Institut für Informatik
Arbeitsgruppe Softwaretechnik
Warburger Str. 100
33098 Paderborn

On- und off-line Visualisierung der Ausführung von Real-Time Statecharts

Studienarbeit
zur Erlangung des Grades

Bachelor of Computer Science

für den Studiengang Informatik

von

Margarete Kudak
Mittelberg 7
33100 Paderborn

vorgelegt bei
Dr. Holger Giese

und
Prof. Dr. Reinhard Keil-Slawik

betreut von
Dipl.-Wirt.-Inf. Matthias Tichy

Paderborn, Juli 2004

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

An der Universität Paderborn wurde im Jahre 1997 in der AG Softwaretechnik unter Leitung von Professor Dr. Wilhelm Schäfer die Softwareentwicklungsumgebung Fujaba [8] entwickelt. Mit Fujaba können verschiedene UML-Diagramme [14] erstellt und Java Code generiert werden. UML steht für Unified Modeling Language und ist eine objektorientierte Modellierungssprache, durch die mit verschiedenen Diagrammen Struktur und Dynamik dargestellt werden können. Unter anderem ist es möglich, Statecharts und Real-Time Statecharts, die auf dem Konzept der Statecharts aufbauen, zu modellieren. Ein UML-Statechart [14] ist auf der Grundlage eines endlichen Automaten aufgebaut und beschreibt ein Verhalten.
Real-Time Statecharts [4] wurden im September 2002 im Rahmen der Diplomarbeit von Sven Burmester in der AG Softwaretechnik entwickelt. Ein Real-Time Statechart ist eine Erweiterung eines UML-Statechart-Modells. Durch Real-Time Statecharts kann das Verhalten von Echtzeitsystemen modelliert werden. Auf diese Themen wird in Kapitel 2 detaillierter eingegangen.
In dieser Studienarbeit soll eine Visualisierung der Ausführung für Real-Time Statecharts entwickelt und implementiert werden. Dabei werden zwei Arten der Visualisierung unterschieden: online und offline. Die online Visualisierung ermöglicht die gleichzeitige Ausführung und Visualisierung des Real-Time Statecharts, während bei der offline Visualisierung zuerst die Ausführung des Real-Time Statecharts und anschließend die Visualisierung der Ausführung vollzogen werden. Die im Folgenden beschriebenen Konzepte sind sowohl für die online, als auch für die offline Visualisierung anzuwenden. Die offline Visualisierung ist im Werkzeug realisiert worden, kann aber sehr einfach für die online Visualisierung angepasst werden.
Die Ausführung der Real-Time Statecharts wird mit modifizierten UML-Sequenzdiagrammen [14] und einer Darstellung des Real-Time Statecharts durch entsprechende, farbliche Markierungen visualisiert. Die Motivation für die Entwicklung einer Visualisierung liegt darin, dass es bei umfassenderen Real-Time Statecharts schwer nachzuvollziehen ist, ob diese korrekt funktionieren. So ist es zum Beispiel möglich, dass es während der Ausführung mehrerer Real-Time Statecharts zu einem Deadlock kommt und die Ausführung nicht mehr weiter läuft. Neben der Erkennung von Deadlocks, unterstützt diese Studienarbeit auch die Erkennung von Modellierungsfehlern seitens des Entwicklers.
Es besteht die Möglichkeit Real-Time Statecharts auf mehreren Systemen parallel auszuführen. Die Kommunikation der Real-Time Statecharts untereinander erfolgt über das Versenden und Empfangen von Nachrichten. Werden diese Nachrichten zum Beispiel über ein Funknetz gesendet, so kann es bei der Übertragung zu einer Verzögerung bzw. sogar zu einer Störung kommen. Durch diese und ähnliche Probleme kann sich das Verhalten der Real-Time Statecharts ändern. Werden solche Real-Time Statecharts zum Beispiel in eingebetteten Systemen eingesetzt, würde das komplette System nicht korrekt funktionieren, was schwerwiegende Folgen haben kann. Mit Hilfe dieser Studienarbeit ist es möglich das Verhalten mehrerer Real-Time Statecharts zu visualisieren, um Probleme die bei der Nachrichtenübertragung auftreten, vorzeitig zu erkennen.
Zudem bieten Real-Time Statecharts mehrere Konstrukte, die es ermöglichen Echtzeitverhalten nachzuempfinden. Gerade bei Echtzeitsystemen ist es sehr wichtig, dass die vorgeschriebenen Zeiten nicht unter- oder überschritten werden, denn eine nicht zeitkonforme Uhr kann drastische Folgen haben. Schließt zum Beispiel der Wasserzufluss eines Beckens nicht rechtzeitig, so wird das Wasserbecken überlaufen und es könnte zu Schäden durch das ausgeflossene Wasser kommen. Es ist aber auch möglich, dass auf Grund nicht korrekt laufender Uhren, das Real-Time Statechart in einem Zustand verweilt und diesen nicht mehr

verlassen kann. In diesem Fall würde sich der Wasserzufluss gar nicht mehr schließen und es käme zu einer Überschwemmung. Anhand des Konzepts, das in dieser Studienarbeit entwickelt wird, ist es möglich zu erkennen, ob das Verhalten des Real-Time Statecharts zeitkonform ist.
Im Folgenden wird die Umsetzung der Visualisierung der Ausführung von Real-Time Statecharts anhand einer Lösungsskizze erläutert.

1.1 Lösungsskizze

Abbildung 1 aus [22] zeigt ein Übersichtsbild, das das Verfahren von der Erstellung des Real-Time Statecharts bis zur Visualisierung zeigt.

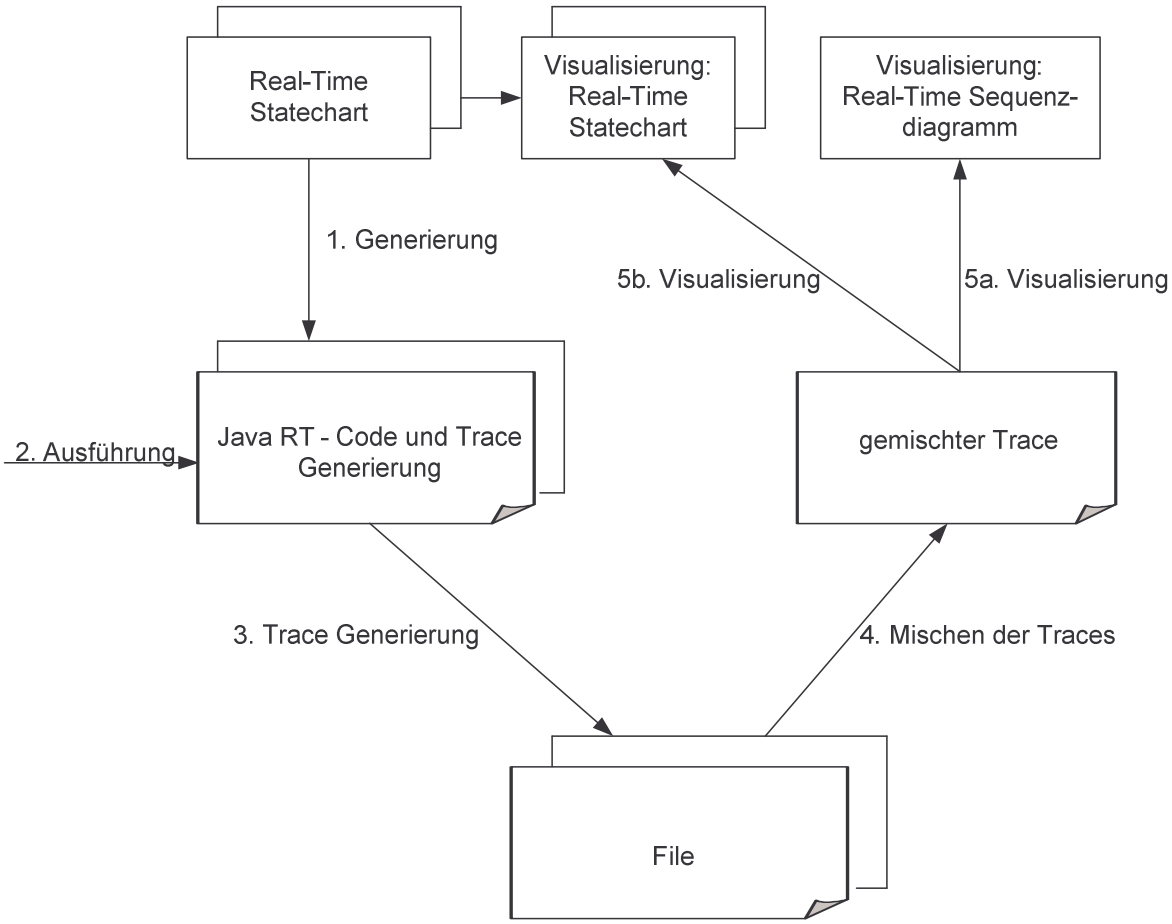

Abbildung 1 - Übersicht

Bisher ist es möglich aus Real-Time Statecharts Java RT-Code zu generieren. In dieser Studienarbeit wird der Java RT-Code um eine Trace Generierung erweitert. Der generierte Trace enthält alle Informationen, die für eine Visualisierung nötig sind. Dazu gehören der Eintritt in einen Zustand, der Austritt aus einem Zustand, das Schalten einer Transition und die jeweiligen Zeitpunkte. Es ist ebenfalls möglich mehrere Real-Time Statecharts zu visualisieren. Dafür werden die generierten Trace Dateien miteinander gemischt, so dass eine einzige Trace Datei entsteht, deren Einträge chronologisch geordnet sind.
Aus der Trace Datei und dem Real-Time Statechart selber kann nun die Visualisierung des Verhaltens in Form eines Real-Time Statecharts und eines modifizierten Sequenzdiagramms angezeigt werden.

1.2 Beispiel

Um dieses Problem näher zu beschreiben, wird im Folgenden anhand eines Beispiels erklärt, wie eine solche Visualisierung auszusehen hat. In diesem Beispiel wird die Problematik anhand von UML-Statecharts erläutert, da sie auf Grund ihrer geringeren Komplexität gegenüber den Real-Time Statecharts für ein einführendes Beispiel besser geeignet sind. Da das Konzept der Real-Time Statecharts auf das Konzept der UML-Statecharts zurückzuführen ist, ist die Übertragung der Problematik und ihrer Lösung auf Real-Time Statecharts möglich.
In Abbildung 2 sind zwei UML-Statecharts dargestellt, die miteinander über ein Funknetz kommunizieren. Die Kommunikation über ein Funknetz ist jedoch oft verzögert.

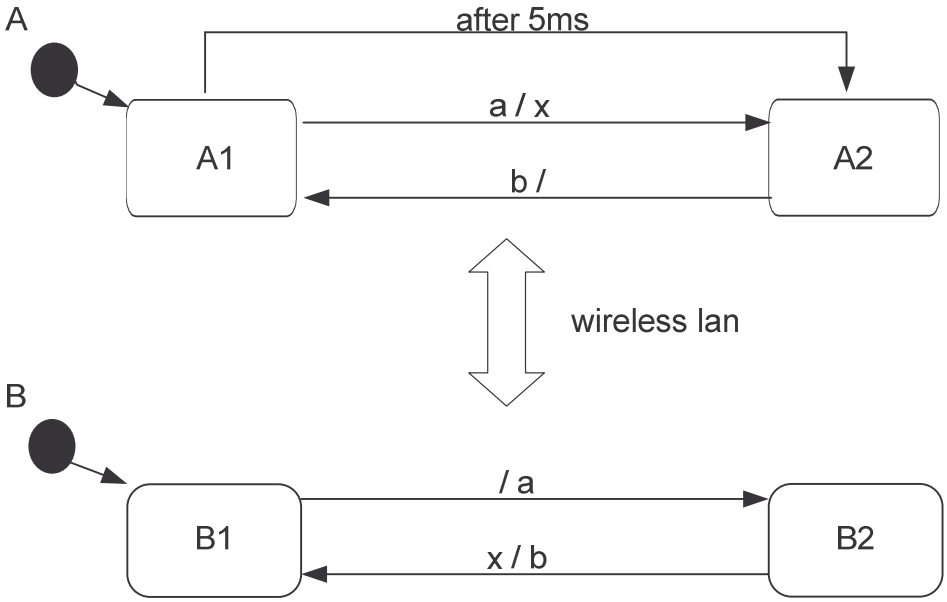

Abbildung 2 - kommunizierende Statecharts

Das UML-Statechart A befindet sich zu Anfang in A1 und B in B1. B1 sendet über wireless lan die Nachricht a an A und geht in den Zustand B2 über. Sobald A die Nachricht empfängt, sendet A die Nachricht x und wechselt in den Zustand A2. Wenn B x erhält, geht B von B2 in B1 über und sendet b an A, worauf A in A1 wechselt. Nun befindet sich A in A1 und wartet auf die Nachricht a von B. Das UML-Statechart B befindet sich in B1 und geht über nach B2. Dies bedeutet, dass die Nachricht a über Funknetz an A geschickt wird.
Dauert diese Übertragung länger als 5ms, so wechselt A in A2 ohne eine Nachricht zu senden. Dies hat zur Folge, dass B auf x wartet und A auf b. Beide UML-Statecharts warten gegenseitig aufeinander und hängen fest. Sie befinden sich also in einer Deadlock Situation.
In größeren Statecharts wäre eine Fehlererkennung, wie hier durchgeführt, nicht mehr möglich. Aus diesem Grund wird in dieser Studienarbeit eine Visualisierung entwickelt, die den Programmierer bei der Validierung unterstützt.

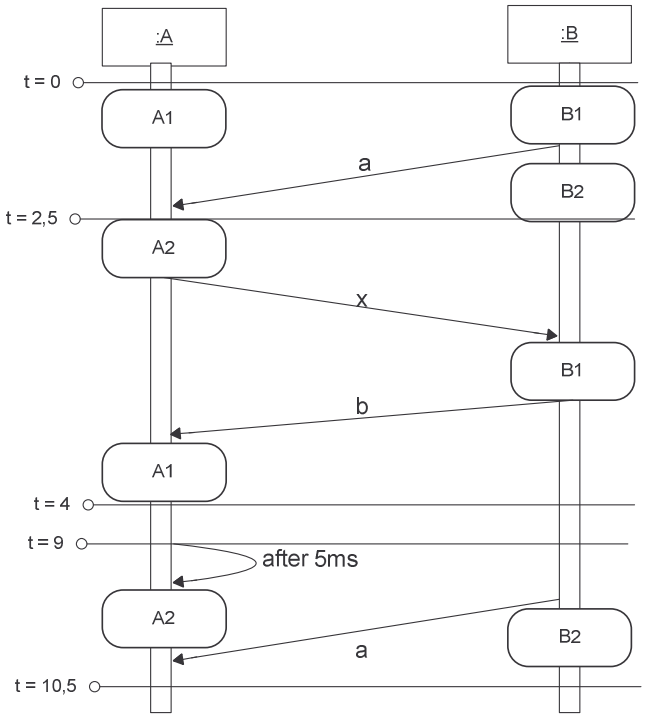

Abbildung 3 - Visualisierung als Sequenzdiagramm

In Abbildung 3 ist ein Sequenzdiagramm dargestellt, dass das Verhalten der vorher beschriebenen UML-Statecharts beschreibt. Jede Instanz eines UML-Statecharts wird durch eine Lebenslinie dargestellt. Die Kommunikation zwischen `A` und `B` wird durch Pfeile symbolisiert. Zudem lässt sich erkennen, zu welchem Zeitpunkt welche Nachricht gesendet wird, da die einzelnen Zeitpunkte `t` mit in die Visualisierung aufgenommen sind. In diesem Sequenzdiagramm ist es sehr leicht zu erkennen, dass `A` mit der Transition `after 5ms` schaltet und die Nachricht `a` zu spät erhält, so dass ein Deadlock entstanden ist.
Eine weitere Form der Visualisierung besteht aus UML-Statecharts, in denen farblich hervorgehoben ist, welche Transition schaltet bzw. in welchem Zustand sich ein UML-Statechart befindet. Der Benutzer hat die Möglichkeit sich einen Zeitpunkt im Sequenzdiagramm auszuwählen und sich dazu das UML-Statechart mit dem aktuellen Zustand bzw. der aktuellen Transition anzeigen zu lassen.

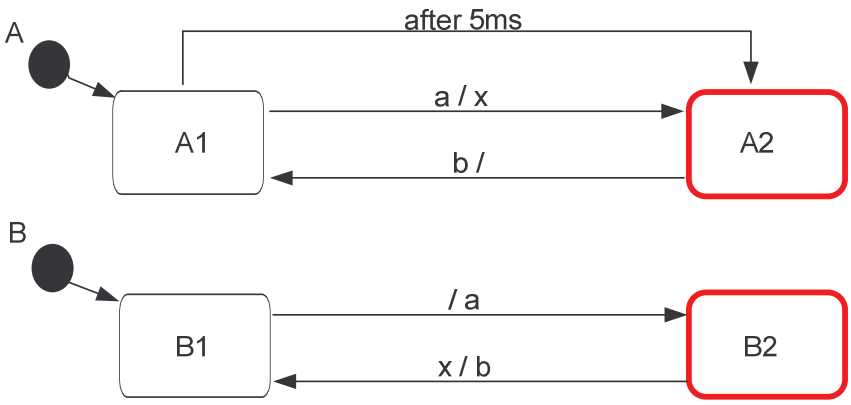

Abbildung 4 - Visualisierung als Statechart

Für das Sequenzdiagramm aus Abbildung 3 und dem Zeitpunkt `t = 10,5` zeigt Abbildung 4 die geforderten UML-Statecharts. Nachdem das UML-Statechart `A` über die Transition `after 5ms` in den Zustand `A2` gewechselt ist, sendet das UML-Statechart `B` die Nachricht `a` und wechselt in den Zustand `B2`. Das UML-Statechart `A` wartet nun auf die Nachricht `b`, während `B` auf die Nachricht `x` wartet und erst dann die Nachricht `b` schickt.
Abbildung 5 zeigt das Schalten der Transition `after 5ms`. Das UML-Statechart `A` wechselt vom Zustand `A1` in den Zustand `A2`, während sich `B` im Zustand `B1` befindet. Diese Situation entspricht dem Zeitpunkt `t = 9` aus dem Sequenzdiagramm aus Abbildung 3.

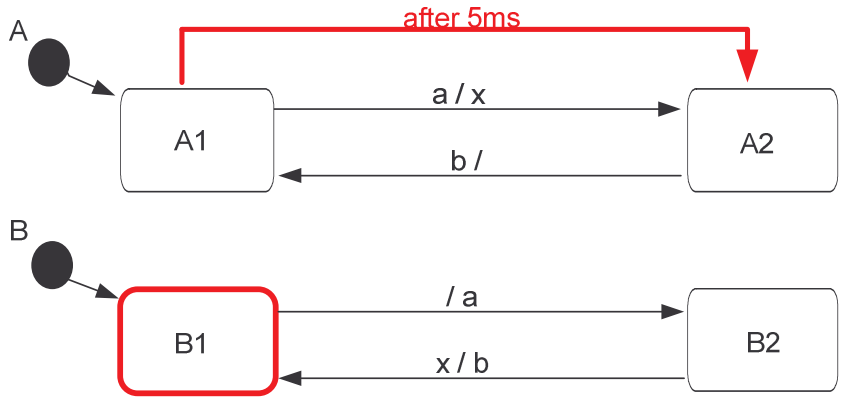

Abbildung 5 - Visualisierung als Statechart

1.3 Ausblick

Im 2. Kapitel werden die einzelnen UML-Diagrammarten, die in dieser Studienarbeit verwendet werden, näher beschrieben. Anschließend werden im 3. Kapitel verwandte Arbeiten diskutiert. Das 4. Kapitel beinhaltet die Generierung der Trace Datei. Im 5. Kapitel wird dann die Visualisierung beschrieben. Im Anschluss daran befinden sich in Kapitel 6 eine Zusammenfassung und ein Ausblick. Die Benutzung der implementierten Software wird im Anhang erläutert.

2. Grundlagen

Im Folgenden werden die für diese Studienarbeit benötigten Grundlagen beschrieben. Dazu gehören neben dem Konzept der verteilten Systeme und der Nebenläufigkeit die beiden Diagrammarten Real-Time Statecharts und UML-Sequenzdiagramme.
UML, als Standard einer objektorientierten Modellierungssprache, bietet die Möglichkeit mittels verschiedenen Diagrammen ein Softwaresystem zu beschreiben. Es existieren zwei verschiedene Kategorien von Diagrammen. Zum einen solche, die die Struktur beschreiben und solche, die das Verhalten spezifizieren. Erstere werden auch statische Diagramme genannt, letztere dynamische Diagramme. Alle hier beschriebenen Diagrammarten sind den dynamischen zuzuordnen.

2.1 Verteilte Systeme

Verteilte Systeme ermöglichen es, die Ausführung mehrerer Real-Time Statecharts durchzuführen, da es möglich ist jedes Real-Time Statechart auf einem eigenen System auszuführen.
Nach [19] ist ein verteiltes System eine Zusammenfassung autonomer Rechner, die durch ein Netzwerk miteinander verbunden sind. Dem Nutzer hingegen stellt sich das verteilte System als einheitliches Rechnersystem dar, da die interne Struktur und die Art der Kommunikation zwischen den einzelnen Rechnern dem Nutzer verborgen bleiben. Somit kann der Anwender alle Applikationen eines verteilten Systems in einer konsistenten und einheitlichen Weise unabhängig von Ort und Zeit der Interaktion nutzen. Zusätzlich bieten verteilte Systeme eine hohe Ausfallsicherheit, da es keine Konsequenzen hat, wenn Teile temporär ausfallen, weil die Arbeit auf andere Rechner verteilt werden kann. Um die Darstellung als einheitliches Rechnersystem zu gewährleisten, organisiert eine Middleware die Rechner. Für weitere Informationen siehe [19].
Die Kommunikation und Koordination zwischen Prozessen in einem verteilten System erfolgt, wie auch bei den Real-Time Statecharts, durch Nachrichtenaustausch.
Für verteilte Systeme existiert keine globale Uhr. Jeder Rechner besitzt eine lokale Uhr. Bei lokalen Uhren ist keine Garantie vorhanden, dass diese alle gleich schnell laufen. Dies ist ein Nachteil von verteilten Systemen, da es nach [13, S. 2 ff.] nur sehr begrenzte Synchronisationsmöglichkeiten für lokale Uhren gibt.
Auf Grund der verteilten Ausführung der Real-Time Statecharts und der Synchronisationsprobleme kann eine Änderung des Verhaltens entstehen. Diese Studienarbeit hilft bei der Lösung dieses Problem durch die Visualisierung der Ausführung. Anhand dieser lässt sich erkennen, ob das tatsächliche Verhalten der Real-Time Statecharts dem modellierten Verhalten entspricht. Allerdings stellt die verteilte Ausführung auch ein Problem für diese Studienarbeit dar, da auf Grund der fehlenden Synchronisationsmöglichkeiten nicht genau bestimmt werden kann, zu welchem Zeitpunkt die Real-Time Statecharts starten. Das Problem wird in Kapitel 4.6 aufgegriffen und genauer erläutert.

2.2 Nebenläufigkeit

Wie in Abschnitt 2.1 schon beschrieben, ist es möglich Real-Time Statecharts auf verteilten Systemen parallel auszuführen und somit Nebenläufigkeit zu erreichen. Des Weiteren kann Nebenläufigkeit auch durch die Ausführung mehrerer Threads auf einem Rechner, wobei jedes Real-Time Statechart durch einen Thread repräsentiert wird, umgesetzt werden.

Hierbei findet die Ausführung in einem gemeinsamen Speicherbereich statt. Zusätzlich existieren in Java RT sowohl periodische, als auch aperiodische Threads. Ein periodischer Thread wird in regelmäßigen Abständen, die von einem Scheduler bestimmt werden, gestartet. Ein aperiodischer Thread hingegen wird in unregelmäßigen, unbekannten Abständen gestartet.
Da sich alle Threads einen gemeinsamen Adressraum teilen, ist eine Synchronisation von Methoden und Variablen erforderlich. Andernfalls kann es zu unerwünschten Ergebnissen kommen, wenn zwei Threads gleichzeitig auf eine Methode bzw. eine Variable zugreifen. Die Trace Generierung muss synchronisiert werden, weil zum Beispiel das Schreiben der Daten in den Trace nicht vom Lesen der Daten aus dem Trace gestört werden darf, da so unter Umständen Daten verloren gehen. Die Synchronisation erfolgt durch einen Monitor [13, S. 241 ff.].
Ein Monitor kapselt einen kritischen Bereich, wie zum Beispiel eine Methode mit Hilfe einer Sperre. Beim Betreten des kritischen Bereichs wird die Sperre atomar gesetzt und beim Verlassen wieder atomar aufgehoben. Greift nun ein anderer Thread auf die für ihn durch die Sperre blockierte Methode zu, so kann er die Methode nicht ausführen und muss warten bis die Sperre wieder atomar aufgehoben wird.
In Java wird das Setzen einer Sperre durch das Schlüsselwort `synchronized` ausgeführt. Es ist möglich `synchronized` auf Methoden und auf Attribute anzuwenden. Das Sperren von Methoden kann jedoch zu Deadlocks führen. Ein Deadlock entsteht, wenn mindestens zwei Prozesse auf den Zugriff von Methoden warten, die jeweils dem anderen Prozess zugeordnet sind. Fordert zum Beispiel Prozess `A` den Zugriff auf die Methode `enqueueElement()` und besitzt den Zugriff auf die Methode `dequeueElement()`, so entsteht ein Deadlock, wenn Prozess `B` den Zugriff auf die Methode `enqueueElement()` besitzt und gleichzeitig den Zugriff auf die Methode `dequeueElement()` anfordert.
Des Weiteren können während einer nebenläufigen Ausführung Race Conditions auftreten. Eine Race Condition liegt vor, wenn die Reihenfolge der Bearbeitung der einzelnen Prozesse zeitlich nicht garantiert werden kann und das Ergebnis mehrerer Ereignisse zusätzlich von der Reihenfolge abhängt. Der Name leitet sich aus dem Rennen, das die Prozesse um die Ressource führen, ab.

2.3 Real-Time Statechart – Diagramm

Im Folgenden wird das Konzept der Real-Time Statecharts anlehnend an [11] vorgestellt. Real-Time Statecharts sind eine Erweiterung des UML-Statechart Modells [14] und modellieren Echtzeitverhalten. Sie besitzen bis auf das After-Konstrukt alle Eigenschaften eines UML-Statechart Modells. Allerdings wurden Uhren in das Modell eingeführt, mit denen das Verhalten des After-Konstrukts nachempfunden werden kann. Dieses Konzept bietet durch die Spezifikation mehrerer Uhren mehr Möglichkeiten, da die Uhren durch das logische „und“ zu einem Ausdruck miteinander verknüpft werden können und zustandsübergreifend existieren.
Mit Real-Time Statecharts wird ebenfalls Verhalten modelliert. Im Gegensatz zu den UML-Statecharts, deren generierter Code keine Echtzeitsituation wieder spiegelt, ist dieses Verhalten jedoch in Real-Time Code umsetzbar. Es existiert eine Java RT Codegenerierung in Fujaba. Näheres dazu findet sich in [5].

Nachfolgend wird die Notation der Real-Time Statecharts vorgestellt.

2.3.1 Zustand

Ein Zustand, wie in Abbildung 6 dargestellt, kann gegenüber den Statecharts weitere Informationen beinhalten. Dazu gehören:

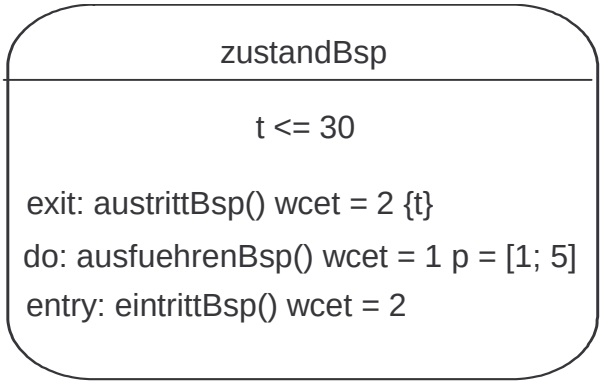

Abbildung 6 - Zustand

- **Zeitinvarianten:** Eine Zeitinvariante bezieht sich auf eine Uhr und gibt an, bis wann ein Zustand spätestens verlassen sein muss. Wird keine Zeitinvariante angegeben, so wird angenommen, dass diese bei unendlich liegt. Ein Real-Time Statechart darf sich niemals in einem Zustand mit abgelaufener Zeitinvariante befinden. Es ist möglich mehre Zeitinvarianten mit dem logischen „und“ zu verknüpfen. Der Zustand aus Abbildung 6 besitzt die Zeitinvariante `t ≤ 30`.

- **Uhren-Reset:** Diese sind mit den `entry` und `exit` Methoden verbunden. Die Uhren können bei Aufruf der beiden Methoden auf Null gesetzt werden.
  Das Konzept der Uhren ist mächtiger als das After-Konstrukt der UML-Statechart Diagramme. Durch die Kombination mehrerer Uhren kann Bezug auf Zeitpunkte genommen werden, die vor dem Eintritt des Zustandes liegen. Bei der `exit` Methode aus dem Zustand aus Abbildung 6 wird die Uhr `t` zurückgesetzt. Dies wird durch `{t}` dargestellt.

- **Worst Case Execution Time (WCET) zu den entry, do und exit Methoden:** Die `WCET` gibt die maximale Ausführungszeit der Methoden an. Dies wird für die Schedulinganalyse benötigt. In dem Zustand aus Abbildung 6 existieren folgende `WCETs`: für die `entry` Methode `eintrittBsp()` 2 Zeiteinheiten, für die `do` Mehthode `ausfuehrenBsp()` 1 Zeiteinheit und für die `exit` Methode `austrittBsp()` 2 Zeiteinheiten.

- **Periodenintervall für die do Methode:** Dieses Intervall gibt an, wie oft die `do` Methdode ausgeführt werden soll. Es wird hierbei eine untere und obere Grenze $[p_{low}, p_{up}]$ angegeben. In dem Zustand aus Abbildung 6 besitzt die `do` Methode `ausfuehrenBsp()` das Periodenintervall `p = [1; 5]`. Das bedeutet, dass `ausfuehrenBsp()` eine Periode von mindestens 1 Zeiteinheit und höchstens 5 Zeiteinheiten enthält.

2.3.2 Transition

Eine Transition ist ein Übergang zwischen zwei Zuständen. Transitionen aus Real-Time Statecharts und UML-Statecharts beinhalten folgende Elemente:

- **Event:** Eine Transition schaltet, wenn ein Event anliegt.
- **Guard:** Ein Guard ist ein boolescher Ausdruck einer Variable. Eine Transition schaltet, wenn der Guard wahr ist. Abbildung 7 zeigt eine Transition, die den Guard `x==2` besitzt.

Abbildung 7 zeigt eine Transition, die gegenüber den Statecharts um folgende Komponenten erweitert werden kann:

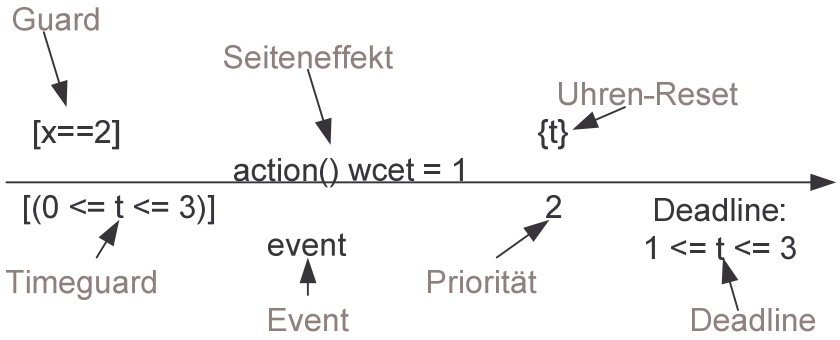

Abbildung 7 - Transition

- **Timeguard:** Zusätzlich zu dem Guard aus dem UML-Statechart Modell, enthält eine Transition einen Timeguard. Dies ist ein boolscher Ausdruck über die angegebenen Uhren. Wird kein Timeguard angegeben, so wird $0 \leq t \leq \infty$ angenommen, wobei `t` eine Uhr ist. Eine Transition kann nur schalten, wenn ein Event anliegt und der Guard und der Timeguard wahr sind. Es ist möglich Timeguards durch das logische „und“ zu verknüpfen. Die Transition, die in Abbildung 7 dargestellt ist, besitzt den Timeguard $0 \leq t \leq 3$. Die Transition kann somit nur schalten, wenn der Wert der Uhr `t` zwischen 0 Zeiteinheiten und 3 Zeiteinheiten liegt.

- **Seiteneffekt:** Diese Methode wird ausgeführt, wenn die Transition schaltet. Beim Schalten der Transition aus Abbildung 7 wird die Methode `action()` als Seiteneffekt ausgeführt.

- **WCET zum Seiteneffekt:** Gibt die maximale Ausführungszeit des Seiteneffekts an. Der Seiteneffekt `action()` der Transition aus Abbildung 7 besitzt die WCET `wcet = 1`. Dadurch spezifiziert der Benutzer, dass die Ausführung von `action()` nicht länger als 1 Zeiteinheit dauert. Benötigt wird die WCET für die Schedulinganalyse.

- **Uhren-Reset:** Zu einer Transition können Uhren angegeben werden, die zurückgesetzt werden, wenn die Transition schaltet. Im Beispiel wird die Uhr `t` zurückgesetzt (`{t}`).

- **Prioritäten:** In nicht parallelen Zuständen kann immer nur eine Transition schalten. Wenn mehrere Transitionen zum selben Zeitpunkt aktiviert werden, muss bestimmt werden, welche Transition schaltet. Um Determinismus zu garantieren, sind Prioritäten

eingeführt worden. Die Transition mit der höchsten Priorität schaltet. In dem Beispiel aus Abbildung 7 hat die Transition die Priorität 2. Existiert eine weitere Transition mit der Priorität 1 und wird diese gleichzeitig mit der Beispiel Transition aktiviert, so schaltet die Transition aus Abbildung 7.

- **Deadline:** Die Deadline gibt an, bis wann der Schaltvorgang beendet sein muss und hat folgende Form: $[d_{low}; d_{up}]$. Der Schaltvorgang darf nicht vor d_{low} beendet sein, muss aber bis zum Zeitpunkt d_{up} beendet sein. Die Transition aus Abbildung 7 besitzt die Deadline $1 \leq t \leq 3$ $(= t \varepsilon [1; 3])$. Der Schaltvorgang darf somit nicht vor 1 Zeiteinheit, muss aber spätestens bei 3 Zeiteinheiten beendet sein.

- **Synchronisationskanal und Synchronisationsart:** Bei parallelen Zuständen können Transitionen, die den gleichen Synchronisationskanal (z.B. a) besitzen, gleichzeitig schalten. Dabei ist eine Transition Sender (dargestellt durch: !) und die andere Transition ist Empfänger (dargestellt durch: ?). Ist Transition 1 der Sender und Transition 2 der Empfänger, so erhält Transition 1 die Information a! und Transition 2 die Information a?.

2.4 UML-Sequenzdiagramm

UML-Sequenzdiagramme [25] zeigen die Interaktion zwischen Objekten in zeitlicher Reihenfolge an. Ein Objekt ist eine Instanz einer Klasse. Diese Klasse wiederum ist ein Real-Time Statechart, das durch Nachrichten mit anderen Real-Time Statecharts kommuniziert. Sie werden durch Rechtecke dargestellt und besitzen eine Lebenslinie. Es existieren aktive und passive Objekte. Aktive Objekte haben eine durchgehende Lebenslinie und können selbstständig Aufrufe tätigen, während passive Objekte nur aktiv sind, wenn sie ein Nachricht empfangen bzw. eine Nachricht versenden. Aktive Objekte werden durch einen stärkeren Rahmen dargestellt. Beide Objekte kommunizieren miteinander über Nachrichten, wobei diese zwischen den einzelnen Lebenslinien verlaufen und den Namen des Aufrufs beinhalten.
Im Beispiel aus Abbildung 2 sind die Real-Time Statecharts A und B aktive Objekte mit einer durchgehenden Lebenslinie. In dieser Studienarbeit werden keine passiven Objekte verwendet, da ein Real-Time Statechart von sich aus Nachrichten schicken kann. Die Pfeile zwischen den beiden Lebenslinien symbolisieren den Nachrichten Austausch.

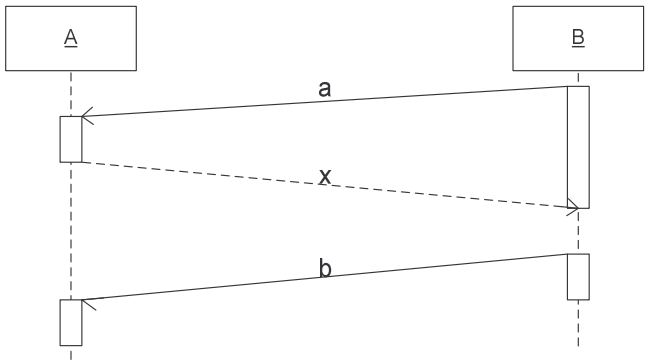

Abbildung 8 – UML-Sequenzdiagramm

Abbildung 8 zeigt ein Sequenzdiagramm mit zwei zeitweilig aktiven Objekte: A und B. Zwischen diesen beiden Objekten werden Nachrichten verschickt. B schickt die Nachricht a() an das Objekt A, worauf dieses mit der Nachricht x() antwortet. Als nächstes schickt B die asynchrone Nachricht b() an A. Auf diese Nachricht muss A keine Antwort geben.

3. Verwandte Arbeiten

Im Folgenden werden zwei verwandte Arbeiten diskutiert. In Kapitel 3.1 wird die Software JaVis, die die Möglichkeit einer Trace Erstellung bietet, vorgestellt. Rational Rose RealTime bietet eine Visualisierung der Ausführung von Statecharts und wird in Kapitel 3.2 näher beschrieben.

3.1 JaVis

JaVis [12] wurde an der Universität Paderborn in der Arbeitsgruppe „Datenbanken und Informationssysteme“ entwickelt. Es dient zur Visualisierung von nebenläufigen Programmen, wobei die Erkennung von Deadlocks im Vordergrund steht. Um ein Java-Programm zu visualisieren, wird eine Trace Datei mit allen benötigten Informationen erstellt. Daraus werden dann UML-Sequenzdiagramme erstellt, um Abläufe verschiedener Java-Threads zu visualisieren. Um die Ursachen für einen Deadlock zu analysieren, werden UML-Kollaborationsdiagramme eingesetzt. Die Erstellung der Trace Datei ist von Interesse für diese Studienarbeit, da aus dem Real-Time Statechart Code und eine Trace Datei generiert werden. Diese Trace Datei beinhaltet alle Informationen zu Zustandswechseln und deren Zeitpunkten.
Das Programm bietet dem Benutzer die Möglichkeit sich Informationen zu einer Java Datei anzusehen. Dazu wählt der Benutzer die zu analysierende Java-Classdatei aus und startet diese. Im Application State Fenster und in der Trace History werden dem Benutzer diverse Informationen zu der Ausführung angezeigt. Zuerst werden das Programm JaVis und die daraus gewonnen Informationen näher beschrieben.

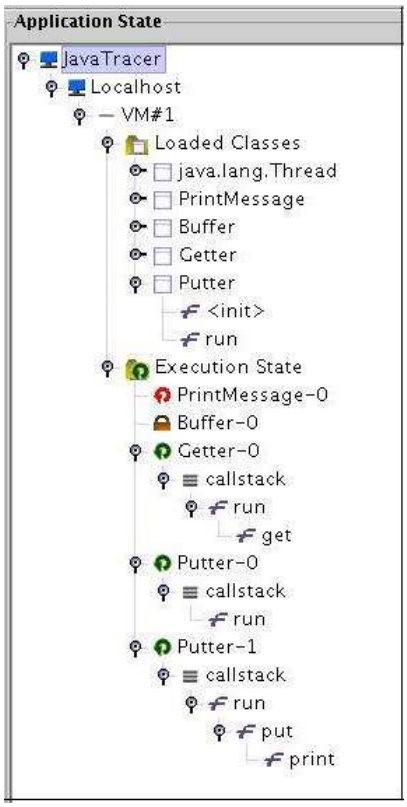

Abbildung 9 - Application State

Der Application State aus Abbildung 9 enthält folgende Informationen:

- `Localhost`: Host, auf dem die Applikation läuft
- `VM#1`: Virtuelle Maschine, die die Applikation ausführt
- `Loaded Classes`: alle geladenen Klassen und aufgerufenen Methoden dieser Klassen
- `Execution State`: Alle Objekte, Threads und Methoden, die aufgerufen werden

Abbildung 9 zeigt an, dass das ausgewählte Java Programm auf dem `Localhost` auf der Virtuellen Maschine `VM#1` läuft. Zusätzlich sind alle geladenen Klasse und deren Methoden aufgeführt. Die Klasse `Putter` z.B. beinhaltet die Methoden `init` und `run`, die mit „`f`“ gekennzeichnet sind. Der `Execution State` zeigt den aktuellen Status der Ausführung an. Alle Objekte, Threads und Methoden sind mit verschiedenen Symbolen gekennzeichnet. Zu `Print-Message-0` und `Buffer-0` existiert kein `callstack`. Somit handelt es sich hierbei um Objekte. `Buffer-0` ist zusätzlich gesperrt. `Getter-0`, `Putter-0` und `Putter-1` sind Threads. `Putter-1` z.B. besitzt einen `callstack` auf dem sich drei Methoden befinden. Anhand der Staffelung ist zu erkennen, dass die Methode `run` als erstes aufgerufen wurde, `run` die Methode `put` aufgerufen hat und diese wiederum die Methode `print`.

Parallel dazu enthält die Trace History aus Abbildung 10 folgende Informationen:

- `Event No.`: es werden alle Ereignisse nummeriert
- `Calling Thread`: Thread, der aufgerufen wird
- `Called Method`: Methode, die aufgerufen wird
- `Called Object`: Objekt, auf dem die Methode aufgerufen wird
- `Enter/Exit`: gibt an, ob eine Methode betreten oder verlassen wird

Trace History

Event No.	Calling Thread	Called Method	Called Object	Enter/Exit
26	Putter-1	freeLock()[synchronized]	PrintMessage-0	Exit
25	Putter-1	freeLock()[synchronized]	PrintMessage-0	Enter
24	Putter-1	print(java.lang.String "Putter ok!")[synchronized]	PrintMessage-0	Enter
23	Putter-1	put(java.lang.String "Test")[synchronized]	Buffer-0	Enter
22	Putter-0	put()[synchronized]	Buffer-0	Exit
21	Putter-0	print()[synchronized]	PrintMessage-0	Exit
20	Putter-0	freeLock()[synchronized]	PrintMessage-0	Exit
19	Putter-0	freeLock()[synchronized]	PrintMessage-0	Enter
18	Putter-1	run()	Putter-1	Enter
17	Putter-0	print(java.lang.String "Putter ok!")[synchronized]	PrintMessage-0	Enter
16	Putter-0	put(java.lang.String "Test")[synchronized]	Buffer-0	Enter
15	Putter-0	run()	Putter-0	Enter
14	Getter-0	get(java.lang.String "Test")[synchronized]	Buffer-0	Enter
13	Getter-0	run()	Getter-0	Enter
12	java.lang.Thread-0	main	Main	Exit
11	java.lang.Thread-0	<init>	Putter-1	Exit
10	java.lang.Thread-0	<init>(Buffer instance of Buffer(id=66))	Putter-1	Enter
9	java.lang.Thread-0	<init>	Putter-0	Exit
8	java.lang.Thread-0	<init>(Buffer instance of Buffer(id=66))	Putter-0	Enter
7	java.lang.Thread-0	<init>	Getter-0	Exit
6	java.lang.Thread-0	<init>(Buffer instance of Buffer(id=66))	Getter-0	Enter
5	java.lang.Thread-0	<init>	Buffer-0	Exit
4	java.lang.Thread-0	<init>(int 3, PrintMessage instance of PrintMessage(id=64))	Buffer-0	Enter
3	java.lang.Thread-0	<init>	PrintMessage-0	Exit
2	java.lang.Thread-0	<init>()	PrintMessage-0	Enter
1	java.lang.Thread-0	main(java.lang.String[] instance of java.lang.String[0] (id=61))	java.lang.Thread-0	Enter

Abbildung 10 - Trace History

Abbildung 10 zeigt einen Ausschnitt der Trace History zu dem oben aufgeführten Application State. Die Trace History speichert alle Methodenaufrufe und listet diese auf. Das erste Event,

das aufgetreten ist, ist der Aufruf der `main`-Methode von `java.lang.Thread-0`. Dieser Aufruf hat in der Log Datei (.trc), die zusätzlich angelegt wird, folgende Gestalt:

- `Localhost_VM#1_main:java.lang.Thread@Localhost_VM#1_ID#1: :PrintMessage@Localhost_VM#1_ID#64:<init>():false:Enter`
- `Localhost_VM#1_main:java.lang.Thread@Localhost_VM#1_ID#1: :PrintMessage@Localhost_VM#1_ID#-1:<init>:false:Exit`

Es finden sich hier, wie schon in der Trace History die Informationen, auf welchem Host (`Localhost`) die virtuelle Maschine (`VM#1`) läuft. Der aufgerufene Thread ist `java.lang.Thread` und besitzt die Id `1`. Der Name des aufgerufenen Objekts ist `PrintMessage`. Zu diesem Objekt werden ebenfalls der Host, die virtuelle Maschine und die Id (`64`) angegeben. Die aufgerufene Methode ist `init`. Da die Methode nicht „`synchronized`“ ist, wird der Parameter `false` mit angegeben. Ist eine Methode „`synchronized`“, so wird der Parameter `true` angegeben. `Enter` bedeutet, dass die Methode betreten wird. Der zweite Eintrag der Log Datei ist fast identisch mit dem vorher gehenden. Durch `Exit` wird deutlich gemacht, dass die Methode verlassen wird.
Die Log Datei wird dazu genutzt um die Deadlock Erkennung mittels UML-Diagrammarten möglich zu machen und wird mit der Java Platform Debugger Architecture (JPDA) [10] erstellt. Ein Debugger ist ein Programm, das den Code eines anderen Programms während dessen Laufzeit schrittweise durchläuft, um Fehler zu finden. Dabei werden Informationen, die der Benutzer selber festlegen kann, in eine Trace Datei geschrieben.
Aus diesen gewonnenen Informationen können nun UML-Diagramme zur Verdeutlichung einer möglichen Deadlock Situation erstellt werden. UML-Sequenzdiagramme sind besonders dazu geeignet, eine zeitliche Reihenfolge wieder zu geben. UML-Kollaborationsdiagramme werden um stereotypisierte Links und Stereotypen für zeitweilig aktive Objekte erweitert, damit die Deadlocksituation besser analysiert werden kann. Es existieren zwei Arten von stereotypisierten Links. Ein Link, der zu einem gesperrten Objekt führt, wird mit „`lock`“ gekennzeichnet, ein Link zu einem angeforderten Objekt wird mit „`acquires`“ verdeutlicht. Objekte, die nur zeitweilig aktiv sind, erhalten die Notiz „`temporary active`“. Es besteht die Möglichkeit diese aus dem Diagramm auszublenden, um die Darstellung so übersichtlicher zu gestalten. Des Weiteren werden Filter eingeführt, die eine Deadlock Situation hervorheben und alle anderen Objekte und Links, die nicht dazu gehören auszublenden. Für weitere Informationen siehe [25].
Einige Konzepte aus JaVis sind für diese Studienarbeit ungeeignet. Die erstellte Log Datei kann in dieser Weise nicht übernommen werden, da keine Zeitstempel mit protokolliert werden. Bei Real-Time Statecharts handelt es sich aber um zeitkritische Systeme, so dass die Information wann was passiert für die Trace Generierung aus einem Real-Time Statechart wichtig ist.
Des Weiteren tritt durch den Einsatz eines Debuggers eine Verzögerung auf. Diese Verzögerung würde bei der Erstellung der Trace Datei das Verhalten des Real-Time Statecharts verfälschen, da dieses auf Grund der Trace Erstellung verzögert ablaufen würde. Zudem besteht die Möglichkeit aus Real-Time Statecharts Java RT-Code zu generieren, so dass es sich anbietet diesen so zu modifizieren, dass dadurch eine Trace Datei erstellt werden kann.
Auch die Darstellung kann so in dieser Studienarbeit nicht übernommen werden. Die Trace History, die die Methodenaufrufe auflistet, und der Application State, der alle Klassen mit den dazugehörigen Methoden anzeigt, werden in dieser Studienarbeit grafisch durch UML-Sequenzdiagramme und Real-Time Statecharts dargestellt.

Zudem benutzt JaVis einen Filter, der alle Objekte und Links herausfiltert, die nichts mit einer Deadlock Situation zu tun haben. Dies kann ebenfalls nicht übernommen werden, da in dieser Studienarbeit alle Informationen angezeigt werden sollen und nicht nur Deadlock Situationen hevorgehoben werden.
Auf der Oberfläche von JaVis ist eine Zeitleiste (siehe Abbildung 11) integriert, die es dem Benutzer ermöglicht den Ablauf zu verlangsamen bzw. zu beschleunigen.

Abbildung 11 - Zeitleiste

Eine solche Zeitleiste ist auch für die Visualisierung des Verhaltens des Real-Time Statecharts sinnvoll, da sich die Zustandswechsel sehr oft im Millisekunden Bereich abspielen und es für den Benutzer schwer nachzuvollziehen ist, wann welche Transition geschaltet hat und der Zustand gewechselt wurde.

3.2 Rational Rose RealTime

Rational Rose RealTime [15] in der Version 2002.05.02 ist eine Entwicklungsumgebung, mit der Real-Time Software modelliert werden kann. Das Tool wurde von der Firma Rational Software Corporation entwickelt, die mittlerweile zu IBM gehört.
Mittels Rational Rose RealTime wird es Entwicklern ermöglicht, Modelle eines Real-Time Softwaresystems auf Basis von UML- und ROOM-Modellen [18] zu erstellen. Das Tool beinhaltet laut [15] folgende Grundfunktionen:

1) Darstellung von UML- und ROOM-Modellen
2) Generieren von Code (C, C++, RTJava) für die erstellten Modelle
3) Ausführen, Testen und Debuggen der Modelle
4) Nutzung von CM-Systemen um die Entwicklung im Team zu unterstützen

Für diese Studienarbeit ist im Besonderen die Funktion des Testens der Modelle von Interesse. Rational Rose RealTime bietet die Möglichkeit Statecharts zu visualisieren. Da die Konzepte der Statecharts auf Real-Time Statecharts übertragen werden können, können Ideen dieser Visualisierung in der Studienarbeit verwendet werden.
Abbildung 12 zeigt ein mit Rational Rose RealTime erstelltes Capsule Diagram. Eine Capsule besitzt alle Eigenschaften einer Klasse und kann um Ports erweitert werden. Ein Port regelt die Kommunikation aufgrund eines Protokolls zwischen zwei Capsules. Das Protokoll gibt die Signale an, die ein Port senden und empfangen kann. Für weitere Informationen siehe [15].
Das Capsule Diagram aus Abbildung 12 besitzt die Capsules `ST`, `StatechartA` und `StatechartB`, sowie das Protokoll `STProtocol`. Die Capsule `ST` ist durch eine Komposition mit den beiden anderen Capsules verbunden und besitzt zwei Ports: `frame` und `timer`. Die Typen dieser beiden Ports sind Standardtypen aus Rational Rose RealTime und nutzen eine Bibliothek mit diversen Funktionen, die von der Software zur Verfügung gestellt wird. Da zwischen den Capsules `ST` und `StatechartA` bzw. `StatechartB` eine Komposition besteht, ist es der Capsule `ST` mittels des Ports `frame` möglich, die beiden anderen Capsules `StatechartA` und `StatechartB` zu erzeugen bzw. zu löschen. Der Port `timer` vom Typ `Timing` ist dafür zuständig, das Signal `timeout` zu empfangen.

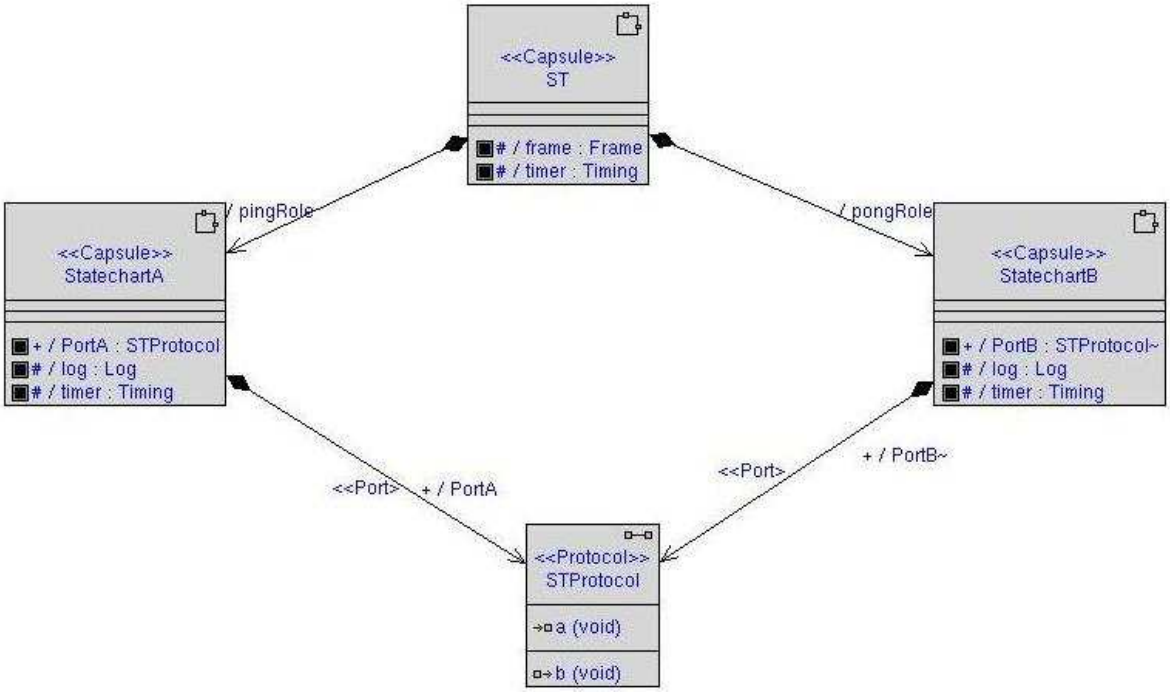

Abbildung 12 - Capsule Diagram

Die beiden Capsules `StatechartA` und `StatechartB` besitzten beide jeweils drei Ports. Die Ports `log` und `timer` sind von Rational Rose RealTime vordefinierte Ports, wobei `log` es ermöglicht Nachrichten auf der Konsole auszugeben. Die Ports `PortA` und `PortB` sind vom Typ `STProtocol`, das ebenfalls in Abbildung 12 dargestellt ist. Das `STProtocol` definiert zwei Signale, das Eingangssignal `a` und das Ausgangssignal `b`. Somit können `PortA` und `PortB` diese beiden Signale senden bzw. empfangen.
Jede dieser Capsules besitzt ein Statechart. Das Statechart von `ST` ist in Abbildung 13 dargestellt und sorgt dafür, dass sich die Capsule bis zu einem `timeout`, das von dem System gesendet und von dem Port `timer` empfangen wird, im Zustand `Running` befindet. Auf Grund der Komposition zwischen `ST` und `StatechartA` bzw. `StatechartB` werden bei einem Zustandswechsel des Statecharts der Capsule `ST` vom Startzustand in den Zustand `Running`, die Capsules `StatechartA` und `StatechartB`, sowie die zugehörigen Statecharts initialisiert. Beim Betreten des Zustands `Shutdown` werden auch hier auf Grund der Komposition die beiden Capsules `StatechartA` und `StatechartB` zerstört.

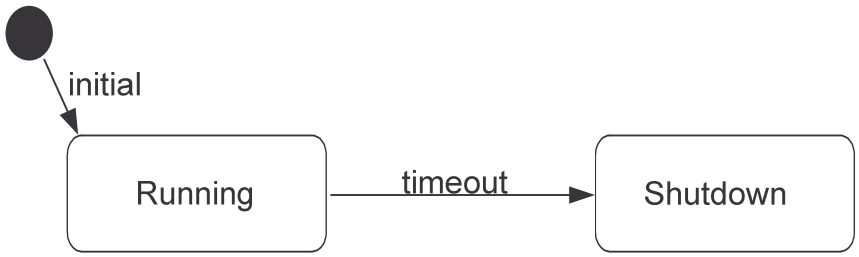

Abbildung 13 - Statechart der Capsule ST

Abbildung 14 zeigt die beiden Statecharts der Capsules `StatechartA` und `StatechartB`. Diese Statecharts sind im Vergleich zu dem Einführungsbeispiel leicht modifiziert, da Rational Rose RealTime nach [1] kein after-Konstrukt bei der Modellierung

von Statecharts zur Verfügung stellt. Das obere Statechart gehört zu der Capsule `StatechartA` und das untere zu der Capsule `StatechartB`.

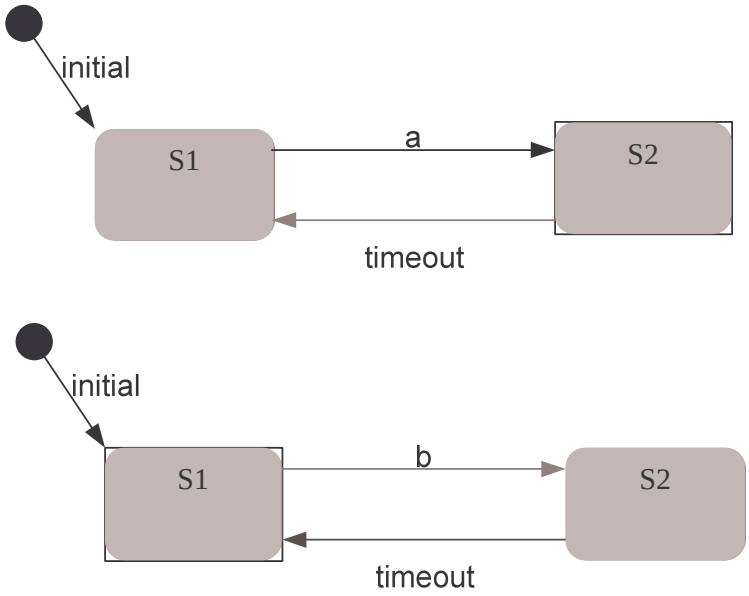

Abbildung 14 - Visualisierung der Statecharts

Wird dieses Modell nun um eine Komponente, die beschreibt wie die Capsules und Protokolle kompiliert werden, und um einen Knoten, der die Ausführung der Komponenteninstanz realisiert, erweitert, so kann das fertige Modell ausgeführt werden.
Zu Anfang befinden sich beide Statecharts im Startzustand. Beim Schalten der Transition `initial` wechseln beide Statecharts in den Zustand `S1`. Hierbei wird das Signal `b` vom `StatechartB` gesendet. Da das Signal über den Port `PortB`, der dem Protokoll `STProtocol` entspricht, gesendet wird, empfängt der `PortA` und somit auch das `StatechartA` das Signal. Damit findet ein weiterer Zustandsübergang statt, so dass sich `StatechartA` in dem Zustand `S2` befindet. Nach einer Zeitspanne von zehn Sekunden, sendet das System das Signal `timeout`, das von `StatechartA` empfangen wird. Als Reaktion darauf sendet `StatechartA` das Signal `a`, welches den Zustandsübergang von `S1` nach `S2` in `StatechartB` auslöst. Nach weiteren 10 Sekunden wird wieder das Signal `timeout` empfangen und `StatechartB` wechselt von `S2` nach `S1`. Während des Schaltvorgangs wird `b` an `StatechartA` gesendet, welches zu einem weiteren Zustandsübergang führt.
Anhand von Abbildung 14, die die Visualisierung der Statecharts `StatechartA` und `StatechartB` zeigt, lässt sich zudem erkennen, dass sich `StatechartA` im Zustand `S1` befindet und der Zustandsübergang durch die Transition `timeout` ausgelöst wurde. `StatechartB` befindet sich in Zustand `S2`, ausgelöst durch die Transition `a`.
Die Visualisierung der Statecharts ist von besonderem Interesse und wird in dieser Studienarbeit für Real-Time Statecharts übernommen. Da das Konzept der Real-Time Statecharts an das der Statecharts angelehnt ist, ist es ebenfalls möglich den aktuellen Zustand bzw. die aktuelle Transition hervorzuheben. Bei der Ausführung der Statecharts in Rational Rose RealTime sind immer der aktuelle Zustand und die Transition, die für diesen Zustandsübergang verantwortlich war, markiert. Dies wird in dieser Studienarbeit nicht übernommen. Es wird immer nur der aktuelle Zustand bzw. nur die aktivierte Transition farblich gekennzeichnet. Der Grund dafür liegt darin, dass bei Statecharts ein

Zustandsübergang statt findet, der keine Zeit benötigt. Bei Real-Time Statecharts hingegen ist durch die Deadline vorgeschrieben, wie lange der Zustandsübergang dauert. Würde die Visualisierung nun sowohl den Zustand, als auch die Transition markieren, kann der Nutzer nicht mehr erkennen, wie lange das Schalten der Transition dauerte.

4. Trace Generierung

Im Folgenden wird die Erstellung einer Trace Datei für eine offline Visualisierung beschrieben. Dazu wird in 4.1 zuerst untersucht, mit welcher Methode eine Trace Datei generiert werden kann. Anschließend wird eine Ist-Analyse des von Fujaba generierten Codes von Real-Time Statecharts durchgeführt. In 4.3 wird beschrieben, um welche Informationen und an welchen Stellen der generierte Code erweitert wird. Kapitel 4.4 beschreibt die generelle Anpassung der Codegenerierung. Die Umsetzung der Trace Erstellung, die den Vorgang vom Schreiben der Trace Daten in ein Array bis zum Schreiben in eine Datei realisiert, wird in 4.5 erläutert. Im Anschluss wird erklärt, in wie fern es möglich ist, Trace Dateien von verschiedenen Real-Time Statecharts zu mischen.

4.1 Möglichkeiten der Trace Generierung

Es existieren verschiedene Möglichkeiten, um eine Trace Datei zu generieren. Java bietet mit der Klasse `Runtime` [23] aus dem Paket `java.lang` eine davon. Mit dem Aufruf der Methode `traceMethodCalls()` werden Methodenein- und austritte protokolliert [25]. Da die Zeitpunkte der Methodenein- und austritte nicht mit gespeichert werden, ist dieses Konzept für die Generierung einer Trace Datei, so wie hier benötigt, nicht anwendbar.
Eine weitere Möglichkeit der Protokollierung bietet die Java Platform Debugger Architektur (JPDA), die für die Erstellung der Trace Datei in JaVis [25] genutzt wird. JPDA [10] besteht aus dem Java Virtual Machine Debugger Interface (JVMDI), dem Java Debug Wire Protocol (JDWP) und dem Java Debug Interface (JDI).
Der Debugger ist ab dem JDK1.3 [10] zu finden und ermöglicht es, andere Programme auf Fehler zu untersuchen. Der JPDA verzögert allerdings die Ausführung. Da in dieser Studienarbeit zeitkritische Systeme behandelt werden, kann diese Art der Protokollierung nicht genutzt werden.
Zudem ist es möglich eine Trace Datei zu erstellen, indem eine Codeerweiterung der Real-Time Statecharts vorgenommen wird. Diese Methode ist für diese Studienarbeit am sinnvollsten. Zum einen existiert bereits eine Codegenerierung für Real-Time Statecharts, die verändert werden kann, und zum anderen findet in diesem Konzept keine so große Zeitverzögerung, wie zum Beispiel beim JDPA, statt. Die Code-Generierung für Real-Time Statecharts [4] wurde im Rahmen einer Diplomarbeit [4] in der AG Softwaretechnik entwickelt und baut auf dem Konzept der Codegenerierung in Fujaba auf.
Für die Codeerweiterung existieren nach [17] zwei grundlegende Methoden: `content-based` bzw. `data-driven execution` und `ordering-based` bzw. `control-driven replay`. Bei der ersten Methode werden alle möglichen Ausführungen des Programms in einer Trace Datei gespeichert. Dies hat zur Folge, dass für die wiederholte Ausführung alle Daten zur Verfügung stehen. Der Nachteil besteht jedoch darin, dass sehr viele Daten gespeichert werden, so dass die Trace Datei immens groß wird. Bei der zweiten Methode, dem `ordering-based` bzw. `control-driven replay`, werden lediglich die Daten in einer Trace Datei abgelegt, die nicht wieder reproduziert werden können. Dies liefert den Vorteil, dass die Größe der Trace Datei im Vergleich zur ersten Methode viel geringer ist.
In der Praxis wird häufig ein Mix aus beiden Methoden eingesetzt, um eine möglichst optimale Trace Datei in Größe und Wahl der Daten für jedes Programm zu erhalten.
In dieser Studienarbeit wird bei der Erstellung der Trace Datei die `content-based` bzw. `data-driven execution` Methode zu Grunde gelegt, da die produzierten Daten bei

jedem Real-Time Statechart unterschiedlich sind, so dass durch die zweite Methode keine Verringerung der Größe der Trace Datei möglich ist.

4.2 IST- Analyse des generierten Codes

Um Java Real-Time Code [2] aus einem Real-Time Statechart zu generieren, müssen vorher folgende Schritte durchgeführt werden:

1) Modellierung des Real-Time Statecharts
2) Bestimmen der fehlenden Wcets durch ein externes Dokument, das z.B. durch ein Wcet-Analyse-Tool erzeugt wurde
3) Analyse und Behebung von Zeitinkonsistenzen
4) Abbildung des Real-Time Statecharts auf verschiedenen periodische und aperiodische Threads

Nun ist es möglich Code für das Real-Time Statechart zu erzeugen. Anschließend wird eine Schedulinganalyse durchgeführt. Diese wird benötigt, wenn mehrere Real-Time Statecharts auf demselben System ausgeführt werden.
Fujaba ermöglicht es Code für nicht-hierarchische und hierarchische Real-Time Statecharts zu erzeugen. Im Folgenden wird die Codegenerierung für nicht-hierarchische Real-Time Statecharts beschrieben.
Bei der Codegenerierung wird für jedes Real-Time Statechart ein Hauptthread erzeugt, der unter Umständen weitere aperiodische Threads starten kann. Der Hauptthread ist für die Aktionen und Reaktionen des Real-Time Statecharts zuständig. Er verwaltet alle Zustände und prüft periodisch, welche Transitionen aktiviert werden können und löst den Schaltvorgang aus. Dabei wird die Periode so kurz gewählt, dass alle Deadlines eingehalten werden. Zusätzlich wird eine weitere Klasse, `Main.java`, erzeugt, die die Real-Time Statecharts im Immortal Speicherbereich startet und aufgrund der Schedulinganalyse jedem Thread seine Priorität zuordnet. Somit wird gewährleistet, dass mehrere Real-Time Statecharts parallel laufen können.
Abbildung 15 zeigt ein stark vereinfachtes Real-Time Statechart, für das im Folgenden der generierte Code vorgestellt wird. Das Real-Time Statechart besteht aus vier Zuständen und vier Zustandsübergängen. Alle Eigenschaften, wie Guard, Event, Deadline etc., sind nicht dargestellt, da sie für die Trace Generierung nicht benötigt werden und nur die Übersichtlichkeit des Real-Time Statecharts stören würden.

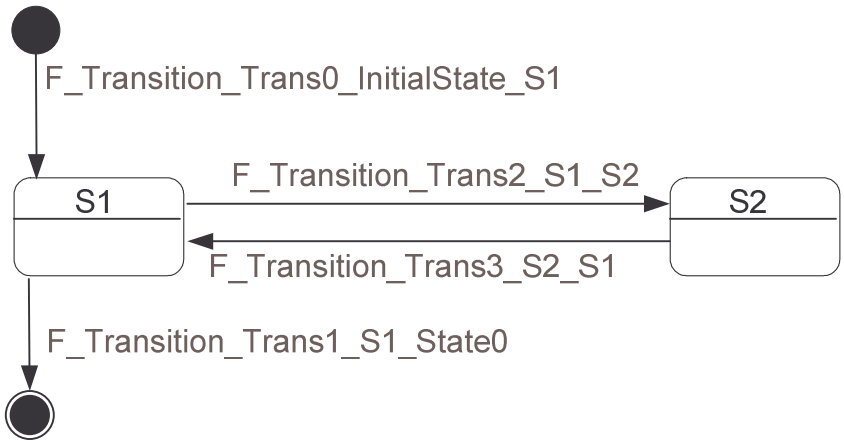

Abbildung 15 - Real-Time Statechart

Die Implementierung der Real-Time Statecharts erfolgt durch Fallunterscheidung. Dabei wird ein Handlerobjekt erzeugt, dass für jeden Zustand und jede Transition eine Konstante definiert.

```
// Konstanten für die einzelnen Zustände
public static final int F_STATE_InitialState = 0;
public static final int F_STATE_S1 = 1;
public static final int F_STATE_S2 = 2;
public static final int F_STATE_State0 = 3;
```

Beispiel 1 - Definition der Zustände

Beispiel 1 zeigt die Definition der Konstanten für die vorhandenen Zustände. Das Real-Time Statechart aus Abbildung 15 besitzt folgende vier Zustände:

1) `InitialState` (Startzustand)
2) `S1`
3) `S2`
4) `State0` (Endzustand)

Analog dazu werden in Beispiel 2 die Konstanten der Transitionen definiert.

```
//Konstanten für die einzelnen Transitionen
public static final int F_TRANSITION_TRANS0_InitialState_S1 =
0;
public static final int F_TRANSITION_TRANS1_S1_State0 = 1;
public static final int F_TRANSITION_TRANS2_S1_S2 = 2;
public static final int F_TRANSITION_TRANS3_S2_S1 = 3;
```

Beispiel 2 - Definition der Konstanten

Das Real-Time Statechart aus Abbildung 15 besitzt folgende vier Zustandsübergänge zwischen den Zuständen aus Beispiel 1:

1) `F_TRANSITION_TRANS0_InitialState_S1`: Transition zwischen dem Startzustand `InitialState` und `S1`
2) `F_TRANSITION_TRANS1_S1_State0`: Transition zwischen dem Zustand `S1` und dem Endzustand `State0`
3) `F_TRANSITION_TRANS2_S1_S2`: Transition zwischen den Zuständen `S1` und `S2`
4) `F_TRANSITION_TRANS3_S2_S1`: Transition zwischen den Zuständen `S2` und `S1`

Der aktuelle Zustand ist in der Integer Variable `state` gespeichert. Durch eine `switch-case` Anweisung kann bestimmt werden, welche Aktion bei den Methoden `entryAction()`, `doAction()` und `exitAction()` ausgeführt wird.

```
//Methode entryAction
public void entryAction(int state)
{
   switch(state){
     case F_STATE_InitialState:
```

```
            //Ausführen der entry()-Methode des Zustandes
            InitialState (Startzustand)
            break;
      case F_STATE_S1:
            //Ausführen der entry()-Methode des Zustandes S1
            break;
      case F_STATE_S2:
            //Ausführen der entry()-Methode des Zustandes S2
            break;
      case F_STATE_State0:
            //Ausführen der entry()-Methode des Zustandes State0
            (Endzustand)
            System.exit(0);
            break;
   }
}
```

Beispiel 3 - entryAction

Beispiel 3 zeigt einen Codeausschnitt der Methode `entryAction(int state)`. Diese Methode wird bei Eintritt in einen Zustand aufgerufen. Dazu wird mittels einer `switch-case` Anweisung überprüft, welcher Zustand der Aktuelle ist. Genutzt werden dafür die vorher definierten Konstanten. Ist der Wert der Integer Variable `state` zum Beispiel `state == 1`, so wird die `entry()`-Methode des Zustandes `S1` aufgerufen. Analog zu der Methode `entryAction(int state)` existieren die Methoden `doAction(int state)` und `exitAction(int state)`. Die Methode `doAction(int state)` wird ausgeführt, wenn sich das Real-Time Statechart in einem Zustand befindet und ist für die Erstellung der Trace Datei nicht von Bedeutung, da sie keine Informationen bezüglich eines Zustandswechsel oder eines Schaltvorgangs einer Transition erhält.
Beispiel 4 zeigt einen Ausschnitt der Methode `exitAction(int state)`.

```
//Methode exitAction
public void exitAction(int state)
{
   switch(state){
   case F_STATE_InitialState:
        //Ausführen der exit()-Methode des Zustandes
        InitialState (Startzustand)
        break;
   case F_STATE_S1:
        //Ausführen der exit()-Methode des Zustandes S1
        break;
   case F_STATE_S2:
        //Ausführen der exit()-Methode des Zustandes S2
        break;
   }
}
```

Beispiel 4 – exitAction

Wie bei der `entry()`-Methode wird auch hier anhand der Konstanten überprüft, welche Methode bei Austritt aus einem Zustand aufgerufen wird. Es existiert keine Abfrage für den

Zustand `F_State_State0`, da Dieser der Endzustand ist und daher nicht mehr verlassen werden kann, wenn er einmal betreten worden ist.
Ebenso wie bei den `entry()`- und `exit()`-Methoden der Zustände, wird über eine `switch-case` Anweisung bestimmt, welche Transition schaltet. Dieser Schaltvorgang wird in der Methode `run(int transition)` in Beispiel 5 ausgeführt.

```
//Methode run
public void run(int transition)
{
   . . .
   switch(transition){
   . . .
     case F_TRANSITION_TRANS2_S1_S2:
     //Entfernen des Events aus der EventQueue
          FRealtimeEvent e1()=eventQueue.dequeueEvent("e1()");
          //Ausführen der exit()-Methode, so dass der aktuelle
          Zustand bzw. in parallelen Real-Time Statecharts die
          aktuellen Zustände verlassen werden
          exitAction(currentState);
          // Ausführen der Action Methode, die beim Schalten
          der Transition durchgeführt wird
          actionF_TRANSITION_TRANS2_S1_S2();
          //Ausführen der entry()-Methode, so dass der neue
          Zustand bzw. in parallelen Real-Time Statecharts die
          neuen Zustände betreten werden
          entryAction(state2);
          //Zurücksetzen der Uhr t
          t0[F_CLOCK_t] = activationTime;
          //Der Schaltvorgang der Transition ist abgeschlossen
          break;
   . . .
   }
}
```

Beispiel 5 - run

In Beispiel 5 findet der Zustandsübergang durch das Schalten der Transition `F_Transition_Trans2_S1_S2` von dem Zustand `S1` in den Zustand `S2` statt. Bei dem Schaltvorgang wird zuerst das Event, das durch das Schalten der Transition ausgelöst wird, aus einer Schlange mit allen Events gelöscht. Danach wird der aktuelle Zustand verlassen. Realisiert wird dieses durch den Aufruf der Methode `exitAction(currentState)`. Anschließend wird die Action Methode, die durch das Schalten der Transition ausgelöst wird, aufgerufen. Nach dem Schalten der Transition wird der neue Zustand ermittelt und durch die Methode `entryAction(state2)` betreten. Zum Schluss wird ein Uhren Reset durchgeführt. In Beispiel 5 wird die Uhr `t` zurückgesetzt.
Damit bei dem Schaltvorgang die Deadlines eingehalten werden und eine Transition nicht zu schnell schaltet, findet eine Überprüfung statt. Gegebenenfalls wird das Schalten der Transition so lange herausgezögert, bis der Zeitpunkt p_{low} der Deadline erreicht ist.
Der Guard und die Timeguards einer Transition werden nicht in der Methode `run` überprüft. Dafür existieren die Methoden `setGuardTime` und `calculateActivationTime`.

```
//Methode setGuardTime
public void setGuardTime(long t)
{
     . . .
     case F_STATE_S1:
          if ( x==0 )
          {
               if ( guardTime[F_TRANSITION_TRANS1_S1_S2] ==
               INFINITY )
               {
                    guardTime[F_TRANSITION_TRANS1_S1_S2] = t;
               }
          }
          else
          {
               guardTime[F_TRANSITION_TRANS1_S1_S2] = INFINITY;
          }
}
```

Beispiel 6 – setGuardTime

Beispiel 6 zeigt einen Ausschnitt der Methode `setGuardTime`. Mittels einer `switch-case` Anweisung wird für den aktuellen Zustand überprüft, ob der Guard wahr ist. Falls ja, wird die Zeit für den Guard `x==0` auf den Wert der Uhr `t` gesetzt.

```
//Methode calculateActivationTime
public void calculateActivationTime(int transition, long
actualTime, int concurrencySlot)
{
     long lowerBound = 0;
     long long1 = 0;
     switch(transition){
          case F_TRANSITION_TRANS3_S1_State0:
          //Bestimmen der unteren und oberen Grenze
               lowerBound = 10000 + t0[F_CLOCK_t0];
               long1 = 10000 + t0[F_CLOCK_t0];
               if ( lowerBound <= actualTime <= long1 )
               {
                    //Timeguard ist wahr
               }
          break;
          . . .
     }
}
```

Beispiel 7 – calculateActiviationTime

Beispiel 7 beinhaltet einen Ausschnitt der Methode `calculateActivationTime`, die die Timeguards einer Transition überprüft. Wie auch bei der Überprüfung der Guards werden anhand einer `switch-case` Anweisung für die aktuelle Transition die untere und obere Grenze durch eine Addition der aktuellen Zeit, die aus der Uhr `t` hervorgeht, überprüft.

4.3 Erweiterung des generierten Codes

Der im vorangegangenen Abschnitt beschriebene und von Fujaba erzeugte Code für Real-Time Statecharts wird nun so erweitert, dass bei der Ausführung von Real-Time Statecharts Einträge, die wichtige Aktionen beschreiben, in einen Trace geschrieben werden. Die Erweiterung findet an Zustandsübergängen und Schaltvorgängen statt. Relevante Methoden sind dafür: `entryAction()`, `exitAction()`, `run()`, `setGuardTime()` und `calculateActivationTime()`.
Neben den Informationen, welcher Zustand verlassen wird und welcher betreten wird, werden zusätzlich Zeitstempel der einzelnen Aktionen mitprotokolliert. Am Anfang jeder Trace Datei wird der Name des Real-Time Statecharts zur Identifizierung angegeben. Somit ist es möglich, Trace Dateien von mehreren Real-Time Statecharts zu mischen, ohne das dabei verloren geht, welche Informationen zu welchem Real-Time Statechart gehören. Zusätzlich wird am Anfang jeder Trace Datei die Startzeit angegeben.

```
<Name>Controller</Name>
```

Beispiel 8 - Protokollierung des Namens

```
<StartTime>1070712186240</StartTime>
```

Beispiel 9 - Protokollierung des Namens und der Startzeit

Beispiel 8 und Beispiel 9 zeigen die Form der Protokollierung des Namens und der Startzeit in Millisekunden, angefangen am 1.Januar 1970. Die aktuelle Startzeit ist von zwei XML-Tags umgeben. Diese Information ist für das Mischen von mehreren Trace Dateien notwendig. Da es durchaus möglich ist Code mit Trace Informationen für mehrere Real-Time Statecharts zu generieren, die parallel ausgeführt werden können, ist es für die Visualisierung der Ausführung von Nöten eine gesamte Trace Datei für alle Real-Time Statecharts zu erstellen. Um eine genaue Abfolge der einzelnen Aktionen der Real-Time Statecharts festzulegen, muss anhand der individuellen Startzeit berechnet werden, was wann durchgeführt wird. Diese Problematik wird in Kapitel 4.6 genauer beschrieben.
Ein Eintritt in eine Methode, der durch `entryAction()` ausgelöst wird, besitzt einen Trace Eintrag, wie er in Beispiel 10 dargestellt ist.

```
<EnterState>
     <actualState>S2</actualState>
     <EnterTime>13</EnterTime>
</EnterState>
```

Beispiel 10 - Protokollierung eines Zustandeintritts

Der Zustandseintritt wird durch das XML-Tag `<EnterState>` gekennzeichnet. `<EnterState>` beinhaltet zwei weitere XML-Tags: `<actualState>` gibt den Zustand an, der gerade betreten wird und `<EnterTime>` protokolliert, wann der Zustand in Abhängigkeit von der Startzeit betreten wurde. In Beispiel 10 wird der Zustand `S2` zum relativen Zeitpunkt 13 betreten. Dabei muss nicht mitprotokolliert werden welcher Zustand verlassen wurde, da dies durch das vorangegangene XML-Tag `<ExitState>` angegeben wird.

```
<ExitState>
     <oldState>S2</oldState>
     <ExitTime>75</ExitTime>
</ExitState>
```

Beispiel 11 - Protokollierung eines Zustandsaustrittes

Beispiel 11 zeigt den Trace Eintrag, der angibt, dass ein Zustand verlassen wurde. Gekennzeichnet wird dieser durch das XML-Tag `<ExitState>`. Zusätzlich wird durch `<oldState>` angegeben welcher Zustand verlassen wird und durch `<ExitTime>` zu welchem Zeitpunkt dieses geschieht. Der Zeitpunkt gibt auch hier wieder an, wie viel Zeit vom Starten des Real-Time Statecharts bis zu dieser Aktion vergangen ist. Somit wird zum relativen Zeitpunkt `75` der Zustand S2 verlassen.
Zusammen reichen Beispiel 10 und Beispiel 11 aus, um zu protokollieren, welcher Zustand wann und zu welchem Zeitpunkt betreten bzw. verlassen wurde. Jedoch ist es darüber hinaus von Interesse, welche Events, Actions, Clock Resets und Timeguards bei einem Zustandsein- oder austritt und dem Schalten einer Transition stattfinden, da diese das Verhalten eines Real-Time Statecharts unter Umständen so beeinflussen können, dass ein Fehler auftritt
Es wäre auch möglich, diese zusätzlichen Informationen aus der fpr-Datei des zu visualisierenden Real-Time Statecharts zu gewinnen. Allerdings wird damit nicht eindeutig definiert, welche Transition wirklich schaltet. Existieren zum Beispiel zwischen zwei Zuständen zwei mögliche Zustandsübergänge, wobei ein Zustandsübergang durch das Event a und der andere Zustandsübergang durch das Event b ausgelöst wird, wird anhand der fpr-Datei nicht klar, welche Transition schaltet. Daher müssen zusätzliche Informationen, die das Schalten der Transitionen beeinflussen, mit in die Trace Datei aufgenommen werden. Zudem werden diese Informationen auch für die Darstellung des Sequenzdiagramms benötigt. Somit werden die Events, Actions, Clock resets und Timeguards mit in der Trace Datei abgespeichert.
Abbildung 16 zeigt das Real-Time Statechart aus Abbildung 15 mit einem Zustandsübergang von `S1` nach `S2`, der durch einen Timeguard ausgelöst wird und einem Zustandsübergang von S2 nach S1, der durch ein Event, einen Guard und einen Timeguard ausgelöst wird. Bei beiden Zustandsübergängen wird beim Schalten der Transition eine Action Methode aufgerufen.

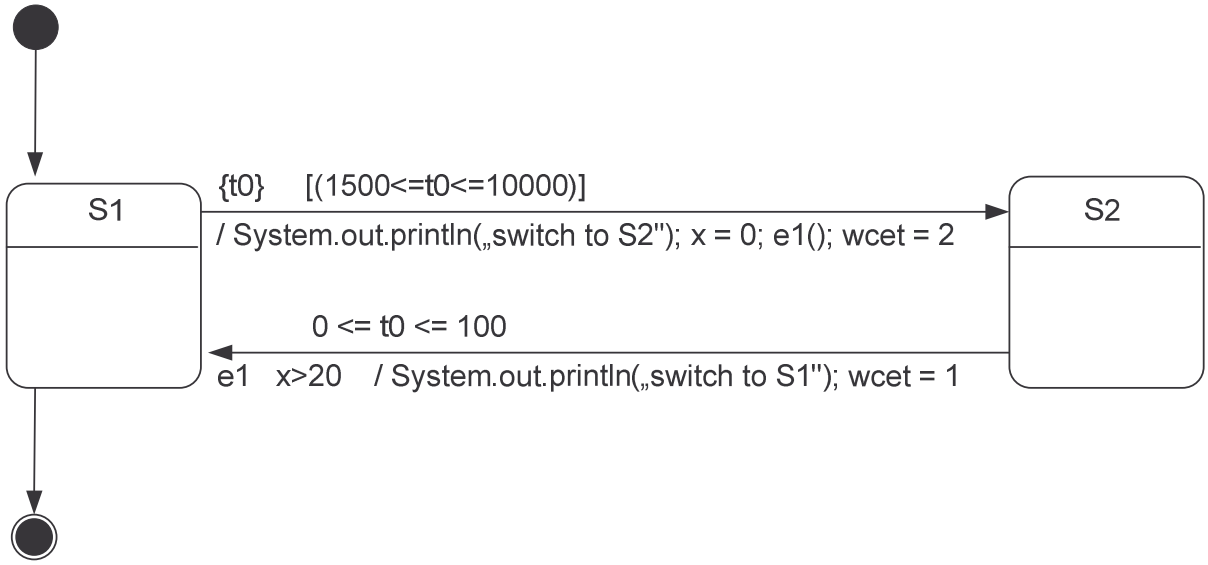

Abbildung 16 - Protokollierung Zustandswechsel

Befindet sich das Real-Time Statechart aus Abbildung 16 im Zustand S2, so kann ein Zustandswechsel stattfinden, wenn der Guard und der Timeguard wahr sind und das Event anliegt. Beispiel 12 zeigt die Protokollierung des Guards. Zum Zeitpunkt `42` liegt der Guard `x > 20` an. Hierbei muss zwingend angegeben werden, bei welcher Transition der Guard wahr wird. Andernfalls wäre es bei der Visualisierung der Ausführung des Real-Time Statecharts nicht möglich festzustellen, was genau bei einer Zustandsaktivierung durchgeführt wird. Daher wird zu Timeguard, Event, Clock Reset und Action ebenfalls die dazugehörige Transition angegeben.

```
<GuardName>TRANS1_S2_S1.x &gt; 20</GuardName>
<GuardTime>42</GuardTime>
```

Beispiel 12 - Protokollierung eines Guards

Anhand von Beispiel 13 ist zu erkennen, dass zum Zeitpunkt `58` eine Überprüfung des Timeguards `1500 <= t0 <= 10000` stattgefunden hat und dieser wahr ist. Damit die Transition schalten kann, muss zusätzlich noch das Event anliegen.

```
<TimeGuardName>
     TRANS2_S1_S2.1500&lt;=t0&lt;=10000
</TimeGuardName>
<Time>58</Time>
```

Beispiel 13 - Protokollierung eines Timeguards

Beispiel 14 zeigt, dass das Event `e1()` zum Zeitpunkt `74` anliegt. Damit ist nun die Transition, die den Zustandsübergang von `S2` nach `S1` reguliert, aktiviert und kann daher auch schalten.

```
<EventName>TRANS1_S2_S1.e1()</EventName>
<EventTime>74</EventTime>
```

Beispiel 14 - Protokollierung eines Events

Bei diesem Zustandsübergang wird zusätzlich der Seiteneffekt ausgelöst. Die Action Methode wird zum Zeitpunkt `78` ausgeführt. Protokolliert wird dabei der Name der Action und der Zeitpunkt des Methodenaufrufs, wie Beispiel 15 zeigt.

```
<ActionName>
     TRANS2_S1_S2.actionF_TRANSITION_TRANS2_S1_S2()
</ActionName>
<ActionTime>78</ActionTime>
```

Beispiel 15 - Protokollierung einer Action

Findet ein Zustandsübergang von S1 nach S2 statt, so wird beim Schalten der Transition zusätzlich noch ein Uhren Reset durchgeführt. Anhand von Beispiel 16 lässt sich erkennen, dass zum Zeitpunkt `157` die Uhr `t0` den Wert `0` annimmt.

```
<ClockResetName>TRANS2_S1_S2.t0</ClockResetName>
<ClockResetTime>157</ClockResetTime>
```

Beispiel 16 - Protokollierung eines Uhren Resets

Nachdem in diesem Kapitel beschrieben wurde, wie die Trace Datei genau aussieht, wird im nächsten Kapitel die Umsetzung der Erweiterung der aktuellen Codegenerierung beschrieben, so dass es möglich ist, eine zusätzliche Trace Datei bei der Ausführung des Real-Time Statecharts zu generieren.

4.4 Anpassung der Codegenerierung

Das Real-Time Statechart Plugin bietet bereits eine Codegenerierung [4], die die Codegenerierung von Fujaba nutzt. Beim Starten des Plugins registriert dieses eine für die Codegenerierung zuständige Klasse in der `Chain of Responsibility` [13]. Die `Chain of Responsibility` beinhaltet alle für die Codegenerierung zuständigen Klassen. Wird seitens des Benutzers die Aufforderung zur Codegenerierung für ein Diagramm gestartet, so wird in der `Chain of Responsibility` anhand des Diagrammtyps die zuständige Klasse für die Codegenerierung identifiziert. Jede dieser Klassen besitzt eine Methode `generateSourceCode()`, die die Codegenerierung durchführt. Zusätzlich werden der `Chain of Responsibility` alle zu generierenden Sprachen mit übergeben. Das Real-Time Statechart Plugin generiert Code für `rtjava`.
Wird die Codegenerierung aus Fujaba heraus aufgerufen, so sucht die Klasse, die durch das ActionEvent ausgelöst wurde, anhand des vorliegenden Diagramms die passende Klasse für die Codegenerierung. Die Klasse `UMLRealtimeStatechartJavaHandler`, die Code für ein Real-Time Statechart generiert, besitzt eine Methode `generateSourceCode()`. In dieser Methode wird der gesamte Code für das Diagram generiert und in einer Datei gespeichert.
Um Code mit zusätzlichen Informationen zu generieren, gibt es eine zweite Klasse im Real-Time Statechart Plugin, die von der zuständigen Klasse für die Codegenerierung erbt und diese ersetzt. Die Methode für die Codegenerierung in der neuen Klasse `RTSCVisualizationJavaHandler` wird ebenfalls durch ein ActionEvent aufgerufen.
Somit hat der Nutzer mit Fujaba die Möglichkeit, Code mit Trace Informationen für ein Real-Time Statechart zu generieren.

4.5 Umsetzung der Trace Erstellung

Um eine Trace Datei mit den vorher beschriebenen Informationen zu erstellen, muss sowohl die Codegenerierung erweitert, als auch ein Mechanismus für die Erstellung der Trace Datei entwickelt werden. Um Informationen aus der Ausführung des Real-Time Statecharts zu gewinnen würde es theoretisch auch ausreichen, diese als Text auf dem Bildschirm auszugeben. Allerdings können die Informationen dann nicht mehr weiter verarbeitet werden und es wäre daher nicht möglich, die Ausführung des Real-Time Statecharts in Fujaba zu visualisieren. Aus diesem Grund wird eine Trace Datei erstellt.
Dabei wird der Inhalt der Trace Datei zuerst in einem speicherplatzbegrenzten Array zwischengespeichert und anschließend in eine Datei geschrieben. Dieser Mechanismus ist erforderlich, da nur begrenzter Speicherplatz zur Verfügung steht und es daher nicht möglich ist, alle Daten während der Ausführung des Real-Time Statecharts komplett in das Array zu schreiben und erst nach der Ausführung in eine Datei zu übermitteln.

Ferner werden die Daten nur dann in die Datei geschrieben, wenn dafür Zeit zur Verfügung steht. Nur so wird garantiert, dass das Echtzeitverhalten nicht beeinflusst wird. Realisiert wird dieses dadurch, dass der Thread, der für die Ausführung des Real-Time Statecharts zuständig ist, eine höhere Priorität besitzt als der Thread, der das Schreiben der Daten organisiert.
Der genaue Ablauf für die Erstellung der Trace Datei ist in Abbildung 17 dargestellt. Die Datei `Main.java` startet den Real-Time Thread und damit die Ausführung des Real-Time Statecharts. Die Java-Klasse `RingPuffer` ist ein Thread und wird von dem Real-Time Thread gestartet. `RingPuffer` verwaltet ein Array mit allen Trace Einträgen und startet den Thread `WriteArrayIntoFile`. Dieser Thread liest die Daten aus dem Array aus und schreibt sie in eine Datei. Ein Datensatz, im Folgenden Element genannt, besteht aus einem Trace Eintrag, der ein komplettes Ereignis, wie z.B. das Anliegen eines Events, beschreibt.

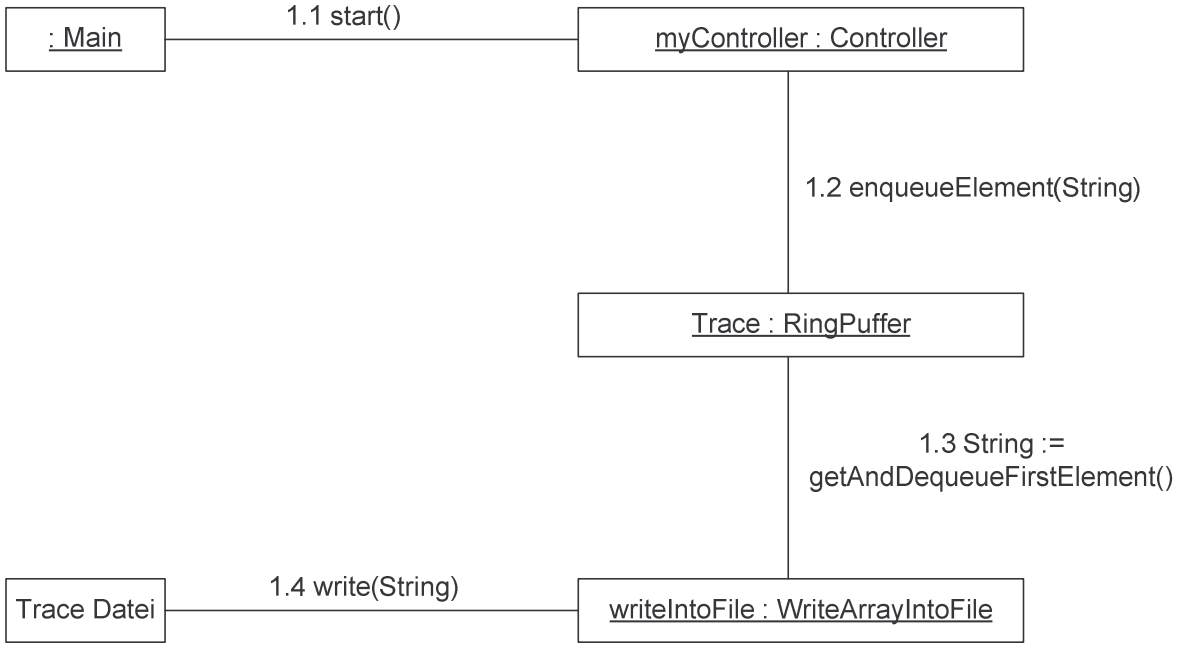

Abbildung 17 - Entstehung der Trace Datei

Abbildung 18 zeigt das zweidimensionale Array aus der Klasse `RingPuffer`, das alle Informationen verwaltet. Die Klasse `FMainThread`, die für die Ausführung des Real-Time Statecharts benötigt wird, initialisiert den Speicherbereich und legt auch dessen Größe fest. Anhand dieses Wertes wird die Größe des Arrays, das durch den zur Verfügung stehenden Speicher in seiner Größe begrenzt ist, bestimmt.
Der Thread `RingPuffer` bietet neben einem Konstruktor, der das Array initialisiert und den Thread `WriteFileIntoArray` startet, eine Methode um Elemente an das Ende des Arrays einzufügen, eine weitere um das älteste Element aus dem Array zu löschen und zurückzugeben und schließlich eine Methode um den Thread zu beenden. Wird ein neues Element im Array eingefügt und es ist kein Platz mehr vorhanden, so wird das älteste Element gelöscht und das neue Element wird an dessen Stelle eingefügt. Das Array arbeitet dabei wie ein Ringpuffer.
Zusätzlich existieren in der Klasse `RingPuffer` zwei Integer-Variablen. Eine zählt die Anzahl der Elemente, die aus Platzgründen aus dem Array gelöscht werden müssen, die Andere zählt die Anzahl aller Elemente. Somit ist es möglich zu berechnen, wie viel Prozent aller Elemente nicht in der Trace Datei vorhanden sind.

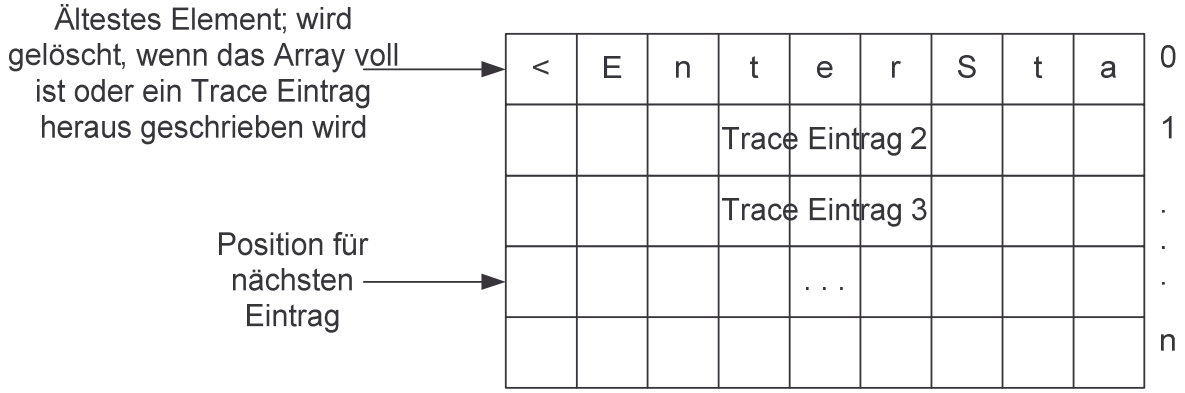

Abbildung 18 - Array mit Trace Einträgen

4.6 Mischen mehrerer Trace Dateien

Da Real-Time Statecharts untereinander kommunizieren können und daher parallel ausgeführt werden, muss ein Mechanismus existieren, der es ermöglicht die Trace Dateien mehrerer Real-Time Statecharts zu mischen und daraus eine gesamte Trace Datei zu erstellen.
Beispiel 17 und Beispiel 18 zeigen einen Ausschnitt zweier Trace Dateien. Um diese Teile zu mischen reicht es nicht aus, dieses anhand ihrer protokollierten Zeit zu vollziehen. Der Grund dafür ist, dass aus den einzelnen Protokollierungen nicht klar wird, welches Real-Time Statechart wann ausgeführt wurde. Somit muss das Mischen mehrerer Trace Dateien in Abhängigkeit von der Startzeit erfolgen.

```
<EventName>TRANS1_S2_S1.e1()</EventName>
<EventTime>1821</EventTime>
```

Beispiel 17 - Ausschnitt Trace Datei Real-Time Statechart 1

```
<GuardName>TRANS1_S2_S1.x > 20</GuardName>
<GuardTime>1708</GuardTime>

<EventName>TRANS1_S2_S1.e1()</EventName>
<EventTime>1900</EventTime>
```

Beispiel 18 - Ausschnitt Trace Datei Real-Time Statechart 2

Allerdings garantiert auch das Mischen in Abhängigkeit von der Startzeit nicht, dass die Real-Time Statecharts wirklich in dieser Reihenfolge ausgeführt werden. Wie in dem Kapitel über Grundlagen beschrieben, besitzen verteilte Systeme keine globale Uhr, sondern jedes System besitzt eine Lokale. Darüber hinaus existieren nur begrenzte Synchronisationsmöglichkeiten für lokale Uhren. Dies hat zur Folge, dass anhand der protokollierten Startzeit nicht angenommen werden kann, dass die angegebene Uhrzeit allgemein gültig ist. Wird also die Ausführung zweier Real-Time Statecharts auf zwei unterschiedlichen Systemen gestartet, wobei die jeweiligen protokollierten Startzeiten identisch sind, ist auf Grund der mangelnden Synchronisationsmöglichkeiten nicht garantiert, dass die Ausführung wirklich zeitgleich startet.
In [9] werden zwei Graphen, der Object Event Graph und der Process Event Graph, vorgestellt, mit denen es möglich wird, kausale Abhängigkeiten zwischen Events herzustellen. Diese Methode ist für diese Studienarbeit von Interesse, da dadurch

festzustellen wäre, welche Protokollierungen voneinander abhängen und somit in eine zeitliche Reihenfolge gebracht werden können. Das Mischen der Trace Dateien geschieht damit nicht nur in Abhängigkeit von der Starzeit und den protokollierten Zeiten, sondern auch anhand der kausalen Abhängigkeiten.
Der Object Event Graph ist ein gerichteter, linearer Graph. Er besteht aus Knoten, die die Reihenfolge der Events repräsentieren und aus beschrifteten Kanten. Die Reihenfolge der Events wird hierbei aus verschiedenen Subgraphen erstellt.
Der Process Event Graph stellt ein Szenario dar und besteht aus Events, die diesem Szenario angehören. Die Events sind durch Knoten und Kanten, die die direkten Verbindungen zwischen den Events repräsentieren, dargestellt. Der Zusammenschluss beider Graphen ergibt den genauen Ablauf der einzelnen Events, die zwischen den verschiedenen Systemen hin und her geschickt werden. Zusätzlich wird in [9] eine Tabelle aufgestellt, die für bestimmte Szenarien beschreibt, wer Sender und wer Empfänger ist und um was für eine Art von Event es sich handelt.
Mit Hilfe dieser beiden Graphen ist es also möglich ein genaueres Mischen von Trace Dateien zu ermöglichen. Allerdings wird in dieser Studienarbeit das Mischen anhand der Startzeiten und der protokollierten Zeiten realisiert, da davon ausgegangen wird, dass der Zeitunterschied zwischen den einzelnen Uhren nicht zu groß ist und somit diese Methode ausreicht.

5. Visualisierung

Im Folgenden wird beschrieben, wie die generierte Trace Datei visualisiert wird. Dabei wird sowohl ein Sequenzdiagramm, als auch ein Real-Time Statechart dargestellt. Zusätzlich hat der Benutzer die Möglichkeit durch eine Interaktion seinerseits zwischen den beiden Diagrammarten zu wechseln.

5.1 Real-Time Sequenzdiagramm

Zurzeit wird an der AG Softwaretechnik die Darstellung von Sequenzdiagrammen in Fujaba entwickelt. Die Notation entspricht dem zurzeit aktuellen UML-Standard 1.5.

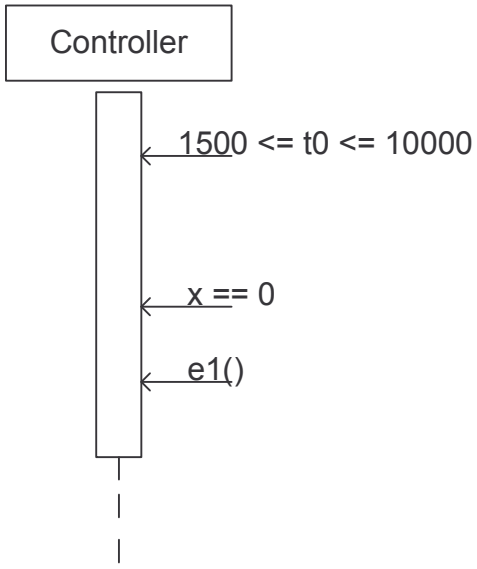

Abbildung 19 - Sequenzdiagramm

Abbildung 19 zeigt ein Sequenzdiagramm wie es im Moment in Fujaba erstellt werden kann. Das Sequenzdiagrammobjekt `Controller` mit der dazugehörigen Lebenslinie repräsentiert das Real-Time Statechart. Die einzelnen Nachrichten entsprechen den Dingen, die einen Zustandsübergang in einem Real-Time Statechart auslösen. Es ist zu erkennen, dass zuerst der Timeguard `1500 ≤ t0 ≤ 10000` anliegt. Kurze Zeit später wird der Guard `x == 0` wahr und das Event `e1()` liegt an.
Anhand dieser Darstellung lässt sich nicht genau erkennen, wann welches Ereignis anliegt bzw. wahr ist. Zudem ist unklar, welche Zustände wann aktuell sind. Um die Darstellung für den Entwickler verständlicher zu machen, werden die Sequenzdiagramme an UML 2.0 [24] angelehnt. Ein Sequenzdiagramm, das der UML 2.0 Notation entspricht, bietet zum einen die Möglichkeit zusätzliche Informationen auf der Lebenslinie anzugeben, zum anderen kann die Zeitdifferenz zwischen zwei Nachrichten angezeigt werden. Diese beiden zusätzlichen Darstellungsmöglichkeiten haben zur Folge, dass sich auf den Lebenslinien zusätzliche Rechtecke befinden, die angeben, in welchem Zustand sich ein Real-Time Statechart befindet. Zudem werden Zeitstempel eingeführt, die verdeutlichen, wann sich welche Aktion ereignet. (siehe Abbildung 3). Für den Zweck dieser Studienarbeit ist es sinnvoller, Zeitpunkte einzuführen, die die einzelnen Aktionen kennzeichnen, als Zeitdifferenzen anzugeben. Daher wird hier leicht von der UML 2.0 Notation abgewichen.

Abbildung 20 zeigt ein um die vorher beschriebenen Elemente erweitertes Real-Time Sequenzdiagramm aus Abbildung 19.
Anhand dieser Darstellung ist zu erkennen, dass sich das Real-Time Statechart `Controller` zuerst in dem Zustand `S1` befindet. Zum Zeitpunkt `t = 1500` liegt der Timeguard `1500 ≤ t0 ≤ 10000` an. `1500` Zeiteinheiten später wird der Guard `x == 0` wahr. Weitere `500` Zeiteinheiten weiter liegt das Event `e1()` an. Kurze Zeit darauf findet der Zustandswechsel statt, so dass sich das Real-Time Statechart zum Zeitpunkt `3750` im Zustand `S2` befindet.

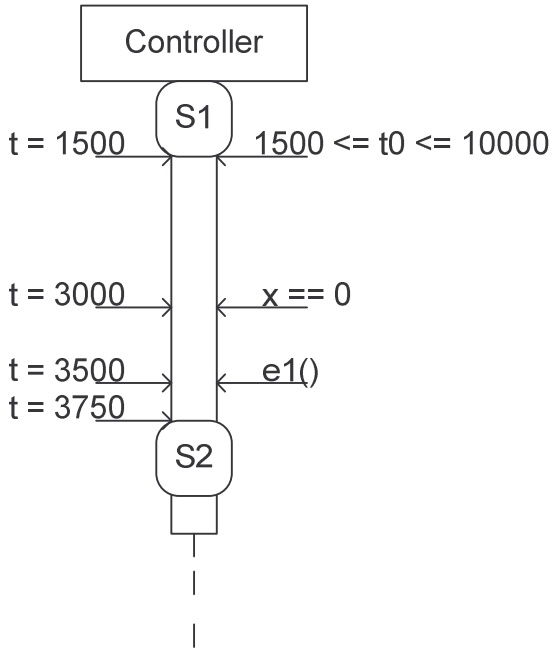

Abbildung 20 – Real-Time Sequenzdiagramm

Um eine Modifikation der Sequenzdiagramme und eine Interaktion zwischen Sequenzdiagrammen und Real-Time Statecharts zu realisieren, wird ein neues Plugin mit dem Namen `RTSCVisualization` entwickelt, das die Verknüpfung zwischen den Sequenzdiagrammen und den Real-Time Statecharts herstellt. Es ist nicht möglich, diese Verknüpfung in einem der bereits vorhandenen Plugins zu realisieren, da so eine Abhängigkeit entstehen würde und es nicht mehr möglich wäre Sequenzdiagramme ohne Real-Time Statecharts zu nutzen.
Abbildung 21 zeigt einen Ausschnitt des Metamodells der Sequenzdiagramme, wie sie zurzeit in Fujaba entwickelt werden können. Da nicht alle Klassen für diese Studienarbeit relevant sind, sind hier lediglich die relevanten Klassen dargestellt. Im Anhang befindet sich in Abbildung 29 das komplette Metamodell der Sequenzdiagramme. Für den Zweck dieser Studienarbeit muss es möglich sein, ein Sequenzdiagramm mit Objekten, einer Lebenslinie und Nachrichten zu erzeugen. Das Sequenzdiagramm selber wird mittels der Klasse `UMLSequenceDiagram` generiert. Dabei muss für jedes Real-Time Statechart ein Sequenzdiagrammobjekt erzeugt werden. Dies ermöglicht die Klasse `UMLSequenceDiagramObject`. Für die Darstellung der Guards, Actions, Clock Resets, Events und Timeguards des Real-Time Statecharts muss je eine Nachricht angelegt werden. Zusätzlich benötigt das Sequenzdiagrammobjekt eine Lebenslinie. Realisiert wird dies durch die Klassen `UMLMessage` und `UMLActivation`.
Um nun das Sequenzdiagramm um die vorher beschriebenen Zustandsinformationen und Zeitstempel zu erweitern, müssen zusätzliche Klassen eingeführt werden. Die Klassen `SeqStateInformation` und `UMSeqStateInformation` sind zuständig für die

Darstellung und das automatische Erneuern der Daten bei einer Änderung der Zustandsinformationen. Die Klasse `SeqStateInformation` stellt `set()`- und `get()`-Methoden der Informationen, die in dem Zustand dargestellt werden sollen, zur Verfügung. Mittels dieser Methoden hat die entsprechende Darstellungsklasse die Möglichkeit, bei einer Änderung der Daten diese auch in der Darstellung zu aktualisieren. Die Darstellungsklasse `UMSeqStateInformation` befindet sich im unparse-Modul des `RTSCVisualization` Plugins und ist daher für die Aktualisierung der Darstellung [21] verantwortlich. Diese Klasse wird über Änderungen durch den FirePropertyChange-Mechanismus [21] von Fujaba benachrichtigt. Diese beiden Klassen befinden sich ebenfalls im `RTSCVisualization` Plugin, da eine Erweiterung des UML-Sequenzdiagramms nur bei gleichzeitiger Visualisierung des Verhaltens eines Real-Time Statecharts gewünscht ist. Würden sich diese Klassen im `UML-SequenceDiagram` Plugin befinden, so hätte dies zur Folge, dass ein UML-Sequenzdiagramm immer mit Zustandsinformationen angezeigt wird.

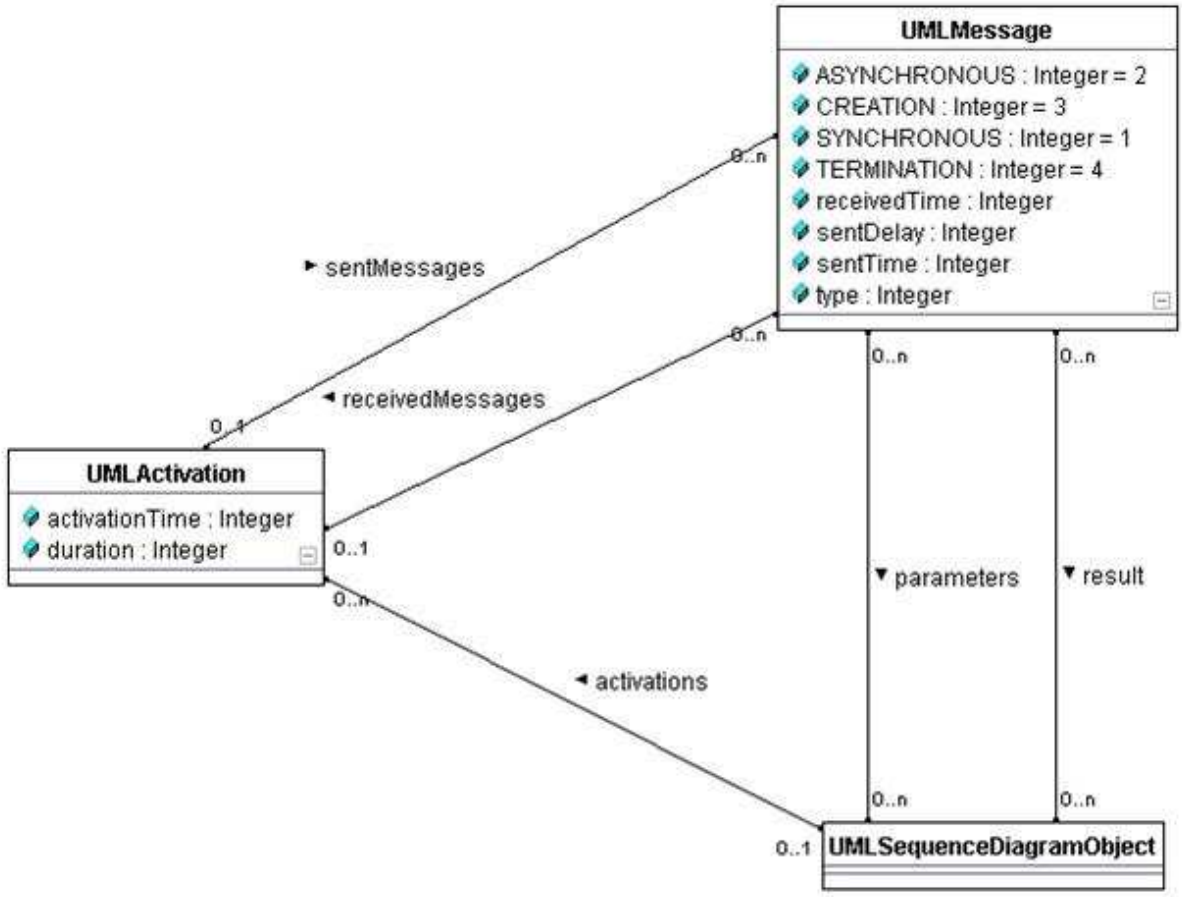

Abbildung 21 – reduziertes Metamodell Sequenzdiagramm

5.2 Real-Time Statechart

Um das Verhalten eines Real-Time Statecharts zu visualisieren, werden die fpr-Datei für die Darstellung des Real-Time Statecharts und die zugehörige Trace Datei für die Visualisierung der Ausführung des Real-Time Statecharts benötigt.

Abbildung 22 zeigt das Real-Time Statechart aus dem Kapitel 4. Der Zustand `S1` besitzt neben der farbigen Hervorhebung zusätzlich zur besseren Lesbarkeit einen breiteren Rand. Dies bedeutet, dass sich das Real-Time Statechart gerade in dem Zustand `S1` befindet.

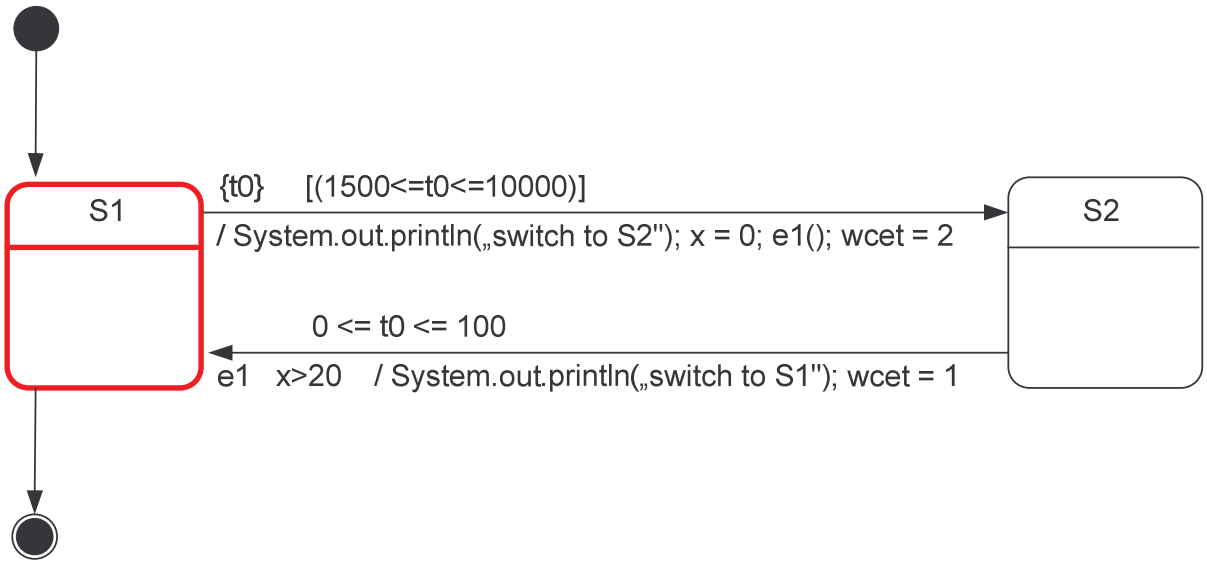

Abbildung 22 - Visualisierung Zustand

Analog zur Visualisierung des aktuellen Zustands werden ebenfalls die Komponenten, die das Schalten einer Transition auslösen, hervorgehoben. Abbildung 23 zeigt das Real-Time Statechart, das sich in dem Zustand `S2` befindet. Zusätzlich ist der Guard `x>20`, der zum Schalten der Transition beiträgt, wahr. Das Event und der Timeguard, die ebenfalls zum Schalten der Transition beitragen, werden auf die gleiche Weise visualisiert.

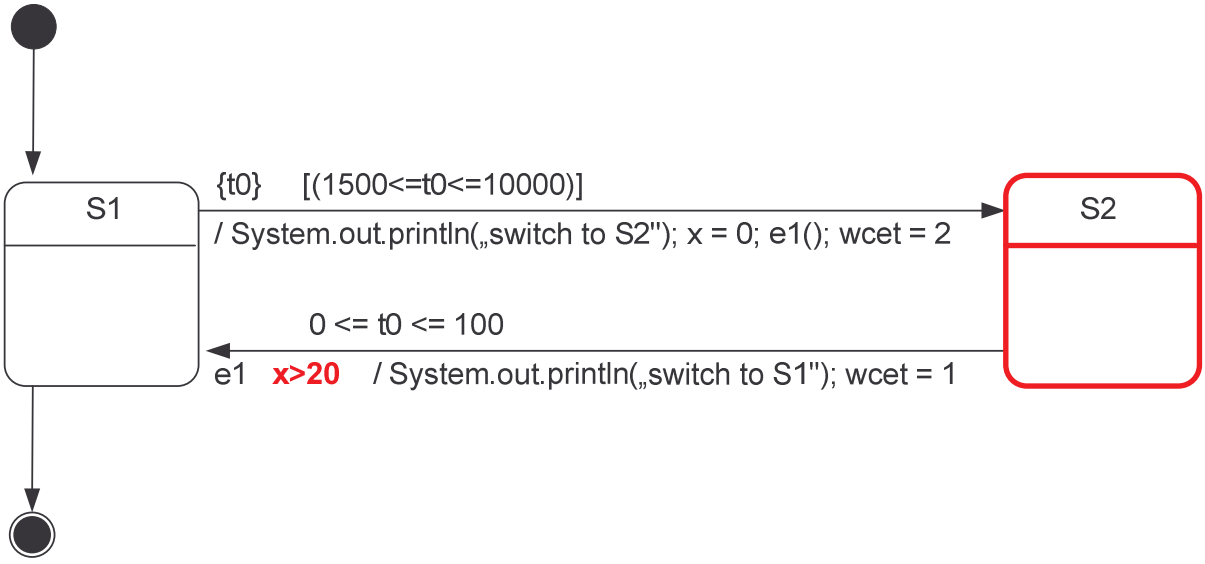

Abbildung 23 - Visualisierung Guard

Das Schalten der Transition wird durch eine rote und breitere Markierung der Transition kenntlich gemacht. Abbildung 24 zeigt, dass die Transition, die vom Zustand `S2` in den Zustand `S1` wechselt, schaltet. Zusätzlich wird beim Schalten der Seiteneffekt ausgelöst. Auch dieser ist hervorgehoben. Da sich die Transition im Schaltvorgang befindet und die

Elemente, die das Schalten der Transition auslösen, den Schaltvorgang an sich nicht beeinflussen, sind der Guard, das Event und der Timeguard nicht hervorgehoben.

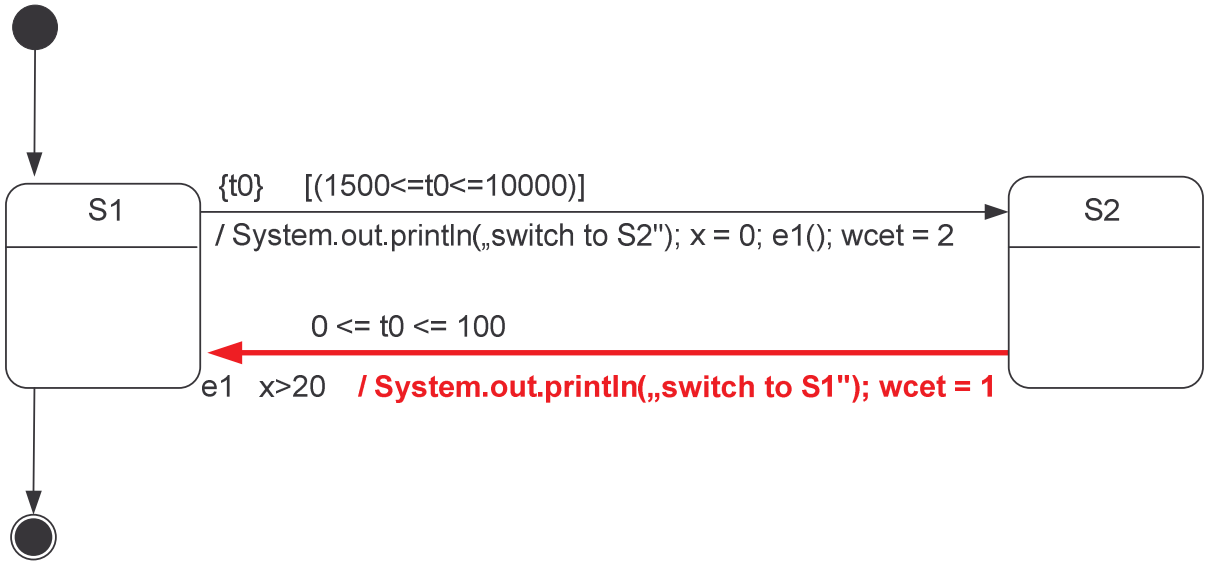

Abbildung 24 - Visualisierung Transition

Diese Visualisierung kann durch den Aufruf einer Methode, die die Farbe von Objekten ändern kann, realisiert werden. Die Klassen `UMLRealtimeTransition`, zuständig für die Darstellung der Transition, und `UMLRealtimeState`, zuständig für die Darstellung der Zustände, erben beide von der Klasse `ASGElement`. Diese wiederum besitzt eine Assoziation zu der Klasse `FSAObject`. FSA steht für `Fujaba Swing Adapter` und implementiert den `Model View Controller` Mechanismus [21], der Änderungen am Diagramm und der Visualisierung konstant hält. Zudem bietet diese Klasse die Möglichkeit, die Farben der einzelnen ASGElemente zu ändern.

5.3 Interaktion zwischen Real-Time Sequenzdiagramm und Real-Time Statechart

Um eine Interaktion zwischen den Sequenzdiagrammen und den Real-Time Statecharts herzustellen, wird ein neues Plugin benötigt, das die Verbindung zwischen diesen beiden Plugins realisiert.
Abbildung 25 zeigt die neue Plugin Struktur. Das im Rahmen dieser Studienarbeit erstellte Plugin `RTSCVisualization` nutzt das Plugin `Real-Time Statechart` für die Darstellung der Real-Time Statecharts und das Plugin `UMLSequenceDiagram` für die Darstellung der Sequenzdiagramme. Zusätzlich benötigt es `NewFujabaApp`, den Kern von Fujaba. Durch diese Plugin Struktur sind die einzelnen Plugins, bis auf `RTSCVisualization`, unabhängig voneinander. Somit müssen keine Änderungen für die Real-Time Statecharts und Sequenzdiagramme vorgenommen werden.

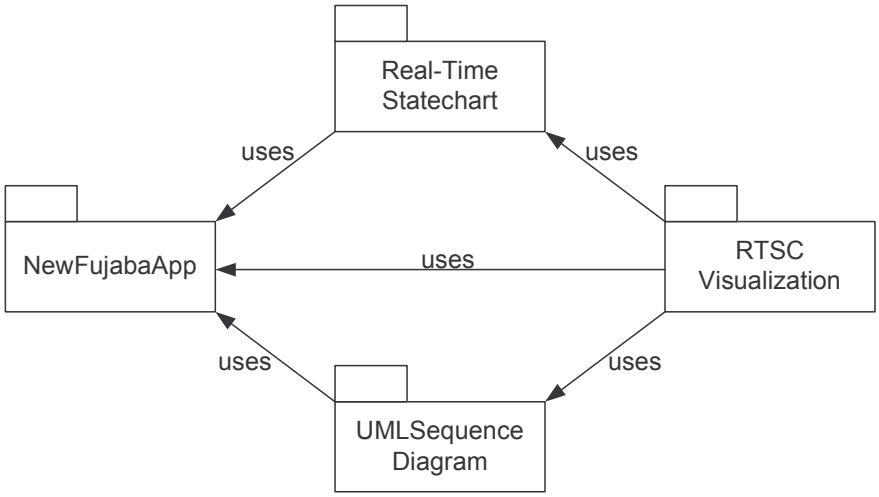

Abbildung 25 - Plugin Struktur

Abbildung 26 zeigt einen Auszug des Metamodells der Visualisierung der Ausführung von Real-Time Statecharts. Das komplette Metamodell ist im Anhang in Abbildung 30 dargestellt. Da der Mechanismus, der im Folgenden vorgestellt wird, auf alle Elemente angewendet werden kann, wird hier nur ein Teil des Metamodells beschrieben.
Das Metamodell `RTSCVisualization` besitzt eine Verbindung zu weiteren Plugins, wie in Abbildung 25 dargestellt.
Das Plugin `NewFujabaApp` beinhaltet unter anderem die Klassen `ASGElement` und `ASGElementRef`. Über den ASG Referenzierungsmechanismus [7] ist es möglich eine Verbindung zwischen Plugins herzustellen ohne dass diese dabei abhängig voneinander werden.
Die Klassen `UMLRealtimeStatechart` aus dem Plugin `RealtimeStatechart` und `UMLSequenceDiagram` aus dem Plugin `UMLSequenceDiagram` erben beide von `ASGElement`.
Das Plugin `RTSCVisualization` nutzt die drei anderen Plugins, indem es Klassen besitzt, die von `ASGElementRef` erben. Die Klasse `ASGElementRef` ermöglicht es, durch einen Methodenaufruf die Verbindung zwischen `UMLRealtimeStatechartVisualization` und `UMLRealtimeStatechart` sowie zwischen `UMLSequenceDiagramVisualization` und `UMLSequenceDiagram` herzustellen. Die Verbindung zwischen den zuständigen Klassen für die Visualisierung der Real-Time Statecharts und der UML-Sequenzdiagramme wird durch eine weitere Klasse, `UMLRTSCToUMLSeqDiag`, hergestellt.
Um diese Verbindung zu realisieren, wird beim Einladen der Trace Datei je ein Objekt der Klassen `UMLSequenceDiagramVisualization` und `UMLRTSCToUMLSeqDiag` erzeugt. Der Konstruktor der Klasse `UMLSequenceDiagramVisualization` erzeugt ein neues UML-Sequenzdiagramm mit dem Namen des Real-Time Statecharts. Die Verbindung zwischen den Klassen `UMLSequenceDiagramVisualization` und `UMLSequenceDiagram` wird durch den Methodenaufruf `setElement (UMLSequenceDiagram)` realisiert. Die Methode `setElement (ASGElement)` wird von der Klasse `ASGElementRef` implementiert. Der Aufruf dieser Methode ist in

UMLSequenceDiagramVisualization möglich, da diese Klasse von ASGElementRef erbt. Zusätzlich wird ein Objekt der Klasse UMLRealtimeStatechartVisualization erzeugt. Hierbei wird ebenfalls über den Aufruf der Methode setElement (ASGElement) die Verbindung zum Real-Time Statechart, das durch das Öffnen der Datei bereits besteht, erzeugt.
Die Verbindung zwischen UMLSequenceDiagramVisualization und UMLRealtimeStatechartVisualization wird durch die Methodenaufrufe setUMLSequenceDiagramVisualization(UMLSequenceDiagramVisualization) und setUMLRealtimeStatechartVisualization (UMLRealtimeStatechartVisualization) der Klasse UMLRTSCToUMLSeqDiag generiert.
Um weitere Klassen aus dem Real-Time Statechart und dem UMLSequenceDiagram Plugin zu verbinden, wird der gleiche Mechanismus verwendet. So ist zum Beispiel die Klasse UMLRealtimeTransition aus Real-Time Statechart mit der Klasse UMLMessage durch eine Vererbung mit ASGElementRef verbunden.

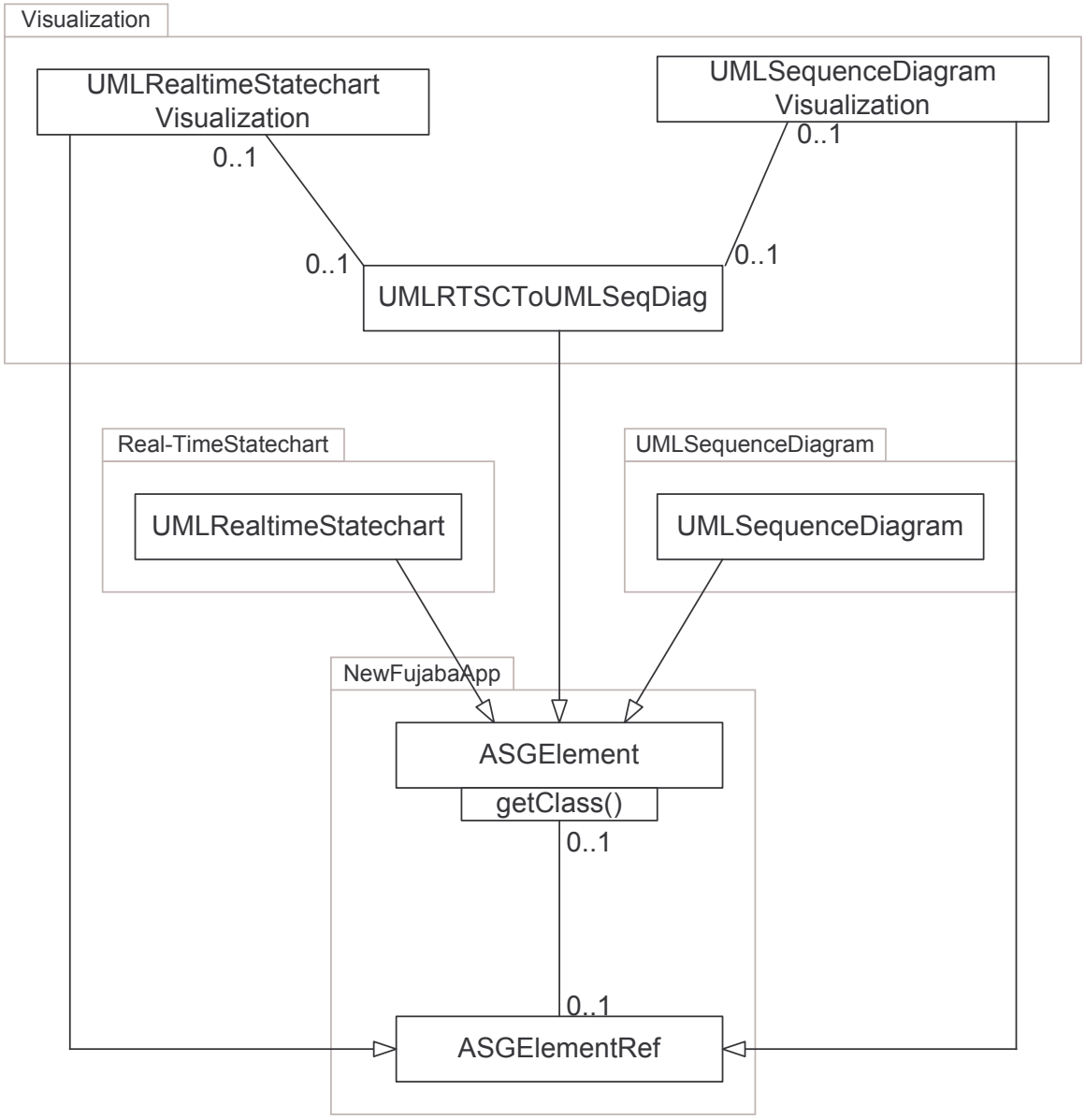

Abbildung 26 - Metamodell Visualization

5.4 Vorstellung der Software

Im Folgenden werden einige Screenshots der implementierten Software gezeigt. Abbildung 27 zeigt einen Ausschnitt der Visualisierung der Ausführung eines Real-Time Statecharts. Anhand der Graphik ist zu erkennen, dass die Transition von `S1` nach `S2` schaltet und dabei der Seiteneffekt `System.out.println(„switch to S2“); x=0; e1(); wcet=2` ausgeführt wird.

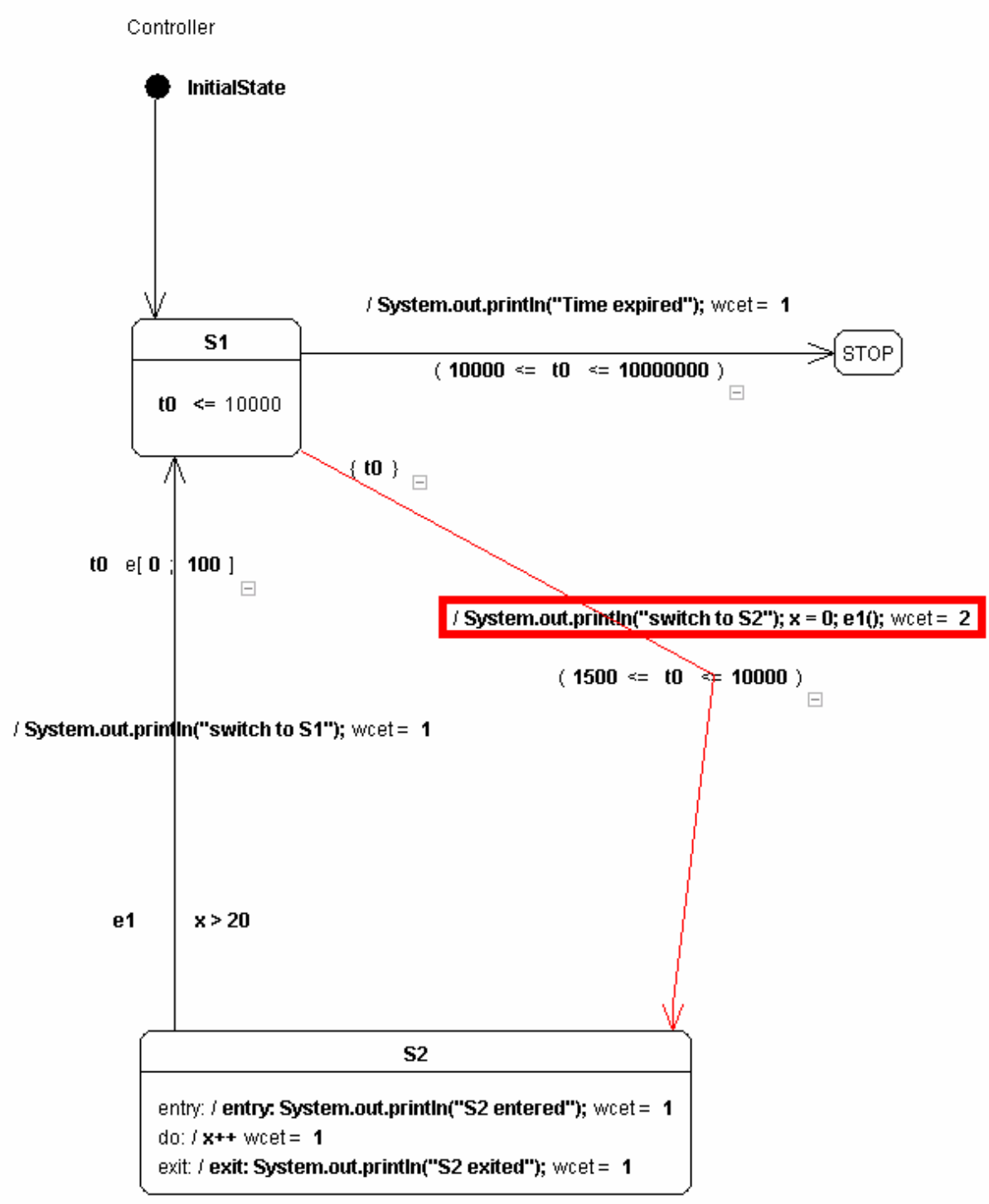

Abbildung 27 - Visualisierung des Schaltvorgangs einer Transition

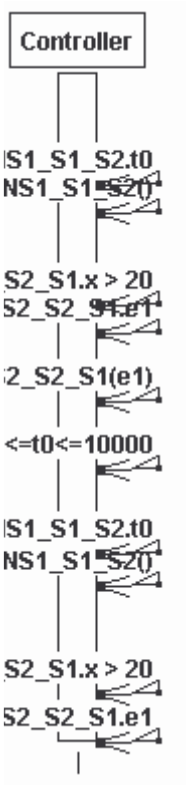

Abbildung 28 - Real-Time Sequenzdiagramm

Abbildung 28 zeigt das Real-Time Sequenzdiagramm, wie es zurzeit aus der Trace Datei des in Abbildung 27 dargestellten Real-Time Statecharts, generiert wird. Jede Nachricht des Real-Time Sequenzdiagramms entspricht einem Event, Guard, Timeguard, Action oder Clock Reset einer Transition des Real-Time Statecharts.

6. Zusammenfassung und Ausblick

In dieser Studienarbeit wurde zuerst motiviert, warum es sinnvoll ist eine Visualisierung der Ausführung von Real-Time Statecharts zu implementieren. Real-Time Statecharts sind ein Konstrukt zur Modellierung von echtzeitfähigen Systemen. Durch die Möglichkeit der Einbringung von Uhren, ermöglichen sie es das Verhalten realer Systeme nachzustellen. Jedoch ist die Modellierung von Real-Time Statecharts nicht trivial und es ist möglich, dass die Konstruktion zu unerwünschtem Verhalten führt. Um das Verhalten eines Real-Time Statecharts zu überprüfen, wurde eine Visualisierung auf Basis von Real-Time Statecharts und Sequenzdiagrammen, die eine weitere Veranschaulichung des Verhaltens von Real-Time Statecharts ermöglichen, erstellt.
Im weiteren Verlauf wurden in Kapitel 2 die Grundlagen für diese Studienarbeit erläutert. Dazu gehören neben den verteilten Systemen und der Parallelität auch die beiden Diagrammarten Real-Time Statecharts und Sequenzdiagramme. Es wurde aufgezeigt, dass die Ausführung von Real-Time Statecharts auf verteilten Systemen einige Probleme mit sich bringt, wie zum Beispiel die Synchronisation von Uhren. Ein weiterer Grund für die Erstellung dieser Studienarbeit ist, dass die parallele Ausführung, die sehr schnell zu einer Deadlock Situation führen kann, besser kontrolliert wird.
Anschließend wurden mit Javis und Rational Rose RealTime zwei Systeme vorgestellt, die die Erstellung einer Trace Datei bzw. die Visualisierung der Ausführung von Statecharts ermöglichen. Diese beiden Systeme reichen jedoch aus den in Kapitel 3 beschriebenen Gründen nicht vollständig aus, um mögliche Fehler bei der Modellierung eines Real-Time Statecharts zu entdecken.
Kapitel 4 bezieht sich auf die Erstellung einer Trace Datei, die alle nötigen Informationen enthält um eine Visualisierung der Ausführung eines Real-Time Statecharts zu ermöglichen. Hierbei wurde die Codegenerierung der Real-Time Statecharts von Fujaba so erweitert, dass es möglich ist, die Daten zuerst in ein speicherplatzbegrenztes Array und anschließend in eine Datei zu schreiben. Dieser Mechanismus wurde gewählt, da verteilte Systeme nur begrenzten Speicherplatz zur Verfügung stellen und die zeitkonforme Ausführung gewährleistet sein muss. Des Weiteren wurde neben der Art der Erstellung der Trace Datei auch diskutiert, welche Informationen in welcher Form vorliegen. Die Trace Datei liegt in XML-Form vor, da Java geeignete Mechanismen zur Verfügung stellt, um XML-Dateien auf Korrektheit zu überprüfen und zeilenweise einzulesen.
Im Anschluss daran wurde in Kapitel 5 beschrieben, wie die Visualisierung in Fujaba verläuft. Dazu wurde ein neues Plugin erstellt, das die Verbindung zwischen den Real-Time Statecharts und den Real- Time Sequenzdiagrammen und damit auch eine Interaktion zwischen beiden Diagrammarten ermöglicht. Zudem wurden die UML-Sequenzdiagramme erweitert und somit dem zukünftigen UML-Standard 2.0 angelehnt, der eine genauere Betrachtung des Ablaufs ermöglicht.
Somit ist es möglich mittels der Entwicklungsumgebung Fujaba ein Real-Time Statechart zu erstellen, daraus eine Trace Datei zu generieren und diese für die Visualisierung der Ausführung zu nutzen, um so Fehler, die bei der Konstruktion des Diagramms bzw. durch die parallele Ausführung auf einem verteilten System entstanden sind, zu erkennen.
Wie in Kapitel 4.6 beschrieben, wäre es möglich, das Mischen mehrerer Trace Dateien anhand eines zusätzlichen Object Event Graphs und eines Process Event Graphs genauer zu realisieren. Zudem kann das gesamte Konzept der Visualisierung der Ausführung von Real-Time Statecharts auch auf andere Modelle angewandt werden. Hierzu gehören UML-Statecharts und Hybride Statecharts[6], die auf dem Konzept der Real-Time Statecharts aufbauen.
Des Weiteren wird zurzeit an der AG Softwaretechnik eine C++-Codegenerierung für Real-Time Statecharts entwickelt. Es ist daher möglich, das Konzept der erweiterten

Codegenerierung auch in C++ umzusetzen, um so ebenfalls eine Visualisierung des Verhaltens des Real-Time Statecharts zu erhalten.

Literaturhinweise:

[1] Lutz Bichler, Ansgar Radermacher, Andreas Schürr: Evaluating UML Extensions for Modeling Real-time Systems, University of the German Federal Armed Forces Munich, Neubiberg, Germany

[2] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, Mark Turnbull, Rudy Belliardi, Doug Locke, Scott Robbins, Pratik Solanki, Dionisio de Niz: The Real-Time Specification for Java™, Addison-Wesley, 2000

[3] Grady Booch, Ivar Jacobson, James Rumbaugh: UML konzentriert, Addison – Wesley, 1998

[4] Sven Burmester: Generierung von Java Real – Time Code für zeitbehaftete UML Modelle, Master´s thesis, University of Paderborn, Paderborn, Germany, September 2002

[5] Sven Burmester, Holger Giese, Wilhelm Schäfer: Code Generation for Hard Real-time Systems from Real-time Statecharts, Tech. Rep. tr-ri-03-244, University of Paderborn, Paderborn, Germany, October 2003

[6] Sven Burmester, Holger Giese, Matthias Tichy: Modeling Reconfigurable Mechatronic Systems with Mechatronic UML, in Porc. of the Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden, June 2004

[7] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, Albert Zündorf: Tool Integration at the Meta-Model Level within the FUJABA Tool Suite, in Proc. of the Workshop on Tool-Integration in System Development (TIS), Helsinki, Finland, (ESEC / FSE 2003 Workshop 3), pp. 51-56, September 2003

[8] Fujaba Home Page www.fujaba.de

[9] Curtis E. Hirschuk and C.Murray Woodside: Logical Clock Requirements for Reverse Engineering Scenarios from a distributed System, IEEE Transactions on Software Engineering, Vol. 28, No. 4, April 2002

[10] Java Home Page java.sun.com

[11] Margarete Kudak: Real-Time Statecharts und Hybride Statecharts, Seminararbeit, University of Paderborn, Paderborn, Germany, January 2004

[12] Katharina Mehner, Annika Wagner: Visualisierung der Synchronisation von Java – Threads mit UML, in Proc. of the Workshop Modellierung 2000, University of Paderborn, Paderborn, Germany, 2000

[13] Susannah Moat: Adapting the Fujaba Code Generation Mechanism, Tech. Rep. 04-247, in Proc. of the Fujaba Days 2003, Kassel, Germany, October 2003

[14] OMG Unified Modeling Language Specification, OMG, March 2003, Version 1.5

[15] Rational Rose Home Page, www.rational.com, Januar 2004

[16] Rational, the software developement company: Rational Rose RealTime, Tutorial, Version: 2003:06.00, Part Number: 800-026117-000, www.rational.com, Mai 2004

[17] Michiel Ronsse, Koen de Bosschere, Mark Christiaens, Jaques Chassin de Kergommeaux, Dieter Kranzlmüller: Record/Replay for Nondeterministic Program Executions, Communications of the ACM, Vol. 46 No.9, September 2003

[18] Bran Selic, Garth Gulkerson, Paul T. Ward: Real-Time Object-Oriented Modelling, John Wiley & Sons Inc., 1994

[19] Andrew S. Tanenbaum, Maarten van Stehen: Distributed Systems, Principles and Paradigms, Prentice-Hall Inc., 2002

[20] Prof. L. Thiele: Digitaltechnik und Rechnerstrukturen 6. Input/Output, Computer Engineering and Networks, Vorlesung, Swiss Federal Insitute of Technology Zurich

[21] Matthias Tichy: How to add a new diagram to Fujaba, Tech. Rep. 04-247, Proc. Of the Fujaba Days 2003, Kassel, Germany, October 2003
[22] Matthias Tichy, Margarete Kudak: Visualization of the execution of Real-Time Statecharts, Tech. Rep. 04-247, in Proc. of the Fujaba Days 2003, Kassel, Germany, October 2003
[23] Unified Modelling Language Homepage www.uml.org, January 2004
[24] UML 2.0 Superstructure Specification, OMG, September 2003, pp. 451
[25] Bernd Weymann: Visualisierung der Synchronisation in Javaprogrammen mit UML, Master`s thesis, University of Paderborn, Paderborn, Germany, 2001

Anhang:

1. XML Schema für Trace Datei

Im Folgenden wird das XML Schema für eine Trace Datei vorgestellt. Ein XML-Schema definiert, wie auch eine DTD, ein XML-Dokument. Jedoch ist die Definition durch ein XML-Schema genauer, da die Datentypen der Elemente definiert werden können. Anhand des nachfolgenden Schemas lässt sich zum Beispiel erkennen, dass das Element `Action` aus den Elementen `ActionName` und `ActionTime` besteht, wobei `ActionName` vom Typ `string` und `ActionTime` vom Typ `long` ist.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
     <xs:element name="Action">
          <xs:complexType>
               <xs:sequence>
                    <xs:element ref="ActionName"/>
                    <xs:element ref="ActionTime"/>
               </xs:sequence>
          </xs:complexType>
     </xs:element>
     <xs:element name="ActionName" type="xs:string"/>
     <xs:element name="ActionTime" type="xs:long"/>
     <xs:element name="ClockReset">
          <xs:complexType>
               <xs:sequence>
                    <xs:element ref="ClockResetName"/>
                    <xs:element ref="ClockResetTime"/>
               </xs:sequence>
          </xs:complexType>
     </xs:element>
     <xs:element name="ClockResetName" type="xs:string"/>
     <xs:element name="ClockResetTime" type="xs:long"/>
     <xs:element name="EnterState">
          <xs:complexType>
               <xs:sequence>
                    <xs:element ref="actualState"/>
                    <xs:element ref="EnterTime"/>
               </xs:sequence>
          </xs:complexType>
     </xs:element>
     <xs:element name="EnterTime" type="xs:long"/>
     <xs:element name="Event">
          <xs:complexType>
               <xs:sequence>
                    <xs:element ref="EventName"/>
                    <xs:element ref="EventTime"/>
               </xs:sequence>
          </xs:complexType>
     </xs:element>
     <xs:element name="EventName" type="xs:string"/>
     <xs:element name="EventTime" type="xs:long"/>
     <xs:element name="ExitState">
```

```
            <xs:complexType>
                  <xs:sequence>
                        <xs:element ref="oldState"/>
                        <xs:element ref="ExitTime"/>
                  </xs:sequence>
            </xs:complexType>
      </xs:element>
      <xs:element name="ExitTime" type="xs:long"/>
      <xs:element name="Guard">
            <xs:complexType>
                  <xs:sequence>
                        <xs:element ref="GuardName"/>
                        <xs:element ref="GuardTime"/>
                  </xs:sequence>
            </xs:complexType>
      </xs:element>
      <xs:element name="GuardName" type="xs:string"/>
      <xs:element name="GuardTime" type="xs:long"/>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="StartTime" type="xs:long"/>
      <xs:element name="Time" type="xs:long"/>
      <xs:element name="TimeGuard">
            <xs:complexType>
                  <xs:sequence>
                        <xs:element ref="TimeGuardName"/>
                        <xs:element ref="Time"/>
                  </xs:sequence>
            </xs:complexType>
      </xs:element>
      <xs:element name="TimeGuardName" type="xs:string"/>
      <xs:element name="Trace">
            <xs:complexType>
                  <xs:choice maxOccurs="unbounded">
                        <xs:element ref="Name"/>
                        <xs:element ref="StartTime"/>
                        <xs:element ref="ExitState"/>
                        <xs:element ref="EnterState"/>
                        <xs:element ref="Action"/>
                        <xs:element ref="ClockReset"/>
                        <xs:element ref="Event"/>
                        <xs:element ref="Guard"/>
                        <xs:element ref="TimeGuard"/>
                  </xs:choice>
            </xs:complexType>
      </xs:element>
      <xs:element name="actualState" type="xs:string"/>
      <xs:element name="oldState" type="xs:string"/>
</xs:schema>
```

2. Metamodell des UML-Sequenzdiagramm Plugins

Abbildung 29 zeigt das vollständige Metamodell der UML-Sequenzdiagramme, wie sie zurzeit mit Fujaba modelliert werden können. Die für diese Studienarbeit wichtigen Klassen sind in Kapitel 5.1 beschrieben.

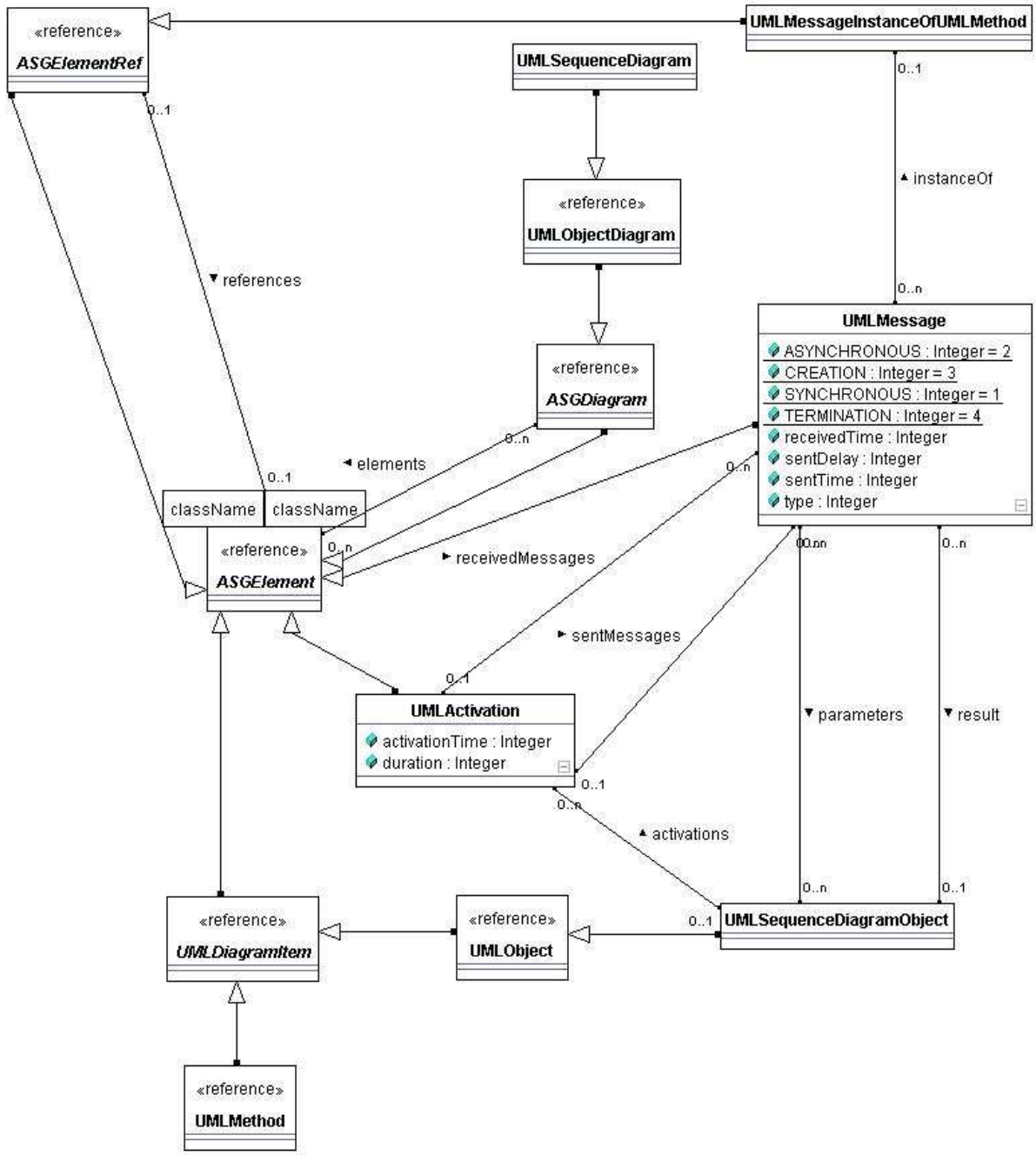

Abbildung 29 - vollständiges Metamodell Sequenzdiagramm

3. Metamodell des RTSCVisualization Plugins

In Abbildung 29 ist das vollständige Metamodell des Plugins das in dieser Studienarbeit entwickelt wurde zu sehen. Zentrale Komponenten sind dabei die Klassen `ASGElement` und `ASGElementRef`, die es ermöglichen; eine Verbindung zwischen zwei unabhängigen Plugins herzustellen. In dem Metamodell sind zur besseren Veranschaulichung auch die Klassen aus den bereits bestehenden Plugins `Real-Time Statechart` und `UMLSequenceDiagram` aufgeführt.
Die zentralen KlasseN des Metamodells sind: `UMLRTSCToUMLSeqDiag`, `StateToSeqStateInformation`, `GuardToMessage`, `TimeguardToMessage`, `EventToMessage`, `ActionToMessage` und `ClockResetToMessage`. Jede dieser Klassen stellt die Verbindung zwischen zwei Klassen her, wobei eine Klasse aus dem Real-Time Statechart Plugin und eine Andere aus dem UML-Sequenzdiagramm Plugin repräsentiert wird. So verbindet zum Beispiel die Klasse `GuardToMessage` den Guard einer Transition und die dazugehörige Nachricht im Real-Time Sequenzdiagramm. Durch diese Verbindung ist es möglich, eine Interaktion zwischen der Nachricht und dem Guard zu erzeugen.
Der weitere Aufbau des Metamodells ist in Kapitel 5.3 erläutert.

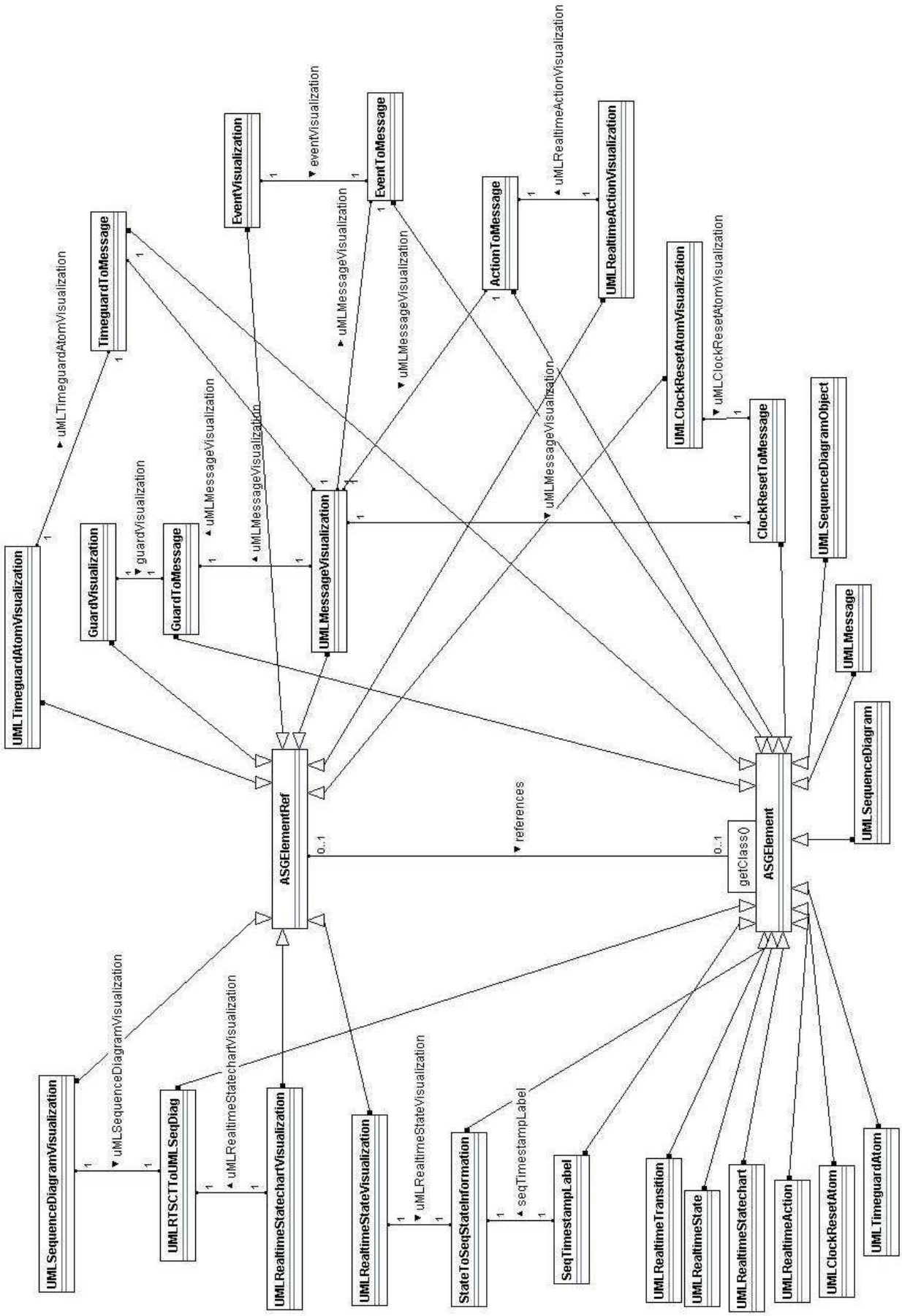

Abbildung 30 - Metamodell RTSCVisualization

4. Beschreibung der Funktionsweise der implementierten Software

Im Folgenden wird die Funktionsweise der im Rahmen dieser Studienarbeit entwickelten Software beschrieben.
Nachdem Fujaba gestartet wurde, muss der Anwender ein Real-Time Statecharts modellieren oder die Datei eines bereits bestehenden Real-Time Statecharts öffnen.

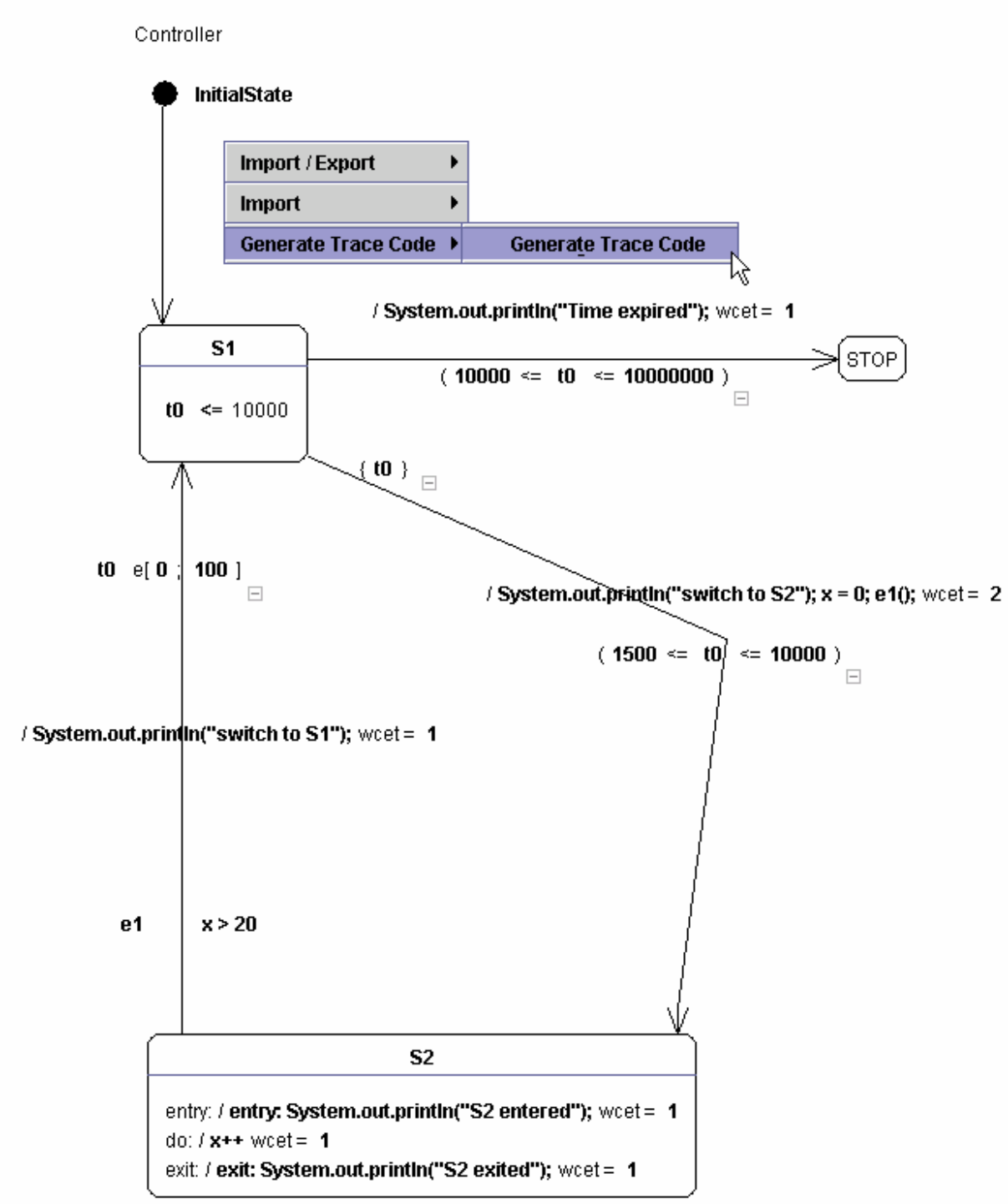

Abbildung 31 - Trace Codegenerierung

Abbildung 31 zeigt ein Real-Time Statechart, das mit Fujaba erstellt worden ist. Durch rechten Mausklick auf die Fujaba Oberfläche innerhalb des Real-Time Statecharts Fensters

öffnet sich ein Dialog, der unter Anderem die Möglichkeit bietet, Java RT-Code mit zusätzlichen Trace Informationen zu generieren. Mittels Bestätigung durch die linke Maustaste auf `Generate Trace Code` wird die Codegenerierung gestartet. Die generierte Klasse findet sich in dem Unterordner `generated` von `NewFujabaApp`. Um den Code auszuführen, ist zusätzlich eine Main Klasse erforderlich. Diese wird durch Betätigung eines Buttons generiert. Abbildung 32 zeigt den Aufruf des Dialogs zur Erstellung der Main Klasse. Der Benutzer wird dabei gefragt, ob er ein zusätzliches Schedule Dokument angeben will. Diese Frage ist mit Ja zu beantworten. Das Schedule Dokument ist in der Ordnerstruktur `../ NewFujabaApp/ plugins/ RealtimeStatechart` zu finden und hat den Namen `schedule.xml`. Es wurde durch die Trace Codegenerierung angelegt. Neben dieser Datei wird kein weiteres Schedule Dokument benötigt. Die hiermit erzeugte Main Klasse befindet sich im Ordner `../NewFujabaApp`.

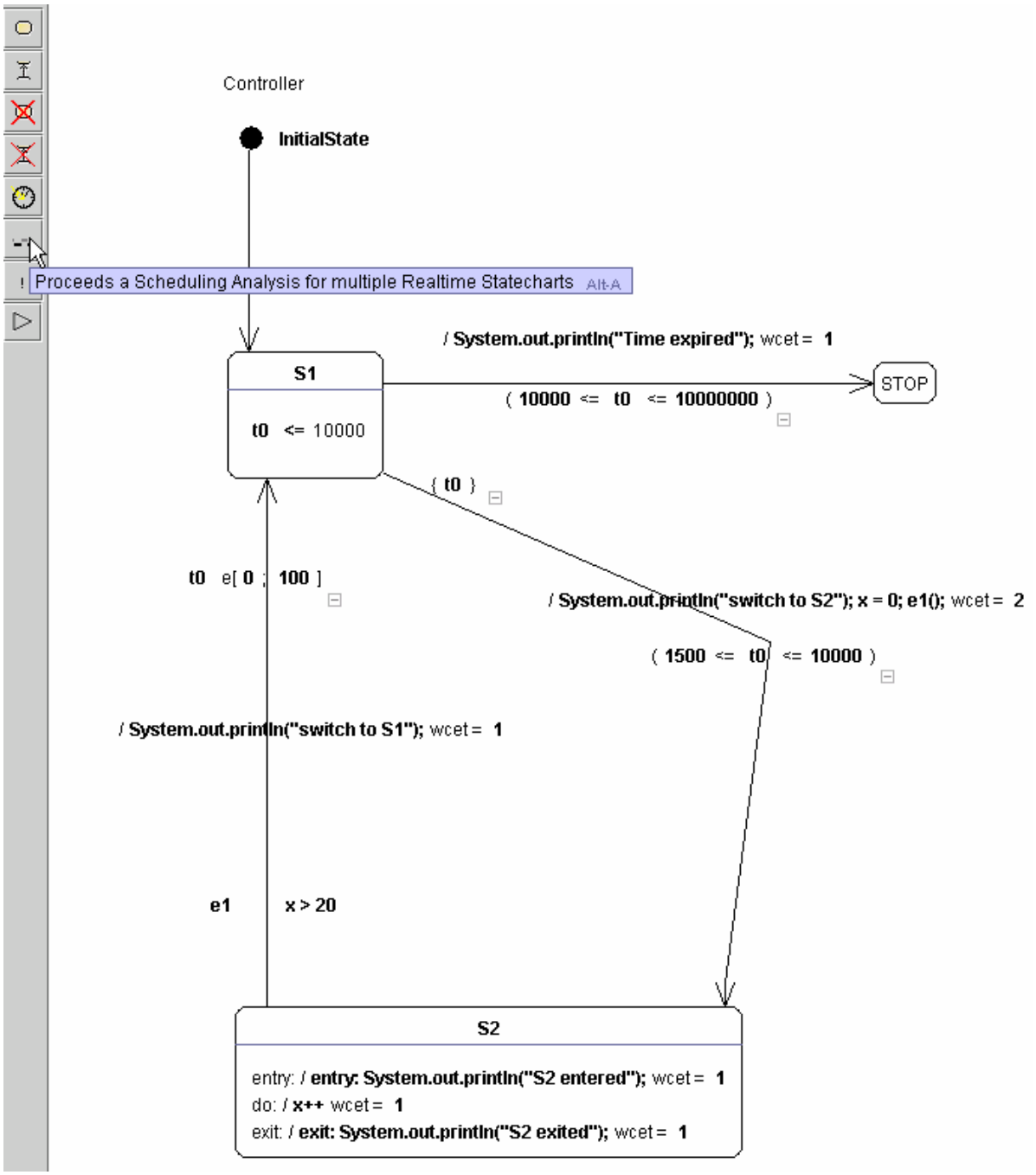

Abbildung 32 - Generierung Main Klasse

Mittels der Main Klasse ist es möglich, die Trace Code Datei auf einem echtzeitfähigem Rechner zu starten. Dazu sind zusätzlich das sdm-Paket und die für die Trace Erstellung benötigten Dateien erforderlich. Beides befindet sich auf der beiliegenden CD.
Nachdem die erzeugten Klassen mit dem Befehl `javac *.java` kompiliert wurden, wird das Real-Time Statechart durch den Aufruf `tjvm –Xbootclasspath= /opt/ timesys/ jtime/ lib/ foundation.jar: /opt/ timesys/ jtime/ btclasses.zip:. Main` gestartet. Während der Ausführung des Real-Time Statecharts wird die Trace Datei mit den Namen `trace.xml` generiert.

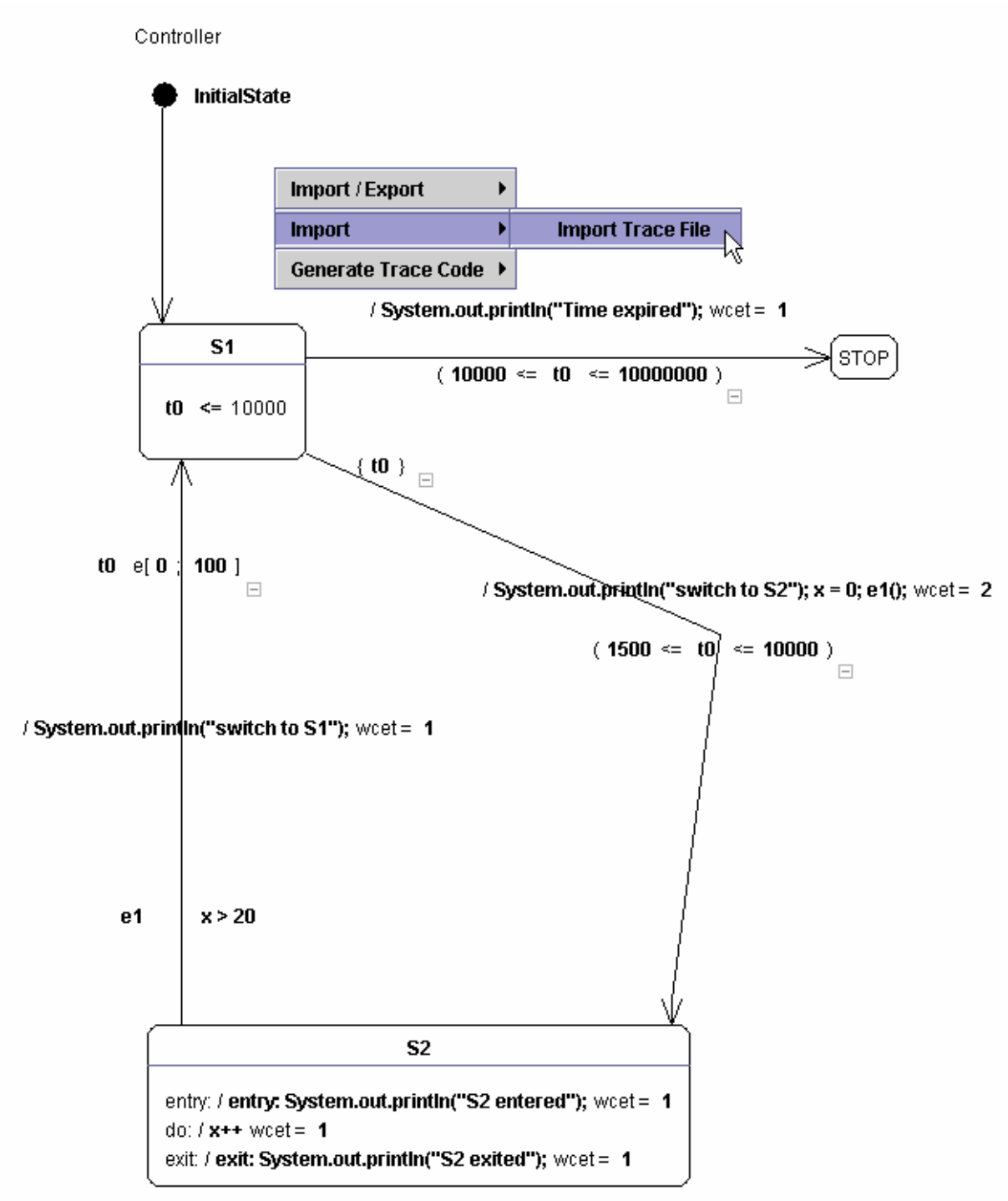

Abbildung 33 - Importieren der Trace Datei

Im nächsten Schritt ist die Trace Datei in Fujaba zu importieren. Durch Betätigung des rechten Mausklicks innerhalb der Fujaba Oberfläche des Real-Time Statechart Fensters öffnet sich ein Dialog, wie in Abbildung 33 zu sehen ist. Durch Bestätigung des Menupunktes `Import Trace File` mit der linken Maustaste, öffnet sich ein Dialog, der es dem Benutzer ermöglicht, die Trace Datei einzuladen. Beim Importieren der Trace Datei wird das Real-Time Sequenzdiagramm in Fujaba erzeugt. Ist dieser Vorgang abgeschlossen, ist es möglich, die Visualisierung der Ausführung zu starten.

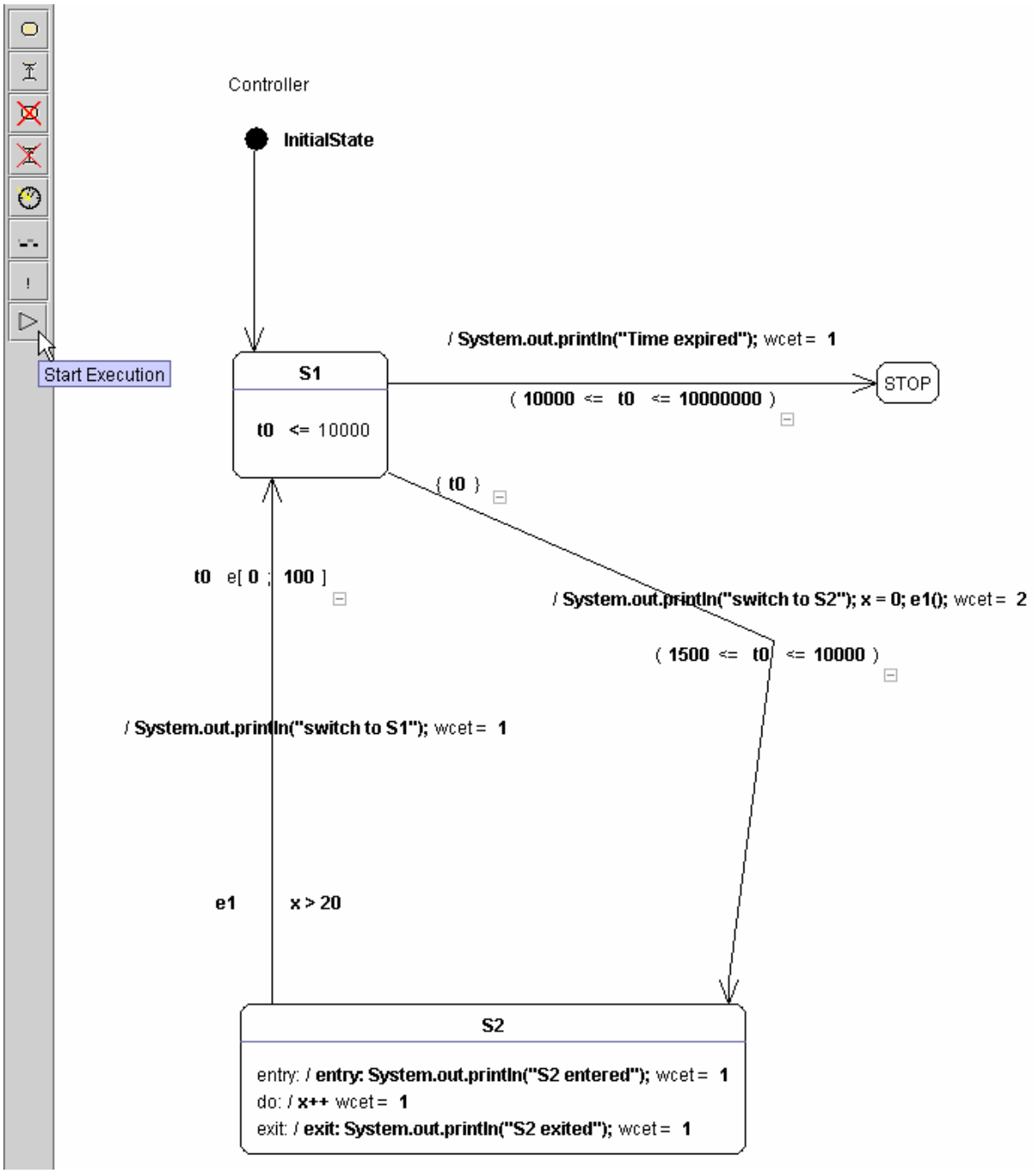

Abbildung 34 - Starten der Ausführung

Abbildung 34 beschreibt den Vorgang zum Starten der Ausführung. Innerhalb der Fujaba Oberfläche befindet sich eine Toolbar, die einen Button beinhaltet, der die Ausführung startet.

Durch einen Mausklick auf die linke Taste wird die Visualisierung gestartet. Nachdem die Trace Datei durchlaufen ist, stoppt die Visualisierung.

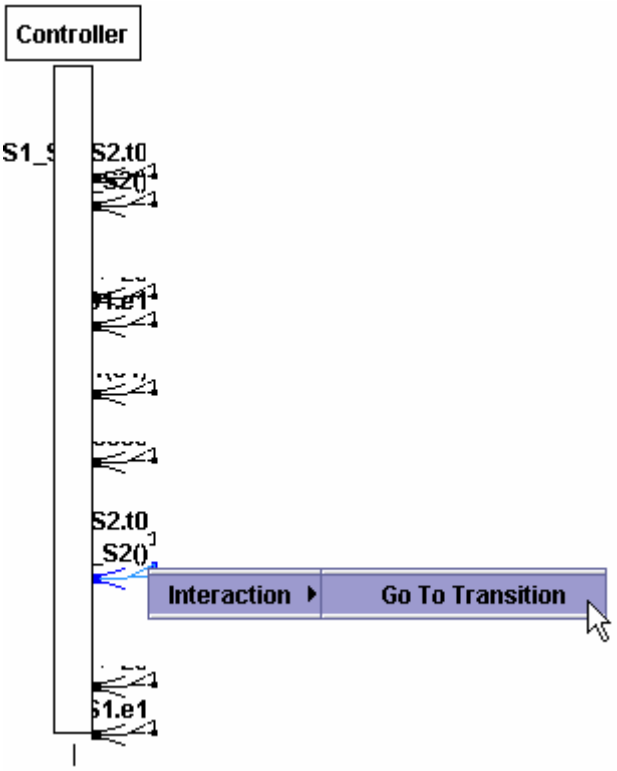

Abbildung 35 – Interaktion

Neben der Visualisierung ist es möglich eine Nachricht im Real-Time Sequenzdiagramm auszuwählen und das dazu passende visualisierte Real-Time Statechart anzeigen zu lassen. Durch den rechten Mausklick auf eine Nachricht in einem Real-Time Sequenzdiagramm öffnet sich ein Fenster, wie in Abbildung 35 gezeigt wird. Durch Bestätigung des Menueintrags `Go To Transition` öffnet sich das Real-Time Statechart, wobei die entsprechende Transition farblich hervorgehoben ist. In Abbildung 36 ist das entsprechende Real-Time Statechart zu Abbildung 35 zu sehen.
Der gleiche Mechanismus kann auch für Zustände angewendet werden.

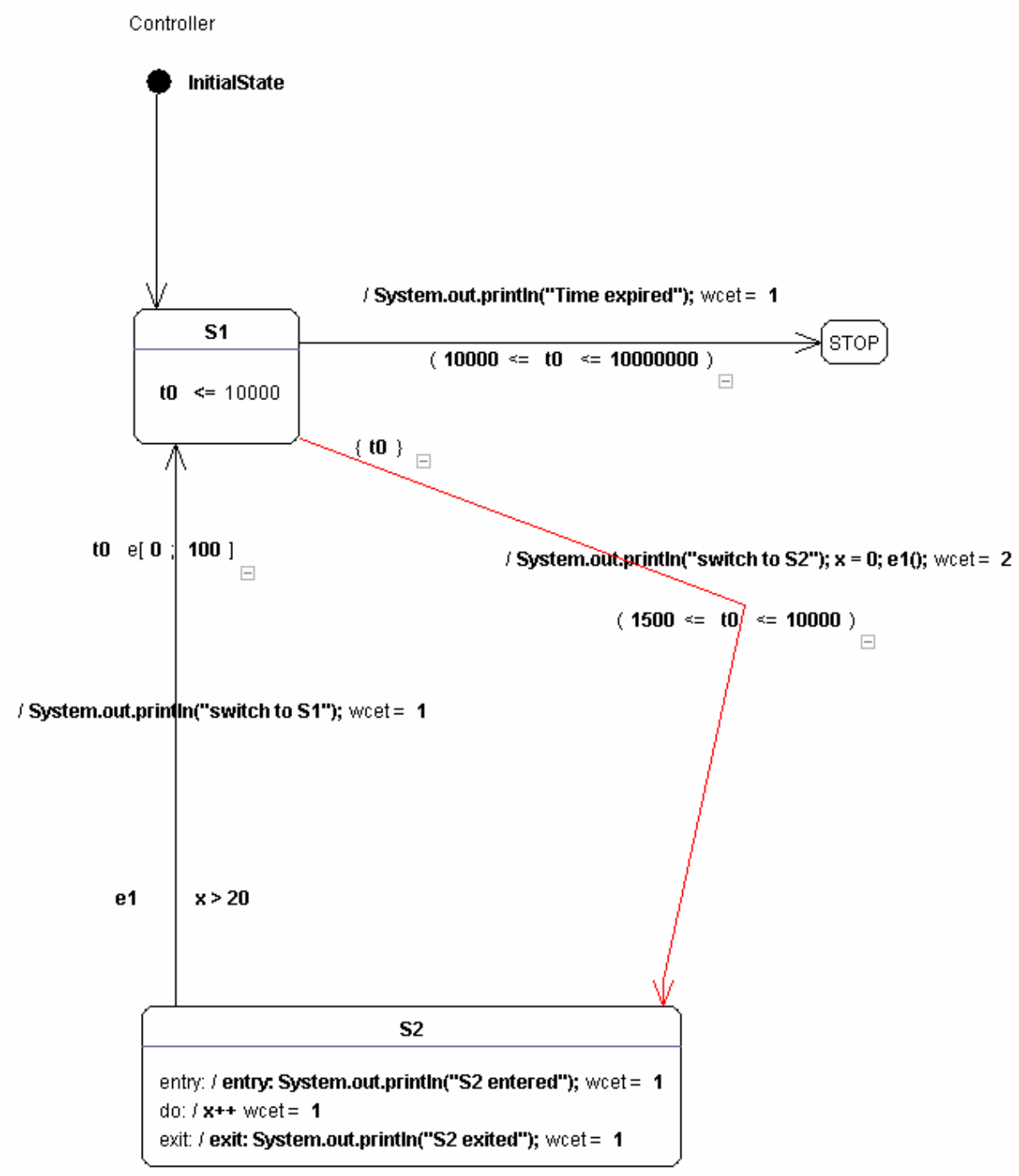

Abbildung 36 - markierte Transition