Vorgenommene Änderungen am Programmcode von *Ptolemy II*

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Im Rahmen meiner Tätigkeit an der TU Darmstadt habe ich verschiedene Änderungen am Programmcode von *Ptolemy II* vorgenommen. Die Möglichkeit, die Simulation durch ein Skript zu steuern, stellt dabei die größte Neuerung dar. Darüber hinaus habe ich der Aktorbibliothek von *Ptolemy II* einige nützliche Aktoren hinzugefügt und kleinere Fehler behoben.

1 Stapelverarbeitung

Jacl [DeJ08] ist ein in der Programmiersprache Java geschriebener Interpreter für die Skriptsprache Tcl. Er ist quelloffen und kann somit um eigene Befehle ergänzt und in eigene Programme eingebunden werden. Mit Hilfe von *Jacl* entstand ein einfaches Programm namens PtolemyBatchApplication, welches von der Kommandozeile aufgerufen wird und als Parameter einen Dateinamen erwartet. Das Programm öffnet die angegebene Datei und versucht, ihren Inhalt als *Tcl*-Skript zu interpretieren und auszuführen. Dabei stehen neben dem üblichen Umfang der Sprache *Tcl* folgende zusätzlichen Befehle zur Verfügung:

`ptloadmodel` lädt ein *Ptolemy-II*-Modell aus einer Datei,

`ptgetparameter` liest den Wert eines Modellparameters aus,

`ptsetparameter` setzt den Wert eines Modellparameters,

`ptrunmodel` startet die Simulation des Modells und

`ptgenerate` erstellt eine gegebene Anzahl von Kopien eines Aktors.

Ferner habe ich einen neuer Datentyp namens TclPtolemyActor definiert. Variablen dieses Typs stellen Referenzen auf einen *Ptolemy-II*-Aktor (bzw. auf ein ganzes Modell, was nur ein Sonderfall eines *Composite Actors* ist).

Der Quelltext aller relevanten *Java*-Klassen findet sich im Verzeichnis `ptolemy/batch`.

1.1 Der Befehl `ptloadmodel`

Der Befehl ptloadmodel erwartet eine Zeichenkette als Parameter, die einen Dateinamen enthält. Der Befehl veranlasst den Interpreter, die angegebene Datei zu öffnen und ihren Inhalt als MoML-Dokument (*Model Markup Language*, eine Beschreibungssprache für *Ptolemy-II*-Modelle) zu interpretieren. Wenn das fehlschlägt, wird eine entsprechende Fehlermeldung ausgegeben. Das Ergebnis des Befehls ist eine Referenz auf das Modell.

1.2 Der Befehl `ptgetparameter`

Der Befehl ptgetparameter erwartet zwei Parameter:

1. Eine Referenz auf ein *Ptolemy-II*-Modell und
2. eine Zeichenkette, die den qualifizierten Namen eines Parameters des Modells enthält.

Der qualifizierte Name eines Parameters besteht aus den durch jeweils einen Punkt (`.`) getrennten Namen aller übergeordneten Aktoren, beginnend mit der höchsten Hierarchieebene, gefolgt von einem Punkt und dem Namen des Parameters[1].

Der Befehl veranlasst den Interpreter, das gegebene Modell nach dem entsprechenden Parameter zu durchsuchen und dessen Wert in Form einer Zeichenkette zurückzugeben. Wenn das fehlschlägt, wird eine entsprechende Fehlermeldung ausgegeben.

1.3 Der Befehl `ptsetparameter`

Der Befehl ptsetparameter erwartet drei Parameter:

1. Eine Referenz auf ein *Ptolemy-II*-Modell,
2. eine Zeichenkette, die den qualifizierten Namen eines Parameters des Modells enthält und
3. eine weitere Zeichenkette, welche den Ausdruck darstellt, der diesem Parameter zugewiesen werden soll.

[1] Beispiel: Der qualifizierte Name des Parameters `factor` des Aktors `Scale`, welcher Teil des Aktors `Cavities` ist, lautet `Cavities.Scale.factor`

Der Befehl veranlasst den Interpreter, das gegebene Modell nach dem entsprechenden Parameter zu durchsuchen und den Ausdruck für dessen Wert entsprechend zu setzen. Wenn das fehlschlägt, wird eine entsprechende Fehlermeldung ausgegeben.

Es können auch Ausdrücke zugewiesen werden, nicht nur konstante Werte. Für Text-Parameter[2] muss der neue Ausdruck mit doppelten Anführungszeichen (`"`) beginnen und enden. Für Listen-Parameter[3] muss der neue Ausdruck mit geschweiften Klammern (`{` bzw. `}`) beginnen bzw. enden und die einzelnen Elemente müssen durch Kommata (`,`) getrennt sein.

Dieser Befehl erzeugt kein Ergebnis.

1.4 Der Befehl `ptrunmodel`

Der Befehl ptrunmodel erwartet eine Referenz auf ein *Ptolemy-II*-Modell als Parameter: Der Befehl veranlasst den Interpreter, die Ausführung des gegebenen Modells zu starten und zu warten, bis die Simulation beendet ist. Wenn der Start der Simulation fehlschlägt, wird eine entsprechende Fehlermeldung ausgegeben. Es wird außerdem unterschieden, ob die Simulation vorzeitig aufgrund eines in der Simulation aufgetretenen Fehlers beendet oder ob sie bis zu ihrem planmäßigen Ende ausgeführt wurde.

Dieser Befehl erzeugt kein Ergebnis.

1.5 Der Befehl `ptgenerate`

Der Befehl ptrunmodel erwartet drei Parameter:

1. Eine Referenz auf ein *Ptolemy-II*-Modell,
2. eine Zeichenkette, die den qualifizierten Namen eines Aktors des Modells enthält und
3. eine ganze Zahl ≥ 1, die angibt, wie viele Instanzen des angegebenen Aktors erstellt werden sollen.

Der qualifizierte Name eines Aktors besteht aus den durch jeweils einen Punkt (`.`) getrennten Namen aller übergeordneten Aktoren, beginnend mit der höchsten Hierarchieebene[4].

Der Befehl veranlasst den Interpreter, das gegebene Modell nach dem entsprechenden Aktor zu durchsuchen, eine entsprechende Anzahl von Kopien dieses Aktors zu erstellen und die Ports dieser Kopien mit den gleichen Relationen zu verbinden, mit denen die Ports des Originals verbunden sind. Wenn das fehlschlägt, wird eine entsprechende Fehlermeldung ausgegeben. Das Ergebnis des Befehls ist eine Liste mit den Namen der erzeugten Instanzen.

Der Name `ptgenerate` ist vom generate-Konstrukt der Sprache VHDL inspiriert, welches eine beliebige Anzahl nebenläufiger Instanzen eines Blocks erzeugt.

Durch Verwendung dieses Befehls kann die Einschränkung umgangen werden, dass die verwendete Version 7 von *Ptolemy II* das dynamische Instanzieren von zeitkontinuierlichen Teilmodellen zu Beginn der Simulation nicht unterstützt. Mit dem MultiInstanceComposite-Konstrukt gibt es zwar die Möglichkeit, Teilmodelle zur Laufzeit mehrfach zu instanzieren, aber das funktioniert mit zeitkontinuierlichen Teilmodellen nicht (siehe Abschnitt 3.2).

2 Neue Aktoren

Im Rahmen meiner Tätigkeit an der TU Darmstadt entstanden mehrere neue Aktoren:

`DeadTime` zeitkontinuierliches Totzeitglied,

`CSVRecordReader` für das Einlesen von Daten aus Dateien im CSV-Format,

`CSVRecordWriter` für die Ausgabe von Daten in Dateien im CSV-Format,

`TappedCircularBuffer` Ringpufferspeicher mit variablen „Anzapfungen“

`CyclicIntegrator` Integrationsglied mit Divisionsrestbildung und

`AntiWindUpIntegrator` Integrationsglied mit Sättigung.

[2] engl. *string parameters*
[3] engl. *array parameters*
[4] Beispiel: Der qualifizierte Name des Aktors `Scale`, welcher Teil des Aktors `Cavities` ist, lautet `Cavities.Scale`

2.1 Der Aktor `DeadTime`

Aktoren dieser Klasse stellen zeitkontinuierliche Totzeitglieder in *Ptolemy II* dar. Diese Aktor-Klasse wurde von Tao Guo im Rahmen seiner Diplomarbeit [Guo08] entwickelt. Aktoren dieser Klasse haben zwei Parameter:

`initialState` (reelle Zahl): Dieser Parameter gibt den Anfangswert der Ausgangsgröße an.

`deadTime` (reelle, positive Zahl): Dieser Parameter gibt die konstante Totzeit an.

Der Eingangswert $x_E(t)$ zum Zeitpunkt t wird zum Ausgangswert $x_A(t + T_T)$ zum Zeitpunkt $t + T_T$, wobei T_T die konstante Totzeit ist. Durch einen Aufruf der Methode `fireAt` der Klasse `Director` im Paket `ptolemy.actor` wird sichergestellt, dass das zeitkontinuierliche Gleichungssystem des Modells zum Zeitpunkt $t + T_T$ erneut ausgewertet wird. Ist also t eine Stützstelle der numerischen Simulation, dann folgt daraus, dass auch $t + T_T$ eine Stützstelle der numerischen Simulation ist. In einem heterogenen Modell gilt das jedoch nicht: Zu jedem beliebigen Zeitpunkt, der kein Vielfaches der Simulationsschrittweite sein muss, kann ein Ereignis ausgelöst werden, welches die Auswertung des zeitkontinuierlichen Gleichungssystems erfordert. Es wird dann eine neue Stützstelle zum Zeitpunkt t eingefügt; daraus folgt aber nicht zwangsläufig, dass auch $t - T_T$ eine Stützstelle der Simulation war. Zur Berechnung des Ausgangswerts $x_A(t)$ zu einem beliebigen Zeitpunkt t muss daher bisweilen zwischen verschiedenen benachbarten Eingangswerten interpoliert werden.

Die Einzelheiten der Implementierung sind in der Diplomarbeit von Tao Guo [Guo08] dokumentiert.

2.2 Der Aktor `CSVRecordReader`

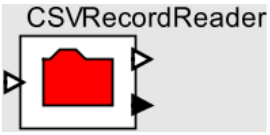

Abbildung 1: Symbol eines CSV-Datenlesers in Vergil

Aktoren dieser Klasse lesen Dateien im CSV-Format [RFC4180] ein und reichen die Daten über einen Ausgabe-Port weiter. Über einen weiteren Ausgabe-Port teilen sie mit, ob weitere Datensätze vorliegen oder nicht. Die einzulesende Datei wird mittels eines Parameters festgelegt. Abbildung 1 zeigt die Darstellung eines solchen Aktors in Vergil.

Die Datei wird zu Beginn der Simulation vollständig in den Speicher eingelesen und dann wieder geschlossen. Auf diese Weise werden Lesezugriffe während der Simulation vermieden und diese somit beschleunigt.

2.2.1 Ports

Aktoren dieser Klasse verfügen über drei Ports:

`trigger` (Eingang): Sobald ein beliebiges Token an diesem Port eintrifft, wird der jeweils nächste Datensatz aus der Datei gelesen.

`output` (Ausgang, Multiport): Sobald ein beliebiges Token am Port `trigger` eintrifft, werden über diesen Port die einzelnen Felder des soeben gelesenen Datensatzes als Zeichenkette ausgegeben. Der erste Kanal erhält das erste Feld, der zweite Kanal das zweite usw.

`endOfFile` (Ausgang): Sobald ein beliebiges Token am Port `trigger` eintrifft, wird über diesen Port ein Boole'scher Wahrheitswert ausgegeben. Falls der soeben gelesen Datensatz der letzte Datensatz der Datei war, wird der Wert WAHR, sonst der Wert FALSCH ausgegeben.

2.2.2 Parameter

Aktoren dieser Klasse haben zwei Parameter:

`fileOrURL` (Zeichenkette): Dieser Parameter enthält den qualifizierten Dateinamen (ggf. einschließlich einer Pfadangabe) der einzulesenden Datei.

`numberOfLinesToSkip` (nichtnegative Ganzzahl): Dieser Parameter gibt die Anzahl der anfangs zu überspringenden Zeilen an, die lediglich die Feldüberschriften enthalten.

2.2.3 Quelltext

Der Quelltext dieser Aktor-Klasse findet sich im Verzeichnis `ptolemy/actor/lib/io`. Dieser Aktor greift auf eine Bibliothek zur Verarbeitung von Daten im CSV-Format zurück; Abschnitt 4 geht auf diese Bibliothek näher ein.

2.3 Der Aktor `CSVRecordWriter`

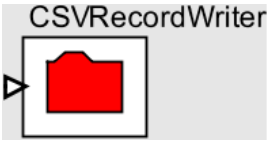

Abbildung 2: Symbol eines CSV-Datenschreibers in Vergil

Aktoren dieser Klasse schreiben Daten in Dateien im CSV-Format [RFC4180]. Der Name der zu schreibenden Datei und der Zugriffsmodus werden mittels Parameters festgelegt. Abbildung 2 zeigt die Darstellung eines solchen Aktors in Vergil.

2.3.1 Ports

Aktoren dieser Klasse verfügen über einen Eingangs-Multiport namens `input`. Sobald Token an diesem Port eintreffen, werden sie in Zeichenketten umgewandelt und zu einem Datensatz zusammengestellt (wobei das Token auf dem ersten Kanal das erste Feld des Datensatz bildet, das Token des zweiten Kanals das zweite Feld usw.). Der Aktor puffert Datensätze im Speicher und schreibt diese erst dann in die Datei, wenn sich eine ausreichende Zahl von Datensätzen angesammelt hat. Auf diese Weise werden viele kleine Schreibzugriffe zu wenigen großen zusammengefasst, was effizienter ist. Falls die Datei neu angelegt wird (darüber entscheidet der Parameter `append`, siehe Abschnitt 2.3.2), so geschieht das zu Beginn der Simulation. Nach dem Ende der Simulation werden die im Speicher verbliebenen Datensätze in die Datei geschrieben und diese dann geschlossen.

2.3.2 Parameter

Aktoren dieser Klasse haben vier Parameter:

`fileOrURL` (Zeichenkette): Dieser Parameter enthält den qualifizierten Dateinamen (ggf. einschließlich einer Pfadangabe) der zu schreibenden Datei. Wenn dieser Dateiname `/dev/null` lautet, also auf das so genannte *Nulldevice* verweist, werden alle eintreffenden Daten verworfen. Das kann sinnvoll sein, um in einem großen Modell die Ausgabe bestimmter Daten zu unterbinden, ohne den Aktor aus dem Modell zu entfernen.

`headlines` (Zeichenkette): Dieser Parameter enthält die an den Anfang der Datei zu stellenden Feldüberschriften.

`append` (Boole'scher Wahrheitswert): Dieser Parameter gibt an, wie verfahren werden soll, wenn die Datei bereits existiert. Der Wert `WAHR` bedeutet, dass die neuen Daten an das Ende der Datei angehängt werden sollen, der Wert `FALSCH`, dass die Datei überschrieben werden soll.

`confirmOverwrite` (Boole'scher Wahrheitswert): Dieser Parameter gibt an, wie verfahren werden soll, wenn die Datei bereits existiert und überschrieben werden soll. Der Wert `WAHR` bedeutet, dass hierfür die Zustimmung des Benutzers erforderlich ist, der Wert `FALSCH`, dass die Datei ohne Nachfrage überschrieben werden kann. Wenn der Dateiname `stdout` (*Standardausgabe* auf die Konsole) oder `/dev/null` lautet, ist die Zustimmung des Benutzers auch dann nicht erforderlich, wenn der Wert dieses Parameters `WAHR` ist.

2.3.3 Quelltext

Der Quelltext dieser Aktor-Klasse findet sich im Verzeichnis `ptolemy/actor/lib/io`. Dieser Aktor greift auf eine Bibliothek zur Verarbeitung von Daten im CSV-Format zurück; Abschnitt 4 geht auf diese Bibliothek näher ein.

2.4 Der Aktor `TappedCircularBuffer`

Aktoren dieser Klasse stellen Ringpufferspeicher mit Anzapfungen dar und werden zur Modellierung von Digitalfiltern mit dünn besetztem Koeffizientenvektor benutzt (zu deren Aufbau und Verwendung siehe [Han06; The07; PGK10; Sur+11; Sam+11; Kli+07]). Abbildung 3 zeigt die Darstellung eines solchen Aktors in Vergil.

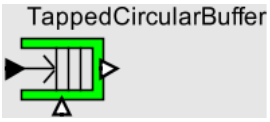

Abbildung 3: Symbol eines Ringpufferspeichers mit Anzapfungen in Vergil

2.4.1 Ports

Aktoren dieser Klasse verfügen über drei Ports:

`input` (Eingang): An diesem Eingang eintreffende Daten werden in den Pufferspeicher geschrieben. Jedes weitere eintreffende Datum verschiebt die bereits im Pufferspeicher befindlichen Daten um eine Stelle. Das älteste im Speicher befindliche Datum wird gelöscht.

`trigger` (Eingang, Multiport): Dieser Eingang erwartet ganzzahlige Werte. Sobald auf irgendeinem Kanal dieses Ports ein Token eintrifft, wird auf dem entsprechenden Kanal des Ports `output` ein im Speicher befindlicher Wert ausgegeben. Das eingehende Token wird dabei als Index in den Speicher verwendet: Trifft der Wert 0 ein, so wird das neueste Datum ausgegeben, der Wert 1 bezeichnet das zweitneueste Datum usw.

`output` (Ausgang, Multiport): Sobald ein Index-Token am Port `trigger` eintrifft, wird über diesen Port das durch den Index ausgewählte Datum aus dem Speicher ausgegeben.

2.4.2 Parameter

Aktoren dieser Klasse haben zwei Parameter:

`defaultValue` : Dieser Parameter gibt den anfänglichen Inhalt aller Speicherzellen an.

`capacity` (positive Ganzzahl): Dieser Parameter gibt die Anzahl der Speicherstellen an.

2.4.3 Quelltext

Der Quelltext dieser Aktor-Klasse findet sich im Verzeichnis `ptolemy/domains/de/lib`.

2.5 Der Aktor `CyclicIntegrator`

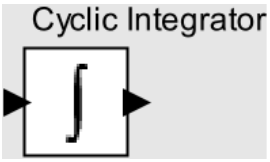

Abbildung 4: Symbol eines Integrators mit Divisionsrestbildung in Vergil

Aktoren dieser Klasse stellen Integrationsglieder mit eingebauter Divisionsrestbildung dar. Diese sind beispielsweise bei der Integration von Frequenzen (bzw. Frequenzdifferenzen) zu (Phasen bzw. Phasendifferenzen) nützlich: Phasen sind auf den Wertebereich von $-\pi$ bis $+\pi$ bzw. von 0 bis 360° begrenzt. Durch eine Divisionsrestbildung durch $2 \cdot \pi$ beispielsweise kann der Wertebereich $\mathbb{R}$ der reellen Zahlen auf das Intervall $[0; 2\pi)$ abgebildet werden. Abbildung 4 zeigt die Darstellung eines solchen Aktors in Vergil.

Aktoren dieser Klasse haben drei Parameter:

`initialState` (reelle Zahl): Dieser Parameter gibt den Anfangswert der Ausgangsgröße an.

`lowerBound` (reelle Zahl): Dieser Parameter gibt den Minimalwert der Ausgangsgröße an.

`upperBound` (reelle Zahl): Dieser Parameter gibt den Maximalwert der Ausgangsgröße an.

Wird der Minimalwert auf $-\infty$ und der Maximalwert auf $+\infty$ gesetzt, so verhält sich dieser Aktor wie ein gewöhnlicher Integrator.

Der Quelltext dieser Aktor-Klasse findet sich im Verzeichnis `ptolemy/domains/ct/lib`.

2.6 Der Aktor `AntiWindUpIntegrator`

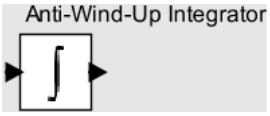

Abbildung 5: Symbol eines Integrators mit Sättigung in Vergil

Aktoren dieser Klasse stellen Integrationsglieder mit eingebauter Sättigung dar. Diese sind bei der Modellierung analoger Regler mit Stellgrößenbeschränkung nützlich. Sobald der Ausgangswert des Integrators den vorgegebenen Maximalwert erreicht, führt eine positive Eingangsgröße nicht zu einer weiteren Zunahme des Ausgangswerts. Ebenso führt eine negative Eingangsgröße nicht zu einer weiteren Abnahme des Ausgangswerts, wenn der Ausgangswert den vorgegebenen Minimalwert erreicht. Abbildung 5 zeigt die Darstellung eines solchen Aktors in Vergil.

Aktoren dieser Klasse haben drei Parameter:

`initialState` (reelle Zahl): Dieser Parameter gibt den Anfangswert der Ausgangsgröße an.

`lowerBound` (reelle Zahl): Dieser Parameter gibt den Minimalwert der Ausgangsgröße an.

`upperBound` (reelle Zahl): Dieser Parameter gibt den Maximalwert der Ausgangsgröße an.

Wird der Minimalwert auf $-\infty$ und der Maximalwert auf $+\infty$ gesetzt, so verhält sich dieser Aktor wie ein gewöhnlicher Integrator.

Der Quelltext dieser Aktor-Klasse findet sich im Verzeichnis `ptolemy/domains/ct/lib`.

3 Fehlerbehebung

Bei der Arbeit mit *Ptolemy II* (Version 7.0.1) zeigten sich vier ärgerliche Fehler:

- Im Laufe der Simulation eines heterogenen Systems werden die Simulationsschrittweiten immer kleiner. Im Extremfall kommt die Simulation zum Stillstand.
- Das MultiInstanceComposite-Konstrukt erlaubt es, Teilmodelle zur Laufzeit mehrfach zu instanzieren, aber das funktioniert mit zeitkontinuierlichen Teilmodellen nicht.
- Halteglieder erster Ordnung (`FirstOrderHold`) verhalten sich nicht identisch zur Kombination aus Halteglied nullter Ordnung (`ZeroOrderHold`) und Integrator.
- Bei der Verwendung von Übertragungsfunktionen im Laplace-Bereich zur Modellierung analoger Regler kam es zu zufälligen Abbrüchen der Simulation mit der Fehlermeldung „Graph is cyclic“.

3.1 Kleiner werdende Schrittweiten

Auch wenn in *Ptolemy II* zur Lösung von Differentialgleichungen ein Algorithmus mit fester Schrittweite gewählt wird, ist die Schrittweite nicht konstant. Der Grund dafür liegt in der Behandlung asynchroner Ereignisse, die zu jedem Zeitpunkt auftreten können — auch zwischen zwei Simulationsschritten. Die Simulationsschrittweite wird in diesen Fällen entsprechend verringert und das Modell zum Zeitpunkt des asynchronen Ereignisses neu ausgewertet.

Der Fehler bestand darin, dass danach die Schrittweite nicht wieder auf den alten Wert gesetzt, sondern die kleinere Schrittweite weiter verwendet wurde. Früher oder später bewirkte dann ein anderes asynchrones Ereignis die erneute Herabsetzung der Schrittweite. Die Schrittweite wurde so im Laufe der Simulation immer kleiner und im Extremfall stand die Simulation irgendwann faktisch still.

Die Lösung des Problems bestand darin, den Rückgabewert der Methode `integratorPredictedStepSize` der Klasse `FixedStepSolver` im Paket `ptolemy.domains.ct.kernel.solver` zu verändern:

Listing 1: Ursprünglicher Quelltext der Methode `integratorPredictedStepSize`

```
120 public final double integratorPredictedStepSize(CTBaseIntegrator integrator) {
121   CTDirector director = (CTDirector) getContainer();
122   return director.getCurrentStepSize();
123 }
```

Listing 2: Modifizierter Quelltext der Methode `integratorPredictedStepSize`

```
120 public final double integratorPredictedStepSize(CTBaseIntegrator integrator) {
121   CTDirector director = (CTDirector) getContainer();
122   return director.getInitialStepSize();
123 }
```

Auf diese Weise wird die Schrittweite immer wieder auf ihren anfänglichen Wert zurückgesetzt.

3.2 Dynamische Instanzierung zeitkontinuierlicher Teilmodelle

Mit dem `MultiInstanceComposite`-Konstrukt bietet *Ptolemy II* die Möglichkeit, ein Teilmodell beliebig oft zu instanzieren. Der Aktor `MultiInstanceComposite` ist ein „Behältnis“ für ein Untermodell und hat einen Parameter `nInstances`, der angibt, wie viele Instanzen dieses Modells erzeugt werden sollen. Das Untermodell muss über einen eigenen *Director* verfügen.

Die Instanzierung zeitkontinuierlicher Teilmodelle ist in der in dieser Arbeit verwendeten Version 7 von *Ptolemy II* mit diesem Konstrukt jedoch nicht möglich: Für zeitkontinuierliche Modelle existieren zwei verschiedene *Directors*, nämlich `CTDirector` und `CTEmbeddedDirector`. Ersterer kann nur auf der höchsten Hierarchieebene verwendet werden, letzterer ist für zeitkontinuierliche Teilmodelle von Modellen anderer Domänen gedacht. Da bereits ein `CTDirector` auf der obersten Ebene des Modells verwendet wird, kann innerhalb des `MultiInstanceComposite` nicht noch ein `CTDirector` verwendet werden. Die Verwendung eines `CTEmbeddedDirector` scheitert dagegen daran, dass dieser erwartet, dass sein *Container* die von der Schnittstellen-Klasse `CTStepSizeControlActor` vorgesehenen Methoden zur Schrittweitensteuerung implementiert. Das `MultiInstanceComposite` implementiert diese Schnittstelle jedoch nicht.

In der seit Oktober 2010 verfügbaren Version 8 von *Ptolemy II* besteht das Problem nicht mehr, da die Domäne CT durch die Domäne *Continuous* mit erweiterter Funktionalität ersetzt wurde. Edward A. Lee schrieb zu diesem Thema in der Diskussionsgruppe ptolemy-hackers (`ptolemy-hackers@lists.eecs.berkeley.edu`) am 18. April 2009:

> The solution for this is to use the ContinuousDirector rather than CTDirector. However, this domain is not quite finished yet. The actor library is incomplete, for example. It has also not been thoroughly tested, and the current scheduler is probably somewhat inefficient.
>
> Basically, CTDirector taught us quite a bit, including how to implement composable continuous-time models correctly. The approach is documented here:
>
> `http://ptolemy.eecs.berkeley.edu/publications/papers/07/unifying/index.htm`
>
> I hope we can finish this within the next few months...

Das Problem kann auch durch die in Abschnitt 1 beschriebene Stapelverarbeitung und den darin enthaltenen Befehl `ptgenerate` (siehe Abschnitt 1.5) umgangen werden [SGK10].

3.3 Halteglieder erster Ordnung

In *Ptolemy II* gibt es Halteglieder „nullter“ und erster Ordnung. Ein Halteglieds nullter Ordnung (`ZeroOrderHold`) hat einen Eingang x_E und sein Ausgang x_A bleibt nach dem Eintreffen eines Datums am Eingang so lange konstant, bis das nächste Datum am Eingang eintrifft. Ein Halteglied erster Ordnung (`FirstOrderHold`) hat zwei Eingänge x_{E1} und x_{E2} und sein Ausgang x_A wächst nach dem Eintreffen von Daten am Eingang linear mit der Rate x_{E2} an, ausgehend vom Startwert x_{E1}. Seien t_1 und t_2 zwei aufeinanderfolgende Zeitpunkte, zu denen Daten am Eingang eintreffen. Dann gilt für für ein Halteglieder nullter Ordnung

$$x_A(t) = x_E(t_1),\ t_1 \leq t < t_2 \tag{1}$$

und für ein Halteglied erster Ordnung

$$x_A(t) = x_{E1}(t_1) + x_{E2}(t_1) \cdot (t - t_1),\ t_1 \leq t < t_2. \tag{2}$$

Der Ausgang $x_A(t)$ ist zum Zeitpunkt t_2 genau dann stetig, wenn $x_{E1}(t_2) = x_{E1}(t_1) + x_{E2}(t_1) \cdot (t_2 - t_1)$. Ein Halteglied erster Ordnung sollte sich in diesem Fall genau so verhalten wie ein Halteglied nullter Ordnung mit nachgeschaltetem Integrationsglied.

Offenbar ist das aber nicht immer der Fall: Die Ausgangswerte eines Halteglieds nullter Ordnung mit nachgeschaltetem Integrationsglied unterscheiden sich unter bestimmten Umständen von denen eines Halteglieds erster Ordnung. Grund dafür ist offenbar, dass die Aktor-Klasse `FirstOrderHold` die Schnittstellen-Klasse `CTWaveformGenerator` implementiert, die Aktor-Klasse `Integrator` jedoch die Schnittstellen-Klasse

`CTDynamicActor`. Das führt dazu, dass Aktoren der beiden Klassen in unterschiedlichen Phasen der Simulation vom *Director* aufgerufen werden, wodurch sich unterschiedliche Ausgangswerte ergeben.

Statt Haltegliedern erster Ordnung verwende ich deshalb stets Halteglieder nullter Ordnung mit nachgeschalteten Integrationsgliedern. Das ist natürlich nur so lange möglich, wie die fraglichen Signale keine Unstetigkeitsstellen aufweisen.

3.4 „Graph is cyclic"

Bei der Verwendung von Aktoren der Klasse `ContinuousTransferFunction`, welche zeitkontinuierliche Übertragungsfunktionen im Laplace-Bereich darstellen, kommt es offenbar zufällig zu einem Abbruch der Simulation mit der Fehlermeldung „Graph is cyclic", wenn ein explizites Lösungsverfahren zum Einsatz kommt. Grund dafür ist, dass *Ptolemy II* zur Laufzeit die angegebe Übertragungsfunktion in ein entsprechendes lineares Teilsystem aus Additions- bzw. Subtraktionsgliedern, Integrations- und Verstärkungsgliedern umsetzt. Dabei kann es vorkommen, dass Verstärkungsglieder mit einer Verstärkung von Null instanziert werden. *Ptolemy II* ist nicht in der Lage, Verstärkungsglieder mit einer Verstärkung von Null aus dem Modell zu entfernen. Liegen diese Verstärkungsglieder im Rückkopplungszweig des Gesamtsystems, so erkennt *Ptolemy II* folglich eine algebraische Schleife, die ein explizites Lösungsverfahren nicht selbsttätig auflösen kann.

4 Verarbeitung tabellarischer Daten im CSV-Format

Im Rahmen verschiedener Kooperationsprojekte mit der GSI entstand eine *Java*-Bibliothek zur Verarbeitung von Daten im CSV-Format [RFC4180]. Die GSI strebt nämlich an, die im Bereich der Hochfrequenz-Regelsysteme verwendeten Dateiformate auf Basis jenes Formats zu standardisieren [KZ11]. Abbildung 6 zeigt das Klassendiagramm dieser Bibliothek. Der Quelltext dieser Bibliothek findet sich im Verzeichnis `gsi/rf/data`.

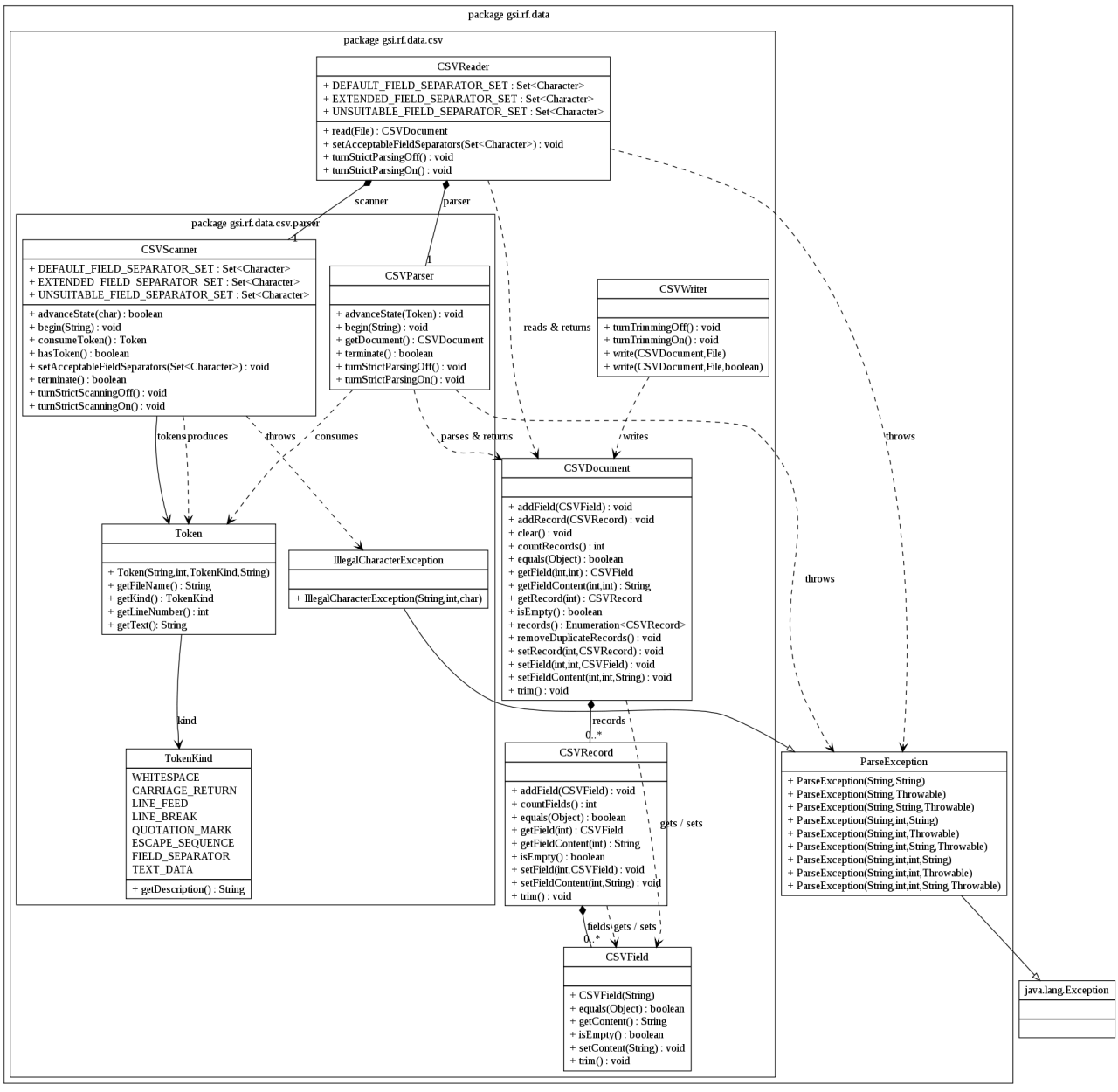

Abbildung 6: UML-Klassendiagramm für die Verarbeitung von Daten im CSV-Format

Literaturverzeichnis

[DeJ08] Mo DeJong. The Tcl/Java Project. 2008. URL: http://tcljava.sourceforge.net.

[Guo08] Tao Guo. "Analyse und Implementierung eines Lösungsverfahrens für Differentialgleichungen mit Totzeit in Ptolemy II". Diplomarbeit. Technische Universität Darmstadt, Fachgebiet Mikroelektronische Systeme, 2008.

[Han06] Liang Han. "Reconfigurable FIR Filter for Large Tap Distances and Low Tap Counts". Bachelor Thesis. Technische Universität Darmstadt, Fachgebiet Mikroelektronische Systeme, 2006.

[Kli+07] Harald Klingbeil, Bernhard Zipfel, Martin Kumm und Peter Moritz. "A Digital Beam-Phase Control System for Heavy-Ion Synchrotrons". In: IEEE Transactions on Nuclear Science 54.6 (2007), S. 2604–2610.

[KZ11] Harald Klingbeil und Bernhard Zipfel. "Data Analysis File Formats for RF Applications". Version 0.80. Mai 2011.

[PGK10] Surapong Pongyupinpanich, Manfred Glesner und Harald Klingbeil. "Implementation of Realtime Pipeline-Folding 64-Tap Filters on FPGA". In: Proceedings of the Conference on Ph. D. Research in Microelectronics and Electronics. 2010, S. 1–4.

[RFC4180] Internet Engineering Task Force, Hrsg. Common Format and MIME Type for Comma-Separated Values (CSV) Files. Request for Comments 4180 (2005).

[Sam+11] Faizal Arya Samman, Pongyupinpanich Surapong, Christopher Spies und Manfred Glesner. "Floating-point-based hardware accelerator of a beam phase-magnitude detector and filter for a beam phase control system in a heavy-ion synchrotron application". In: Proceedings of the International Conference on Accelerators and Large Experimental Physics Control Systems. 2011, S. 683–686.

[SGK10] Christopher Spies, Manfred Glesner und Harald Klingbeil. "Statusbericht zum GSI-Projekt 'Systemmodellierung und Vernetzung der digitalen Mehr-Kavitäten-Regelung für das FAIR-Projekt' — A Standard Scenario for Future Design Space Exploration". Berichtszeitraum Januar bis Juni 2010. Dez. 2010.

[Sur+11] Pongyupinpanich Surapong, Christopher Spies, Manfred Glesner und Harald Klingbeil. "Design of frequency-variable digital filters for beam phase control". In: GSI Scientific Report (2011), S. 340.

[The07] Alexander Theisen. "Implementierung, Optimierung und Systemintegration eines FIR-Filters in VHDL". Bachelor Thesis. Technische Universität Darmstadt, Fachgebiet Mikroelektronische Systeme, 2007.