte dann folgenden Nachteil: Beschreibt ein Zeitguard ein Intervall der Länge l, muss für den Hauptthread als Periode $p \approx l/2$ gewählt werden, um trotz der auftretenden Verzögerungen sicher zu stellen, dass der Thread einmal innerhalb des Intervalls ausgeführt wird. Aufgrund der Form der Zeitguards ($\prod_{t_i \in C}(a_i < t_i < b_i)$) wäre die Bestimmung von l vor der Laufzeit zu aufwendig, sobald im Zeitguard mehrere Uhren enthalten sind. Abhängig von der Zeitdifferenz mit der die Uhren laufen, kann l beliebig klein oder auch 0 werden (siehe Kapitel 4.1.1). Dies würde zu $p \approx 0$ führen.

Stattdessen überprüft der Hauptthread, ob *seit der letzten Ausführung* eine vom aktuellen Zustand ausgehende Transition aktiviert war. Ist dies der Fall, wird "rückwirkend" zu dem Aktivierungszeitpunkt geschaltet.

5.3. Reaktives Verhaltens

Der Hauptthread ist für Aktionen und Reaktionen des Realtime Statecharts zuständig. Wie sein Ablauf im Detail aussieht wird, erst anhand eines Beispiels und anschließend allgemein beschrieben.

Beispiel: In diesem Beispiel soll das oben beschriebene Wasserbecken eine durch eine Zeitschaltuhr gesteuerte Bewässerung steuern. Abb. 5.1 zeigt einen Ausschnitt aus dem dafür notwendigen Realtime Statechart, und Abb. 5.2 zeigt den zugehörigen ExHTA, an dem das reaktive Verhalten erläutert wird.

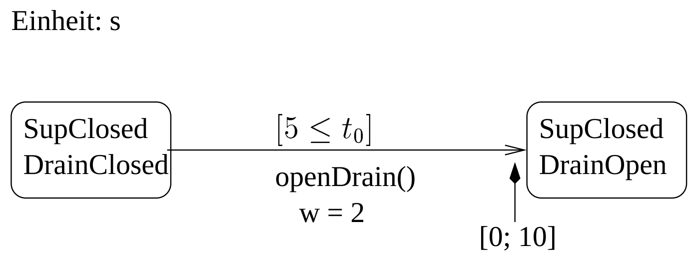

Abbildung 5.1.: Steuerung durch Zeitschaltuhr

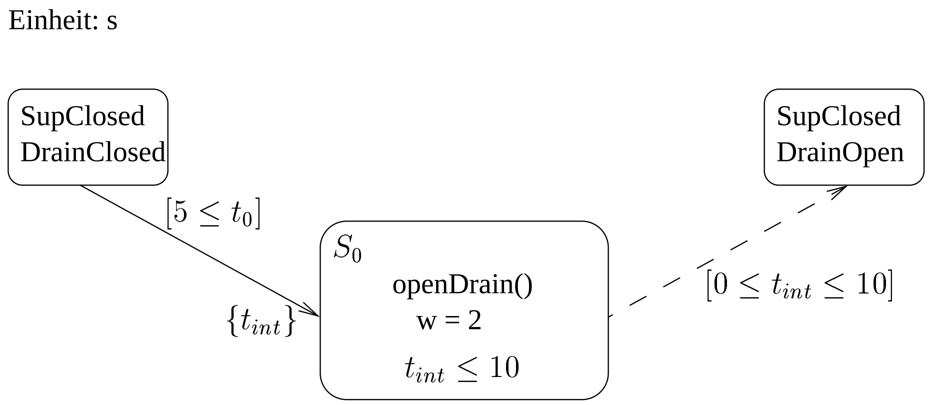

Abbildung 5.2.: ExHTA zu 5.1

Zu Beginn der Ausführung des Hauptthreads befinde sich der ExHTA in der Location `SupClosedDrainClosed`. Die Periode des Hauptthreads sei $p = 3$ und $t_0 \geq 5s$ gelte ab dem absoluten Zeitpunkt $t = 150s$. Aufgrund der möglichen Verzögerungen bei periodischen Threads würde die aktivierte Transition im Worst Case erst zum Zeitpunkt $t = 156s$ erkannt werden. Der so genannte *virtuelle Schaltzeitpunkt* liegt allerdings bei $t = 150s$, da sie urgent-Verhalten zeigt. Der Zeitpunkt, zu dem t_{int} zurückgesetzt wird, ist ebenfalls $t^0_{int} = 150s$. An dieser Stelle würde ein nicht-urgent-Verhalten den Nicht-Determinismus fördern. Die

Invariante der Location S_0 läuft somit zum Zeitpunkt $t = t^0_{int} + 10 = 160s$ ab. Da die Location erst bei $t = 156s$ betreten wird, bleibt für die Aktion `openGate()` nur noch eine relative Deadline von $160s - 156s = 4s$.

Angenommen die Aktion `openGate()` wurde schließlich zum Zeitpunkt $t = 158s$ ($t_{int} = 8s$) beendet, dann schaltet zu dem Zeitpunkt auch die Transition von S_0 nach `SupClosedDrainOpen`. Die Transition ist zwar schon bei $t = 150s$ aktiviert, schaltet aber nicht urgent und somit erst nach Beendigung der Aktion `openGate()`. Es wird also zum virtuellen Schaltzeitpunkt $t = 158s$ in die Location `SupClosedDrainOpen` geschaltet, womit auch die Ausführung einer Periode des Hauptthreads beendet ist.

Wurde bei der Deadline ein relatives oder absolutes Zeitintervall angegeben, dessen untere Grenze ungleich 0 ist (siehe Kapitel 4.1.1), kann es vorkommen, dass die Transition von der Actionlocation zur Statelocation erst in der Zukunft aktiviert wird. Da sie jedoch weder an ein Ereignis noch an einen Guard gekoppelt ist, kann dieser Zeitpunkt im Voraus bestimmt werden. Die Transition schaltet nach wie vor zum virtuellen Zeitpunkt, der dem Aktivierungszeitpunkt entspricht. Allerdings liegt jener in diesem Fall in der Zukunft. Der Zeitpunkt $t_{reached}$, der angibt wann die aktuelle Location erreicht wurde, wird auf den virtuellen Schaltzeitpunkt dieser Transition gesetzt: $t_{reached} = t_{virt}$. Konzeptionell ist die Ziellocation in dem Fall dann noch nicht erreicht, weshalb die Ausführung des Realtime Statecharts blockiert wird solange $t < t_{reached}$ gilt .

Überblick Zu Beginn einer jeden Periode befindet sich der ExHTA in einer Statelocation. Während einer Periode führt er 6 Schritte aus:

1. Zeitinvariantencheck. Es wird der Zeitpunkt bestimmt, zu dem die Invariante abläuft.

2. Aktivierungsüberprüfung. Von den von der aktuellen Statelocation wegführenden Transitionen werden Guard und Zeitguard analysiert, und die Eventqueue wird überprüft. Ist eine Transition aktiviert, wird der Zeitpunkt der Aktivierung bestimmt. Eine Transition gilt als aktiviert, wenn sie seit der letzten Überprüfung mindestens einmal hätte schalten können.

3. Transitionsauswahl. Aus der Menge der aktivierten Transitionen wird die zuerst aktivierte ausgewählt. Liegt ihr Aktivierungszeitpunkt vor Ablauf der Zeitinvarianten, wird sie geschaltet. Ist keine Transition aktiviert, wird die `do()`-Operation der aktuellen Statelocation ausgeführt. Für sie wurde eine eigene Periode spezifiziert. Je nachdem in welchem Verhältnis sie zu der Periode des Hauptthreads steht, muss die Ausführung nicht bei jedem

Durchlauf stattfinden.[1] Die Schritte 4 bis 6 finden nur statt, falls eine Transition schaltet.

4. Locationwechsel. Es wird der ausgewählten Transition gefolgt, und die mit der Transition assoziierten Uhren werden zurückgesetzt. Die Ziellocation ist eine Actionlocation, von der die Aktion gestartet wird.

5. Aktivierungsüberprüfung. Da der ExHTA aus einer Abbildung eines Realtime Statecharts entstanden ist, führt von der aktuellen Actionlocation genau eine Transition weg (siehe Kapitel 4.2.2). Mit dieser Transition sind weder Guard noch Ereignis assoziiert, sondern lediglich ein Zeitguard. Da vor dem Start des Automaten Zeitinkonsistenzen beseitigt wurden, ist sichergestellt, dass die Zeitinvariante der Actionlocation der der Ziellocation entspricht oder "härter" ist. Aufgrund dessen und wegen der Tatasache, dass der Zeitguard bei der Auflösung der Inkonsistenzen minimal eingeschränkt wurde, ist garantiert, dass diese eine Transition entweder aktiviert ist oder in der Zukunft aktiviert wird. Sollte sie nicht aktiviert sein, kann ihr nächster Aktivierungszeitpunkt im Voraus berechnet werden.

6. Locationwechsel. Die einzige Transition wird geschaltet. Liegt ihr Aktivierungszeitpunkt in der Zukunft, wird dieser im ExHTA „vermerkt“ und bei der Aktivierungsüberprüfung in der nächsten Periode berücksichtigt. Da die Ziellocation eine Statelocation ist, ist die Forderung, jede Periode in einer Statelocation zu beginnen, erfüllt.

In den folgenden Kapiteln werden einzelne Punkte ausführlicher beschrieben.

5.3.1. Schaltverhalten

Beim Schaltvorgang werden zuerst Transitionen auf Aktivierung überprüft, anschließend aus mehreren aktivierten Transitionen eine ausgewählt und schließlich wird ermittelt, zu welchem Zeitpunk geschaltet wird.

Eine Transition ist aktiviert, wenn es seit der letzten Überprüfung einen Zeitpunkt gab, an dem

1. der Guard wahr war
2. das Ereignis in der Eventqueue vorlag und
3. der Zeitguard wahr war.

[1] Es muss allerdings sichergestellt werden, dass die `do()`-Operation einer Location mindestens einmal ausgeführt wurde, bevor die Location wieder verlassen wird. Hierzu sei auf Kapitel 5.3.4 verwiesen.

Die einzelnen Überprüfungen werden genau in dieser Reihenfolge vorgenommen, da die Betrachtung des Zeitguards komplexer und somit rechenzeitintensiever ist, als die des Guards und Ereignisses. Wird festgestellt, dass die Guardbedingung nicht erfüllt ist, muss auf Ereignis und Zeitguard nicht mehr geprüft werden.

Guard Der Guard ändert sich ausschließlich aufgrund von Ausführungen der `do()` Methode oder der Seiteneffekte. Das bedeutet: Ist der Guard wahr, war er das schon seit der letzten Aktionsausführung. Daher gibt es zu jeder Location einen Zeitpunkt t_{guard}, der angibt, wann die `do()`-Methode zum letzten Mal beendet wurde, bzw. seit wann der Wert des Guards gilt.[2]

Ereignis Das Anliegen eines Ereignisses wird dadurch implementiert, dass ein Objekt vom entsprechenden Typ an eine Eventqueue angefügt wird. Zu jedem Ereignis wird zusätzlich der Auftrittszeitpunkt t_{event} festgehalten.

Zeitguard Zu bestimmen, wann der Zeitguard wahr war, ist etwas aufwendiger, besteht aber aus einer nach oben festen Anzahl von Additionen und Vergleichen:

Aufgrund der Restriktionen aus Kapitel 4.1.1 hat ein Zeitguard die Form $\prod_i a_i \leq t_i \leq b_i$. Sei t_i^0 der absolute Zeitpunkt, zu dem die Uhr t_i auf 0 zurückgesetzt wurde, dann liefert der Ausdruck $a_i \leq t_i \leq b_i$ im Zeitintervall $[a_i+t_i^0,\, b_i+t_i^0]$ `true` zurück. Die Konjuktion $\prod_i a_i \leq t_i \leq b_i$ liefert dann im Intervall $[\max_i(a_i + t_i^0),\, \min_i(b_i + t_i^0)]$ `true`. Es werden nun folgende Abkürzungen eingeführt: $tg_{low} := \max_i(a_i+t_i^0)$ und $tg_{up} := \min_i(b_i+t_i^0)$. Der Zeitguard war dann also in dem Intervall $[tg_{low},\, tg_{up}]$ wahr, falls $tg_{low} \leq tg_{up}$ gilt.

Zeitinvariantencheck Die Zeitinvarianten werden ähnlich wie die Zeitguards überprüft. Eine Invariante der Form $\prod_i t_i \leq T_i$ läuft zu dem Zeitpunkt $t_{stop} = \min_i(T_i + t_i^0)$ ab.

Aktionsausführung Wird in eine Actionlocation gewechselt, wird sofort die Aktion gestartet. Da die Location nicht vor dem Ende der Aktion verlassen werden darf, wird der Zeitpunkt $t_{actionEnd}$, zu dem die Aktion beendet wurde, mitprotokolliert.

[2] Wenn der Guard von `true` nach `false` oder von `false` nach `true` wechselt, dann aufgrund einer Ausführung der `do()`-Methode. Zwischen 2 `do()`-Ausführungen ändert sich der Guard nicht.

Aktivierung und Locationwechsel Seien nun t_{guard}, t_{event} und das Intervall bestimmt, zu dem der Zeitguard wahr war, dann ist die Transition aktiviert, wenn es einen Zeitpunkt gibt, zu dem sowohl der Guard wahr war als auch das Ereignis anlag und zu dem ebenfalls der Zeitguard aktiviert war. Der Zeitpunkt t_{act}, zu dem die Transition das erste Mal (seit der letzten Überprüfung) aktiviert war, ist der kleinste dieser Zeitpunkte. Handelt es sich bei der Transition um eine urgent schaltende, gibt t_{act} nicht nur den Aktivierungszeitpunkt an, sondern auch den Schaltzeitpunkt. Wie bereits im obigen Beispiel angedeutet, besitzt jede Statelocation ein Attribut $t_{reached}$, dass angibt, seit wann sich der ExHTA in der Location aufhält. Der Aktivierungszeitpunkt t_{act} darf natürlich nicht vor $t_{reached}$ der Quelllocation liegen:

$$t_{act} = \min \{t | \, t \geq t_{guard} \wedge t \geq t_{event} \wedge t \geq t_{reached} \wedge t \in [tg_{low}, tg_{up}] \, ; \, tg_{low} \leq tg_{up}\} \tag{5.1}$$

Eine Transition, die von einer Actionlocation wegführt, ist weder an ein Ereignis noch an einen Guard gebunden. Da sie nicht urgent schaltet –also nicht bevor die Aktion der Quell-Actionlocation beendet ist– wird in der Formel 5.1 für jenen Fall $t_{reached}$ durch $t_{actionEnd}$ ersetzt:

$$t_{act} = \min \{t | \, t \geq t_{actionEnd} \wedge t \in [tg_{low}, tg_{up}] \, ; \, tg_{low} \leq tg_{up}\} \tag{5.2}$$

So wird für jede den aktuellen Zustand verlassende Transition der Zeitpunkt, zu dem sie frühstens schalten könnte, ermittelt. Wie in Kapitel 4.1.1 beschrieben, schaltet die Transition, die zuerst aktiviert war. Die Prioritäten der Transitionen sind nur dann wichtig, wenn 2 Transitionen zum selben Zeitpunkt aktiviert werden.

In den obigen Abschnitten wurde bestimmt, welche Transition schalten soll und zu welchem Zeitpunkt sie schalten sollte (Erinnerung: Dieser Zeitpunkt liegt in der Vergangenheit zwischen der letzten Ausführung des Hauptthreads und dem aktuellen Zeitpunkt). Auch wenn die Transition "jetzt" nicht mehr schalten kann, wird sie rückwirkend zu diesem Zeitpunkt geschaltet. Wir sagen: "Der *virtuelle Schaltzeitpunkt* entspricht dem Aktivierungszeitpunkt." (Ein Beispiel hierzu befindet sich im folgenden Kapitel (5.3.2)). Die mit der Transition assoziierten Uhren werden zum virtuellen Schaltzeitpunkt auf 0 zurückgesetzt.

5.3.2. Distanz der virtuellen Schaltzeitpunkte in Bezug auf die Periode

Bisher ist eine wichtige Frage offen geblieben: Kann es passieren, dass zwischen 2 Ausführungen des Hauptthreads mehrere virtuelle Schaltzeitpunkte liegen? Dabei seien nun die Transitionen von Actionlocations zu Statelocations außer Acht gelassen, da diese nicht die Ausführung einer Aktion innerhalb der aktuellen Periode veranlassen. Die Frage wird anhand eines weiteren Beispiels verdeutlicht und beantwortet:

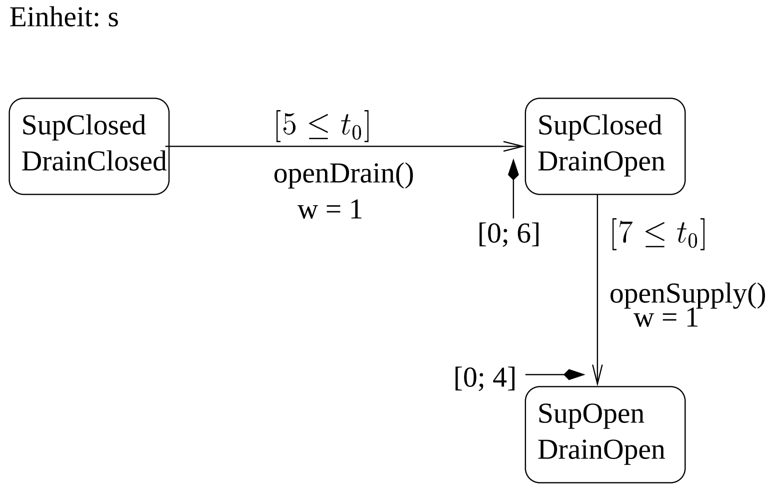

Abbildung 5.3.: Beispiel für Abb. 5.4

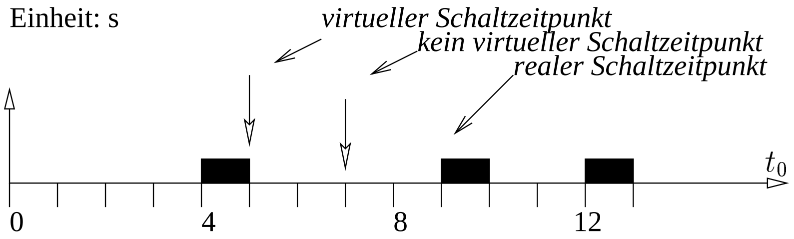

Abbildung 5.4.: Nicht-mögliche Folge von Aktivierungszeitpunkten

Kann es sein, dass das Realtime Statechart aus Abb. 5.3 bewirkt, dass –wie in Abb. 5.4 dargestellt– `openDrain()` zum Zeitpunkt $t_0 = 5s$ und `openSupply()` bei $t_0 = 7s$ gestartet werden soll, obwohl aufgrund der Periode (diese sei als $p = 3s$ angenommen) der Hauptthread zu $t_0 = 4s$ ausgeführt wird und dann erst wieder bei $t_0 = 9s$? Das hätte zur Folge, dass innerhalb einer Periode sowohl `openDrain()` als auch `openTap()` ausgeführt werden müssten. Die Antwortet lautet: “Nein”, weil `openDrain()` zum virtuellen Zeitpunkt $t_0 = 5s$ gestartet wird, aber erst “viel später” (in diesem Beispiel zum Zeitpunkt $t_0 = 10s$) beendet wurde. Somit liegt der Aktivierungszeitpunkt der zweiten Transition nicht bei $t_0 = 7s$, sondern bei $t_0 = 10s$, wodurch sie problemlos in der nächsten Periode behandelt werden kann. Die Deadlines waren lang genug, so dass sie beide eingehalten wurden. Wären die Deadlines kürzer gewesen, müsste eine kürzere Periode gewählt werden (siehe hierzu Kapitel 5.4). Dieser Zusammenhang zwischen Periode und Deadline führt im Übrigen dazu, dass kein Zeno-Verhalten auftreten kann (vgl. 2.4.2 und 4.3). Wäre eine relative Deadline $0s$ (was eine Wcet von $w = 0s$

vorraussetzt), müsste als Periode ebenfalls $p = 0s$ gewählt werden, was theoretisch eine unendlich häufige Ausführung zur Folge hätte, praktisch aber nicht möglich ist. Da die Wcets aber immer größer als 0 sind (aufgrund der minimalen Ausführungszeiten der Transitionen), muss auch die Deadline größer als 0 gewählt werden, wodurch sich eine Periode ergibt, die das Zeno-Verhalten ausschließt.

5.3.3. Hierarchie, Parallelität und Synchronisation

Hierarchisch geschachtelte Zustände bewirken, dass sich der Automat in mehreren Zuständen befindet. In der Menge der aktuellen Zustände befindet sich dann der innerste und alle hierarchisch weiter aussen liegenden Zustände. Bei der Feststellung, welche Transition zu Schalten ist, werden die Transitionen aller aktuellen Zustände berücksichtigt.

Befindet sich der Automat in einem parallelen AND-State, können mehrere Aktionen nebenläufig ausgeführt werden. In dem Fall finden die in Kapitel 5.3.1 beschriebenen Berechnungen und Überprüfungen pro parallelen Zweig einmal statt. Das hat zur Folge, dass die hierfür benötigte Zeit mit steigendem Grad an Parallelität ebenfalls zunimmt.

Gilt eine Transition, die über einen Kommunikationskanal synchronisiert wird, nach obigen Überprüfungen als aktiviert, schaltet sie erst, wenn eine passende Transition ebenfalls aktiviert ist. Dabei spielt das Ende des Zeitraumes, in dem die Transitionen aktiviert waren, ebenfalls eine Rolle, da sich diese Intervalle überschneiden müssen. Das Ende des Aktivierungsintervalls, das hier mit t_{end} bezeichnet wird, wird ähnlich wie der Beginn t_{act} bestimmt (siehe Kapitel 5.3.1). Seien 2 sich synchronisierende Transitionen in den Intervallen $[t^i_{act}, t^i_{end}]$, $i = 1, 2$ aktiviert, muss $t^1_{end} \geq t^2_{act} \vee t^2_{end} \geq t^1_{act}$ gelten. Die Aktivierungszeitpunkte der beiden Transitionen werden dann abgeglichen, indem jener der früher aktivierten Transition auf den der späteren gesetzt wird: $t^1_{act} = t^2_{act} = \max(t^1_{act}, t^2_{act})$.

Des Weiteren muss bei Parallelität und Synchronisation beachtet werden, dass das Schalten einer Transition evtl. das Schalten einer anderen ausschließt. So können z.B. im Realtime Statechart aus Abb. 5.5[3] die Transitionen T_1, T_2, T_3 und T_7 parallel zueinander schalten, T_5 kann aber nicht schalten, während z.B. T_1 ausgeführt wird.

Im Folgenden werden die Regeln, wann sich 2 Transitionen gegenseitig ausschließen, anhand des Realtime Statecharts aus Abb. 5.5 erläutert.

Regel I Ist der Quellzustand einer Transition hierarchisch, schließt sich diese Transition mit allen Transitionen aus, die innerhalb des Quellzustands sind.

[3] Um eine bessere Übersicht zu erreichen, wurde ein Großteil der Annotationen ausgeblendet. Stattdessen wurden die Transitionen mit Namen versehen.

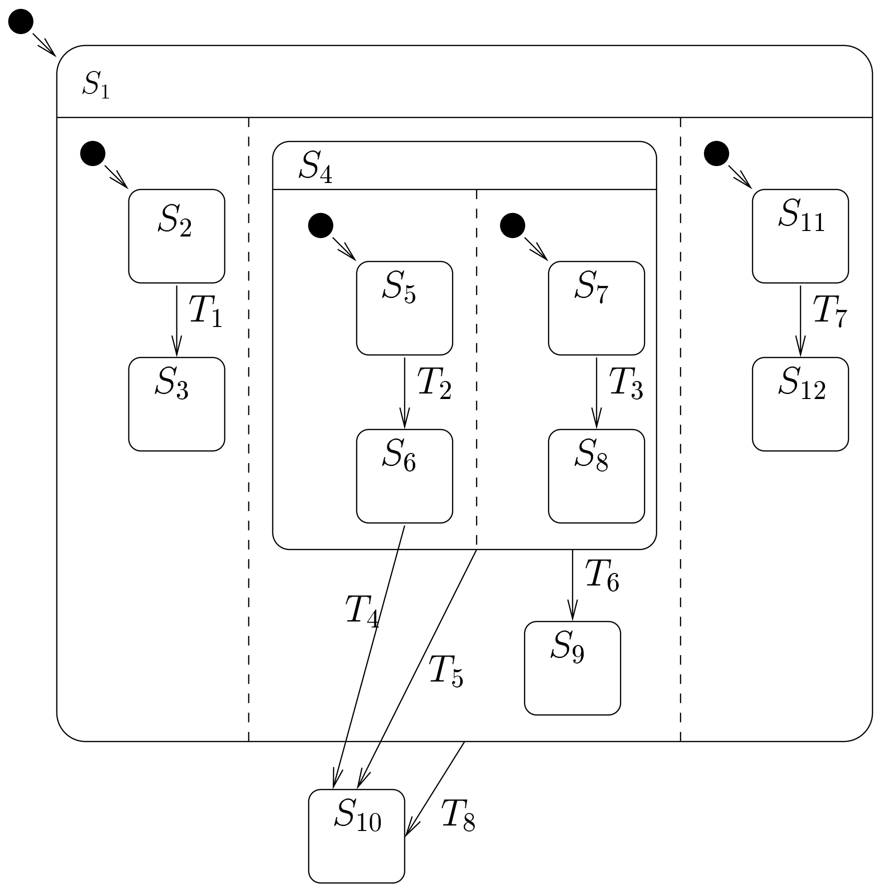

Abbildung 5.5.: Hierarchie und Parallelität

Beispiel: Während T_5 schaltet, dürfen u.a. T_2 und T_3 nicht schalten.

Regel II Eine Transition schließt sich mit allen weiteren Transitionen aus, die von ihrem Quellzustand wegführen. Befindet sich der Quellzustand innerhalb eines hierarchischen Zustands, gilt das ebenso für alle Transitionen, die von den übergeordneten Zuständen wegführen.

Beispiel: Während T_5 schaltet, darf u.a. T_8 nicht schalten.

Regel III Befindet sich der Quellzustand einer Transition innerhalb eines hierarchischen Zustands, aber auf unterschiedlicher Hierarchieebene wie der Zielzustand, dann gehe ausgehend von diesem Quellzustand in der Hierarchie nach oben bis der Zustand erreicht ist, der auf derselben Hierarchieebene liegt, wie der Zielzustand. Eine Transition schließt sich mit allen übrigen Transitionen aus, deren Quellzustand innerhalb dieses "übergeordneten hierarchischen Quellzustands" liegt.

Beispiel: Während T_4 schaltet, dürfen u.a. T_1, T_3, T_5, T_6 und T_7 nicht schalten.

Repräsentation und Analyse Die Menge der aktuellen Zustände kann als Baum dargestellt werden, der Hierarchie und Parallelität wiederspiegelt. So zeigt z.B. Abb. 5.6 einen möglichen Zustandsbaum des Realtime Statecharts aus Abb. 5.5 und wie er sich durch Schalten der Transition T_1 ändert. Die Entfernung eines Knotens von der Wurzel spiegelt die Hierarchieebene wieder, in der sich der Zustand befindet. Der Weg zur Wurzel führt über alle Zustände, in denen er geschachtelt ist. Die Anzahl der Blätter gibt sozusagen den *Parallelitätsgrad* an und beschreibt, wie viele Transitionen maximal parallel schalten können. Ein Pfad von der Wurzel zu einem Blatt wird als *Parallelitätsschacht* bezeichnet. Im Beispiel aus Abb. 5.6 a) ergeben sich 4 Schächte: (S_1, S_2), (S_1, S_4, S_5), (S_1, S_4, S_7) und (S_1, S_{11}).

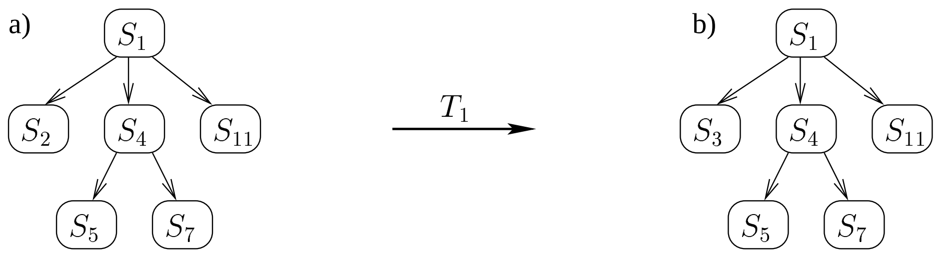

Abbildung 5.6.: Zustandsbäume

Befindet sich das Realtime Statechart im Zustand (bzw. in den Zuständen) aus Abb. 5.6 a), können verschiedene Transition parallel schalten. Da sich (wie oben beschrieben) das Schalten verschiedener Transitionen ausschließt, wird eine Menge namens `startedTransition` eingeführt, in der alle laufenden Transitionen abgelegt werden. Bevor eine weitere Transition ausgeführt wird, muss überprüft werden, ob sie mit den in der Menge enthaltenen Transitionen im Konflikt steht. Für diese Überprüfung wird eine Methode namens `isExecutable` generiert. Angenommen, T_1 werde gerade geschaltet, dann befindet sich ein Eintrag in der Menge `activeTransition`:

...	T_1	...	...	...

Nach Beendigung des Schaltvorgangs von T_1 wird der Eintrag wieder gelöscht. Wurde über eine Deadline spezifiziert, dass die Transition nicht vor einem bestimmten Zeitpunkt beendet sein darf, wird sie auch erst zu diesem Zeitpunkt ausgetragen.

Im Weiteren muss festgestellt werden, welche Transition zuerst aktiviert war. Dazu müssen zunächst alle aktivierten Transitionen aller Zustände des Baumes bestimmt werden. Dabei empfiehlt es sich, die Transitionen nach Parallelitätsschächten zu trennen. So ergibt sich eine Tabelle, in deren ersten Spalte ausschließlich aktivierte Transitionen sind, die als Quelle Zustände des ersten Parallelitätsschachtes haben. Für die weiteren Spalten gelten gleichartige Überlegungen. Nach dem Erstellen dieser `enabledTransition`-Tabelle sollen die einzelnen Spalten sortiert sein, was durch ein direktes Einfügen erreicht wird. Die Tabelle könnte dann z.B. wie folgt aussehen:

	T_4	T_3	T_7
	T_5		
	T_6		

Im ersten Parallelitätsschacht ist keine Transition aktiviert, weshalb er leer bleibt. Die Transitionen T_5 und T_6 gehören zwar genauso zum dritten Parallelitätsschacht wie zum zweiten, werden aber nur einmal aufgeführt. T_4 ist die zuerst aktivierte Transition ihrer Spalte. Die maximale Breite der Tabelle ergibt sich aus der maximalen Anzahl an Parallelitätsschächten. Die maximale Länge einer Spalte ist die Anzahl aller Transitionen im jeweiligen Parallelitätsschacht.

Um die Transition zu finden, die von allen als erstes aktiviert wurde, müssen nun lediglich diejenigen der ersten Zeile miteinander verglichen werden. In diesem Beispiel sei T_4 die gesuchte Transition. Ergibt eine Überprüfung mittels `isExecutable`, dass es keine Konflikte mit zur Zeit laufenden Transitionen gibt, wird sie (bzw. der Seiteneffekt) gestartet und die Transition in die Menge der Laufenden aufgenommen. Hier steht aber ein Schalten von T_4 im Wiederspruch zu T_1. Somit wird T_4 nicht geschaltet. Von den übrigen Transitionen wird erneut die zuerst aktivierte ausgewählt und in gleicher Weise mit ihr verfahren.

Synchronisation Transitionen, die synchronisiert werden, werden genauso wie die übrigen Transitionen behandelt. Lediglich im Anschluss an die Aktivierungsüberprüfung wird eine passende Transition gesucht. Dazu wird eine Sprungtabelle (`waitingTransition`) eingeführt, in der Transitionen aufgeführt sind, die aktiviert wurden, aber wegen eines fehlenden Synchronisationspartners nicht schalten konnten. Wird für eine Transition ein Partner gesucht, wird in dieser Tabelle gesucht. Falls keine passende Transition gefunden wurde, wird die gerade betrachtete in die Tabelle aufgenommen, ansonsten wird die Partnertransition aus der Tabelle entfernt und beide Transitionen werden gestartet. Die Tabelle ist so aufgebaut, dass es pro Synchronisationskanal eine Spalte gibt. Eine Transition wird immer an die ihres Kanales entsprechende Spalte angefügt. Bei der Suche nach passenden Transitionspartnern muss also lediglich eine Spalte durchsucht werden. Dabei wird nicht nur auf die Synchronisationsart ('Senden' oder 'Empfangen') geachtet, sondern auch –wie oben beschrieben– auf die Aktivierungsintervalle. Auch wenn die einzelnen Spalten nach den Aktivierungszeitpunkten sortiert bleiben, muss deswegen eine Spalte gegebenenfalls bis zum Ende durchsucht werden.

Beispiel Nun seien die Mengen und Tabellen für das Beispiel aus Abb. 5.5 wie folgt belegt:

`enabledTransition:`

T_1 a!	T_2 a?	T_3	T_7 b!
	T_5		
	T_6		

`activeTransition:` leer

`waitingTransition:`

channel a	channel b
leer	leer

Die Transitionen T_1, T_2 und T_7 können nur durch Kommunikation über die Kanäle `a` bzw. `b` mit einem passenden Synchronisationspartner schalten. Sei die Reihenfolge der Aktivierungen T_2, T_3, T_1, T_7, T_5, T_6, dann wird in der "channel a"-Spalte der `waitingTransition`-Matrix nach einem Partner für T_2 gesucht. Da es dort keinen (passenden) Eintrag gibt, wird T_2 aus `enabledTransition` entfernt und in `waitingTransition` eingefügt. Dadurch "rutschen" die nachfolgenden Transitionen nach:

`enabledTransition:`

T_1 a!	T_5	T_3	T_7 b!
	T_6		

`activeTransition:` leer

`waitingTransition:`

channel a	channel b
T_2 a?	leer

Zwar ist mit T_1 ein passender Synchronisationspartner aktiviert, T_1 darf aber nicht vor T_3 schalten, weshalb T_1 (noch) nicht betrachtet wird. Die nächste aktivierte Transition ist T_3 . Da `activeTransition` leer ist und T_3 keinen Synchronisationspartner benötigt, wird diese Transition ausgeführt:

`enabledTransition:`

T_1 a!	T_5		T_7 b!
	T_6		

`activeTransition:`

T_3

`waitingTransition:`

channel a	channel b
T_2 a?	leer

Nun ist T_1 an der Reihe. Da eine passende Transition in `waitingTransition` vorhanden ist (vorrausgesetzt die Aktivierungszeitintervalle stimmen überein), werden beide Transitionen geschaltet und in die `activeTransition`-Menge aufgenommen:

`enabledTransition:`

	T_5		T_7 b!
	T_6		

`activeTransition:`

T_3	T_2	T_1

`waitingTransition:`

channel A	channel B
leer	leer

Für T_7 wird genauso verfahren, allerdings wird sich in diesem Durchlauf kein Synchronisationspartner finden. Die Transition wird aber in der nächsten Periode erneut betrachtet und schaltet dann gegebenenfalls.

5.3.4. Verzögerung

Nachdem der Ablauf des Hauptthreads festgelegt ist, wird erneut die maximal mögliche Verzögerung betrachtet – also die Zeit, die zwischen der Transitionsaktivierung (also dem virtuellen Schaltzeitpunkt) und dem Aktionsstart verstreicht. Zu Beginn einer jeden Periode werden die Guards, Zeitguards und die Eventqueue überprüft. Die Zeit, die hierbei verstreicht, sei mit w_{guard} bezeichnet und ist im Beispiel aus Abb. 5.7 als Zeitblock 'g' dargestellt. Ist eine Transition aktiviert, wird geschaltet und die Aktion der Ziellocation ausgeführt (`a()`). Ansonsten wird entweder `do()` ausgeführt oder bis zum Beginn der nächsten Periode gewartet. Da die Location nicht verlassen werden soll, bevor wenigstens einmal `do()` ausgeführt wurde, wird bei der jeweils *ersten* Ausführung nach Locationwechsel zuerst `do()` ausgeführt, anschließend die Transitionsüberprüfung durchgeführt und gegebenenfalls die Aktion gestartet.

Zur größten Verzögerung kommt es, wenn, wie in Abb. 5.7 dargestellt, im Hauptthread die Aktivierungsüberprüfung vor der `do()`-Methode durchgeführt wird und eine Transition unmittelbar nach Beginn der Aktivierungsüberprüfung (Zeitpunkt t) aktiviert wurde, aber gerade so spät, dass dies nicht mehr vom Hauptthread erkannt wurde. Deshalb wird die Aktion erst in der folgenden Periode ausgeführt. Im Kapitel 5.4 wird erklärt werden, dass, wenn eine Aktion von der Ausführungslänge (Wcet) in die Periode des Hauptthreads passt, sie dann auch von ihm ausgeführt wird (`a()`). Ist sie dafür zu lang, wird sie in einen aperiodischen Thread ausgelagert, der dann vom Hauptthread gestartet wird (`startA()`). Befindet sich der Automat in einem parallelen Zustand, wird jede Aktion in Form eines aperiodischen Threads ausgeführt.

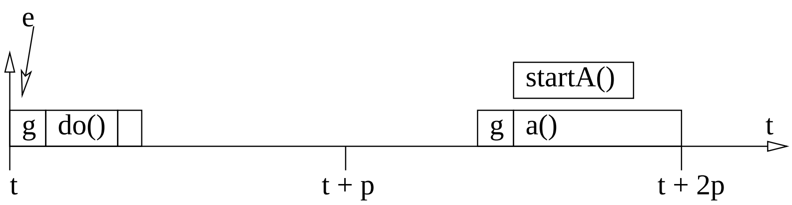

Abbildung 5.7.: Verzögerung

Die Zeit, um die sich der Start im Worst Case verzögert, ist $t_{delay} = 2p - w_{a()}$, bzw. $t_{delay} = 2p - w_{a()} + w_{startA()}$, wenn ein aperiodischer Thread gestartet werden muss. $w_{startA()}$ bezeichnet dabei eine Konstante, die angibt, wie viel Zeit zum Starten eines aperiodischen Threads benötigt wird. Allgemein gilt: Je kleiner die Periode, desto kleiner die Verzögerung.

5.3.5. Dauer der Aktivierungsüberprüfung

Bei Analysen und Berechnungen muss die Dauer der Aktivierungsüberprüfung, also dem Test, welche Transitionen aktiviert sind und der Auswahl, welche der aktivierten Transitionen schalten soll, berücksichtigt werden. Die Wcets für die Aktivierungsüberprüfung sind implementierungsabhängig und hängen außerdem von der Dauer der "elementaren" Operationen wie z.B. Vergleiche, Zuweisungen etc. ab. Diese *Systemkonstanten* müssen wie die Wcets der einzelnen Aktionen angegeben werden. Sind diese Werte bekannt, können die Wcets der Aktivierungsüberprüfung als Summe über diese Systemkonstanten ausgedrückt werden.

Da jeder Zustand eine unterschiedliche Anzahl von ausgehenden Transitionen besitzt, sind auch die Wcets bei jedem Zustand verschieden. Somit kann für jeden Zustand folgende Formel als Wcet für die Feststellung, welche Transitionen aktiviert sind, angegeben werden:

$$w_{activationProof} = \sum_{i=0}^{system.length()-1} k[i] * system[i] \tag{5.3}$$

Dabei sei *system* ein Array in dem die Systemkonstanten abgelegt sind und k ein Array, das individuell für jeden Zustand beschreibt welche Konstante mit welchem Faktor in die Wcet einfließt. k ist nicht nur für jeden Zustand unterschiedlich, sondern hängt außerdem von der Umsetzung des Realtime Statecharts ab (siehe Kapitel 6).

Der Fall von Hierarchie und Parallelität wird anhand des Beispiels aus Abb. 5.5 erläutert:

Ziel sei es w^{S4}_{guard} –die Zeit für eine Aktivierungsüberprüfung von Zustand S_4 – zu bestimmen. Dazu werden in einem ersten Schritt die Werte für die Zustände S_5, S_6, S_7 und S_8 bestimmt (wobei letzterer 0 ist, da er keine ausgehenden Transitionen besitzt). In S_4 werden nicht nur die Transitionen T_5 und T_6 überprüft, sondern zusätzlich jeweils einer der Zustände innerhalb dieses parallelen Zustands. Auch hier wird wieder der "Worst Case" angenommen, da davon ausgegangen werden muss, dass sich das Realtime Statechart gerade in dem Zustand mit der größeren Wcet aufhält:

$$\begin{aligned} w^{S4}_{activationProof} &= \max\left(w^{S5}_{guard}, w^{S6}_{guard}\right) \\ &+ \max\left(w^{S7}_{guard}, w^{S8}_{guard}\right) + \sum_{i=0}^{system.length()-1} k\,[i] * system\,[i] \end{aligned}$$

Hinzu kommt noch die Zeit, die für die Auswahl der zu schaltenden Transition(en) benötigt wird:

$$w^{S4}_{guard} = w^{S4}_{activationProof} + \sum_{i=0}^{system.length()-1} r\,[i] * system\,[i]$$

r bezeichnet ein Array, ähnlich wie k, hier jedoch für die Auswahl der zu schaltenden Transitionen.

5.4. Partitionierung

In den vorangegangenen Kapiteln wurde des Öfteren der periodische Hauptthread erwähnt und Verzögerungen abhängig von seiner Periode berechnet. Wie seine Periode bestimmt/gewählt wird, wurde noch nicht beschrieben.

Wie aus Abb. 5.8 ersichtlich, bestimmen die Deadlines der einzelnen Aktionen im Wesentlichen die Periode. Hier seien zunächst die Ausführungszeiten vernachlässigt und es wird gezeigt, wie die Periode in Abhängigkeit von der Deadline beschaffen sein muss, damit die aktivierte Transition vor Deadlineablauf erkannt wird. In Kapitel 5.4.2 wird der allgemeine Fall, wenn die Ausführungszeiten nicht vernachlässigt werden, betrachtet. Ist die Periode zu groß ($p > d/2$), kann es vorkommen, dass die Deadline bereits verstrichen ist, bevor der Hauptthread überhaupt von der Aktivierung "erfahren" hat. Wird die Periode kurz gewählt ($p < d/2$), liegt die Deadline garantiert hinter der nächsten Ausführung des Hauptthreads. Hier sieht man bereits, dass die Periode also hinreichend kurz gewählt werden muss, damit Deadlines eingehalten werden können. Gleichzeitig soll die Periode möglichst groß sein, damit der Hauptthread eine große Ausführungszeit erhalten kann und das System nicht durch viele Kontextwechsel belastet wird. Je kürzer (bei einer feststehenden Wcet w) die Periode p wird, desto größer wird der Quotient w/p, was im Hinblick auf die Schedulability vermieden werden sollte. Somit wird die Periode so kurz wie nötig und so lang wie möglich gewählt.

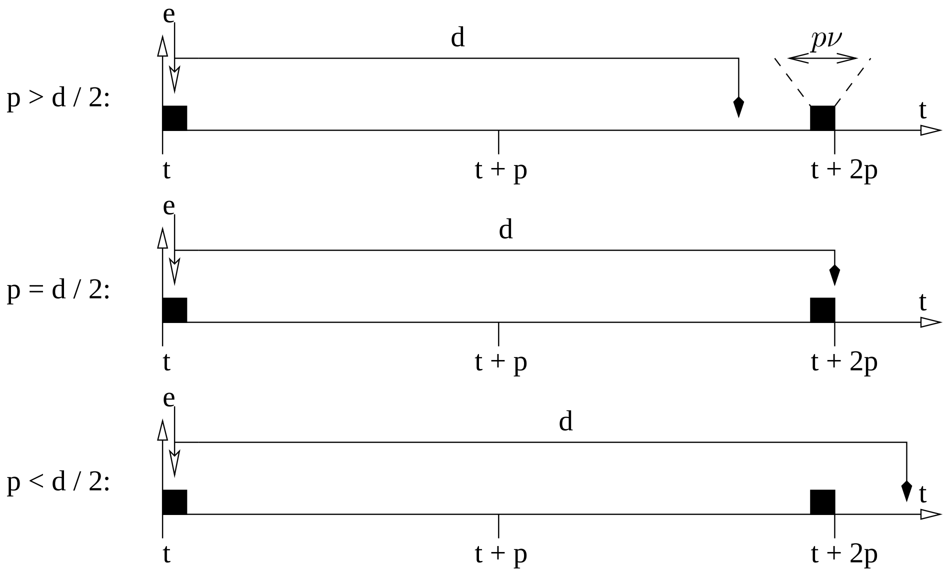

Abbildung 5.8.: Zusammenhang zwischen Deadline und Periode

5.4.1. Deadlineberechnung

In Kapitel 4.1.1 wurden 2 verschiedene Arten vorgestellt, Deadlines zu spezifizieren. Neben Deadlines, die Aktionen zugeordnet werden können, existieren noch Invarianten, die zu den Zuständen gehören und in ähnlicher Weise die Zeit vorgeben, die eine Aktion maximal benötigen darf. Abb. 5.9 zeigt ein Beispiel, bei dem die Transition im Worst Case (also spätestens) schaltet, wenn $t_0 = 5ms$ oder $t_1 = 10ms$ ist. Betrachtet man die Zeitinvariante, stehen der Aktion `a()` dann noch mindestens $\min(12ms - 5ms, 15ms - 10ms, 6ms) = 5ms$ für die Ausführung zur Verfügung. Im Worst Case ist die relative Deadline für `a()` also $d = 5ms$. Die Zeitverschiebung, mit der die einzelnen Uhren laufen, ist für diese Berechnung unerheblich, da z.B. t_0 nicht nach dem Zeitpunkt $t_0 = 5ms$ schaltet. Die Zeitverschiebung gegenüber Uhr t_1 kann höchstens bewirken, dass die Transition früher schaltet, nicht aber später.

Allgemein wird die Länge der relative Deadline d für den Worst Case wie folgt bestimmt:

Gibt es im ExHTA eine Actionlocation l mit der Invarianten $\prod_{t_i \in C}(t_i \leq T_i)$ und der Aktion `a()`, sei der Zeitguard der zu l hinführenden Transition $\prod_{t_j \in C}(a_j < t_j < b_j)$ (Erinnerung: Zu einer Actionlocation führt genau eine Transition,

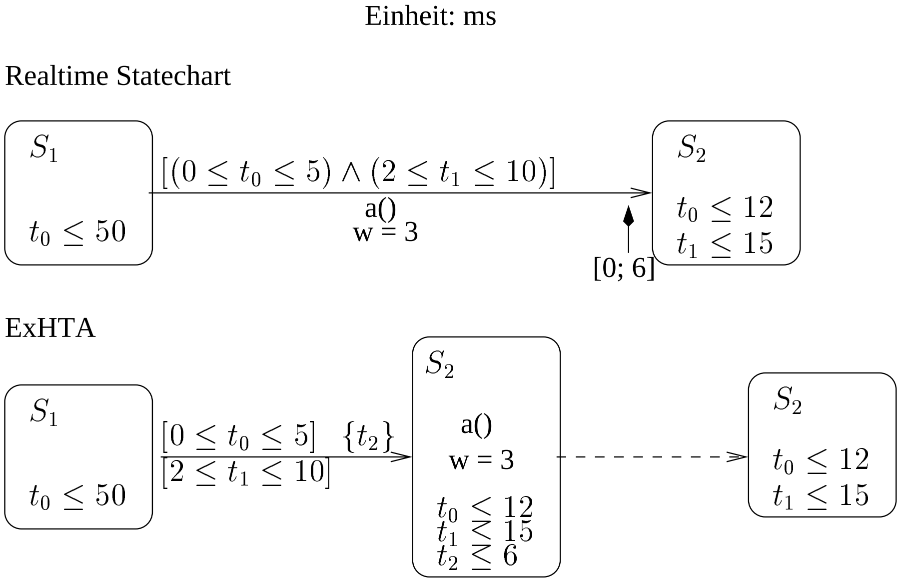

Abbildung 5.9.: Bestimmung der Deadline aus dem Extended Hierarchical Timed Automata

vgl. Kapitel 4.2.2) und werden beim Schalten dieser Transition die Uhren der Menge CR zurückgesetzt, dann beträgt die relative Deadline für `a()` im Worst Case:

$$d = \min_j \left(\{T_j | t_j \in CR\} \cup \{T_j - b_j | t_j \notin CR\} \right) \tag{5.4}$$

Wichtig ist zu bemerken, dass an dieser Stelle keine Zeitkonflikte, insbesondere keine schwachen Zeitguards, mehr vorliegen dürfen. Diese sollten vor dem Prozess der Partitionierung behoben worden sein.

5.4.2. Periodenbestimmung

Wie in Kapitel 5.3 ausführlich beschrieben und auch in Abb. 5.7 dargestellt, verbraucht der Hauptthread Zeit für die Aktivierungsüberprüfung, die `do()`- und die Aktionsausführung.[4] Bezüglich der Wcets dieser einzelnen Operationen werden 4 Fälle unterschieden. Dabei sei vorrausgesetzt, dass immer $w_{do()} + w_{guard} + w_{startA} \leq \nu p$ gilt. Da für `do()` gefordert wird, verhältnismäßig kurz zu sein und die Guardauswertung und das Starten eines aperiodischen Threads ebenfalls relativ schnell geschieht, ist diese Vorraussetzung im Regelfall immer erfüllt. Der Auslastungsfaktor ν berücksichtigt, dass nicht mehr als ein bestimmter Anteil einer Periode

[4] Es wird eine beliebige Transition betrachtet, die mit einer Aktion `a()` assoziiert ist. Wird von einer `do()`-Methode gesprochen, ist die des Quellzustands gemeint.

für die Ausführung zur Verfügung steht. Es muss berücksichtigt werden, dass bei der jeweils ersten Ausführung des Hauptthreads nach einem Locationwechsel, die `do()`-Methode aufgerufen wird, die Transitionen auf Aktivierung überprüft werden *und* gegebenenfalls noch die Aktion ausgeführt wird. Bei den übrigen Ausführungen findet lediglich die Aktivierungsüberprüfung und die Ausführung von `do()` *oder* der Aktion statt. Somit ergeben sich 4 verschiedene Fälle:

1. $w_{do()} + w_{guard} + w_{a()} \leq \nu p$. Alle Teile passen in eine Periode des Hauptthreads.

2. $w_{do()} + w_{guard} + w_{a()} > \nu p$. Zumindest bei der ersten Ausführung nach einem Locationwechsel muss `a()` in einen aperiodischen Thread ausgelagert werden.

3. $w_{guard} + w_{a()} \leq \nu p$. Wenn `do()` nicht ausgeführt wird, muss keine Aktion aus dem Hauptthread ausgelagert werden.

4. $w_{guard} + w_{a()} > \nu p$. Auch wenn `do()` nicht ausgeführt wird, muss `a()` ausgelagert werden.

Das Ziel des ersten Schrittes der Analyse ist, ein Intervall der Form $p^i_{min} \leq p \leq p^i_{max}$, $i = 1, \ldots, 4$ zu bestimmen. Dabei sind dann durch das Intervall die Werte für p bestimmt, die in dem jeweiligen Fall zu einem Treffen der Deadline führen würden. Anschließend werden die 4 Formeln, die für jede Transition aufgestellt werden, kombiniert, um den maximalen Wert für die Periode zu bestimmen, die für *jede* Transition in *jedem* der 4 Fälle die Deadline trifft.

zu 1.) In diesem Fall werden `do()`-Methode, Aktivierungsüberprüfung und Aktionsausführung vom Hauptthread ausgeführt. Dies muss –wenn überhaupt– nur im ersten Zyklus nach dem Schalten der Vorgänger-Transition geschehen. Abb. 5.10 zeigt den Fall, in dem, nach dem Schaltvorgang der Transition mit der Aktion a_0, in den aktuellen Zustand gewechselt wurde. Der nach unten gerichtete Pfeil kennzeichnet das Ende des Seiteneffektes und den Zeitpunkt des Erreichens des neuen Zustands. Sei sofort bei Erreichen des Zustands eine Transition aktiviert, kennzeichnet der Pfeil ebenfalls den Aktivierungszeitpunkt. Da der Scheduler dafür sorgt, dass der nun auszuführende Seiteneffekt `a()` spätestens zum Ende der zweiten Periode beendet wird, folgt für die Deadline: $d \geq 2p - (w_{g0} + w_{a0()})/\nu$ und somit mit der Vorbedingung

$$p^1_{t,min} := \frac{w_{do()} + w_{guard} + w_{a()}}{\nu} \leq p \leq \frac{d\nu + w_{g0} + w_{a0()}}{2\nu} =: p^1_{t,max} \tag{5.5}$$

Für eine Worst Case Betrachtung wird aus allen Vorgänger-Transitionen diejenige mit dem geringsten Wert für $w_{g0} + w_{a0()}$ ausgewählt. Der Index t berücksichtigt, dass für jede Transition ein eigener Wert berechnet wird.

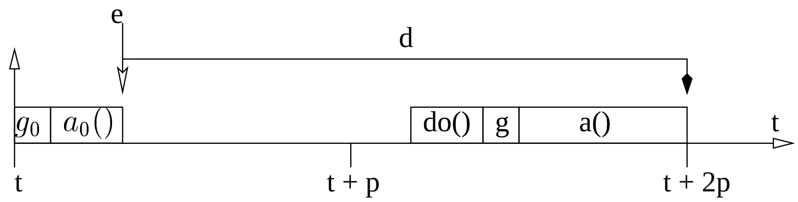

Abbildung 5.10.: Periodenbestimmung – Fall 1

zu 2.) In diesem Fall ist die Aktion, die ausgeführt werden soll, zu lang für den Hauptthread. Aufgrund dessen wird ein neuer aperiodischer Thread erzeugt, der vom Hauptthread gestartet wird und dann die Aktion ausführt. Nach dem Erzeugen und Starten (was eine nicht zu vernachlässigende Menge an Zeit w_{startA} benötigt) wird der Hauptthread nach wie vor periodisch ausgeführt, jedoch führt er keine weiteren Aktionen aus, weshalb sich seine Wcet auf $w_{g'}$ ändert.[5] Mit Hilfe von Abbildung 5.11 kann der Zeitpunkt, zu dem die Aktivierungsüberprüfung ausgeführt wird und somit auch der des Startes des aperiodischen Threads bestimmt werden. Der periodische Hauptthread ist mit einer Wcet von $w = \nu p$ beim Scheduler angemeldet. Dieser sorgt dafür, dass der Hauptthread spätestens νp vor Ende einer jeden Periode gestartet wird. Auch wenn die Wcet von $w = \nu p$ nicht vollständig ausgeschöpft wird, beginnt die Aktivierungsüberprüfung in Abb. 5.11 zum Zeitpunkt $t + 2p - \nu p$. Nach dem Start des aperiodischen Threads steht ihm noch die Zeit t_{rest} bis zum Ende der Periode zur Verfügung. Diese Zeit wird nicht ausreichen, um den aperiodischen Thread zu beenden (ansonsten wäre die Vorbedingung nicht erfüllt und es gelte Fall 1), so dass er in den folgenden Perioden fortgesetzt werden muss. Da seine Deadline –und somit auch seine Minimum Interarrival Time– größer als die Periode des Hauptthreads ist, erhält der aperiodische eine niedrigere Priorität und wird somit genau einmal pro Periode unterbrochen (siehe Kapitel 2.1.2 und Abb. 5.12). Auch in der letzten Periode, also in der, in der er beendet wird, muss in der Worst Case Betrachtung davon ausgegangen werden, dass der Hauptthread so früh ausgeführt wird, dass er ihn noch einmal unterbricht.

Eine mathematische Beschreibung der oben beschriebenen Zusammenhänge findet in Anhang B statt und führt –wie in Fall 1– zu einer Ungleichung der Form $p^2_{t,min} \leq p \leq p^2_{t,max}$, die im Absatz „Auswahl aus möglichen Perioden“ in diesem Kapitel weiterverwendet wird.

zu 3.) Dieser Fall ähnelt Fall 1. Da die Transition hier jedoch nicht im ersten Zyklus nach einem Zustandswechsel geschaltet wird, liegt der Worst Case vor, wenn

[5]Im Fall von Parallelität kann es dennoch vorkommen, dass der Hauptthread `do()` Operationen ausführen muss und aperiodische Threads startet. Dann wird die dafür benötigte Zeit in $w_{g'}$ berücksichtigt. Um eine bessere Übersicht zu erreichen, wird sie hier in den Graphiken nicht dargestellt.

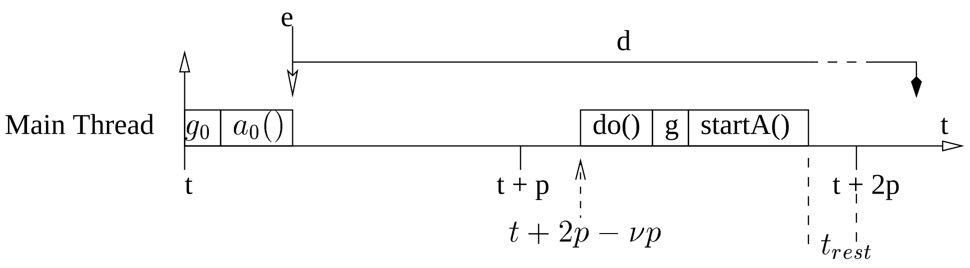

Abbildung 5.11.: Periodenbestimmung – Fall 2

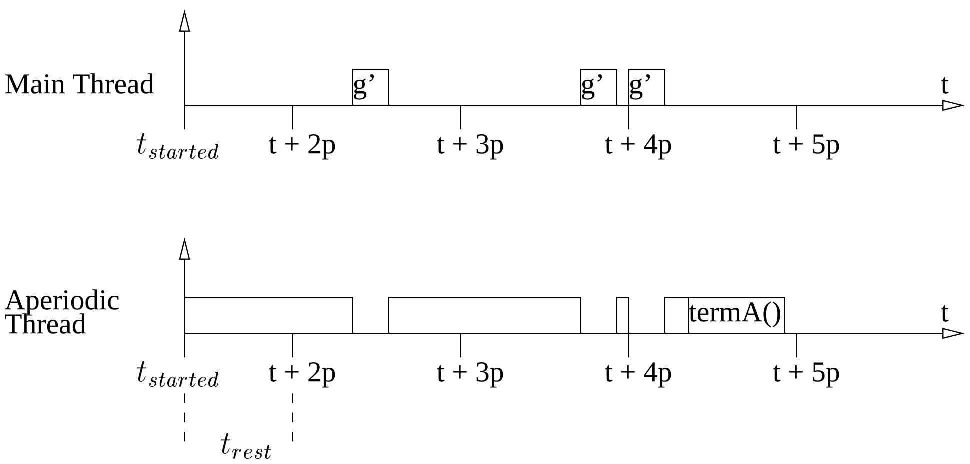

Abbildung 5.12.: Unterbrechung eines aperiodischen Threads

eine Transition unmittelbar nach Beginn der Aktivierungsüberprüfung aktiviert wird, aber dennoch zu spät, um erkannt zu werden (Zeitpunkt t in Abb. 5.13).

Somit folget für die Deadline: $d \geq 2p$ und mit der Vorbedingung:

$$p^3_{t,min} := \frac{w_{a()} + w_{guard}}{\nu} \leq p \leq \frac{d}{2} =: p^3_{t,max} \tag{5.6}$$

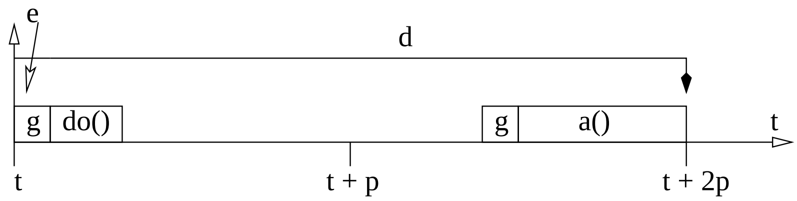

Abbildung 5.13.: Periodenbestimmung – Fall 3

zu 4.) Auch dieser Fall ähnelt Fall 2 und wird in Anhang B mathematisch analysiert. Für die Periode ergibt sich wieder eine Ungleichung der Form $p^4_{t,min} \leq p \leq p^4_{t,max}$.

Auswahl aus möglichen Perioden Für jeden der 4 Fälle existiert ein Intervall, in dem die Periode liegen muss. Für jede Transition (Aktion) t können also 4 Ungleichungen der Form

$$\begin{aligned}
p^1_{t,min} &\leq p \leq p^1_{t,max} \\
p^2_{t,min} &\leq p \leq p^2_{t,max} \\
p^3_{t,min} &\leq p \leq p^3_{t,max} \\
p^4_{t,min} &\leq p \leq p^4_{t,max}
\end{aligned}$$

aufgestellt werden. Gesucht ist ein Wert für p, der es ermöglicht, *jede* Aktion bis zum Ende ihrer Deadline auszuführen, sowohl wenn sie in derselben Periode wie die `do()`-Aktion ausgeführt wird (Fall 1 und 2) als auch wenn sie ohne `do()` ausgeführt wird (Fall 3 und 4). Des Weiteren wird nicht irgendein Wert gesucht, sondern der maximale:

Sei $A_t := \left\{p | \left(p^1_{t,min} \leq p \leq p^1_{t,max}\right) \vee \left(p^2_{t,min} \leq p \leq p^2_{t,max}\right)\right\}$

und $B_t := \left\{p | \left(p^3_{t,min} \leq p \leq p^3_{t,max}\right) \vee \left(p^4_{t,min} \leq p \leq p^4_{t,max}\right)\right\}$

dann folgt für p:

$$p = \max\left(\bigcap_{t \in T} (A_t \cap B_t)\right) \tag{5.7}$$

Anschaulich bedeutet dies, dass es für jede Transition und jeden der 2 Fälle (mit/ohne `do()`) 2 Zeitintervalle gibt, die die Menge der jeweils zulässigen Werte für p beschreiben (A_t bzw. B_t). Um eine Periode zu bestimmen, die für jede Transition zulässig ist, also in jeder Menge enthalten ist, wird die Schnittmenge gebildet und schließlich das Maximum dieser Schnittmenge ausgewählt. Dieses Verfahren liefert garantiert eine Lösung (somit ist es vollständig), da in jeder Menge der Wert $p_{min} = (w_{do()} + w_{guard()} + w_{startA})/\nu$ enthalten ist.

5.4.3. Perioden der `do()`-Operation

Für die Ausführung der `do()`-Operation eines Zustands wurde keine feste Periode, sondern ein Intervall von möglichen Perioden spezifiziert, so dass $p_{do} \in [p_{low}, p_{up}]$ gelten soll. Ist die Periode p des Hauptthreads so beschaffen, dass ein Vielfaches von ihr in das Intervall passt, wird dieses Vielfache für p_{do} gewählt:

$$p_{do} = p \left\lceil \frac{p_{low}}{p} \right\rceil, \text{falls}\, p \left\lceil \frac{p_{low}}{p} \right\rceil \leq p_{up} \tag{5.8}$$

Bei dem Verfahren treten Probleme auf, wenn das Realtime Statechart aus einer Vielzahl von Zuständen besteht für deren `do()`-Operationen unterschiedliche Perioden spezifiziert wurden. Um dann die Formel 5.8 anwenden zu können, muss die Vorbedingung für *jeden* Zustand erfüllt sein:

$$p \left\lceil \frac{p^1_{low}}{p} \right\rceil \leq p^1_{up} \wedge p \left\lceil \frac{p^2_{low}}{p} \right\rceil \leq p^2_{up} \wedge \ldots \tag{5.9}$$

Daher wird die Formel 5.7 zur Bestimmung der Periode des periodischen Hauptthreads so abgeändert, dass aus den Mengen A_t und B_t alle p herausgenommen werden, für die 5.9 nicht gilt. Je breiter die Intervalle $\left[p^i_{low}, p^i_{up}\right]$ sind, desto schwächer wird die Vorbedingung. Bei der Frage nach der Vollständigkeit dieses Verfahrens, kann die Formel $\lceil x \rceil < x + 1$ ausgenutzt werden. Es soll für jeden Zustand p$\left\lceil \frac{p_{low}}{p} \right\rceil \leq p_{up}$ gelten. Das ist erfüllt, wenn $p\left(\frac{p_{low}}{p} + 1\right) \leq p_{up} \Leftrightarrow p \leq p_{up} - p_{low}$. Die in Kapitel 5.4.2 bestimmte untere Grenze für die Periode lautet $p_{min} = (w_{do()} + w_{guard} + w_{startA})/\nu$. Unter der Vorraussetzung, dass die Breite der spezifizierten Intervalle mindestens p_{min} beträgt, existiert auf jeden Fall eine Wert (p_{min}), der alle Bedingungen erfüllt. Nach Bestimmung von p kann für jeden Zustand p_{do} mit der Formel 5.8 bestimmt werden.

5.5. Ereignis Behandlung

In den vorangegangenen Kapiteln wurde des Öfteren eine Eventqueue angesprochen. Nach welchem Konzept Ereignisse in die Queue eingetragen werden, wird in diesem Kapitel diskutiert. Dabei orientiert sich das hier vorgestellte Konzept an dem aus [Köh00]. Da Ereignisse in Seiteneffekten erzeugt werden (vgl. 4.1.1), wird –wie in [Köh00] beschrieben– das Erzeugen eines Ereignisses durch den Aufruf einer gleichnamigen Methode modelliert und implementiert. In dieser Methode wird eine Ereignis-Instanz von einer Ereignis-Factory erzeugt, mit Namen und Auftrittszeitpunkt versehen und in die Queue eingetragen.

Zusätzlich ist es möglich, der Ereignis-Instanz Parameter zu übergeben. Dies wird modelliert und implementiert, indem der entsprechenden Methode Parameter übergeben werden. Somit wird bei der Codeerzeugung für jedes Ereignis eine Methode generiert. Da der Type der Parameter zum Synthesezeitpunkt bekannt sein muss, müssen alle Ereignisse einmalig mit Anzahl und Typ der Parameter vom Anwender spezifiziert werden.

5.6. History

Wird ein hierarchischer Zustand, der als Deep History-Zustand gekennzeichnet ist, verlassen, wird der bis dahin aktuelle Zustand in einem Attribut namens `history` abgelegt. Bei Wiedereintritt über den History-Operator wird dann nicht in den Initialzustand, sondern in den Zustand, der im `history`-Attribut gespeichert ist, gewechselt.

Beispiel Betrachtet wird das Beispiel aus Abb. 5.14. Das Realtime Statechart befinde sich in den Zuständen S_6 und S_8 (und in deren übergeordneten Zuständen S_0, S_2, und S_4) sowie in S_9, dann hat der Zustandsbaum (siehe Kapitel 5.3.3) die in Abb. 5.15 dargestellte Form. Durch Schalten einer der Transitionen T_3, T_4 oder T_5 wird der als History-Zustand gekennzeichnete Zustand S_2 verlassen. Der Ausschnitt des Zustandsbaumes, der die gerade verlassenen Zustände beinhaltet, wird in S_2 als `history` gespeichert (siehe Abb. 5.16).

Im Fall von Shallow History würden die hierarchisch weiter unten befindlichen Zustände (S_6 und S_8) auf die entsprechenden Startzustände (S_5 und S_7) gesetzt werden.

5.7. Schedulinganalyse

Bei der Partitionierung eines Realtime Statecharts wird u.a. der Auslastungsfaktor berücksichtigt. Somit wird sichergestellt, dass es niemals mehr als einen bestimmten Prozentsatz an Rechenleistung des Prozessors für sich beansprucht. Laufen mehrere Realtime Statecharts parallel auf demselben System, muss sichergestellt werden, dass jede theoretisch mögliche Kombination von periodischen und

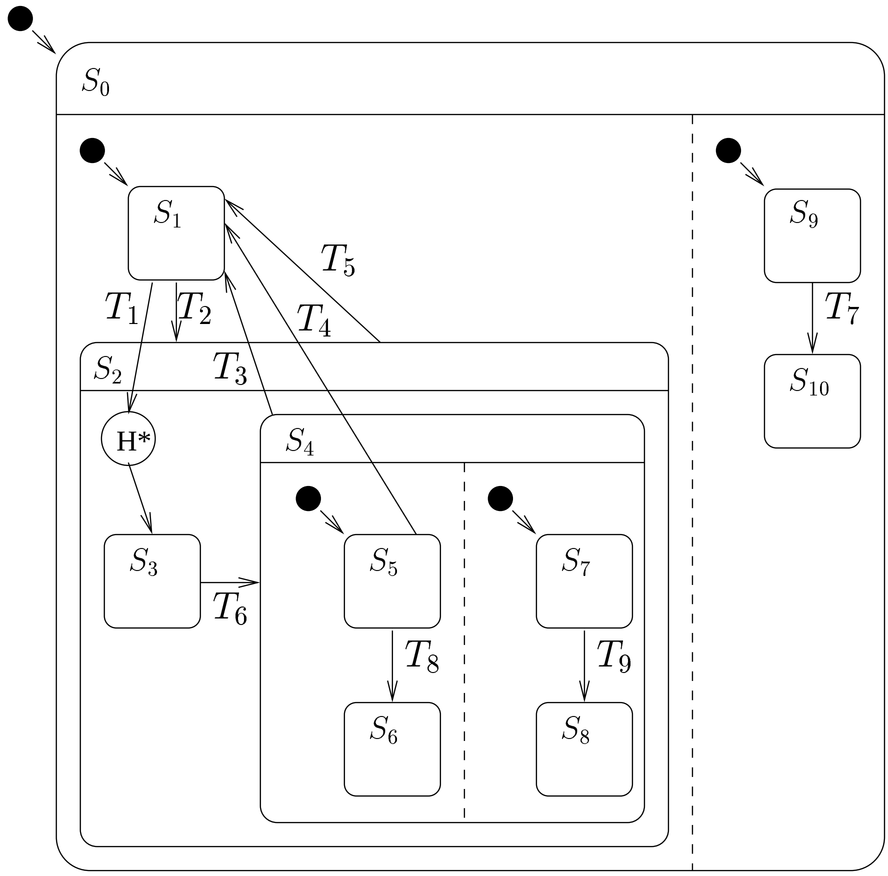

Abbildung 5.14.: History

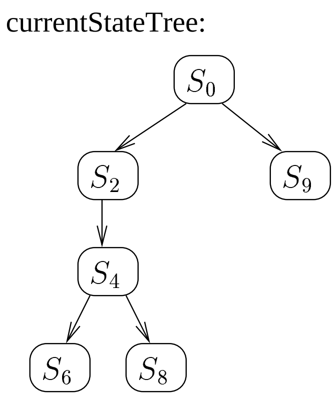

Abbildung 5.15.: Zustandsbaum zu Abb. 5.14

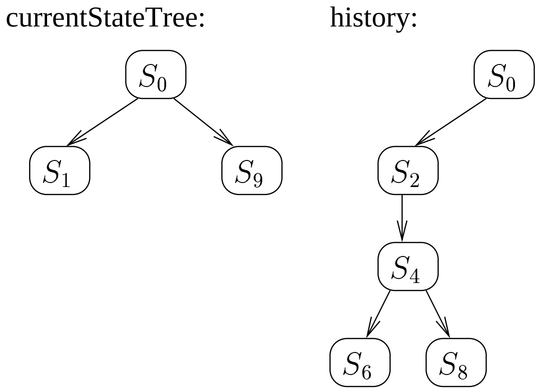

Abbildung 5.16.: Zustandsbaum und `history`-Attribut zu Abb. 5.14

aperiodischen Threads schedulbar ist. Um solch eine Schedulinganalyse durchführen zu können, müssen alle dafür notwendigen Eigenschaften eines jeden Realtime Statecharts bekannt sein. Dazu wird bei der Codeerzeugung zusätzlich ein weiteres XML-Dokument erzeugt, in dem aufgeführt ist, welche periodischen und aperiodischen Threads gestartet werden können, welche zueinander parallel laufen können und welchen Wert jeweils die Deadline, Wcet, maximale Blockierungszeit (aufgrund von Ressourcenzugriffen) und ggf. die Periode haben. Die Blockierungszeit berechnet sich dabei nach Formel 2.5 aus Kapitel 2.1.4 und hängt davon ab, welches Protokoll der Anwender zur Vermeidung von Prioritätsinversion gewählt hat.

Bei der Analyse wird dann zu jedem Realtime Statechart dieses zugehörige *Schedule Document* geladen, und anschließend auf Basis der Antwortzeitanalyse (siehe Kapitel 2.1.2) bestimmt, ob es sicher ist diese Realtime Statecharts parallel laufen zu lassen. Da die Prioritäten nach dem Rate Monotonic Ansatz vergeben werden -also eine kürzere Periode eine höhere Priorität erhält (bei aperiodischen Threads wird die Deadline anstatt der Periode betrachtet)- können diese erst vergeben werden, wenn die Menge *aller* Threads, die schließlich gestartet werden können, bekannt sind. Das bedeutet, dass zu dem Zeitpunkt, zu dem der Code und das Schedule-Dokument erzeugt werden, die Prioritäten noch nicht feststehen. Da aber im Fall einer erfolgreichen Schedulinganalyse der erzeugte Code nicht mehr geändert werden soll, wird eine weitere Klasse erzeugt, die jedem Thread seine Priorität zuordnet. Da außerdem an dieser Stelle spezifiziert wurde, welche Realtime Statecharts parallel laufen sollen, wird in diese neu synthetisierte Klasse die Startmethode[6] generiert, in der die Realtime Statecharts instanziiert und gestartet werden. Bevor diese Klasse synthetisiert werden kann, muss die Schedulinganalyse durchgeführt werden. Dazu werden in einem ersten Schritt die Prioritäten vergeben. Anschließend werden alle möglichen Kombinationen von Threads, die zueinander parallel laufen können, gebildet. Jede Kombination wird

[6]`public static void main(String[] args)` in Java

einzeln der Schedulinganalyse unterzogen, denn wenn jeder dieser Einzeltests erfolgreich ist, ist sichergestellt, dass die Realtime Statecharts problemlos parallel gestartet werden können.

5.8. Zusammenfassung

In diesem Kapitel wurde gezeigt, wie Echtzeitverhalten, das durch ein Realtime Statechart spezifiziert wurde, auf verschiedene Threads abgebildet wird. Dabei entsteht ein periodischer Thread, der das gesamte Modell „verwaltet“ und mehrere aperiodische Threads, die für die Aktionsausführung verantwortlich sind. Abb. 5.17 zeigt einen Überblick über den gesamten Ablauf:

Zunächst wird das Verhalten mit Hilfe von Realtime Statecharts modelliert. Für die im Modell verwendeten Operationen werden die Wcets durch ein externes Tool bestimmt und in einem XML-Dokument zur Verfügung gestellt. Werden bei einer anschließenden Prüfung Zeitinkonsistenzen erkannt, muss das Modell „nachgebessert“ werden. Nachdem alle Inkonsistenzen beseitigt wurden, wird das Realtime Statechart auf Threads abgebildet, von denen außerdem die Priorität und von den periodischen die Periode berechnet wird.

Im nächsten Kapitel wird die Codesynthese am Beispiel von Realtime-Java gezeigt. Neben dem Quellcode wird das Schedule-Dokument erzeugt, das für eine Schedulinganalyse benötigt wird, falls mehrere Realtime Statecharts parallel ausgeführt werden sollen. Ergibt die Schedulinganalyse, dass die Menge von Realtime Statecharts nicht schedulbar ist, müssen die einzelnen Modelle angepasst bzw. korrigiert werden.

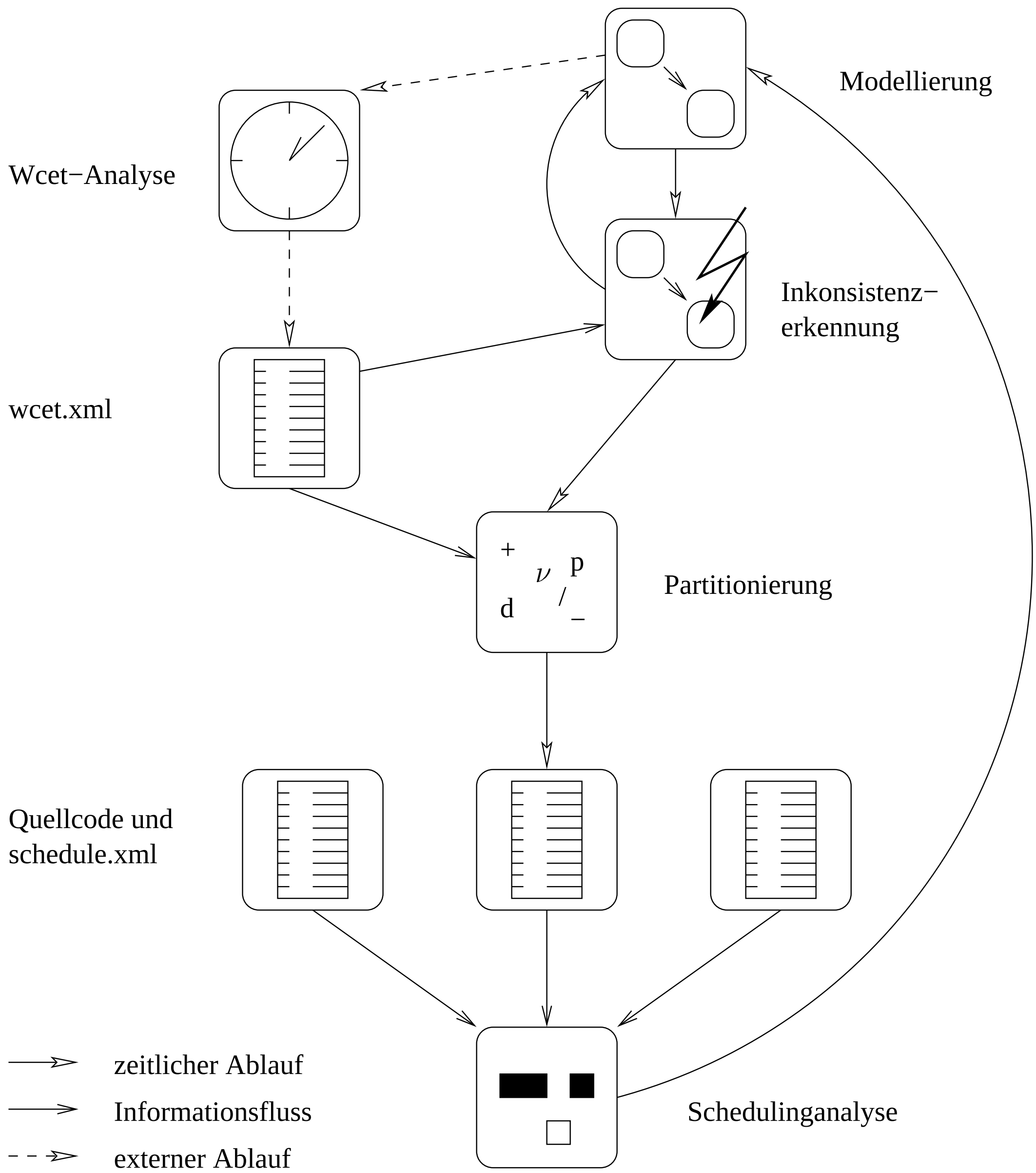

Abbildung 5.17.: Von der Modellierung zur Schedulinganalyse

6. Codesynthese

In den Kapiteln 4 und 5 wurde beschrieben, wie ein Realtime Statechart auf eine Menge von Threads abgebildet werden kann und wie sich aus den Annotationen des Modells die Parameter der Threads errechnen. In diesem Kapitel wird für diese Threads, basierend auf den Ergebnissen der vorherigen Kapitel und am Beispiel von Realtime Java, die Codesynthese beschrieben. Die Codesynthese wurde in die Entwicklungsumgebung Fujaba (From UML to Java And Back Again) [Fuj] integriert. Die graphische Repräsentation der Realtime Statecharts wurde ebenfalls, jedoch nicht im Rahmen dieser Arbeit, in Fujaba realisiert.

6.1. Integration in das Case-Tool 'Fujaba'

Die integrierte Entwicklungsumgebung Fujaba bot bereits vor Beginn dieser Arbeit die Möglichkeit nicht-zeitbehaftete Statecharts zu modellieren und aus dem Modell Code zu erzeugen. Das Modell des Statecharts wird in einer Datenstruktur in Fujaba abgelegt. Diese Datenstruktur wird als Metamodell bezeichnet. Das Metamodell der Realtime Statecharts orientiert sich an dem der Statecharts, wurde jedoch in ein eigenes Package (`umlrt.realtimestatechart`) ausgelagert. Diese Entscheidung wurde getroffen, da es zum einen unabhängig von der Struktur der Statecharts ist und zum anderen, um die gesamte Realtime Statechart-Erweiterung unabhängig von den Statecharts in ein eigenständiges Modul zusammenfassen zu können und Fujaba als Plugin zur Verfügung zu stellen.

Im Fujaba ist ein Statechart einer Klasse zugeordnet, was durch eine Assoziation zwischen den Klassen `UMLStatechart` und `UMLClass` modelliert war. Diese Klassen befinden sich im `uml`-Package. Ein RealtimeStatechart –das durch die Klasse `UMLRealtimeStatechart` repräsentiert wird– soll ebenfalls einer Klasse zugeordnet sein. Eine Möglichkeit wäre `UMLRealtimeStatechart` von `UMLStatechart` erben zu lassen, was allerdings bewirken würde, dass es möglich wäre, mit einem Realtimestatechart das Modell eines Statecharts zu assoziieren (Beispiel: Ein `UMLStatechart` hat eine Assoziation zu einem `UMLComplexState`. Ein `UMLRealtimeStatechart` soll stattdessen eine Assoziation zu einem `UMLComplexRealtimeState` besitzen). Aufgrund dessen wurde das Metamodell wie in Abb. 6.1 erweitert.

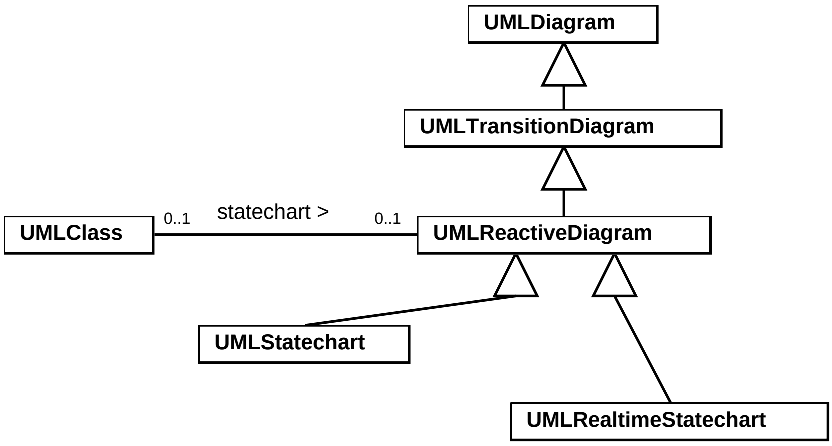

Abbildung 6.1.: Anbindung der Realtime Statecharts an das Metamodell

Die Assoziation von `UMLClass` zu `UMLStatechart` wurde durch die Assoziation zu der neu eingeführten Klasse `UMLReactiveDiagram` ersetzt. Von dieser neuen Klasse erben sowohl `UMLStatechart` als auch das sich in einem anderen Package befindliche `UMLRealtimeStatechart`. Somit bleibt das `uml`-Package vom `umlrt`-Package unabhängig. Aufgrund der Strukturierung von Fujaba ist es erforderlich, dass jede Diagrammart von `UMLDiagramm` erbt.

Im Metamodell von Fujaba finden sich Instanzen verschiedener Diagrammtypen wieder, die wiederum Instanzen unterschiedlicher diagrammspezifischer Klassen referenzieren. Bei der Codesynthese werden alle diese Objekte von einer *Chain of Responsibility* [GHJV94] verwaltet (`OOGenStrategyHandler` in Abb. 6.2). Soll z.B. für ein `UMLFile` Code generiert werden, wird der `UMLFileOOHandler` aufgerufen. Dieser synthetisiert entsprechenden Code und startet die Codegenerierung für die mit ihm assoziierten Klassen (die Klassen, die innerhalb des Files deklariert wurden). Das bewirkt, dass der `UMLClassOOHandler` aufgerufen wird. Dieser wiederum muss seine `UMLMethod`-Instanzen behandeln usw. Um die Codegenerierung für ein `UMLRealtimeStatechart` zu ermöglichen, wird beim Laden des Plugins der `UMLRealtimeStatechartOOHandler` in diese Chain of Responsibility eingefügt.

Im Rahmen dieser Arbeit ist das Package `uml.realtimestatechart` entstanden, das die Klassen des Metamodells und zur Codegenerierung enthält. Die Klassenstruktur wurde in die bestehende Umgebung integriert.

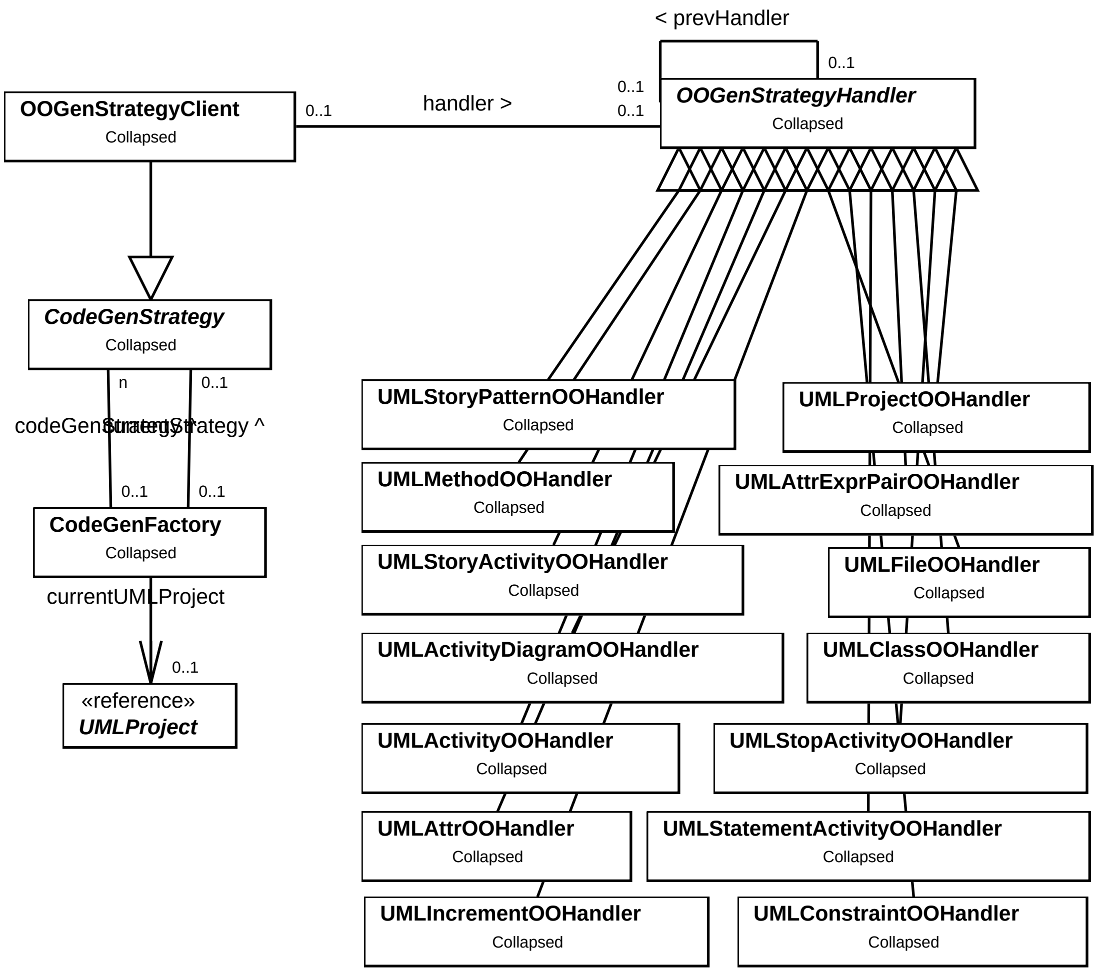

Abbildung 6.2.: Codegenerierung in Fujaba

6.2. Das Metamodell

Das Metamodell beinhaltet –wie bereits in Kapitel 6.1 erläutert– die Informationen des Realtime Statecharts, die beim Modellieren spezifiziert wurden.

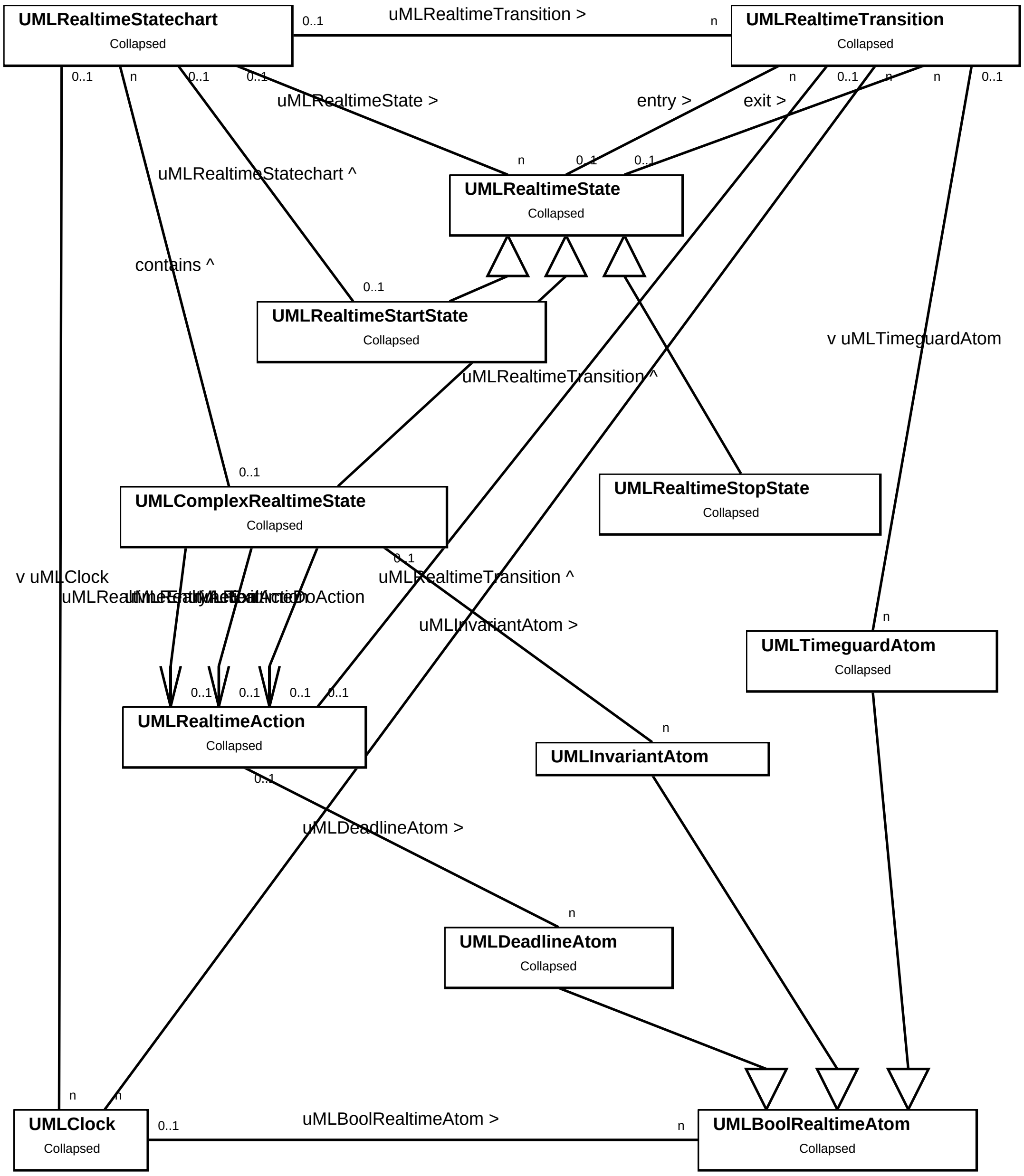

Abbildung 6.3.: Metamodell der Realtime Statecharts (Ausschnitt)

Abb. 6.3 zeigt einen Ausschnitt aus dem Klassendiagramm des Metamodells. Basis eines Realtime Statecharts ist die Klasse `UMLRealtimeStatechart`. Diese ist mit einer Menge von Zuständen (`UMLRealtimeState`), Transitionen (`UMLRealtimeTransition`) und Uhren (`UMLClock`) assoziiert. Ein Zustand kann mehrere eingehende und mehrere ausgehende Transitionen haben, während eine Transition aus genau einem Quell- und einem Zielzustand besteht, was über die beiden Assoziationen `entry` und `exit` modelliert ist. Es existieren 3 verschiedene Zustände: `UMLRealtimeStartState` (von dem genau einer einem `UMLRealtimeStatechart` zugeordnet ist), `UMLRealtimeStopState` und `UMLComplexRealtimeState`. Letzterer kann im Fall von Hierarchie oder Parallelität wiederum ein oder mehrere `UMLRealtimeStatecharts` enthalten (vgl. Assoziation `contains`). Außerdem sind mit einem `UMLComplexRealtimeState` drei `UMLRealtimeActions` verknüpft, die die `entry()`-, `exit()`- und `do()`- Methoden repräsentieren.

Eine `UMLRealtimeTransition` besitzt neben Attributen, die Guard und Ereignis repräsentieren, einen Seiteneffekt in Form einer weiteren `UMLRealtimeAction`, eine Menge von Uhren, die beim Schaltvorgang zurückgesetzt werden, sowie einen Zeitguard. Die spezielle Form, die für den Zeitguard gefordert wird (siehe Kapitel 4.1.1), wird durch die Assoziationen zum `UMLTimeguardAtom` modelliert. Ein `UMLTimeguardAtom` besitzt 2 Attribute sowie eine Assoziation mit einer `UMLClock` und entspricht dadurch einem Ausdruck der Form $a_j \leq t_j \leq b_j$. Über die gleiche Struktur werden Zeitinvarianten an einen `UMLComplexRealtimeState` und Deadlines an eine `UMLRealtimeAction` gebunden.

`UMLRealtimeAction` erbt von `UMLAction` aus dem `uml`-Package (nicht in Abb. 6.3 dargestellt). Dadurch ist es möglich, als Aktion eine Zeichenkette oder eine Methode zu spezifizieren. Letztere kann selbst wiederum als Aktivitäts- oder Story Diagramm (siehe Kapitel 2.3.2 und 2.3.3) spezifiziert sein.

In Kapitel 2.1.4 wurde der Effekt der Prioritätsumkehrung vorgestellt. Der Effekt tritt auf, wenn Methoden, die durch verschiedene `UMLRealtimeActions` repräsentiert werden, auf gemeinsamen Ressourcen operieren. Dann müssen die Ressourcen-Zugriffe durch den Einsatz so genannter Monitore[1] synchronisiert werden. Der Anwender kreiert pro Monitor eine Klasse, in deren Methoden er die kritischen Bereiche auslagert (siehe Abb. 6.4).

Für jede Methode, die in einem Monitor spezifiziert wird, wird eine Wcet mit angegeben. Je nach eingesetztem Protokoll zur Vermeidung von Prioritätsumkehrung ergibt sich die für eine Schedulinganalyse notwendige maximale Blockierungszeit eines Threads (einer `UMLRealtimeAction`) aus den Wcets der Monitor-Operationen, der Anzahl der `UMLRealtimeActions` und der Anzahl der Monitore (vgl. Formel 2.5 in Kapitel 2.1.4). Da die RTSJ nur die Implementierung des

[1] In Java wird z.B. ein Monitor implementiert, indem die, sich gegenseitig ausschließenden, Bereiche durch das Schlüsselwort `synchronized` eingeleitet werden. Zu ausführlichen Informationen über Monitore und deren Implementierung sei auf [Kas] verwiesen.

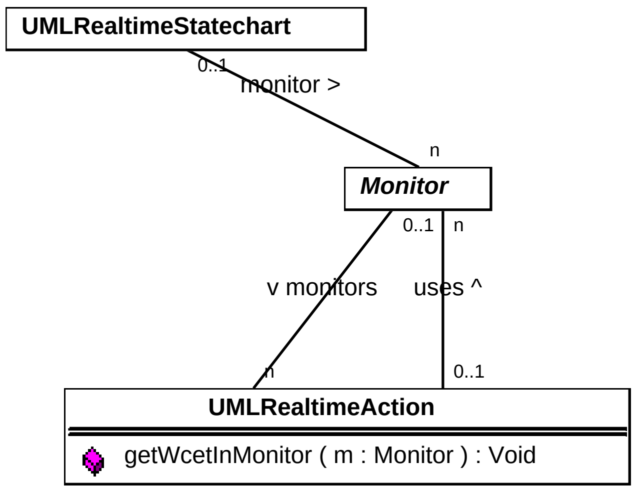

Abbildung 6.4.: Metamodell-Ausschnitt für Monitore

Priority Inheritance Protokolls vorschreibt, wird dieses hier eingesetzt.

6.3. Implementierungsvarianten

Es existieren 3 verbreitete Möglichkeiten einen Automaten zu implementieren (siehe [Dou99, Köh00]):

- durch State Pattern
- durch State Table Pattern
- durch Fallunterscheidungen.

State Pattern Beim Statepattern gibt es 2 verschiedene Implementierungsvarianten. Bei der klassischen Umsetzung existiert eine abstrakte Oberklasse, von der pro Zustand eine Klasse erbt und jeweils die entsprechenden `entry()`- , `do()`- und `exit()`-Methoden, sowie die Methode `handleEvent(FEvent Event)` implementiert (siehe Abb. 6.5). Jeder Zustand ist durch genau ein Objekt seiner entsprechenden Klasse repräsentiert, wovon eins über die `current`-Assoziation von der Handlerklasse (`Statechart` im Beispiel) referenziert wird. Ein Ereignis wird vom aktuellen Zustand abgearbeitet, indem auf dem `current`-Objekt `handleEvent(FEvent Event)` aufgerufen wird. Diese Methode führt gegebenenfalls Seiteneffekt sowie `entry()`, `do()` und `exit()` aus und setzt die `current`-Referenz auf den Zielzustand. Der Nachteil dieser Methode ist, dass neben der Handlerklasse, die den Automaten repräsentiert, für jeden Zustand eine Klasse synthetisiert

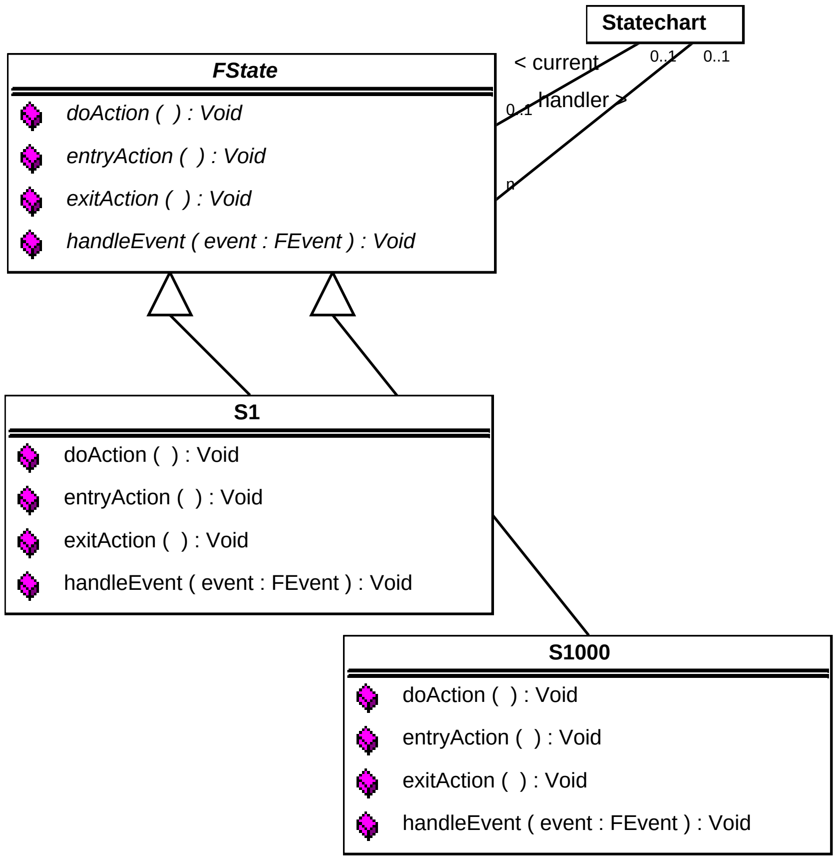

Abbildung 6.5.: State Pattern mit Vererbung

wird, wodurch die Anzahl durch die Virtuelle Maschine (VM) zu ladenden Klassen sehr groß werden kann. Alternativ dazu kann man auch mit einer einzigen nicht-abstrakten Zustandsklasse auskommen, wie in Abb. 6.6 dargestellt. `handleEvent` und die `entry()`- , `do()`- und `exit()`-Methoden der Klasse `FState` werden dann mit Verwendung der `java.lang.reflect` Klassen, wie in Abb. 6.7, implementiert.

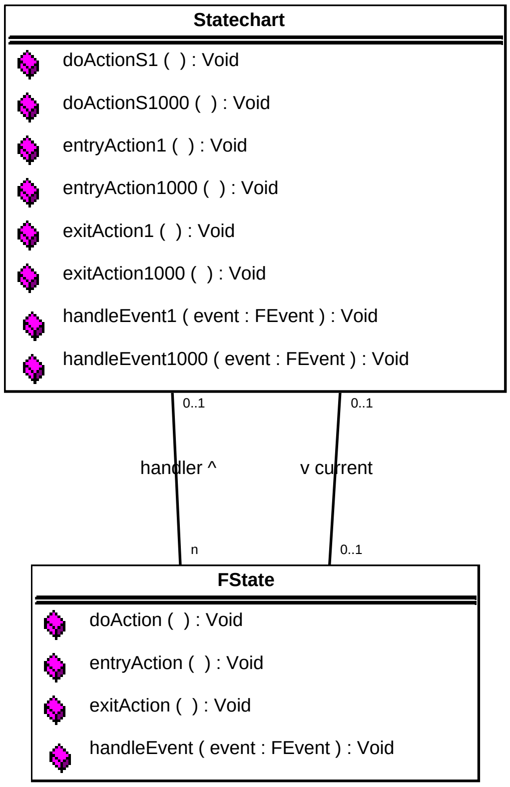

Abbildung 6.6.: State Pattern mit `reflect`-Klassen

```
FState(String entryName, String doName, String exitName,
       String handleEventName) {
  entryMethod = handler.getClass().getMethod(entryName, null);
  //do, exit and handleEvent similar
}
Method entryMethod;
public void entry() {
  entryMethod.invoke(handler, null);
}
//do, exit and handleEvent similar
```

Abbildung 6.7.: Verwendung der Reflektion

In der Handlerklasse werden für jeden Zustand 4 entsprechende Methoden synthetisiert, deren Namen dem Konstruktor `FState` übergeben werden. Der Vorteil dieses Verfahrens ist, dass lediglich die Handlerklasse synthetisiert werden muss. An der Anzahl der zur Laufzeit benötigten Objekte ändert sich jedoch nichts. Nachteilig wirkt sich aus, dass für jeden Zustand 4 Methoden in die Handlerklasse generiert werden, wodurch diese sehr groß werden kann.

State Table Pattern Eine Zustandstabelle (State Table) kann wie eine $n \times m$-Matrix aufgefasst werden, wobei n die Anzahl der Zustände und m die Anzahl der Ereignisse entspricht. Jedes Element der Matrix referenziert entweder genau ein `FTransition`-Objekt oder ist leer (`null`). Nur, wenn für den Zustand n' eine ausgehende Transition, die mit dem Ereignis m' verknüpft ist, existiert, dann ist der Eintrag (n', m') ein Transitionsobjekt. Jenes Objekt wiederum hält eine Referenz auf den Zielzustand und beinhaltet je eine Methode, die den Guard und den Seiteneffekt repräsentiert [Dou99]. Da ein Zustand mehrere ausgehenden Transitionen mit verschiedenen Guards, Seiteneffekten und Zielen, aber mit demselben Ereignis enthalten kann, ist in einem Transitionsobjekt nicht nur eins, sondern jeweils eine Menge dieser Repräsentanten abgelegt. Die verschiedenen Methoden können wiederum durch Vererbung oder durch Einsatz der `java.lang.reflect`-Klassen implementiert werden (vgl. State Pattern). Abb. 6.8 zeigt, wie die Zustandstabelle in [Köh00] realisiert wurde. Die Matrix wurde durch 2 “1 : n” Assoziationen ersetzt: Die Handlerklasse hält n `FState`-Objekte, welche wiederum jeweils m (für jedes Ereignis ein) `FTransition`-Objekt referenziert. Im Rahmen jener Arbeit wurde die Codeerzeugung für nicht-zeitbehaftete Statecharts nach dem State Table Pattern mit den `reflect`-Klassen umgesetzt und in das Case-Tool Fujaba integriert.

Fallunterscheidung Bei der Implementierung durch Fallunterscheidung wird in der Laufzeitumgebung nur das Handlerobjekt erzeugt. In diesem Objekt wird für jeden Zustand eine Konstante definiert. In einer Variablen ist dann der jeweils aktuelle Zustand abgelegt. Somit kann durch eine Fallunterscheidung bestimmt werden, welche Aktion bei Ausführung von `entry()`, `do()` oder `exit()` ausgeführt werden muss (siehe Abb. 6.9).

Da der `switch`-Befehl nur auf `int`-Daten arbeitet,[2] und die Auswertung von `switch-case`-Statements optimiert wird, indem hier die einzelnen `switch`-Fälle “direkt als Index in eine effizient darstellbare Tabelle von Sprungzielen” [LY97] übersetzt werden, wird als Datentyp für die Konstanten `int` gewählt.

Nachteilig ist bei dieser Variante, dass die einzelnen Methoden und somit die Handlerklasse sehr groß wird. Der Vorteil ist, dass auch zur Laufzeit nur ein einziges Objekt erzeugt wird.

[2] `switch` arbeitet ebenfalls auf `byte`-, `char`- und `short`-Typen, repräsentiert diese aber intern als `int` (siehe [LY97]).

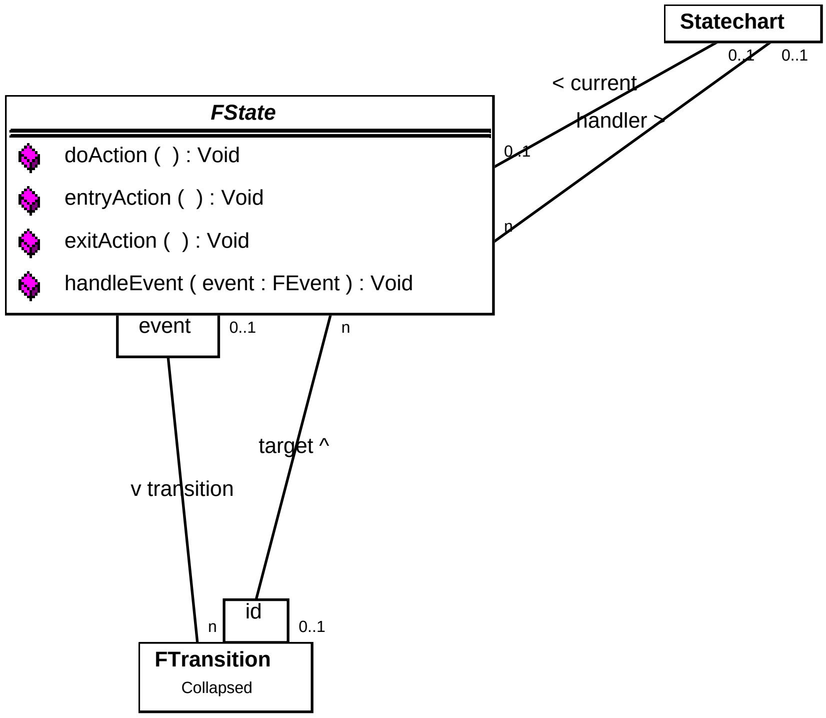

Abbildung 6.8.: State Table Pattern

```
//A Constant for every state
public static void int F_STATE1 = 0;
public static void int F_STATE2 = 1;
public static void int F_STATE3 = 2;
//The current state
private int current;
public void entry() {
  switch(current) {
    case F_STATE1:
      //code for the 'entry()'-Method of State1
    break;
    case F_STATE2:
      //code for the 'entry()'-Method of State2
    break;
    case F_STATE3:
      //code for the 'entry()'-Method of State3
    break;
  }
}
```

Abbildung 6.9.: Fallunterscheidung

Einsatz der Implementierungsvarianten in Echtzeitsystemen Die vorgestellten Varianten unterscheiden sich in verschiedenen Punkten: Die Java Virtuelle Maschine (JVM) muss eine unterschiedliche Anzahl von Klassen unterschiedlicher Größe laden, eine unterschiedliche Anzahl von Objekten erstellen und durch verschiedene Mechanismen auf die verschiedenen Codefragmente der einzelnen Methoden zugreifen. Welche Implementierungsvariante die beste bezüglich der Performanz ist, hängt somit von der Implementierung der JVM ab. Für das in dieser Arbeit verwendete System wurde ein Test durchgeführt.[3] Getestet wurde die JVM "tjvm" der Firma TimeSys [Tim], die u.a. die RTSJ implementiert, und Suns JVM der Standard Edition 1.4.0_01-b03.

Bei dem Test wurde angenommen, man habe ein Statechart mit 1000 Zuständen, die jeweils eine `entry()`-Methode haben, deren Methodenrumpf jedoch leer ist. Es wurde jeweils ein Test für die Vererbung, für den Einsatz der `reflect`-Klassen und für die Fallunterscheidung durchgeführt. In den Tests der beiden Varianten des State Patterns wurden zunächst die 1000 Objekte erstellt, die die Zustände repräsentieren. Die dafür benötigte Zeit sowie der verbrauchte Speicher wurden gemessen, wobei die Zeit an dieser Stelle eine untergeordnete Rolle spielt, da die Erstellung der Datenstruktur einmaliger Initialisierungsaufwand ist. Bei der Umsetzung durch Fallunterscheidung muss keine solche Datenstruktur erstellt werden, weshalb hierfür keine Messwerte vorliegen. Anschließend wurde unabhängig von jeglicher Automatensemantik von jedem Zustand die `entry()`-Methode aufgerufen. Es wurden also keine Transitionen erzeugt oder Ereignisse verarbeitet o.ä. Dabei wurden Zeit- und Speicherverbrauch für die 1000 Methodenaufrufe gemessen. 40 Durchläufe ergaben für Suns JVM folgende (gerundeten) Durchschnittswerte:

Suns JVM	Vererbung	`reflect`	Fallunterscheidung
$mem_{handlerClass}$	50 203 Bytes	48 532 Bytes	39 774 Bytes
$mem_{datastructure}$	156 800 Bytes	90 992 Bytes	-
t_{init}	2 867 ms	298 ms	-
$mem_{methodCalls}$	0 Bytes	40 352 Bytes	0 Bytes
$t_{methodCalls}$	27 ms	37 ms	1 ms

Dabei haben die Bezeichner folgende Bedeutung:

$mem_{handlerClass}$: Größe der Handlerklasse[4]

$mem_{datasructure}$: Speicherverbrauch der Datenstruktur

[3] Als System wurde ein Pentium-III MMX mit 450 MHz und 512 MByte Speicher, sowie Red Hat Linux 7.1 eingesetzt, wobei der Kernel durch den TimeSys Linux-Kernel [Tim] ausgetauscht wurde.

[4] Bei der Vererbungsvariante besteht die Handlerklasse ausschließlich aus Anweisungen zum Aufbau der Datenstruktur und zum Aufrufen der Methoden. In der `reflect`-Handlerklasse stehen 1000 Methodendeklarationen, während bei der Fallunterscheidung nur 1 Methode mit 1000-fachem `switch-case` deklariert wird.

t_{init}: Zeitverbrauch für das Erzeugen der Datenstruktur

$mem_{methodCalls}$: Speicherverbrauch für 1000 Methodenaufrufe.

$t_{methodCalls}$: Zeitverbrauch für 1000 Methodenaufrufe.

Zu bemerken ist, dass bei Verwendung der `java.lang.reflect`-Klassen ein Methodenaufruf Speicher verbraucht. Die Fallunterscheidung ist mit Abstand die schnellste Variante. Derselbe Test lieferte für "tjvm":

TJVM	Vererbung	`reflect`	Fallunterscheidung
$mem_{handlerClass}$	50 203 Bytes	48 532 Bytes	39 774 Bytes
$mem_{datastructure}$	*OutOfMemory*	259 608 Bytes	-
t_{init}	-	175 ms	-
$m_{methodCalls}$	-	0 Bytes	0 Bytes
$t_{methodCalls}$	-	6 ms	1 ms

Für die Vererbungsvariante des State Patterns liegen keine Messwerte vor, da das Testprogramm mit einem `OutOfMemoryError` abbrach. Das macht diese Methode der Umsetzung für den Einsatz in dieser Arbeit unbrauchbar. Bemerkenswert ist, dass bei Einsatz der `reflect`-Klassen kein Speicher bei den Methodenaufrufen verbraucht wird. Vermutlich wurde der dafür benötigte Speicher bereits bei der Initialisierung allokiert, wodurch sich der hohe Speicherverbrauch an dieser Stelle erklären lässt. Wie zu erwarten war, hat die Implementierung durch die Fallunterscheidung die um ein Vielfaches bessere Performanz. Aufgrund dessen wird in dieser Arbeit Code nach dieser Variante synthetisiert.

6.4. Threads und Speicherverwaltung

Wie in Kapitel 5 beschrieben, wird das Realtime Statechart durch einen periodischen Hauptthread verwaltet, der unter Umständen weitere aperiodische Threads startet. Da in dieser Arbeit Java Code erzeugt wird, muss der Einsatz des Garbage Collectors berücksichtigt werden. Alle Überlegungen aus den Kapiteln 4 und 5, um Zeitschranken einzuhalten, wären hinfällig, wenn ein Thread jederzeit für eine unbestimmte Zeit vom Garbage Collector unterbrochen werden könnte. In Kapitel 2.2.2 wurde der `NoHeapRealtimeThread` vorgestellt, der immer eine höhere Priorität als der Garbage Collector hat und somit niemals von ihm unterbrochen wird, bzw. von ihm am Starten gehindert wird. Das Starten eines `NoHeapRealtimeThreads` ist nicht so trivial wie das Starten eines "normalen" Threads (`java.lang.Thread`), da diese Objekte nicht im Heap Speicherbereich allokiert werden dürfen. Da der Hauptthread bis zur Terminierung aktiv ist, bietet es sich an, ihn im Immortal Memory unterzubringen. Im Immortal Memory dürfen `NoHeapRealtimeThreads` jedoch nur aus `RealtimeThreads` heraus allokiert werden, nicht aus normalen `Threads` heraus (das gilt im Übrigen auch für das Scoped Memory). In Abb. 6.10 ist die Klassenstruktur der Laufzeitumgebung dargestellt, die u.a. den Startvorgang verwaltet. Dabei wird im Heap Speicherbereich ein `RealtimeThread` (`FStarter`) erzeugt, der nichts anderes tut, außer den `NoHeapRealtimeThread` `FMainThread` im Immortal Memory zu erzeugen und zu starten. Um die Logik der Threads, also den Inhalt der `run()`-Methoden, einfach ändern bzw. erweitern zu können, wurde sie in die `Runnable`-Objekte `FStartLogic` und `FPeriodicLogic` ausgelagert. Aus demselben Grund wurde die Klasse `FAperiodicStartLogic` eingeführt, in der beschrieben ist, wie aus dem periodischen Hauptthread aperiodische gestartet werden (s.u.).

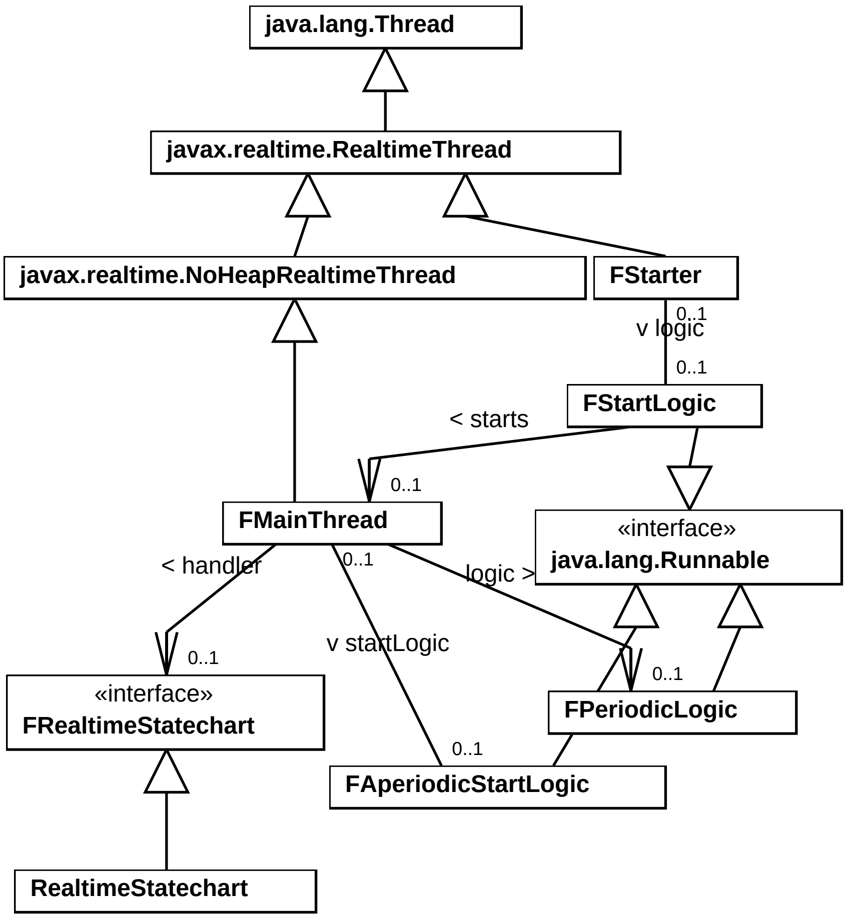

Abbildung 6.10.: Speichermanagement

Die Instanzen der Klasse `FMainThread` werden im Immortal Memory allokiert. Das wird dadurch sichergestellt, dass alle Konstruktoren private Zugriffsrechte haben und somit Instanzen nur durch die Methode `getNewInstance()` erzeugt werden können (siehe Abb. 6.11).

Durch die `if`-Abfrage wird sichergestellt, dass die Methode aus einem Realti-

```
private static FMainThread instance;
public static FMainThread getNewInstance() {
  if (Thread.currentThread() instanceof RealtimeThread) {
    ImmortalMemory.instance().enter(new Runnable() {
      public void run() {
        FPeriodicLogic periodicLogic = new FPeriodicLogic();
        instance = new FMainThread(null, ImmortalMemory.instance(),
                                   periodicLogic);
      }
    });
  }
  else
  {
    throw new RuntimeException("FMainThread::getNewInstance()
                                must be called from a
                                RealtimeThread");
  }
  return instance;
}
```

Abbildung 6.11.: Instanziierung im Immortal Speicherbereich

meThread heraus aufgerufen wurde. Dies ist wichtig, da ansonsten das folgende Standardkonstrukt, welches ein Objekt im Immortal Speicherbereich plaziert, eine `IllegalStateException` verursacht. Ist die Bedingung nicht erfüllt, wird eine `RuntimeException` mit der Fehlerursache als `message`-Attribut erzeugt.

Zur Laufzeit wird das Realtime Statechart im Wesentlichen von 2 Objekten repräsentiert: Eine Instanz der Handlerklasse, in deren Methoden und Attributen die Daten des Realtime Statecharts wiederzufinden sind, und eine Instanz der Logik des Hauptthreads, in der der Ablauf festgelegt ist, wie er in Kapitel 5.3 beschrieben wurde, und der für jedes Realtime Statechart gleich ist. Der Typ des Handlerobjekts entspricht dem Klassennamen, der mit dem Realtime Statechart assoziiert ist. Somit sind Name und Oberklassen der Handlerklasse für jedes Realtime Statechart individuell und nicht im Hauptthread bekannt. Um dennoch vom Hauptthread auf das Handlerobjekt zugreifen zu können, ist der Einsatz des Interfaces `FRealtimeStatechart` notwendig (siehe Abb. 6.10).

Eine Aufgabe des Hauptthreads ist es, aperiodische Threads zu starten. Das geschieht über die Method `executeAperiodicThread` aus Abb. 6.12.

Die Logik der aperiodischen Threads steht in der `run()`-Methode des `FRealtimeStatechart`-Objekts namens `logic`. Durch die Anweisung `scopedMemory.enter(aperiodicStartLogic)` wird dafür gesorgt, dass ein neues Objekt vom Typ `FAperiodicThread` im Scoped Memory erzeugt wird. Die `run()`-Methode von `aperiodicStartLogic` (die durch den `enter`-Aufruf ausgeführt wird) sieht wie in Abb. 6.13 dargestellt aus.

```
public void executeAperiodicThread(FRealtimeStatechart logic,
                                   long deadline,
                                   int transition) {
  aperiodicStartLogic.setLogic(logic);
  aperiodicStartLogic.setMemoryArea(scopedMemory);
  aperiodicStartLogic.setDeadline(deadline);
  aperiodicStartLogic.setTransition(transition);
  scopedMemory.enter(aperiodicStartLogic);
}
```

Abbildung 6.12.: Auszug aus der Klasse FMainThread

```
public void run() {
  PriorityParameters pp = new PriorityParameters(
                          logic.getPriorityForScheduling(transition));
  long deadline = logic.getDeadline(transition);
  AperiodicParameters ap = new AperiodicParameters(null,
                              new RelativeTime(deadline, 0), null,
                              new DeadlineMissHandler(transition));
  FAperiodicThread aperiodic = new FAperiodicThread(pp, ap, memoryArea,
                                  logic, transition);
  aperiodic.start();

  try {
   Thread.sleep(logic.getFMainThread()
               .getHandler().SLEEP_AFTER_THREAD_START);
  } catch (InterruptedException e) {e.printStackTrace();}
}
```

Abbildung 6.13.: Auszug aus der Klasse FAperiodicStartLogic

Zunächst werden Objekte im Scoped Speicherbereich erzeugt, die die Priorität und die Deadline des zu startenden Threads repräsentieren. Ein `DeadlineMissHandler` wird aufgerufen, falls die Deadline nicht getroffen wird. In dem Handler kann dann angemessen auf dieses Ereignis reagiert werden oder es wird eine `Exception` erzeugt. Die übrigen Parameter, die dem Konstruktor übergeben werden, sind jene, die vom Hauptthread vor dem `enter`-Aufruf gesetzt wurden.[5] Die `run()`-Methode des aperiodischen Thread-Objektes führt lediglich die ihm übergebene Logik aus (Abb. 6.14).

```
public void run() {
  aperiodicLogic.run();
}
```

Abbildung 6.14.: `run()`-Methode der Klasse `FAperiodicThread`

Durch den `enter`-Aufruf wird der Reference Count des Scoped Memory um 1 erhöht, durch das Starten des aperiodischen Threads ebenfalls um 1. Wird die `run()`-Methode von `aperiodicStartLogic` beendet, wird außerdem die `enter`-Methode beendet, wodurch der Reference Count um 1 erniedrigt wird. Nach Beendigung des aperiodischen Threads wird der Reference Count erneut um 1 erniedrigt. Daraufhin liegt der Reference Count wieder bei 0, der Finalizer des `FAperiodicThread`-Objekts wird aufgerufen und der allokierte Speicher vom Garbage Collector freigegeben, was zu einem deterministischen Worst Case Verhalten führt. Im Fall von Parallelität kann es vorkommen, dass bei Beendigung eines aperiodischen Threads noch weitere laufen. Dann würde der Reference-Count erst auf 0 fallen, wenn alle aperiodischen Threads beendet sind. Durch ein ständiges Erzeugen und Starten aperiodischer Threads könnte das verhindert werden, was zu einem Speicherleck führen würde. Um das zu verhindern, existiert pro Parallelitätsschacht ein eigener Scoped Speicherbereich.

6.5. Logik des Hauptthreads

Das Sequenzdiagramm aus Abb. 6.15 zeigt die Interaktion zwischen dem periodischen Hauptthread (`periodic`) und der Handlerklasse (`handler`). Zu Beginn

[5] Die Spezifikation schreibt nicht vor, nach solch einem Start eines Threads im Scoped Speicherbereich ein `Thread.sleep` einzufügen. Auf dem in dieser Arbeit verwendeten System ist dies allerdings zwingend notwendig, da ansonsten die VM mit der Fehlermeldung `Process received signal 11, suspending` stehen bleibt. Dies ist vermutlich auf einen Fehler in der Implementierung der VM zurückzuführen. Auch wenn sicherlich Systeme existieren (oder existieren werden), die nicht diesen Fehler haben, soll das an dieser Stelle als Hinweis für Programmierer erwähnt sein. Auf dem hier verwendeten System beträgt die benötigte Verzögerung $3ms$.

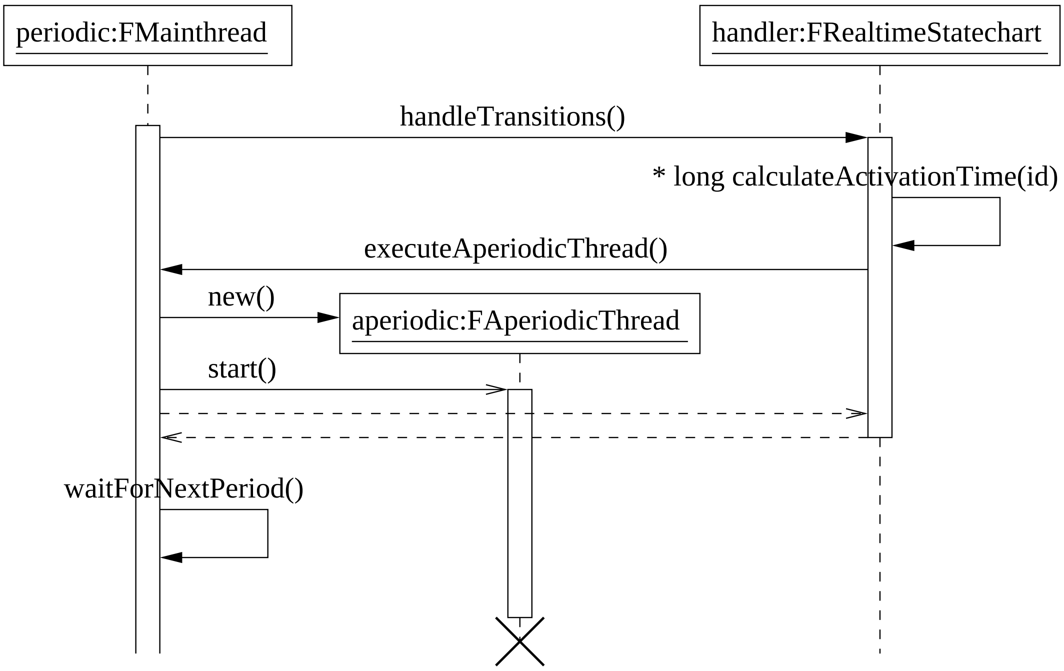

Abbildung 6.15.: Sequenzdiagramm

einer Periode ruft periodic die Methode handleTransitions() auf, in der die Aktivierungszeiten aller Transitionen des aktuellen Zustands berechnet werden. Die zuerst aktivierte Transition wird dann ausgeführt. Dazu wird in der Regel ein aperiodischer Thread vom Hauptthread kreiert und gestartet. An dieser Stelle ist bemerkenswert, dass der `start()`-Aufruf asynchron erfolgt, danach also 2 Threads parallel laufen. Nach Beendigung des Durchlaufs wartet `periodic` auf den Beginn der nächsten Periode. Der aperiodische Thread läuft bis zur Terminierung, woraufhin seine Destruktion von der Speicherverwaltung durchgeführt wird. In Abb. 6.16 ist der Ablauf eines Periodendurchlaufs von periodic im Detail beschrieben.

Solange der Hauptthread nicht angehalten (beendet) wird, läuft er in Form einer Endlosschleife. Zu Beginn einer jeden Ausführung wird die absolute (System-) Zeit bestimmt und festgehalten. Alle Berechnungen und Vergleiche, die im Folgenden Durchlauf ausgeführt werden, beziehen sich auf diesen Zeitpunkt. Das bewirkt, dass ein Durchlauf, der in der Realität eine bestimmte Zeit dauert, scheinbar zu genau diesem Zeitpunkt ausgeführt wurde. Wie in Kapitel 5.3 beschrieben, kann es vorkommen, dass ein Zustandswechsel durchgeführt wird, obwohl der Aktivierungszeitpunkt in der Zukunft liegt. In dem Fall wird der Aktivierungszeitpunkt in der Variablen `locationReached` abgespeichert. Der Automat soll erst nachdem dieser Zeitpunkt verstrichen ist wieder reagieren. Anschließend werden eine Reihe von verschiedenen Fällen unterschieden: Bei der ersten Ausführung nach einem Zustandswechsel muss definitiv die `do()`-Methode ein-

```
while (mainThread.isRunning()) {
  actualTime = FMainThread.getTime();
  if (actualTime >= mainThread.getLocationReached()) {
    //only once in every state
    if (!mainThread.isDoExecutedInCurrentState()) {
      rtsc.doAction();
      actualTime = FMainThread.getTime();
      rtsc.setGuardTime(actualTime);

      anyTransitionExecuted = rtsc.handleTransitions(actualTime);
      if (!anyTransitionExecuted) {
        //only in this case the value is changed to 'true',
        //as in the other case a new state is entered in
        //which do has not been executed, yet.
        mainThread.setDoExecutedInCurrentState(true);
      }
    }
    else
    {
      anyTransitionExecuted = rtsc.handleTransitions(actualTime);
      if (anyTransitionExecuted) {
        //only in this case the value is changed to 'false',
        //as in the other case no new state is entered and
        //the value must still be 'true'.
        mainThread.setDoExecutedInCurrentState(false);
      }
      else
      {
        rtsc.doAction();
        actualTime = FMainThread.getTime();
        rtsc.setGuardTime(actualTime);
      }
    }
  }
  mainThread.waitForNextPeriod();
} //while (mainThread.isRunning()
```

Abbildung 6.16.: Der periodische Hauptthread mal ausgeführt werden und anschließend gegebenenfalls noch der Seiteneffekt der schaltenden Transition. Ansonsten wird entweder `do()` oder der Seiteneffekt ausgeführt (siehe Kapitel 5.3.4). Die Seiteneffektsausführung wird von der Methode `handleTransition(long actualTime)` übernommen, die in der Handlerklasse implementiert und im zugehörigen Interface `FRealtimeState` definiert ist. Diese Methode liefert einen boolschen Wert zurück, der angibt, ob mindestens eine Transition geschaltet, also ein Zustandswechsel stattgefunden hat (Aufgrund von parallelen AND-States können auch mehrere Transitionen geschaltet werden). Eine ausführliche Beschreibung der Methode findet in Kapitel 6.6 statt.

Wie in Kapitel 5.3.1 beschrieben, ist für die Bestimmung des Aktivierungszeitpunktes einer Transition wichtig, ab wann der Zeitguard gilt, also wann die `do()`-Methode beendet wurde. Somit wird der Zeitpunkt nach Ende der Methode mitprotokolliert, was natürlich nur notwendig ist, wenn der Wert des Guards von `false` auf `true` gewechselt ist. Die dafür verwendete Methode `setGuardTime(long time)` ist ebenfalls in der Handlerklasse implementiert.

Da das Schaltverhalten u.a. davon abhängig ist, ob im aktuellen Zustand `do()` bereits ausgeführt wurde, wird dies in dem Attribut `doExecutedInCurrentState` in der Klasse `FMainThread` abgelegt. Wurde im Rahmen der Ausführung von `handleTransitions(long actualTime)` ein Zustandswechsel vorgenommen, was durch die variable `anyTransitionExecuted` angezeigt wird, erhält jenes Attribut den Wert `false`. Fand kein Zustandswechsel statt, wurde `do()` ausgeführt und das Attribut soll ggf. von `false` nach `true` wechseln.

Das Scheduling des Hauptthreads wird durch die Anweisung `waitForNextPeriod()` vorgenommen, womit der Durchlauf beendet ist.

6.6. Aufbau der synthetisierten Handlerklasse

Jede Handlerklasse besteht aus denselben Methoden. Der Inhalt der Methoden ist für jedes Realtime Statechart individuell und spiegelt die Struktur wieder. Der aktuelle Zustand wird durch das Attribut namens `currentState` vom Typ Integer repräsentiert.[6] Weitere Attribute werden für die Realisierung der in Kapitel 6.3 beschriebenen Fallunterscheidung generiert. So werden z.B. für das Realtime Statechart aus Abb. 6.17 die in Abb. 6.18 aufgeführten Konstanten erzeugt.

Dabei entspricht `TRANS0_S1_S2` dem Namen der Transition im Metamodell. Da es im Realtime Statechart nicht notwendig ist, Namen für Transitionen anzugeben,

[6] Da sich das Realtime Statechart aufgrund von Hierarchie und Parallelität in einer Menge von Zuständen aufhalten kann, reicht eine einzige Integervariable nicht aus. Um ein einfacheres Verständnis zu erreichen, werden zunächst Hierarchie und Parallelität nicht berücksichtigt und erst in Kapitel 6.7 beschrieben.

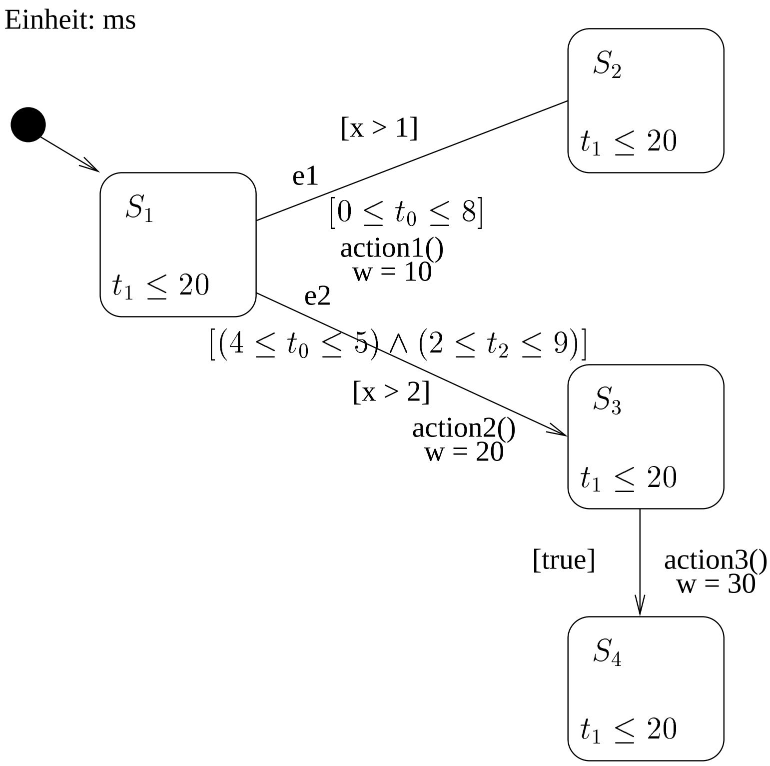

Abbildung 6.17.: Einfaches Realtime Statechart

```
public static final int F_STATE_S1 = 0;
public static final int F_STATE_S2 = 1;
public static final int F_STATE_S3 = 2;
public static final int F_STATE_S4 = 3;
public static final int F_TRANSITION_TRANS0_S1_S2 = 0;
public static final int F_TRANSITION_TRANS1_S1_S3 = 1;
public static final int F_TRANSITION_TRANS2_S3_S4 = 2;
```

Abbildung 6.18.: Konstanten für Zustände und Transitionen wird bei der Erzeugung der Laufzeitdatenstruktur des Metamodells gegebenenfalls einer automatisch generiert. Die Uhren werden ebenfalls durch konstante Attribute repräsentiert (siehe Abb. 6.19).

```
public static final int F_CLOCK_T0 = 0;
public static final int F_CLOCK_T1 = 1;
public static final int F_CLOCK_T2 = 2;
```

Abbildung 6.19.: Konstanten für Uhren

handleTransitions(long actualTime) In jedem Periodendurchlauf des Hauptthreads wird die Methode `handleTransitions(long actualTime)` aufgerufen. Ihr Aufbau ist in Abb. 6.20 dargestellt.

Zu Beginn der Methode wird geprüft, ob die Invariante abgelaufen ist. In dem Fall wird eine Exception erzeugt. Durch Abfangen der Exception kann angemessen auf dieses Ereignis reagiert werden. Ansonsten werden alle aktivierten Transitionen in einen `Vector` namens `enabledTransition` eingefügt. Repräsentiert wird jede Transition von einem Objekt, in dem u.a. die Aktivierungszeit gespeichert ist. Dabei werden nur die Transitionen berücksichtigt, die `currentState` als Quellzustand haben und deren Aktivierungszeitpunkt zwischen den Zeitpunkten $t_{locationReached}$ (abgelegt als Attribut in der Klasse `FMainThread`; siehe auch Kapitel 5.3.1) und `actualTime` liegt. Abb. 6.21 zeigt, dass `actualTime` den Zeitpunkt zu Beginn eines jeden Ausführungszyklus beschreibt.

Durch direktes Einfügen in den `enabledTransition`-Vektor wird sichergestellt, dass die Transitionen nach den Aktivierungszeitpunkten sortiert sind. Wurde keine aktivierte Transition gefunden, kann die Methode beendet werden. Das keine Transition geschaltet hat, wird durch den Rückgabewert `false` angezeigt. Ansonsten wird die am frühsten aktivierte Transition bestimmt und in der Variablen `earliestTransition` abgelegt.

Beim Schalten der Transition muss unterschieden werden, ob der Schaltvorgang innerhalb des Hauptthreads oder als eigener aperiodischer Thread ausgeführt werden soll. Das hängt zum einen davon ab, was die Analysen aus Kapitel 5.4 ergaben, zum anderen, ob im aktuellen Zustand bereits `do()` ausgeführt wurde. Je nachdem was `executeInMainThread` liefert, wird der Seiteneffekt im Hauptthread oder in einem aperiodischen Thread ausgeführt.

addEnabledTransitions(int state, long actualTime) In dieser Methode findet sich ein Teil der Struktur des Realtime Statecharts wieder. So wird für alle Transitionen, die den als Parameter übergebenen Zustand (in der Regel ist das der aktuelle Zustand `currentState`) als Quellzustand haben, die Aktivierungsüberprüfung durchgeführt.

```
public boolean handleTransitions(long actualTime) {
  if (actualTime > getInvariantExpiryTime())
  {
    throw new RuntimeException(„Invariant missed!“);
  }
  else {
    addEnabledTransitions(currentState, actualTime);
    if (enabledTransition.size() == 0) {
      return false;
    }
    earliestTransition = (FRealtimeTransition)
                          enabledTransition.get(0);
    startedTransition.add(earliestTransition);
    if (logEnabled)
    {
      logBuffer.add(earliestTransition.getId() + „ enabled at “
                  + earliestTransition.getActivationStart());
    }
    doExecuted = getFMainThread().isDoExecutedInCurrentState();
    if (((!doExecuted) && executeInMainThread(
                               FMainThread.WITH_DO_ACTION))
        ||
        ((doExecuted) && executeInMainThread(
                               FMainThread.WITHOUT_DO_ACTION)))
    {
      run(earliestTransition.getId());
    }
    else {
      getFMainThread().executeAperiodicThread(
                               earliestTransition.getId());
    }
    anyTransitionExecuted = true;
  }
  return anyTransitionExecuted;
}
```

Abbildung 6.20.: handleTransitions(long actualTime)

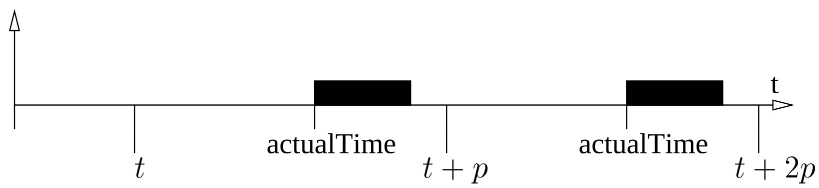

Abbildung 6.21.: Die Variable `actualTime`

Für das Realtime Statechart aus Abb. 6.17 würde sich der Code aus Abb. 6.22 ergeben. Im Zustand S_1 müssen 2 Transitionen überprüft werden, während in S_3 lediglich eine und in S_2 und S_4 keine Transition überprüft werden muss.

```
private void addEnabledTransitions(int state, long actualTime) {
  switch(state){
    case F_STATE_S1:
      calculateActivationTime (
                        F_TRANSITION_TRANS0_S1_S2, actualTime);
      calculateActivationTime (
                        F_TRANSITION_TRANS1_S1_S3, actualTime);
      break;
    case F_STATE_S3:
      calculateActivationTime (
                        F_TRANSITION_TRANS2_S3_S4, actualTime);
      break;
  }
}
```

Abbildung 6.22.: addEnabledTransitions(int state, long actualTime)

calculateActivationTime(int transition, long actualTime) Hier wird geprüft, ob die als Parameter übergebene Transition bis zum Zeitpunkt `actualTime` aktiviert war. Ist dies der Fall, wird der `Vector enabledTransition` um ein Objekt erweitert, welches die Aktivierungsdaten (Integer-ID, Beginn und Ende des Aktivierungsintervalls) enthält (vgl. Abb. 6.23).

Der Hauptteil der Methode besteht aus einer `switch-case` Anweisung, in der sich die in den Zeitguards spezifizierten Konstanten wiederfinden, um die in Kapitel 5.3.1 beschriebenen Werte $tg_{low} := \max_j(a_j + t_j^0)$ und $tg_{up} := \min_j(b_j + t_j^0)$ zu berechnen. Zuvor wird durch den Aufruf `getEventTimeOf(transition)` bestimmt, ob das zu der Transition gehörende Ereignis anliegt und zu welchem Zeitpunkt es auftrat. Liegt es nicht an, wird der Wert der Konstanten `INFINITY` zurückgeliefert. In jedem System sind die darstellbaren Zeitpunkte beschränkt. Der Wert INFINITY ist der größte dieser Zeitpunkte.[7] Ebenso liefert `guardTime[transition] INFINITY`, falls der Guard nicht wahr ist. Dies bewirkt, dass der berechnete Aktivierungszeitpunkt in der Zukunft liegt und die Transition somit nicht als aktiviert erkannt wird. Ergeben alle Werte, dass die Transition schalten kann, wird sie in den `Vector enabledTransition` eingefügt. Die Methode `FRealtimeTransition::fireBefore(FRealtimeTransition transition)` stellt fest, welche der beiden betrachteten Transitionen zuerst schaltet. Dabei wird zuerst

[7] Die Granularität der Uhren ist 1 Millisekunde. Da sie die Uhrzeiten im Datentyp `long` (64 Bit mit Vorzeichen) repräsentiert werden, liegt der Zeitpunkt `INFINITY` ca. 292 278 994 Jahre nach Start des Realtime Statecharts.

```
public void calculateActivationTime(int transition, long actualTime) {
  lowerBound = 0;
  upperBound = INFINITY;
  eventTime = getEventTimeOf(transition);
  guardTime = guardTime[transition];
  long0 = getFMainThread().getLocationReached();
  if ( guardTime > long0 ) {
    long0 = guardTime;
  }
  if ( eventTime > long0 )
  {
    long0 = eventTime;
  }
  if ( long0 > actualTime ) {
    switch(transition){
      case F_TRANSITION_TRANS0_StartState_S1: break;
      case F_TRANSITION_TRANS1_S1_S2:
        //lowerBound = max_j (a_j + t_0_j);
        lowerBound = 0 + t0[F_CLOCK_0];
        //upperBound = min_j (b_j + t_0_j);
        upperBound = 8 + t0[F_CLOCK_0];
      break;
      case F_TRANSITION_TRANS3_S3_S4:
      ...
    }
    if ( lowerBound > long0 ) {
      long0 = lowerBound;
    }
    if ( (long0 <= upperBound) && (long0 <= actualTime) ) {
      realtimeTransition = FRealtimeTransition.newInstance(transition,
                                  activationTime, upperBound,
                                  concurrencySlot);
      int i;
      for ( i = 0;i < enabledTransition.size();i++ ) {
        if (realtimeTransition.fireBefore((FRealtimeTransition)
                                  enabledTransition.get(i)))
        {
          enabledTransition.insertElementAt(realtimeTransition, i);
        }
      }
      if ( i == enabledTransition.size() ) {
        enabledTransition.add(realtimeTransition);
      }
   }
  } // if (long0 > actualTime)
}
```

Abbildung 6.23.: calculateActivationTime(int transition, long actualTime) der Aktivierungszeitpunkt, bei dessen Gleichheit die Priorität betrachtet. Durch das direkte Einfügen ist sichergestellt, dass der Vektor immer in der Schaltreihenfolge sortiert ist.

getInvariantExpiryTime() Die Formel zur Bestimmung des Zeitpunktes, zu dem die Zeitinvariante des aktuellen Zustands abläuft, ähnelt der zur Bestimmung des Aktivierungszeitpunktes einer Transition. Daher wurde diese Methode nach dem gleichen Prinzip wie `calculateActivationTime(int transition, long actualTime)` erstellt.

executeInMainThread(int transition, boolean withDoAction) In Kapitel 5.3.1 ist beschrieben, dass einige Aktionen im Hauptthread ausgeführt werden können, und diejenigen, die dafür zu lang sind, in einen aperiodischen Thread ausgelagert werden. Des Weiteren hängt die Entscheidung davon ab, ob bereits Zeit für die Ausführung der `do()`-Methode verstrichen ist. Die Methode `executeInMainThread(int transition, boolean withDoAction)` dient sozusagen als "Look-Up-Table" für diese Entscheidung. Je nach Eigenschaften des Zielsystems, der Seiteneffekte und der übrigen Aktionen, könnte die Methode für das RealtimeStatechart aus Abb. 6.17 so aussehen, wie in Abb. .6.24 dargestellt.

```
public boolean executeInMainThread(int transition, boolean withDoAction){
  switch(transition){
    case F_TRANSITION_TRANS1_S1_S2:
      if (withDoAction == FMainThread.WITH_DO_ACTION) {
        return false;
      }
      else {
        return true;
      }
    break;
    case F_TRANSITION_TRANS3_S3_S4:
      if (withDoAction == FMainThread.WITH_DO_ACTION) {
        return false;
      }
      else {
        return true;
      }
    break;
    ...
  }
  return false;
}
```

Abbildung 6.24.: executeInMainThread(int transition, boolean withDoAction)

Der Vorteil dabei ist, dass die Berechnungen, die einen verhältnismäßig hohen Aufwand haben, bereits zum Zeitpunkt der Codegenerierung, also vor der Laufzeit des Realtime Statecharts, durchgeführt werden können.

Evaluierung Bevor die zuerst aktivierte Transition (`earliestTransition` in Abb. 6.20) schaltet, wird in einen nicht-blockierenden Puffer (`logBuffer`) eine Nachricht über die Aktivierung der Transition eingetragen. Zusätzlich existiert ein niedrig-priorisierter nicht-Realtime Thread, der diesen Puffer ausliest und die Nachrichten in eine Log-Datei schreibt. Zusätzlich werden zu Beginn und Ende der Ausführung des Seiteneffektes die Nachrichten „`[id] started/ended at [time]`“ in den Log-Puffer eingefügt. Diese Evaluierungsfunktionalität kann über das Flag `logEnabled` aktiviert und deaktiviert werden.

6.7. Erweiterung um Hierarchie und Parallelität

Aufgrund von Hierarchie und Parallelität kann sich ein Realtime Statechart nicht nur in einem, sondern in einer Menge von Zuständen befinden, die in Form eines Baumes, der Hierarchie und Parallelität wiederspiegelt, angeordnet werden können (siehe Kapitel 5.3.3). In [Köh00] wird vorgeschlagen, das Integer-Attribut `currentState` zu einem Integer-Array `currentState[MAX_DEPTH]` zu erweitern, bei dem in jedem Eintrag eine Menge von Zuständen abgelegt werden kann. Dann werden in `currentState[0]` die hierarchisch höchsten Zustände abgelegt, in `currentState[1]` die Subzustände der Einträge aus `currentState[0]` usw. Bei solch einer Darstellung der aktuellen Zustände ginge allerdings die Baumstruktur verloren, weshalb die Zustände in einer dynamischen Tabelle[8] abgelegt werden.

Der Zustandsbaum aus Abb. 5.6 a) würde dann wie folgt repräsentiert werden:

S1	S1	S1	S1
S2	S4	S4	S11
	S5	S7	

Jede Spalte stellt dann einen Parallelitätsschacht dar. Verringert sich der Grad der Parallelität (z.B. durch Schalten der Transition T_6) verringert sich auch die Anzahl der Spalten in dieser `currentState`-Tabelle. Die maximale Breite der Tabelle

[8]Eine Tabelle wurde in Java Code umgesetzt, indem jede Spalte durch eine Instanz der Klasse `IntVector` repräsentiert wird. `IntVector` hat dieselben Eigenschaften und Methoden wie `java.util.Vector`, nimmt aber anstatt von Objekten nur `int`-Werte auf. Die einzelnen `IntVector`-Objekte werden in einer Instanz der Klasse `Vector` zur Tabelle zusammengefasst. `Vector` hat dieselben Eigenschaften wie `java.util.Vector`, wurde jedoch neu implementiert, um Aussagen bezüglich Aufwand und Speicherverbrauch der einzelnen Methoden treffen zu können. Die Struktur hat den Vorteil, dass die Tabelle eine dynamische Größe hat und auch Spalten unterschiedlicher Länge möglich sind.

(also die größtmögliche Anzahl an Parallelitätsschächten) ist `MAX_CONCURRENCY`. Die maximale Länge ergibt sich aus der maximalen Tiefe `MAX_DEPTH` innerhalb hierarchischer Zustände. Eine (pessimistische) Abschätzung für die maximale Anzahl an Einträgen der Tabelle ist somit `MAX_CONCURRENCY * MAX_DEPTH`.

Die `startedTransition-Menge` wird als `Vector` umgesetzt und die `waiting-Transition`-Tabelle ebenso wie die `currentState`-Tabelle (vgl. Kapitel 5.3.3).

Im Folgenden sei noch beschrieben, wie der synthetisierte Code für die beiden Varianten der Methode `isExecutable`, die regelt, welche Transitionen parallel zueinander ausgeführt werden dürfen, aussieht.

isExecutable(FRealtimeTransition transition) isExecutable prüft, ob es einen Eintrag im oben beschriebenen `startedTransition`-Vektor gibt, der ein Schalten von `transition` ausschließt. Dazu wird eine weitere Hilfsfunktion (`mutalExclusive(int transition1, int transition2)`) verwendet.

isExecutable(FRealtimeTransition transition1, FRealtimeTransition transition2) Werden bei einer Synchronisation 2 Transitionen gestartet, müssen diese Transition nicht nur mit den bereits gestarteten Transitionen auf gegenseitgen Ausschluss geprüft werden, sondern ebenfalls mit sich selbst. Diese Variante der Methode verwendet dieselbe Hilfsfunktion `mutalExclusive`.

mutalExclusive(int transition1, int transition2) Ähnlich wie die Methode `executeInMainThread(int transition, boolean withDoAction)` wie ein "Look-Up-Table" für die Entscheidung, ob eine Transition in einen aperiodischen Thread ausgelagert werden muss oder nicht, fungiert, dient diese Methode als "Look-Up-Table" um einen gegenseitigen Ausschluss festzustellen. Ein Ausschnitt dieser Methode für das Realtime Statechart aus Abb. 5.5 ist in Abb. 6.25 dargestellt.

Auch hier findet die Bestimmung welche Transitionen sich gegenseitig Ausschließen vor der Laufzeit des Realtime Statecharts statt. Dabei werden die Überlegungen aus Kapitel 5.3.3 berücksichtigt.

6.8. Worst Case Execution Time

Zu allen in einem Realtime Statechart vorkommenden Aktionen (Seiteneffekte, `entry`()-, `exit`()- und `do`()- sowie Monitor-Operationen) muss eine Worst Case Execution Time angegeben werden. Da Wcets immer vom unterliegenden System abhängig sind, ändern sich diese, wenn das Realtime Statechart von einem System

```
public boolean mutalExclusive(int transition1, int transition2) {
  switch(transition1) {
    case F_TRANSITION_T1:
      switch(transition2) {
        case FTRANSITION_T4:
        case FTRANSITION_T5:
        case FTRANSITION_T8:
          return (true) ;
      }
    break;
    case F_TRANSITION_T3:
      switch(transition2) {
        case FTRANSITION_T4:
        case FTRANSITION_T5:
        case FTRANSITION_T6:
        case FTRANSITION_T8:
          return (true) ;
      }
    break;
    ...
    return false;
  }
}
```

Abbildung 6.25.: mutalExclusive(int transition1, int transition2)

```xml
<wcet>
  <action id="RealtimeStatechartentryState1"
     wcet="7" unit="ms" />
  <action id="RealtimeStatechartexitState1"
     wcet="10" unit="ms" />
  <action id="RealtimeStatechartdoState1"
     wcet="3" unit="ms" />
  <action id="RealtimeStatechartactionTrans1"
     wcet="40" unit="ms" />
  <action id="RealtimeStatechartactionTrans2"
     wcet="23" unit="ms" />
  <systemconstant name="INTEGER_ASSIGNMENT"
     wcet="1" unit="ms"/>
  <systemconstant name="LONG_INTEGER_ASSIGNMENT"
     wcet="1" unit="ms"/>
  <systemconstant name="INTEGER_ADDITION"
     wcet="2" unit="ms"/>
  <systemconstant name="INTEGER_COMPARISON"
     wcet="1" unit="ms"/>
  <systemconstant name="GET_METHOD_CALL"
     wcet="2" unit="ms"/>
  <systemconstant name="SET_OR_ADD_METHOD_CALL"
     wcet="3" unit="ms"/>
  <systemconstant name="OBJECT_ASSIGNMENT"
     wcet="3" unit="ms"/>
  <systemconstant name="TYPE_CAST"
     wcet="1" unit="ms"/>
  <systemconstant name="START_APERIODIC_THREAD"
     wcet="6" unit="ms"/>
  <systemconstant name="END_APERIODIC_THREAD"
     wcet="5" unit="ms"/>
  <systemconstant name="SLEEP_AFTER_THREAD_START"
     sleep="3" unit="ms"/>
</wcet>
```

Abbildung 6.26.: wcet.xml

auf ein Anderes portiert wird. Eine Verringerung der Wcets hat keinen negativen Einfluss auf die Schedulability. Eine Vergrößerung der Wcets bewirkt jedoch, dass das Realtime Statechart eine andere Periode erhält, was wiederum dazu führt, dass die Einträge im "Look-up Table" der Methode `executeInMainThread(int transition, boolean withDoAction)` nicht mehr stimmen. Außerdem kann es erneut zu Zeitinkonsistenzen kommen. Deshalb muss nach Anpassen der Wcets erneut der Prozess der Codegenerierung durchlaufen werden. Um dem Anwender die Anpassung zu erleichtern, kann er zu jedem Realtime Statechart ein XML-Dokument angeben, in dem alle Wcets aufgeführt sind. Dieses Dokument hat die in Abb. 6.26 dargestellte Form. Dabei entsprechen die `id`-Einträge den Namen der zum Realtime Statechart gehörigen Klasse (ggf. in Verbindung mit dem Packagenamen) zuzüglich der Namen, die zu jeder `UMLRealtimeAction`-Instanz angegeben werden müssen. Portiert der Anwender nun das Realtime Statechart, braucht er lediglich die Einträge im XML-Dokument zu ändern und ggf. erneut die Codegenerierung zu starten. Die Einbindung des externen Dokumentes ermöglicht es auch, dass z.B. ein Tool zur automatischen Bestimmung der Wcets (durch Bytecode-Analyse / Tests) eingesetzt werden kann und die Ergebnisse in dieser Form abspeichert.

Neben den Wcets für die Aktionen sind auch die Maximalen Ausführungszeiten für die zur Verwaltung des Realtime Statecharts eingesetzten Java-Operationen, wie z.B. Vergleich von Integer-Werten, Zuweisungen, Starten von aperiodischen Threads etc. von Bedeutung. Diese Systemkonstanten bestimmen die Wcets für die Auswertung der aktivierten Transitionen (siehe Kapitel 5.3.5). Die Werte können ebenfalls in dem XML-Dokument aufgeführt werden. Die Systemkonstante `SLEEP_AFTER_THREAD_START` stellt einen Sonderfall dar, da sie keine Wcet angibt. Sie gibt die Zeit an, die der Hauptthread nach dem Starten eines aperiodischen Threads warten soll, bevor er fortfährt (siehe Kapitel 6.4).

Eine Analyse des synthetisierten Codes zur Überprüfung welche Transitionen aktiviert sind, ergibt für die Worst Case Execution Time eine Obergrenze, die von verschiedenen Zustandsabhängigen Konstanten bestimmt ist: Somit bezeichnet P die maximale Parallelität, die innerhalb des betrachteten Zustands vorkommen kann, D die maximale Hierarchie-Tiefe innerhalb des Zustands, T die maximale Anzahl an Transitionen, die *gleichzeitig* aktiviert sein können und C die Anzahl der Uhren, die innerhalb des Zustands bzw. an wegführenden Transitionen auftreten. Seien die Systemkonstanten -wie in Kapitel 5.3.5 beschrieben- in einem Array namens `system` abgelegt, ergibt sich (für einen Zustand ohne Subzustände, siehe 5.3.5) folgende Abschätzung:

$$\begin{aligned} w^S_{activationProof} = {} & (3T + P + 1)\, system\,[\mathsf{INTEGER\_ASSIGNMENT}] \\ & + ((C + 4)\, T + CT)\, system\,[\mathsf{LONG\_INTEGER\_ASSIGNMENT}] \\ & + (T^2 + 2PD + P)\, system\,[\mathsf{INTEGER\_ADDITION}] \\ & + (2T^2 + (C + 6)\, T + PD + 2P)\, system\,[\mathsf{INTEGER\_COMPARISON}] \end{aligned}$$

$$
\begin{aligned}
&+ (T^2 + 7T + 5C)\, system\, [\mathsf{GET\_METHOD\_CALL}] \\
&+ (T + P)\, system\, [\mathsf{SET\_OR\_ADD\_METHOD\_CALL}] \\
&+ (3T + P)\, system\, [\mathrm{OBJECT}\_\mathsf{ASSIGNMENT}] \\
&+ (2T^2 + T + P)\, system\, [\mathsf{TYPE\_CAST}]
\end{aligned}
$$

Eine ähnliche Betrachtung kann für die Auswahl der zu schaltenden Transitionen getroffen werden, die dann liefert:

$$
\begin{aligned}
w^S_{guard} = w^S_{activationProof} \\
&+ (4T + P + 4)\, system\, [\mathsf{INTEGER\_ASSIGNMENT}] \\
&+ (0)\, system\, [\mathsf{LONG\_INTEGER\_ASSIGNMENT}] \\
&+ (T^2 + 3T + P)\, system\, [\mathsf{INTEGER\_ADDITION}] \\
&+ (T^2 + 6T + 6P - 1)\, system\, [\mathsf{INTEGER\_COMPARISON}] \\
&+ (4T + 11P - 3)\, system\, [\mathsf{GET\_METHOD\_CALL}] \\
&+ (4T)\, system\, [\mathsf{SET\_OR\_ADD\_METHOD\_CALL}] \\
&+ (T^2 + T + 2P + 1)\, system\, [\mathrm{OBJECT}\_\mathsf{ASSIGNMENT}] \\
&+ (2T + 3P)\, system\, [\mathsf{TYPE\_CAST}] \\
&+ (T)\, system\, [\mathsf{START\_APERIODIC\_THREAD}]
\end{aligned}
$$

6.9. Behandlung von Zeitinkonsistenzen

Um die in Kapitel 4.3.1 beschriebenen Zeitinkonsistenzen zu erkennen und sie gegebenenfalls wie in Kapitel 4.3.2 automatisch zu beheben, wurden die Klassen `UMLRealtimeStatechart`, `UMLRealtimeTransition` und `UMLComplexRealtimeState` des Metamodells (siehe Kapitel 6.2) um verschiedene Attribute und Methoden erweitert. Die "Hauptmethode", die einen vollständigen Check auf Inkonsistenzen durchführt, ist `UMLRealtimeStatechart::handleTimeInconsistencies()`. In ihr wird auf jedem `UMLComplexRealtimeState`, der –auch unter Berücksichtigung von Hierarchie und Parallelität– zu dem Realtime Statechart gehört, `hierarchicalInvariantCheck()` sowie `handleWeakInvariants()` aufgerufen. Schwache Zeitguards werden in der Methode `UMLRealtimeTransition::handleWeakTimeguards()` behandelt. Der Aufbau dieser Methoden ähnelt sich sehr. In 6.27 ist ein (vereinfachter) Auszug aus `handleWeakTimeguards()` gezeigt.

Zu Beginn wird ein `Iterator` über alle existierenden Uhren erzeugt, da die Überprüfungen für jede Uhr einmal durchgeführt werden muss. Innerhalb der folgenden `while`-Schleife wird aus der Datenstruktur des Metamodells zuerst der zur Uhr gehörige Zeitguard und –über die `UMLRealtimeAction`– die Deadline des Seiteneffektes der Transition geholt. Liegen diese dann in Form von `UMLTimeguardAtom`- und `UMLDeadlineAtom`-Objekten vor, können die Vergleiche und Berechnungen aus den Kapiteln 4.3.1 und 4.3.2 durchgeführt werden. Kann ein aufgedeckter

```
Iterator iter = sourceState.getRealtimeStatechart().iteratorOfAllUMLClocks();
while (iter.hasNext()) {
  UMLClock clock = (UMLClock) iter.next();
  UMLTimeguardAtom timeguardAtom = getFromUMLTimeguardAtom(clock);
  UMLRealtimeAction action = getUMLRealtimeAction();
  UMLDeadlineAtom deadlineAtom = null;
  long wcet = 0;
  if (action != null) {
    deadlineAtom = action.getFromUMLDeadlineAtom(clock);
    wcet = action.getWcet();
  }
  ...
}
```

Abbildung 6.27.: handleWeakTimeguards()

Konflikt automatisch behoben werden, wird über die Methode `isFinal()`, die in jedem `UMLBoolRealtimeAtom` vorhanden ist, geprüft, ob der Benutzer eine automatische Konfliktauflösung erlaubt oder nicht.

Zur weiteren Konfliktbehandlung existieren die Methoden:

- `UMLRealtimeTransition::getLiveCapability()`, in der eine Überprüfung auf tote Transitionen durchgeführt wird und die eines der Attribute `ALIVE`, `DEAD` oder `INSECURE` zurückliefert.
- `UMLComplexRealtimeState:getLiveCapability()`, die mit Hilfe der gleichnamigen Methode aus `UMLRealtimeTransition` feststellt, ob ein Time Deadlock vorliegt.
- `UMLRealtimeStatechart::getTimeConformity()`, die eines der Attribute `CONFORM`, `INCONSISTENT` oder `INSECURE` zurückliefert und dabei obige Methoden verwendet.

Über das Attribut `securityLevel` kann der Anwender angeben, ob er mit einer unsicheren "Zeitkonformität" zufrieden ist, oder ob er ein zeitkonformes Modell fordert.

6.10. Scheduling mehrerer Realtime Statecharts

In Kapitel 5.7 wurde beschrieben, dass zu jedem Statechart ein Dokument generiert wird, in dem alle zu einer Schedulinganalyse notwendigen Daten aufgeführt sind. Wie auch das Dokument, in dem die Wcets aufgeführt sind, wird auch in diesem *Scheduling Document* als Dateiformat XML eingesetzt. Die Datei hätte dann für das Realtime Statechart aus Abb. 5.5 die in Abb. 6.28 dargestellte Struktur.

```
<realtimestatechart>
  <mainthread period=”4” wcet=”3” minet=”1” blocking=“0“ />
  <concurrency>
    <aperiodicthread name=”T1” deadline=”9” wcet=”4” blocking=“1” />
    <aperiodicthread name=”T2” deadline=”44” wcet=”18” blocking=“1” />
    <aperiodicthread name=”T3” deadline=”100” wcet=”45” blocking=“0” />
    <aperiodicthread name=”T7” deadline=”120” wcet=”87” blocking=“0” />
  </concurrency>
  <concurrency>
    <aperiodicthread name=”T1” deadline=”9” wcet=”3” blocking=“0” />
    <aperiodicthread name=”T6” deadline=”10” wcet=”4” blocking=“0” />
    <aperiodicthread name=”T7” deadline=”10” wcet=”4” blocking=“0” />
  </concurrency>
  <concurrency>
    <aperiodicthread name=”T4” deadline=”90” wcet=”11” blocking=“0” />
  </concurrency>
  <concurrency>
    <aperiodicthread name=”T5” deadline=”40” wcet=”33” blocking=“0“ />
  </concurrency>
  <concurrency>
    <aperiodicthread name=”T8” deadline=”9” wcet=”5” blocking=“0” />
  </concurrency>
</realtimestatechart>
```

Abbildung 6.28.: schedule.xml

In den Attributen des Tags `mainthread` sind die Daten des periodischen Hauptthreads aufgeführt. Die *Minimum Execution Time (minet)* gibt an, wie viel Zeit er maximal benötigt, wenn er nur die Aktivierungsüberprüfung durchführt und aperiodische Threads startet, aber keine Seiteneffekte selbst ausführt. Innerhalb der `concurrency`-Tags sind die Threads aufgeführt, die zueinander parallel ausgeführt werden können. So repräsentieren die 4 Einträge des ersten Tags die Transitionen T_1, T_2, T_3 und T_7, die des nächsten Tags T_1, T_6 und T_7, während die übrigen für T_4, T_5 und T_8 stehen. Die Anzahl der `concurrency` Tags steigt exponentiell mit dem Grad der Parallelität, da jedes einzelne Tag eine Kombinationsmöglichkeit von möglichen parallel auszuführenden Threads wiederspiegelt. Aufgrund der Gesetze der Kombinatorik kann dieser exponentielle Aufwand nicht verringert werden.

Um eine Schedulinganalyse durchzuführen, müssen zunächst von jedem Realtime Statechart die Scheduling-Dokumente geladen werden. Dies wird von der Klasse `SchedulingAnalyser` vorgenommen. Nachdem dort alle Scheduling-Dokumente angemeldet wurden, werden die Prioritäten nach dem Rate Monotonic Ansatz (siehe 2.1.2 und 5.7) vergeben. Anschließend wird durch den Aufruf `analyze() throws SchedulingException` eine Antwortzeitanalyse (siehe Kapitel 2.1.4) durchgeführt. Können die Realtime Statecharts geschedult werden, wird die Klasse namens `Main` zur Vergabe der Prioritäten und zum Starten der Realtime Statecharts erzeugt (siehe Kapitel 5.7), um daraufhin `analyze()` mit dem Rückgabewert `true` zu beenden. Ansonsten wird eine `SchedulingException` geworfen, in deren `message`-Attribut die Menge von Threads aufgeführt ist, die diese `Exception` verursacht haben.

6.11. Zusammenfassung

In diesem Kapitel wurde gezeigt, wie ein Realtime Statechart in Realtime Java implementiert wird, und wie das Modell der Realtime Statecharts in Fujaba integriert wurde. Bei der Implementierung galt es zum einen die Möglichkeiten von Realtime Java –wie Speicherverwaltung und (No Heap) Realtime Threads– umzusetzen, zum anderen musste eins der bekannten Muster zur Implementierung von Statecharts ausgewählt und auf Realtime Statecharts angepasst werden. Zusätzlich mussten Methoden zur Behandlung von Zeitinkonsistenzen und für die Schedulinganalyse implementiert werden.

7. Zusammenfassung und Ausblick

Ein erstes Teilziel dieser Arbeit war die Spezifikation von Echtzeitverhalten durch Erweiterung von UML-Diagrammen zu ermöglichen. Dazu wurde zunächst erarbeitet, durch welche Attribute (Deadlines, etc.) Echtzeitsysteme beschrieben werden. Anschließend wurden verschiedene existierende Diagramme, die zur Verhaltensmodellierung eingesetzt werden, daraufhin betrachtet, wie diese Attribute in den bekannten Modellen umgesetzt werden können. Da keine zufriedenstellenden Modelle existieren, mit denen sowohl komplexes Verhalten als auch Echtzeitanforderungen spezifiziert werden können, wurde ein neues Automatenmodell, das der Realtime Statecharts, kreiert. Das Realtime Statechart kombiniert dabei die Vorteile anderer Modelle, um komplexes zeitbasiertes Verhalten modellieren zu können.

Bei der Beschreibung von Zeitverhalten kann immer eine ungünstige oder vom Anwender undurchdachte Kombination von Zeitschranken ein nicht realisierbares Verhalten hervorrufen. Dies macht eine an den Modellierungsprozess anschließende Überprüfung auf Inkonsistenzen nötig, die gegebenenfalls eine automatische oder manuelle Anpassung der Zeitkonstanten zur Folge hat.

Die Definition des Realtime Statechart Modells wurde im Hinblick auf das Gesamtziel, Code für Echtzeitsysteme zu generieren, vorgenommen. Deshalb wird in den Diagrammen –neben den Informationen zur Verhaltensspezifikation– auch die Angabe von Annotationen, die für die Beschreibung realer physikalischer Systeme notwendig sind (wie z.B. Wcets), gefordert. Dennoch wurde das Modell so flexibel gehalten, dass es um High-Level Konstrukte erweitert werden kann.

Bei der Umsetzung des Modells werden dann alle Operationen periodischen und aperiodischen Threads zugeordnet. Bei dieser Abbildung wurde darauf geachtet, dass sie unabhängig von der für die Implementierung verwendeten Programmiersprache ist. Die Konzepte der Abbildung können in jeder Sprache, die Multitasking erlaubt, umgesetzt werden und wurden am Beispiel von Realtime Java implementiert. Dadurch wird eine hohe Portabilität aufgrund der von Java bekannten Plattformunabhängigkeit geboten.

Während der Codesynthese wird ein zusätzliches Dokument, in dem die Daten aller partitionierten Threads aufgelistet sind, generiert. Entschließt sich der Anwender verschiedene Realtime Statecharts parallel auf demselben System laufen zu lassen, kann er mit Hilfe dieser Dokumente eine Schedulinganalyse durchführen. Die einzelnen Dateien werden von einem Analysator geladen, der aufgrund der Daten, die für das Scheduling notwendigen Prioritäten bestimmt und eine Aussage bezüglich der Schedulability trifft. Ist diese negativ, muss an den Modellen "nachmodelliert" werden, andernfalls weist er den einzelnen Operatonen der einzelnen Realtime Statecharts die Prioritäten zu und erzeugt eine weitere Klasse zum Start der synthetisierten Threads. Alle beschriebenen Funktionalitäten wurden in die Entwicklungsumgebung Fujaba integriert.

Ausblick

Da in Echtzeitsystemen, die aus einer Vielzahl von Threads bestehen, häufig Preemption Threshold Scheduler anstelle von Priority Schedulern eingesetzt werden, ist eine Erweiterung der Konzepte zur Implementierung um dieses Schedulingverfahren eine weitere Aufgabe. Außerdem ist es sinnvoll, die Umsetzung der Konzepte auch für andere Sprachen –wie C/C++ oder Ada– vorzunehmen und ein Wcet-Analyse Tool zu integrieren. Für den Bereich der Inkonsistenzprüfung könnte ein Modelchecker eingesetzt werden, um für unsicher eingestufte Transitionen zu entscheiden, ob sie sicher oder inkonsistent sind.

Abbildungsverzeichnis

Literaturverzeichnis

[aJi] aJile Systems Home Page: http://www.ajile.com/.

[BBF$^+$00] Greg Bollella, Ben Brosgol, Steve Furr, Savid Hardin, Peter Dibble, James Gosling, and Mark Turnbull. *The Real-Time Specification for Java^{TM}*. Addison-Wesley, 2000.

[BBW00] Guillem Bernat, Alan Burns, and Andy Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (Euromicro-RTS 2000)*, 2000.

[BR99] Loic P. Briand and Daniel M. Roy. *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*. IEEE Press, 1999.

[But97] Giorgio C. Buttazzo. *Hard real time computing systems : predictable scheduling algorithms and applications*. Kluwer international series in engineering and computer science : Real-time systems. Kluwer Academic Publishers, 1997.

[BW01] Alan Burns and Andy Wellings. *Real-Timer Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*. Addison-Wesley, 3rd edition, 2001.

[Dib02] Peter C. Dibble. *Real-Time Java Platform Programming*. Prentice Hall, March 2002.

[DM01] A. David and M.O. Moeller. From HUPPAAL to UPPAAL: A translation from hierarchical timed automata to flat timed automata. In *TechReport BRICS RS-01-11, Department of Computer Science, University of Aarhus*, 2001.

[Dou99] Bruce Powel Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Addison-Wesley, May 1999.

[Erp00] Edwin Erpenbach. *Compilation, Worst-Case Execution Times and Scheduability Analysis of Statechart Models.* Ph.D.-thesis, University of Paderborn, Department of Mathematics and Computer Science, February 2000.

[Fuj] Fujaba Home Page: http://www.fujaba.de/.

[GB] H. Giese and S. Burmester. Extended Hierarchical Timed Automata. *TechReport, Universität Paderborn, geplante Veröffentlichung: Ende Oktober 2002.*

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Har87] D. Harel. STATECHARTS: A Visual Formalism for complex systems. *Science of Computer Programming*, 3(8):231–274, 1987.

[Har98] D. Harel. *Modeling reactive systems with statecharts.* McGraw-Hill, 1998.

[HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. In *Proc. of IEEE Symposium on Logic in Computer Science*. IEEE Press, 1992.

[Jos96] Mathai Joseph. *Real time systems : specification verification and analysis.* Prentice Hall international series in computer science. Prentice Hall, 1996.

[Kas] Uwe Kastens. Parallele Programmierung in Java: http://www.upb.de/fachbereich/ag/agkastens/ppjava/.

[KHK02] Saehwa Kim, Seongsoo Hong, and Tae-Hyung Kim. Perfecting preemption threshold scheduling for object-oriented real-time system design: from the perspective of real-time synchronization. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 223–232. ACM Press, 2002.

[Köh00] H.J. Köhler. Codegenerierung für UML-Collaborations-, -Sequenz- und -Statechart-Diagramme. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, 2000.

[KP92] Y. Kesten and A. Pnueli. Timed and Hybrid Statecharts and their Textual Representation. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[LPY97] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Springer International Journal of Software Tools for Technology*, 1(1), 1997.

[LY97] T. Lindholm and F. Yellin. *Java^{TM}: Die Spezifikation der virtuellen Maschine*. Addison-Wesley, 1997.

[Obj01] Object Management Group. *OMG Unified Modeling Language Specification, Version 1.4*, September 2001. OMG document ad/01-09-67.

[OTI] O T I Home Page: http://www.oti.com/.

[PAS94] A. Peron and A.Maggiolo-Schettini. Transitions as Interrupts: A New Semantics for Timed Statecharts. In *Proceedings of TACS '94*, volume 789 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[QNX] Q N X Home Page: http://get.qnx.com/.

[Ram] F. J. Rammig. Grundlagen der Technischen Informatik: http://www.upb.de/cs/ag-rammig/www/courses/gti/.

[RSRS99] B. Rumpe, M. Schoenmakers, A. Radermacher, and A. Schürr. UML + ROOM as a Standard ADL? In F. Titsworth, editor, *Engineering of Complex Computer Systems, ICECCS'99 Proceedings*. IEEE Computer Society, 1999.

[RTJ] Realtime for Java^{TM} Expert Group: http://www.rtj.org.

[SKW00] M. Saksena, P. Karvelas, and Y. Wang. Automatic Synthesis of Multi-Tasking Implementations from Real-Time Object-Oriented Models. In *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Newport Beach, California, 2000.

[SR98] Bran Selic and Jim Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Techreport, ObjectTime Limited, 1998.

[Sun] Java Home Page: http://java.sun.com/.

[Tim] Timesys Home Page: http://www.timesys.com/.

[Zün01] A. Zündorf. *Rigorous Object Oriented Software Development*. Habilitation-Thesis, University of Paderborn, 2001.

A. Notation

Formel- und Diagrammzeichen Wie in Kapitel 2.1.1 beschrieben, wird das zeitbehaftete Verhalten eines Threads durch eine Menge von Parametern beschrieben. Da in dieser Arbeit die Parameter in verschiedenen mathematischen Formeln und Beziehungen auftauchen, sei in der folgenden Tabelle ein kurzer Überblick gegeben, nach welchem Schema die Bezeichner vergeben werden.

Parameter	Bezeichner
Thread	τ
Worst Case Execution Time (Wcet)	w
Startzeit	s
Deadline	d
Periode	p
Priorität	δ
Minimum Interarrivaltime	m
Antwortzeit	r

Diagramme setzen sich ebenfalls aus verschiedenen Komponenten zusammen, die aus Gründen der Übersichtlichkeit abgekürzt bzw. besonders gekennzeichnet werden (die Begriffe werden in Kapitel 4 an geeigneter Stelle ausführlich erklärt). In der folgenden Tabelle sind diese Komponenten und alle übrigen Abkürzungen aufgelistet.

Komponente	Kennzeichnung
Zustand / Location	S
Ereignis	e
Guard / Zeitguard	[...]
Uhr	t
Aktion / Seiteneffekt	...()
Auslastungsfaktor	ν

Boolsche Algebra In der boolschen Algebra sind eine Vielzahl von unterschiedlichen Schreibweisen verbreitet. Um eine logische 'UND'-Beziehung zwischen den boolschen Ausdrücken a und b darzustellen, werden in dieser Arbeit folgende Schreibweisen verwendet: $a \wedge b$ bzw. $a + b$.

Ein logisches 'ODER' wird durch $a \vee b$ bzw. $a * b$ oder ab dargestellt. Insbesondere bietet die Verwendung, der aus der Arithmetik bekannten Multiplikations- und Additionsschreibweise, die Möglichkeit längere Ausdrücke abzukürzen. So kann z.B. ein Ausdruck der Form $a_1 \wedge a_2 \wedge \ldots \wedge a_n$ durch $\prod_{j=1}^{n} a_j$ abgekürzt werden.

B. Periodenbestimmung

In Kapitel 5.4 wurde beschrieben, welche Faktoren bei der Periodenbestimmung für den Hauptthread berücksichtigt werden müssen. Dabei wird zwischen 4 Fällen unterschieden. Aufgrund der Komplexität des zweiten und vierten Falls wurde auf eine mathematische Betrachtung in Kapitel 5.4 verzichtet. Diese wird hier nachgeliefert. Für alle 4 Fälle gilt die Vorbedingung $w_{do()} + w_{guard} + w_{startA} \leq \nu p$. Im Übrigen unterscheiden sich die 4 Fälle wie folgt:

1. $w_{do()} + w_{guard} + w_{a()} \leq \nu p$. Alle Teile passen in eine Periode des Hauptthreads.

2. $w_{do()} + w_{guard} + w_{a()} > \nu p$. Zumindest bei der ersten Ausführung nach einem Locationwechsel muss `a()` in einen aperiodischen Thread ausgelagert werden.

3. $w_{guard} + w_{a()} \leq \nu p$. Wenn `do()` nicht ausgeführt wird, muss keine Aktion aus dem Hauptthread ausgelagert werden.

4. $w_{guard} + w_{a()} > \nu p$. Auch wenn `do()` nicht ausgeführt wird, muss `a()` ausgelagert werden.

zu 2.) In diesem Fall wird ein neuer aperiodischer Thread erzeugt. Das Erzeugen und Starten des Threads benötigt die Zeit w_{startA}. Die maximale Ausführungszeit des periodischen Hauptthreads verkürzt sich auf $w_{g'}$. Der periodische Hauptthread ist und bleibt mit einer Wcet von $w = \nu p$ beim Scheduler angemeldet, was bewirkt, dass spätestens νp vor Ende einer jeden Periode mit der Ausführung begonnen wird. Wie aus Abb. B.1 ersichtlich beginnt die Aktivierungsüberprüfung zum Zeitpunkt $t+2p-\nu p$. Daraus ergibt sich für den Startzeitpunkt des aperiodischen Threads $t_{started} = t+2p-\nu p+(w_{do()}+w_{guard()}+w_{startA()})/\nu$. Nach dem Start steht ihm noch die Zeit $t_{rest} = t+2p-t_{started} = \nu p-(w_{do()}+w_{guard()}+w_{startA()})/\nu$ bis zum Ende der Periode zur Verfügung. Die Deadline –und somit auch die Minimum Interarrival Time– des aperiodischen Threads ist größer als die Periode des Hauptthreads, weshalb der aperiodische eine niedrigere Priorität erhält und deswegen genau einmal pro Periode unterbrochen wird (siehe Kapitel 2.1.2 und Abb. B.2).

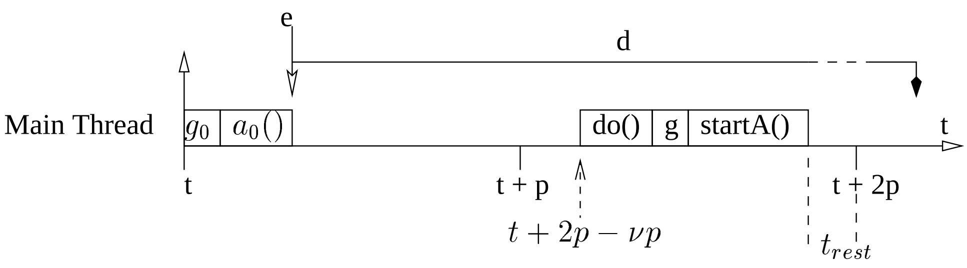

Abbildung B.1.: Periodenbestimmung – Fall 2

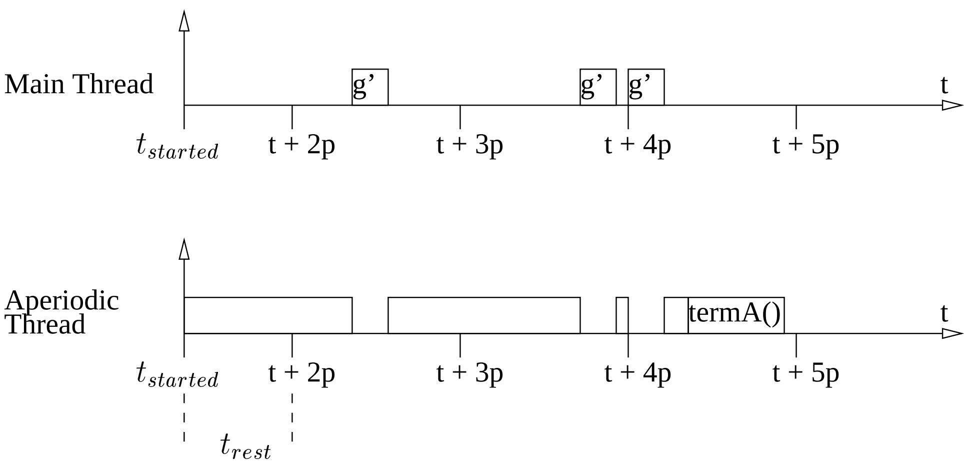

Abbildung B.2.: Unterbrechung eines aperiodischen Threads

Auch in der letzten Periode, also in der, in der er beendet wird, muss in der Worst Case Betrachtung davon ausgegangen werden, dass der Hauptthread so früh ausgeführt wird, dass er ihn noch einmal unterbricht. Angenommen, er wird k-mal für ein Zeitintervall der Länge $w_{g'}$ unterbrochen, dann liegt das Ende des aperiodischen Threads bei
$t_{end} = t + 2p + \frac{w_{a()}+w_{termA()}-t_{rest}}{\nu} + k\frac{w_{g'}}{\nu}$.
$\Rightarrow\ t_{end} = t + 2p + \frac{\nu w_{a()}+\nu w_{termA()}-\nu^2 p+w_{do()}+w_{guard}+w_{startA()}}{\nu^2} + k\frac{w_{g'}}{\nu}$.
$\Rightarrow\ t_{end} = t + p + \frac{\nu w_{a()}+\nu w_{termA()}+w_{do()}+w_{guard}+w_{startA()}}{\nu^2} + k\frac{w_{g'}}{\nu}$
Bei der Bestimmung von k muss berücksichtigt werden, dass nach k Perioden k mal $w_{g'}$ sowie der Seiteneffekt $w_{a()}$ (abzüglich der Zeit t_{rest}) ausgeführt wurde:
$\frac{w_{a()}-\nu t_{rest}}{\nu} + k\frac{w_{g'}}{\nu} \leq kp$
$\Leftrightarrow k\left(p - \frac{w_{g'}}{\nu}\right) \geq \frac{w_{a()}-\nu t_{rest}}{\nu}$
$\Leftrightarrow k \geq \frac{w_{a()}-\nu t_{rest}}{p\nu - w_{g'}} = \frac{w_{a()}-\nu^2 p+w_{guard}+w_{startA()}}{p\nu - w_{g'}}$
Somit ergibt sich für k:

$$k = \left\lceil \frac{w_{a()} - \nu^2 p - w_{guard} - w_{startA()}}{p\nu - w_{g'}} \right\rceil \tag{B.1}$$

Für die Deadline muss $d \geq t_{end} - t - (w_{g0} + w_{a0()})/\nu$ (vgl. Abb. B.1) gelten. Auch hier wird wieder die Vorgänger-Transitionen, die den geringsten Wert für $w_{g0} + w_{a0()}$ liefert, ausgewählt. Durch Einführen der Substitutionen α, β und γ ergibt sich die Ungleichung $d \geq p + \alpha + k\beta - \gamma \ \Leftrightarrow\ p \leq d - \alpha - k\beta + \gamma$ mit
$\alpha = \frac{\nu w_{a()}+\nu w_{termA()}+w_{do()}+w_{guard}+w_{startA()}}{\nu^2}$, $\beta = \frac{w_{g'}}{\nu}$, $\gamma = \frac{w_{g0}+w_{a0()}}{\nu}$.
k kann als $k = \left\lceil \frac{\nu p - \varphi}{p+\lambda} \right\rceil$ mit $\varphi = \frac{w_{a()}-w_{guard}-w_{startA()}}{\nu}$, $\lambda = \frac{w_{g'}}{\nu}$ dargestellt werden. Einsetzen liefert die Ungleichung B.2.

$$p \leq d - \alpha + \gamma - \beta \left\lceil \frac{\nu p - \varphi}{p + \lambda} \right\rceil \tag{B.2}$$

Die Lösung dieser Ungleichung ist nicht trivial, da sich p auf beiden Seiten der Ungleichung und insbesondere innerhalb der Gauß-Klammern befindet. Eine Anwendung numerischer Verfahren (wie z.B. dem Bisektionsverfahren) liefert eine Lösung p_{equ} für die entsprechende Gleichung. $0 \leq p \leq p_{equ}$ sind alle positiven Lösungen der Ungleichung. In Verbindung mit den Vorbedingungen folgt für p:

$$p^2_{t,min} := \frac{w_{do()} + w_{guard} + w_{startA}}{\nu} \leq p < \min\left(p_{equ}, \frac{w_{do()} + w_{guard} + w_{a()}}{\nu}\right) =: p^2_{t,max} \tag{B.3}$$

zu 4.) Hier ändern sich nur einige Konstanten gegenüber den Gleichungen aus Fall 2. Das Starten des aperiodischen Threads ist zum Zeitpunkt $t_{started} = t + 2p - \nu p + (w_{guard} + w_{startA()})/\nu$ beendet (vgl Abb. B.3). Die verbleibende Zeit in dieser Periode ist dann $t_{rest} = t + 2p - t_{started} = \nu p - (w_{guard} + w_{startA()})/\nu$, wodurch sich die Formel für t_{end} wie folgt ändert:

$t_{end} = t + 2p + \frac{w_{a()}+w_{termA()}-t_{rest}}{\nu} + k\frac{w_{g'}}{\nu}$

$\Rightarrow\ t_{end} = t + 2p + \frac{\nu w_{a()}+\nu w_{termA()}-\nu^2 p+w_{guard}+w_{startA()}}{\nu^2} + k\frac{w_{g'}}{\nu}$

$\Rightarrow\ t_{end} = t + p + \frac{\nu w_{a()}+\nu w_{termA()}+w_{guard}+w_{startA()}}{\nu^2} + k\frac{w_{g'}}{\nu}$

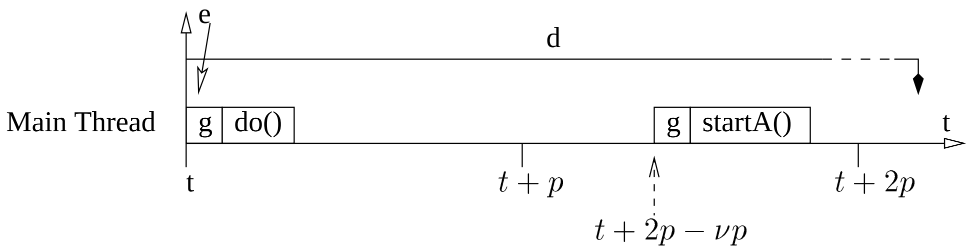

Abbildung B.3.: Periodenbestimmung – Fall 4

Die Bestimmung von k ändert sich nicht. Es kann weiterhin $k \geq \frac{w_{a()}-\nu^2 p+w_{guard}+w_{startA()}}{p\nu-w_{g'}}$ und mit obigen Substitutionen $k = \left\lceil\frac{\nu p-\varphi}{p+\lambda}\right\rceil$ verwendet werden.

Für die Deadline muss $d \geq t_{end} - t$ (vgl Abb. B.3) gelten. Durch Einführen der Substitutionen α' und β ergibt sich die Ungleichung $d \geq p + \alpha' + k\beta \Leftrightarrow p \leq d - \alpha' - k\beta$ mit

$\alpha' = \frac{\nu w_{a()}+\nu w_{termA()}+w_{guard}+w_{startA()}}{\nu^2}$, $\beta = \frac{w_{g'}}{\nu}$.

Hier führt Einsetzten zu der Ungleichung B.4.

$$p \leq d - \alpha' - \beta\left\lceil\frac{\nu p - \varphi}{p + \lambda}\right\rceil \tag{B.4}$$

Auch hier liefert ein numerisches Verfahren eine Lösungsmenge der Form $0 \leq p \leq p_{equ}$. In Verbindung mit den Vorbedingungen folgt für p:

$$p^4_{t,min} := \frac{w_{do()} + w_{guard} + w_{startA}}{\nu} \leq p < \min\left(p_{equ}, \frac{w_{guard} + w_{a()}}{\nu}\right) =: p^4_{t,max} \tag{B.5}$$

C. Realtime-Java Referenz Implementierungen

Während der Vorarbeiten zu dieser Arbeit lag lediglich eine Referenz Implementierung (RI) der RTSJ von der Firma OTI [OTI] vor. Später folgte eine von TimeSys [Tim]. Javaprozessorhersteller aJile Systems [aJi] arbeitet an der Entwicklung eines Javaprozessors, der die RTSJ implementiert und hat die Fertigstellung für Ende 2002 angekündigt.

OTI Die RI lag in der Version 1.4Beta vor. In dieser Version sind u.a. eine VM (j9) und ein Compiler (j9c) für das Echtzeitbetriebssystem QNX [QNX] enthalten, welches zum Zeitpunkt des Testens in der Version 6.1.0 vorlag. Ein Test der Klassen und Methoden auf Funktionalität brachte allerdings einige Fehler zum Vorschein. Wie erwähnt, handelte es sich hierbei um eine Beta-Version. Nachfragen bezüglich eines Updates blieben unbeantwortet und auf der Homepage der Firma finden sich inzwischen keine Hinweise mehr zur RI, weshalb die Vermutung nahe liegt, dass die Arbeit an diesem Projekt eingestellt worden ist.

Die Speicherverwaltung funktioniert in der Version einwandfrei. Anhand mehrerer Beispiele wurden die einzelnen Klassen getestet und ausschließlich Übereinstimmungen bezüglich Spezifikation und Funktionalität festgestellt. Leider können in den einzelnen Speicherbereichen nur Objekte erzeugt werden, deren Konstruktoren keine Parameter übergeben werden (`public Object newInstance(Class type)`). Das ist allerdings darauf zurückzuführen, dass zu dem Zeitpunkt die RTSJ noch nicht in der endgültigen Version vorlag und dort zu dem Zeitpunkt keine weiteren Methoden zur Objektinstanziierung spezifiziert waren. Das Erzeugen von asynchronen Ereignissen und deren Behandlung funktioniert ebenfalls, allerdings mit einer Einschränkung, da in der Klasse `AsyncEvent` die Zählvariable, die die Anzahl der assoziierten `AsyncEventHandler` angibt, falsch gesetzt wird. Das Scheduling funktioniert nur begrenzt. Weist man den `RealtimeThreads` Prioritäten zu, bestimmen diese zwar die Asuführungsreihenfolge, ein Ablauf der Deadline führt allerdings nicht zum Aufruf des `missHandlers`. Die RTSJ schreibt keine Implementierung der Kostenüberwachung und der Feasibility Analyse vor, worauf bei der RI auch verzichtet wurde. Allerdings verlangt die Spezifikation, dass die Methoden zur Feasibility Analyse, die einen boolschen Rückgabewert haben (z.B. `PriorityScheduler.isFeasible()`), in dem Fall `true` zurückliefern. Diese Methoden liefern allerdings standardmäßig `false` zurück. Die Rümpfe der Methoden `schedulePeriodic()`, `deschedulePeriodic()` und `waitForNextPeriod()` aus der Klasse `RealtimeThread` sind leer, wodurch die Verwendung periodischer Threads verhindert wird. Die Klasse `MonitorControl`, deren Unterklassen `PriorityCeilingEmulation` und `PriorityInheritance` sowie die Klasse `AsynchronouslyInterruptedException` und deren Unterklasse `Timed` stellen ebenfalls nur ein Grundgerüst ohne Funktionalität dar, um konform mit der Spezifikation zu sein, obwohl lediglich auf eine Implementierung der `PriorityCeilingEmulation`-Klasse verzichtet werden darf. Zusätzlich enthalten viele Klassen folgende Methode: `public java.lang.String toString() { return "Null"; }` Das verstößt zwar gegen keine Spezifikation, jedoch gegen Konventionen.

Wer lediglich die Funktionalität des MemoryManagements und der asynchronen Events nutzen möchte, wird mit dieser RI keine Probleme haben. Für alle anderen Bereiche –insbesondere für jene dieser Arbeit– ist diese Version ungeeignet.

TimeSys Diese RI liegt in der Version 1.0 für Linux-Betriebssysteme vor. Neben der Virtuellen Maschine (tjvm) bietet TimeSys einen Echtzeit-Kernel an, der u.a. den benötigten Priority Scheduler bereitstellt. Als Linux Distribution wird Red Hat Linux 7.1, 7.0 oder 6.2 empfohlen. Der Echtzeit-Kernel basiert auf der Version 2.4.7 des Linux-Kernels. Als Compiler wird 'javac' aus Suns SDK empfohlen. In dieser Arbeit konnte Red Hat Linux 7.1 sowie die SDKs 1.3 und 1.4 ohne Probleme verwendet werden. Um die Funktionalität des Schedulings nutzen zu können muss man auf dem Linux-System als User `root` eingeloggt sein. Auf eine Implementierung der Kostenüberwachung beim Scheduling und des Priority Ceiling Emulation Protokolls wurde in Übereinstimmung mit der Spezifikation verzichtet. Die Feasibility Analyse wurde zwar implementiert, basiert allerdings auf der Formel $\sum_i w_i/p_i \leq 1$, deren Erfüllung zwar notwendig, aber nicht hinreichend ist (vgl. Kapitel 2.1.3 und 2.1.4). Mit Ausnahme des Bugs, der in einigen Fällen beim Starten von `NoHeapRealtimeThreads` auftritt (Dieser Fehler und die Möglichkeit ihn zu umgehen wurde in Kapitel 6.4 beschrieben), funktioniert diese RI einwandfrei.

Neben der RI bietet TimeSys das so genannte *RTSJ Technology Compability Kit (TCK)* an. Es wird empfohlen, das TCK nach einer Installation der RI zu verwenden, um sie auf Funktionalität zu überprüfen. Allerdings werden dabei nur rudimentäre Eigenschaften überprüft, so dass die Aussagekraft dieser Tests diskussionswürdig ist.