PEARL-News

Ausgabe 24 November 2006

Mitteilungen der GI-Fachgruppe 'real-time' Echtzeitsysteme und PEARL

ISSN 1437-5966

Impressum

Herausgeber	GI-Fachgruppe 'real-time' Echtzeitsysteme und PEARL URL: http://www.real-time.de
Sprecher	Dr. P. Holleczek Universität Erlangen-Nürnberg, Regionales Rechenzentrum Martensstraße 1, D-91058 Erlangen Telefon: 09131/85-27817 Telefax: 09131/30 29 41 E-Mail: holleczek@rrze.uni-erlangen.de
Stellvertreterin	Prof. Dr.-Ing. B. Vogel-Heuser Universität Kassel FB16-Elektrotechnik/Informatik Wilhelmshöher Allee 73, D-34121 Kassel Telefon: 0561/804-6020 E-Mail: vogel-heuser@uni-kassel.de
Redaktion	Prof. Dr. R. Baran HAW Hamburg, Fakultät Technik und Informatik E-Mail: baran@cpt.haw-hamburg.de Berliner Tor 7, D-20099 Hamburg
ISSN	1437-5966

Redaktionell abgeschlossen am 24. November 2006

Inhaltsverzeichnis

1 Editorial

Ab der vorliegenden Ausgabe betreuet ein neuer Redakteur die PEARL News. Daraus zu schlussfolgern, dass diese nun weniger auf die Unterstützung und die Beiträge aus dem Kreis der Mitglieder der GI-Fachgruppe Echtzeitsysteme angewiesen seien als seine Vorgänger, wäre allerdings vollkommen falsch. Diese Zeitschrift soll schließlich nicht nur Mitteilungsblatt sein, sonder auch eine Plattform für den Informations- und Meinungsaustausch zwischen allen an den Fragen der Echtzeitprogrammierung Interessierten. Diskussionsstoff bzw. offene Fragen gibt es auf unserem Gebiet reichlich. Eine dieser Fragen wird in dem folgenden Beitrag von Prof. Pareigis über die Echtzeitaspekte bei der Implementation von hierarchischen Automaten angesprochen. Vielleicht bringt dieser Artikel neuen Diskussionsstoff in den Arbeitskreis systematische Software-Entwicklung / Modellierung. Auch Ideen für ein neues Logo der Fachgruppe sind sehr willkommen. Ich möchten Sie, liebe Leserinnen und Leser, daher ausdrücklich ermuntern, auch in Zukunft die PEARL News durch Ihre Beiträge mit zu gestalten.

Reinhard Baran

2 Reisestipendien

2.1 Die Gewinner

Die Gewinner der von der GI-FG Echtzeitsysteme in 2006 ausgelobten Reisestipendien stehen fest - eine wirkungsvolle Vertretung der deutschen Echtzeit-Szene durch Nachwuchs-Wissenschaftler ...

Ausgeschrieben waren Reisestipendien zu internationalen Tagungen über Echtzeitsysteme. Den Gewinnern winkten Erstattungen von bis zu Eur 500.

Die Fachgruppe gratuliert den Gewinnern!

Peter Holleczek (Sprecher der FG)

Birgit Vogel-Heuser (Stellvertreterin)

2.2 Korte

Herr Korte ist Student der Informatik mit Schwerpunkt Eingebettete Systeme an der Universität Oldenburg. Der prämierte Beitrag zur Maple Conference 2006 in Waterloo, Canada, 23.-26.07.2006, ‚Design and implementation of a Maple-Package for the predictability of real-time systems‘.

2.3 Lehnhoff

Herr Lehnhoff ist Doktorand am Lehrstuhl für Betriebssysteme und Rechnerarchitektur des Fachbereichs Informatik an der Universität Dortmund. Der prämierte Beitrag zur 18th Euromicro Conference on Realtime Systems (ECRTS'2006), 5.-7.7.2006, Dresden, lautet: ‚Real-Time Multi-Agent Support for Decentralized Management of Electric Power‘.

2.4 Skambraks

Herr Skambraks ist wissenschaftlicher Mitarbeiter am Lehrstuhl für Informationstechnik der FernUniversität Hagen. Der prämierte Beitrag zur 11th IEEE International Conference on Emerging Technologies

and Factory Automation (EFTA'2006), 20.- 22.9.2006, Prag, lautet: ‚An Architecture for Runtime State Restoration after Transient Hardware-Faults in Redundant Real-Time Systems'.

3 Aus dem Arbeitskreis: Echtzeit in der Ausbildung und PEARL

Auf der diesjährigen FGL-Sitzung wurde entschieden, daß die Leitung des Arbeitskreises nach der Emeritierung von Herrn Thiele auf mich übergehen solle. An dieser Stelle möchte ich Herrn Thiele für seine Arbeit zu danken. Unter seiner Leitung des Arbeitkreises ist es zum Beispiel gelungen die DIN-Norm zu erreichen, den Sprachreport mit starker Hilfe der Fa. Werum in englischer Sprache aufzulegen.

- Nachstehende Arbeitspunkte wurden laufend verfolgt:
- Wartung der Literaturliste zu PEARL
- Wartung der Homepage des Arbeitskreises
- Wartung der Liste von Institutionen mit PEARL-Anteilen in der Ausbildung
- Vertretung von PEARL beim DIN über dessen Ausschuß NI-22
- Wartung der Liste mit Industrieprojekten mit (wesentlichen) PEARL-Anteilen

Ich würde mich freuen, wenn sich mehrere Schultern finden diese Arbeiten zu übernehmen.

Aus Gesprächen bei den Nachsitzungen bei Ömaünd der Tagungsrückreise per Bahn sind mir noch folgende interessante Punkte in Gedächtnis geblieben:

- Lehrbuch zum Thema SSoftwareengineering von Echtzeitsytemen"
- PEARL für Realtime Linux

Sie haben sicherlich weitere Ideen für Betätigungsfelder, die wir beim diesjährigen Treffen am 30.11. am Rande des Workshops besprechen können. Ich fände es schön ein konkretes neues Ziel vor Augen zu haben, das im Arbeitskreis von einigem Mitgliedern zusammen verfolgt wird. Vielleicht gelingt es auch neue Mitstreiter für eine konkrete neue Aufgabe zu finden. Interessierte Leser sind herzlich eingeladen sich direkt bei mir zu melden.

Der DIN Ausschuß NI-22 hatte in diesem Herbst wegen Terminprobleme der Mitglieder keine Sitzung. Ein Lebenszeichen von PEARL habe ich mit einem kleinen Statusbericht abgegeben.

R. Müller mueller@fh-furtwangen.de

4 Implementation von hierarchischen Zustandsautomaten in Echtzeitsystemen

4.1 Einleitung

Zustandsautomaten sind ein verbreitetes Werkzeug zur Verhaltensmodellierung von reaktiven Systemen. Üblicherweise werden hierbei hierarchische Automaten verwendet, deren Zustände Entry- und Exit Code besitzen, die Unterzustände speichern können (History-Funktionalität) und die auf *extended state variables* zugreifen können (siehe [4],[7]).

Es existieren verschiedene Vorschläge zur Implementation von Zustandsautomaten ([1],[3],[6]). Alle Vorschläge enthalten Kompromisse zwischen Performanz, Speicherbedarf und Erweiterbarkeit.

In unserem Beitrag diskutieren wir eine Implementationsmethode für hierarchische Zustandsautomaten, welche einige der typischen Implementationsprobleme behebt und besonders geeignet für Echtzeitsysteme ist. Wir verwenden für den Zustandswechsel ein spezielles Sprachkonstrukt der Sprache C++,

den `placement new`-Operator. Die Hierarchie des hierarchischen Automaten bilden wir unmittelbar auf die Vererbungshierarchie der Zustände ab. Ein dabei entstehendes Problem der Ausführung von `exit`-Funktionen lösen wir durch Verwendung einer geeigneten Aspektklasse.

In Abschnitt 4.2 geben wir zunächst kurz einen Überblick über die vorhandenen Implementationsmethoden und deren Problematik. Danach beschreiben wir unsere Methode für den Zustandswechsel.

In Abschnitt 4.3 ist die Implementationsmethode eines hierarchischen Automaten beschrieben, zusammen mit dem aspektorientierten Ansatz zur Implementation von `exit`-Funktionen.

4.2 Zustandswechsel

In [6] findet man eine ausführliche Diskussionen zu unterschiedlichen Implementationsmöglichkeiten von hierarchische Automaten. Dort werden auch nicht-objektorientierte Ansätze untersucht. Als optimale Lösung wird Vererbung in Kombination mit `switch-case` Anweisungen vorgeschlagen. Ein Vergleich der verschiedenen Entwurfsmuster für Zustandsautomaten und deren Bewertung findet man in [1].

Unser Ansatz kommt ohne `switch-case`-Anweisungen aus und ist damit nicht nur übersichtlicher sondern auch performanter, insbesondere bei hierarchischen Automaten mit sehr vielen Zuständen. Zudem muss die Kapselung der Kontext-Klasse nicht mit `friend` aufgebrochen werden. Letzteres ist eine typische Problematik beim Zustandsentwurfsmuster von Gamma et al [3].

Wir verwenden als Grundlage das Zustandsentwurfsmuster (state design pattern) von [3] welches wir im folgenden kurz beschreiben. Ein vereinfachtes UML-Diagramm haben wir in Abbildung 1 dargestellt.

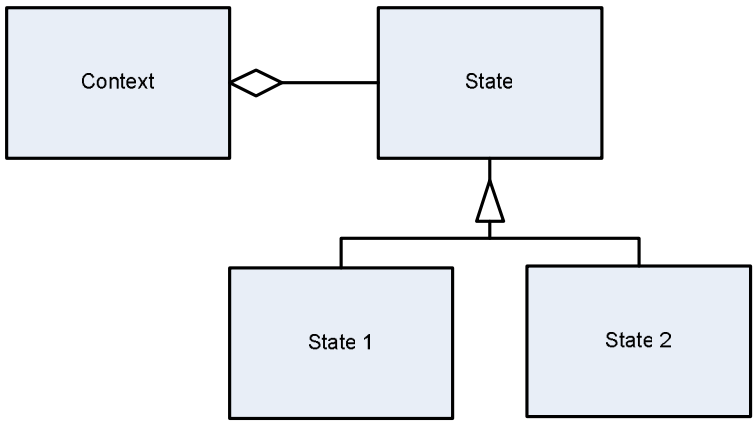

Abbildung 1: UML-Diagramm für das Zustandspattern nach [3]

Zustände werden polymorph genutzt. Eine Kontext-Klasse (oder Aktor-Klasse) aggregiert einen Zustand, der verschiedene Ausprägungen haben kann. Signale werden verarbeitet, indem die entsprechende (z.B. gleichnamige) polymorphe Operation der Zustand-Basisklasse aufgerufen wird. Abhängig von der konkreten Zustandsklasse wird der entsprechende Transitionscode ausgeführt. Die Implementation arbeitet intern mit virtuellen Funktionstabellen (vtbl) und ist damit sehr performant.

Um den Zustand zu wechseln, muss der Variablen der Kontextklasse, die den aktuellen Zustand hält, der neue Zustand zugewiesen werden. Üblicherweise möchte man die Zuweisung nicht in der Kontextklasse machen, sondern in den Zuständen selbst (genauer: im Transitionscode). Diese sollen nämlich ihre Folgezustände kennen. Durch diese Anforderung ergibt sich die Schwierigkeit, dass den Zuständen die entsprechende Variable in der Kontextklasse bekannt sein muss. Dies erfordert eine Abhängigkeit von den Zuständen zurück zur Kontextklasse.

In unserem Ansatz geschieht der Zustandswechsel dadurch, dass im Transitionscode des aktuellen Zustands der Folgezustand an die Speicherstelle des aktuellen Zustands geschrieben wird. Der aktuelle Zustand überschreibt sich also selbst mit dem Folgezustand. In C++ ist dies mit dem `placement new` Operator möglich. Dieser erlaubt es, ein neues Objekt zu erzeugen und als Argument die Speicherstelle zu übergeben an der das Objekt erzeugt werden soll. Eine Transition, die vom Zustand `StateOld` zum Zustand `StateNew` wechselt, sieht dann so aus

```
#include "StateOld.h"
#include "StateNew.h"

void StateOld::change2New()
{
    new (this) StateNew;
}
```

Neben der Einfachheit dieses Codes, in der unmittelbar der Folgezustand deutlich wird, umgeht man auch die Nachteile, welche sich sonst durch Aufbrechen der Kapselung und Singletonbildung ergeben, wie es in [3] oder [6] geschieht.

Außerdem ergibt sich ein Speicherplatzvorteil: die Objektgröße eines jeden Zustandsobjekts ist 4 Byte. Dies ergibt sich durch die Art der Datenhaltung. Daten können entweder statisch in der Zustandsbasisklasse gehalten werden oder in der Kontextklasse wie in [5] beschrieben. Diese 4 Byte enthalten (vom Anwender verborgen) den Zeiger auf die virtuelle-Funktionstabelle des aktuellen Zustandsobjekts. Bei dem Vorgang `new (this) StateNew;` werden also nur 4 Byte kopiert. Im Gegensatz zu den anderen Implementationen müssen aber nicht alle möglichen Zustände im Speicher (heap) gehalten werden: sie werden jedes mal neu erzeugt. Allerdings immer wieder an der gleichen Stelle, so dass auch keine Speicherverwaltung in Anspruch genommen wird.

Diese sehr starke Vereinfachung bei der Implementation des Zustandswechsels hat zur Folge, dass sich auch andere Eigenschaften besonders einfach implementieren lassen.

4.3 Hierarchische Automaten

Für die Implementation von hierarchischen Automaten werden oft verschiedene Entwurfsmuster kombiniert. Das Composite Pattern aus [3] wird zusammen mit dem State Design Pattern (SDP) verwendet, um die Schachtelung von Zuständen in einem hierarchischen Automaten zu implementieren. Beispiele für solche Kombinationen sind das Composite State Pattern oder auch das Hierarchical Statechart Pattern. Ein Überblick über verschiedene Möglichkeiten wird in [1] zusammenfassend dargestellt. Viele dieser Techniken haben den Nachteil, dass die Struktur aufgrund der Mischung der verschiedenen Muster teilweise kompliziert wird, was die Anwendung weiterhin erschwert. Das Gleiche trifft für die Implementation eines hierarchischen Automaten aus [6] zu.

Wir wenden die Vererbungstechnik auf das SDP an. Auf diese Weise kann zusätzlich zur Verwendung der notwendigen Laufzeitpolymorphie die für hierarchische Automaten typische Verhaltensvererbung realisiert werden. Hierbei spiegelt die entstehende Baumstruktur die Schachtelung der Zustände wieder. Das Verständnis und damit die Anwendung des Musters werden im Vergleich zu den anderen Mustern vereinfacht. In Abbildung 4.3 ist ein einfacher hierarchischer Automat dargestellt zusammen mit der zugehörigen Klassenstruktur.

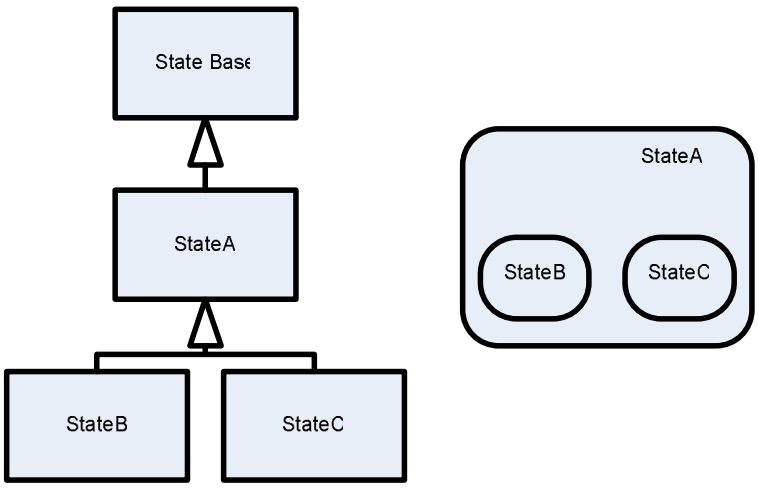

Abbildung 2: Hierarchischer Zustandsautomat und Implementation durch Vererbung

In [2] wird ähnlich wie bei uns Vererbung für die Implementation von hierarchischen Automaten angewendet. Eine angenehme Eigenschaft dieser Vorgehensweise ist insbesondere, dass nicht immer alle Signal-Operationen auf allen Hierarchieebenen implementiert werden müssen. Ist ein Signal in einem inneren Zustand nicht definiert, so wird automatisch die Signal-Operation des äußeren Zustands aufgerufen. Ist sie dort auch nicht definiert, so wird in der Zustandshierarchie der Nächste genommen.

Eine Problematik ergibt sich jedoch, wenn `exit`-Funktionen beim Verlassen eines jeden Zustands aufgerufen werden sollen. Man könnte jetzt alle Signal-Operationen aller Zustände definieren und die `exit`-Funktion explizit aufrufen. Der Code dazu sieht so aus:

```
void InnererZustand::SignalXY()
{
    exit(); // dies ist der exit-code des Zustands
    new (this) AeussererZustand; // Zustandswechsel
    SignalXY(); // Signal an Superzustand delegieren
}
```

Hier wird die `exit`-Funktion des inneren Zustands ausgeführt, der Zustand gewechselt und dann die Signal-Operation erneut aufgerufen.

Wir schlagen ein Design vor, welches der aspektorientierten Programmierung entstammt: Es wird eine einzige generische Klasse `ExitAspect` geschaffen, die alle Signal-Operationen implementiert und jeweils nur eine Funktion `notImplemented()` aufruft. Diese Funktion `notImplemented()` ist in jedem Zustand einmal implementiert und enthält den oben angegebenen Dreizeiler mit dem entsprechenden Superzustand. Die `ExitAspect`-Klasse sieht etwa so aus:

```
template<class STATE>
class ExitAspect : public STATE
{
  public:
     void SignalA(){notImplemented();}
     ...
     void SignalZ(){notImplemented();}
}
```

Die Zustände erhalten diesen Aspekt durch die Typdefinition

```
typedef ExitAspect<AeussererZustand> AeussererExitZustand;
```

Die Signal-Operationen, die keinen eigentlichen Code haben, müssen nicht implementiert werden. Sie werden automatisch von der `ExitAspect`-Klasse aufgefangen.

Dieses Prinzip funktioniert auch über mehrere Hierarchieebenen hinweg. Zwischen je zwei Hierarchieebenen muss eine ExitAspect-Klasse eingeschoben werden. Dies haben wir in folgender Abbildung 4.3 dargestellt. Die ExitAspect-Klassen sind generisch, damit nur eine ExitAspect-Definition nötig ist, um alle Hierarchiestufen abzudecken.

4.4 Ausblick

In [5] wird eine template-basierte Implementationstechnik für *extended state variables* vorgeschlagen. Dort wird auch gezeigt, dass sich die *history*-Funktionalität einfach einarbeiten lässt. Letzteres ergibt sich aus der Realisierung der Hierarchie durch Vererbungstechnik. Eine kleine Beispielanwendung zeigt auch, dass sich die vorgeschlagenen Implementationstechniken für automatische Codeerzeugung zum Beispiel aus XML-Dateien eignen.

Zur Zeit untersuchen wir, ob sich die beschriebenen Konzepte sinnvoll auf andere Programmiersprachen übertragen lassen. In Java gibt es zum Beispiel keinen *placement new* Operator, und zur Lösung der Echtzeitproblematik müssen hier andere Wege gegangen werden. Ein weiteres Thema ist die geschickte Handhabung von Ausnahmen in hierarchischen Zustandsautomaten, welche Echtzeitsysteme steuern. Die *run to completion* Philosophie in Zustandsautomaten widerspricht der Verwendung von sprachinternen Ausnahmemechanismen wie `catch` und `throw`.

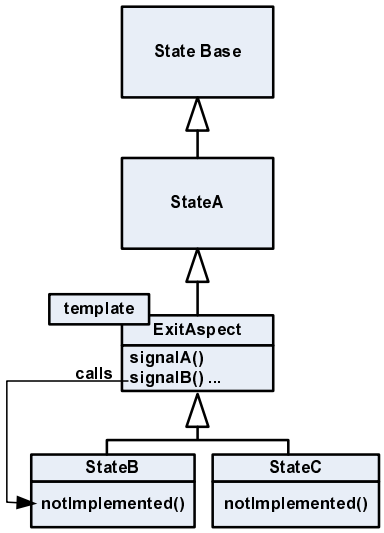

Abbildung 3: Hierarchischer Zustandsautomat mit aspektorientierter Exit-Funktion. Nicht benutzte Signale müssen im Zustand auch nicht implementiert werden. Sie werden von der ExitAspect-Klasse abgefangen. Diese sorgt durch den `notImplemented()`-Aufruf für das Ausführen der `exit`-Funktion und für den Zustandswechsel.

Literatur

[1] Paul Adamczyk. The anthology of the finite state machine design patterns. *EuroPLoP 03*, 2003.

[2] EventHelix. Hierarchical state machine . *http://www.eventhelix.com/RealtimeMantra/HierarchicalStateMachine.htm.*

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison_Wesley, 1995.

[4] David Harel, E. Gery, and M. Politi. *Object-Oriented Modeling with Statecharts.* Weizmann Institute of Science, Dept. of Applied Mathematics and Computer Science, Rehovot, Israel, 1994.

[5] Nico Manske. *Eine einfache, schnelle und speicherschonende Technologie zur Implementation des Zustands-Entwurfsmusters.* Hochschule für Angewandte Wissenschaften, Hamburg. Department Informatik, 2006.

[6] Miro Samek. *Practical statecharts in C/C++: Quantum programming for embedded systems.* CMP Publications, Inc., Manhasset, NY, USA, 2002.

[7] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object Oriented Modeling.* John Wiley & Sons, 1994.

Stephan Pareigis
Hochschule für Angewandte Wissenschaften Hamburg
Department Informatik
pareigis@informatik.haw-hamburg.de

5 Abstracts von Abschlussarbeiten

Auf der FGL-Sitzung am 11.5.2005 wurde die Idee geboren, Abstracts von Thesisarbeiten und Dissertationen aus den Einrichtungen unserer Fachgruppenmitglieder mit starkem Bezug zu PEARL oder Echtzeitproblemen als festen Bestandteil aufzunehmen. Damit verfolgen wir das Ziel, die Breite der aktuellen Forschungsarbeiten in diesem spannenden Gebiet darzustellen.

Der Umfang der Abstracts sollte ca 15 Textzeilen umfassen. Die Einreichung soll direkt an die Redaktion per E-Mail mit den Angaben über Verfasser, Hochschule, Kontaktadresse (EMail) und evtl. eine URL für weitere Informationen erfolgen. Als Textformat wird seitens der Redaktion LaTeX bevorzugt.

Die Leser der PEARL-News werden gebeten, auf geeignete Arbeiten zu achten und die betreffenden Abstracts mit den ergänzenden Daten an die Redaktion zu übermitteln.

5.1 Ereignisabhängigkeitsanalyse für Multiratensysteme

Für die Analyse der zeitlichen Korrektheit von eingebetteten Echtzeitsystemen werden Modelle benötigt, mit denen es möglich ist die Aktivierungsraten der Softwaretasks präzise darzustellen. Für diese Diplomarbeit wird das Ereignisstrommodell nach Gresser verwendet, bei dem die Aktivierungen durch das Auftreten von Ereignissen beschrieben werden und dabei die worst-case Szenarien untersucht werden.

Die Ereignisabhängigkeitsanalyse wird dazu verwendet, um anhand des externen Ereignisstroms und des Kontrollflusses eines Tasks dessen resultierenden ausgehenden Ereignisstrom zu berechnen, der wiederum andere abhängige Tasks aktivieren kann.

In der Diplomarbeit soll eine erweiterte Ereignisabhängigkeitsanalyse entwickelt werden, um auch so genannte Multiratensysteme korrekt untersuchen zu können. Diese Systeme finden Einsatz in eingebetteten Systemen für die digitale Signalverarbeitung und unterscheiden sich in der Art der Aktivierung von den bisher untersuchten Systemen dadurch, dass für die einmalige Aktivierung eines Multiratentasks gleich mehrere Ereignisse verbraucht werden.

Mario Korte
Carl von Ossietzky Universität Oldenburg
Fakultät II, Department für Informatik