Kapitel 3

Modellgetriebene Entwicklung von Echtzeitsystemen (inkl. Werkzeuge)

Inhalt

- Fokus: Konzepte und Werkzeuge zur Modellierung **und** Generierung von Code für Echtzeit- und eingebettete Systeme

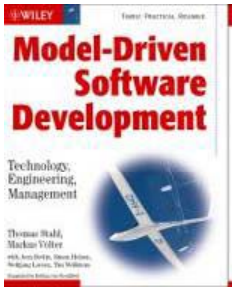
- Motivation
- Grundsätzlicher Aufbau, „Modelle“ und „Models of Computation“
  - Werkzeug Ptolemy
- Zeitgesteuerte Systeme
  - Werkzeug Giotto
- Synchrone Sprachen (Esterel, Lustre)
  - Synchroner Datenfluss: EasyLab
  - Reaktive Systeme: Werkzeuge Esterel Studio, SCADE

Fokus dieses Kapitels

- Voraussetzungen an Werkzeuge für ganzheitlichen Ansatz:
  - Explizite Modellierung des zeitlichen Verhaltens (z.B. Fristen)
  - Modellierung von parallelen Abläufen
  - Modellierung von Hardware und Software
  - Eindeutige Semantik der Modelle
  - Berücksichtigung von nicht-funktionalen Aspekten (z.B. Zeit*, Zuverlässigkeit, Sicherheit)
- Ansatz zur Realisierung:
  - Schaffung von domänenspezifischen Werkzeugen (Matlab/Simulink, Labview, SCADE werden überwiegend von spezifischen Entwicklergruppen benutzt)
  - Einfache Erweiterbarkeit der Codegeneratoren oder Verwendung von virtuellen Maschinen / Middleware-Ansätzen

* Zeit wird zumeist als nicht-funktionale Eigenschaft betrachtet, in Echtzeitsystemen ist Zeit jedoch als funktionale Eigenschaft anzusehen (siehe z.B. Edward Lee: Time is a Resource, and Other Stories, May 2008) http://chess.eecs.berkeley.edu/pubs/426/Lee_TimeIsNotAResource.pdf

Literatur



- Sastry et al: Scanning the issue – special issue on modeling and design of embedded software, Proceedings of the IEEE, vol.91, no.1, pp. 3-10, Jan 2003
- Thomas Stahl, Markus Völter: Model-Driven Software Development, Wiley, 2006
- Ptolemy: Software und Dokumentation http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm
- Benveniste et al.: The Synchronous Languages, 12 Years Later, Proceedings of the IEEE, vol.91, no.1, pp. 64-83, Jan 2003
- Diverse Texte zu Esterel, Lustre, Safe State Machines: http://www.esterel-technologies.com/technology/scientific-papers/
- David Harrel, Statecharts: A Visual Formalism For Complex Systems, 1987
- Henzinger et al.: Giotto: A time-triggered language für embedded programming, Proceedings of the IEEE, vol.91, no.1, pp. 84-99, Jan 2003

Hinweis: Veröffentlichungen von IEEE, Springer, ACM können Sie kostenfrei herunterladen, wenn Sie den Proxy der TUM Informatik benutzen (proxy.in.tum.de)

Begriff: Modell

- Brockhaus:
  Ein **Abbild** der Natur unter der Hervorhebung für **wesentlich** erachteter **Eigenschaften** und Außerachtlassen als nebensächlich angesehener Aspekte. Ein M. in diesem Sinne ist ein Mittel zur Beschreibung der erfahrenen Realität, zur Bildung von Begriffen der Wirklichkeit und Grundlage von Voraussagen über künftiges Verhalten des erfassten Erfahrungsbereichs. Es ist um so realistischer oder wirklichkeitsnäher, je konsistenter es den von ihm umfassten Erfahrungsbereich zu deuten gestattet und je genauer seine Vorhersagen zutreffen; es ist um so **mächtiger**, je **größer** der von ihm beschriebene **Erfahrungsbereich** ist.
- Wikipedia:
  Von einem **Modell** spricht man oftmals als Gegenstand wissenschaftlicher Methodik und meint damit, dass eine zu untersuchende Realität durch bestimmte Erklärungsgrößen im Rahmen einer wissenschaftlich handhabbaren Theorie abgebildet wird.

Modellbasierte Entwicklung

- Wir beobachten im Bereich eingebettete Systeme seit längerem einen Übergang von der klassischen Programmentwicklung zur einer Vorgehensweise, bei der Modelle (für physikalische Prozesse, für die Hardware eines Systems, für Verhalten eines Kommunikationsnetzes, usw.) eine zentrale Rolle spielen
- Modelle werden typischerweise durch verknüpfte grafische Notationselemente dargestellt (bzw. definiert)
- **Funktions-Modelle** sind dabei (meist Rechnerausführbare) Beschreibungen der zu realisierenden Algorithmen
- In unterschiedlichen Phasen der Systementwicklung kann ein Modell unterschiedliche Rollen annehmen – und etwa zum Funktionsdesign, zur Simulation, zur Codegenerierung verwendet werden
- Im günstigsten Fall kann ein System grafisch notiert („zusammengeklickt“) werden und unmittelbar Code erzeugt werden, der dann auf einem Zielsystem zur Ausführung gebracht wird (Beispiel: EasyKit)
- Diese Entwicklung steht im Einklang mit der generellen Geschichte der Informatik, die immer mächtigere (Programmier-) Werkzeuge auf immer abstrakterem Niveau geschaffen hat

Vorteile Modellbasierter Entwicklung

- Für die modellbasierte Entwicklung sprechen diverse Gründe:
  - Modelle sind häufig einfacher zu verstehen als der Programmcode (graphische Darstellung, Erhöhung des Abstraktionslevels)
  - Vorwissen ist zum Verständnis der Modelle häufig nicht notwendig:
    - Experten unterschiedlicher Disziplinen können sich verständigen
  - Systeme können vorab simuliert werden. Hierdurch können Designentscheidungen vorab evaluiert werden und späte Systemänderungen minimiert werden.
  - Es existieren Werkzeuge um Code automatisch aus Modellen zu generieren:
    - Programmierung wird stark erleichtert
    - Ziel: umfassende Codegenerierung (Entwicklung konzentriert sich ausschließlich auf Modelle)
  - Mittels formaler Methoden kann
    - die Umsetzung der Modelle in Code getestet werden
    - das Modell auf gewisse Eigenschaften hin überprüft werden

Modellbasierte Ansätze sind erfolgreich …

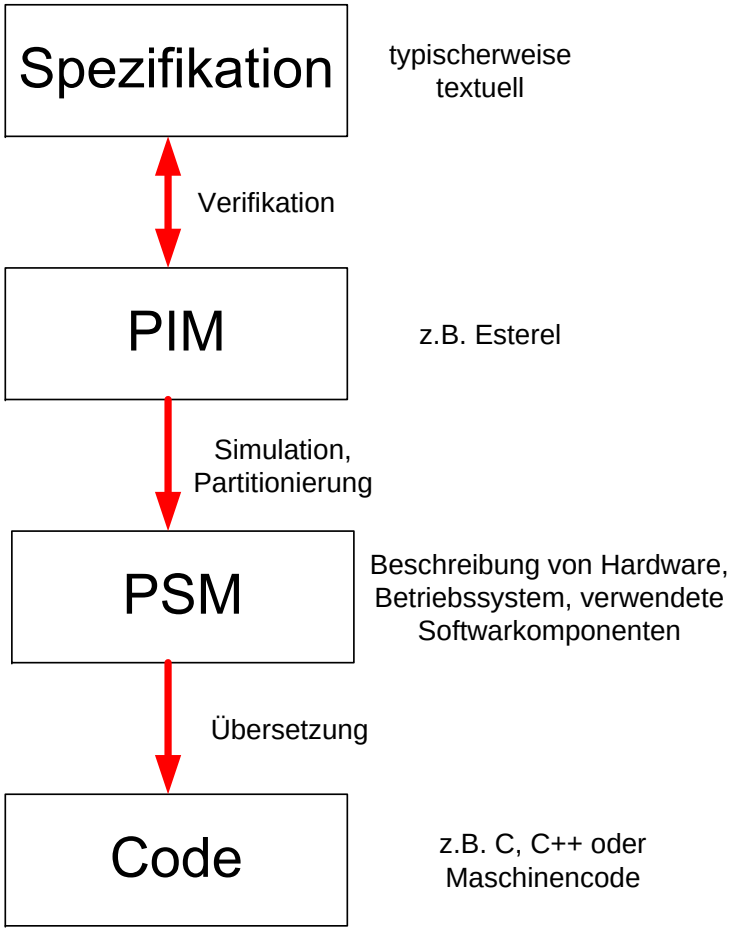
… sie haben unter anderem zu Folgendem beigetragen:

- Ablösen des klassischen „Komponenten-Kopie-Ansatzes“ durch das Prinzips der **Klasse-Instanz-Beziehung**
- Definition von **domänenspezifische Architekturmodellen**: Schichtenmodelle, Bushierarchien
- Konstruktion von **hybriden Umgebungsmodellen**: Abstraktere und erweiterte Modelle der Umgebung
- Definition von Entwurfsebenen: Funktionsarchitektur, logische Architektur, technische Architektur (analog zur Strukturierung der Hardware-Entwicklungsschritte: Architektur – Implementierung – Realisierung, (Gene Amdahl, IBM, 1964), siehe VL-homepage)
- Vermittlung von Modellen und Techniken zur Unterstützung der Anforderungsanalyse und -definition
- Vermittlung von Modellen und Techniken zur Beschreibung von Varianten und Produktlinien

(siehe http://www4.informatik.tu-muenchen.de/~schaetz/MBEES/index.html)

Beispiel OMG: Model-Driven Architecture (MDA)

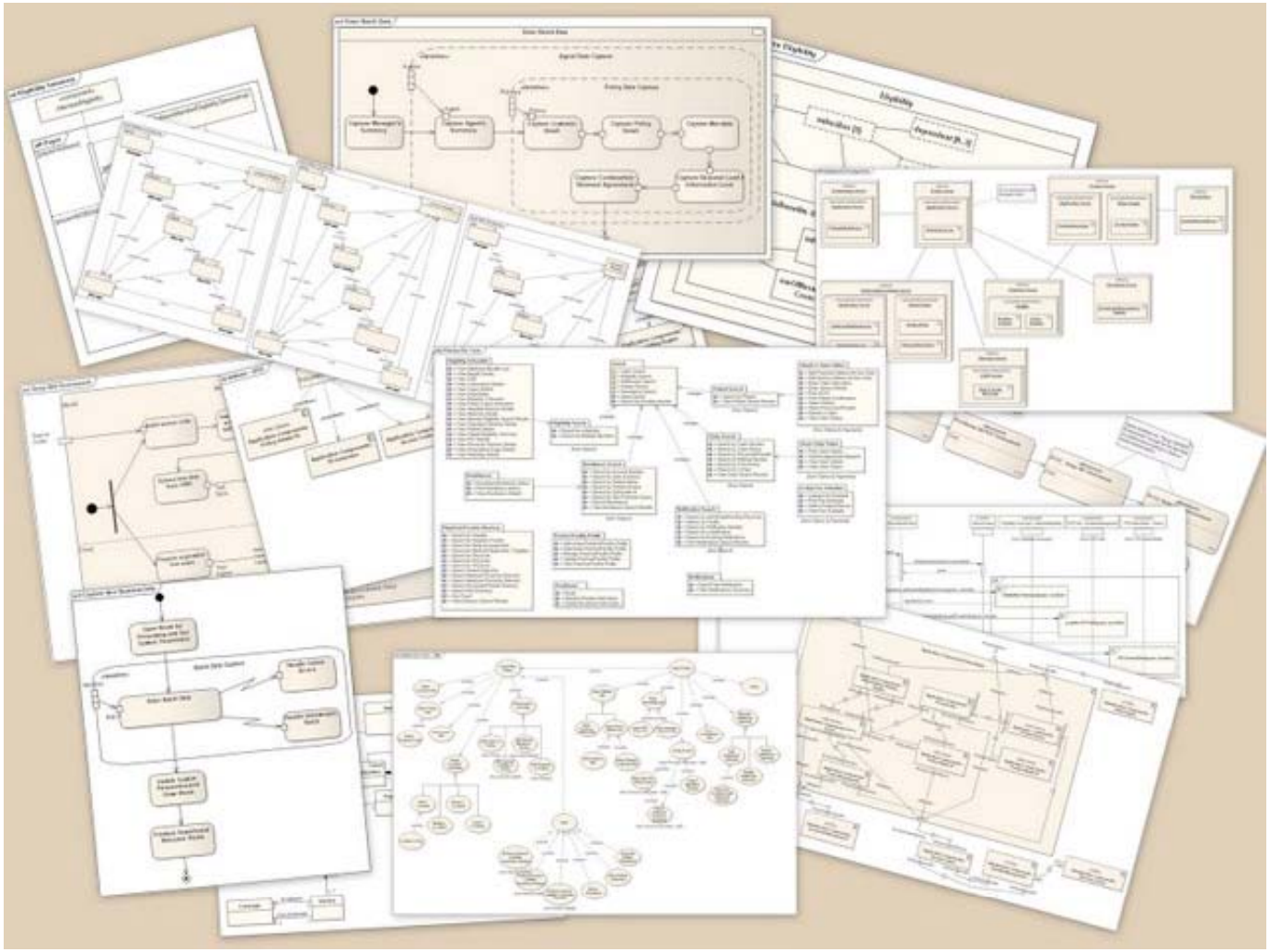
- Die Entwicklung des Systems erfolgt in diversen Schritten:
  - textuelle Spezifikation
  - PIM: platform independent model
  - PSM: platform specific model
  - Code: Maschinencode bzw. Quellcode
- Aus der Spezifikation erstellt der Entwickler das plattformunabhängige Modell
- Hoffung: weitgehende Automatisierung der Transformationen PIM → PSM → Code (Entwickler muss nur noch notwendige Informationen in Bezug auf die Plattform geben)
- http://www.omg.org/mda



MDA im Kontext von Echtzeitsystemen

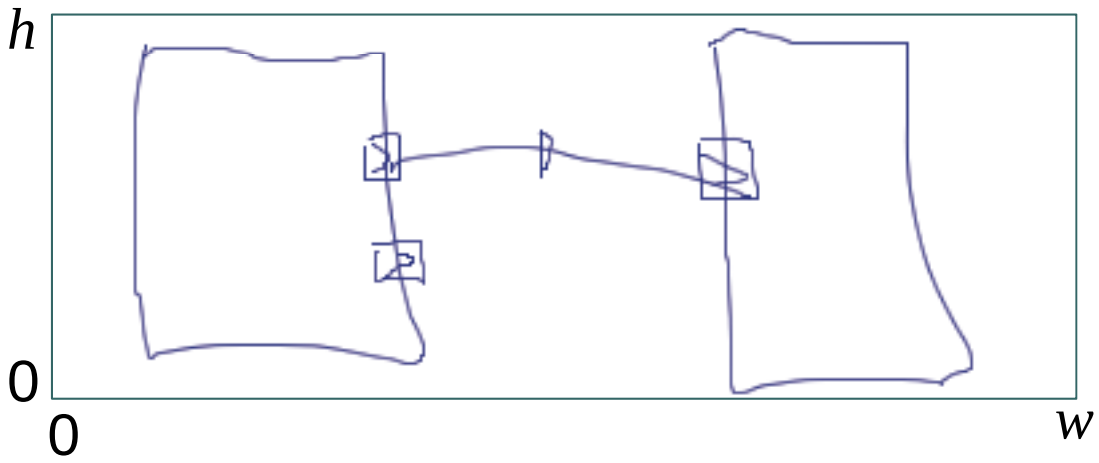
- In Echtzeitsystemen / eingebetteten Systemen ist bei einem umfassenden Ansatz ein Hardwaremodell (z.B. Rechner im verteilten Systen, Topologie) schon in frühen Phasen (PIM) notwendig
- Das plattformspezifische Modell (PSM) erweitert das Hardware- & Softwaremodell um Implementierungskonzepte, z.B.
  - Implementierung als Funktion/Thread/Prozess
  - Prozesssynchronisation

One way to build heterogeneous models: UML : Unified?



[Image from Wikipedia Commons. Author: Kishorekumar 62]

The Truly Unified Modeling Language *TUML*



A *model* in TUML is a function of the form

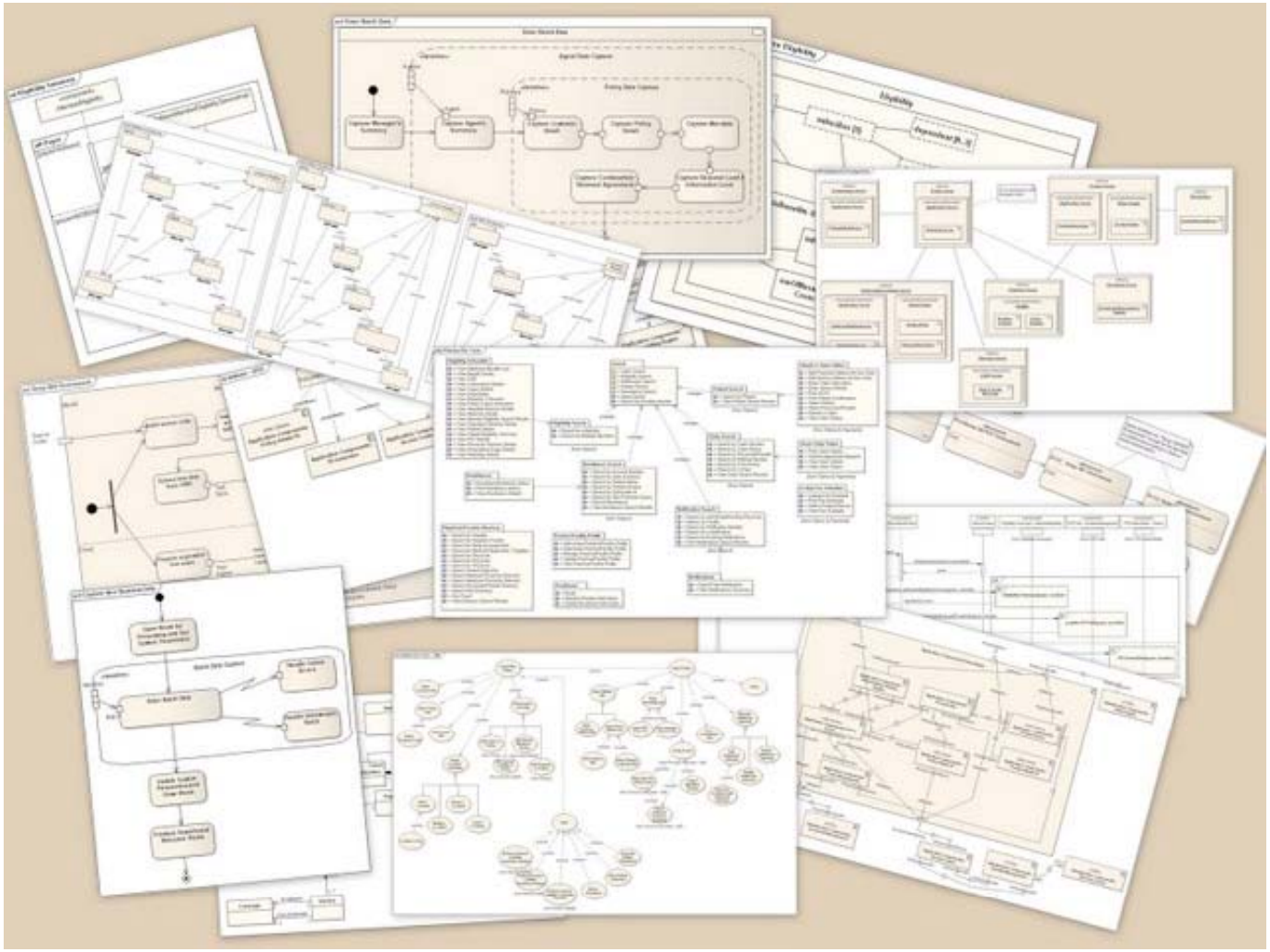
$$f : [0, w] \times [0, h] \rightarrow \{0, 1\}, \quad w, h \in \mathbb{N}$$

(notice how nicely formal the language is!)

Tools already exist.

With the mere addition of a *TUML profile*, every existing UML notation is a special case!

Examples of TUML Models



[Image from Wikipedia Commons. Author: Kishorekumar 62]

My Claim

Modeling languages that are not executable, or where the execution semantics is vague or undefined are not much better than TUML.

We can do better.

Useful Modeling Languages with Strong Semantics

Useful executable modeling languages impose *constraints* on the designer.

The constraints may come with benefits.

We have to stop thinking of constraints as a universal negative!!!

Freedom from choice!!!

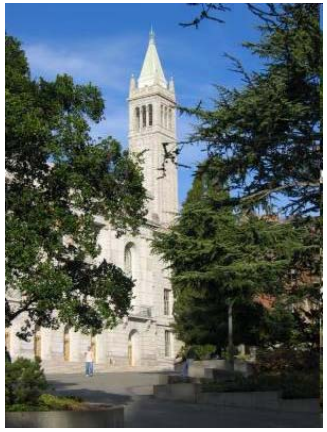
Modellierung von Echtzeitsystemen

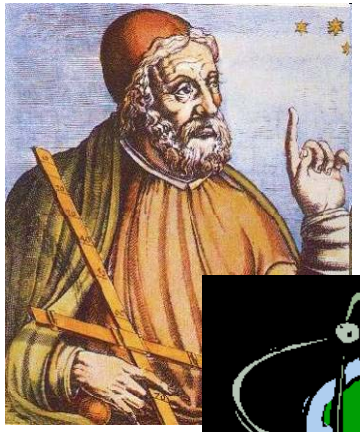
Aktoren, Ausführungsmodelle

Werkzeuge: Ptolemy

Ptolemy

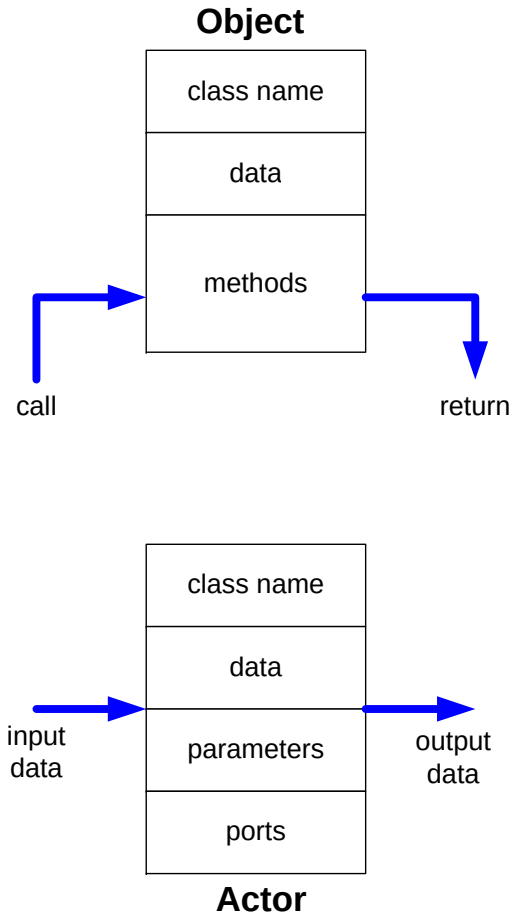
- Das Ptolemy*-Projekt an der UC Berkeley untersucht verschiedene Modellierungs-methodiken für eingebettete Systeme mit einem Fokus auf verschiedene Ausführungs-modelle (Models of Computation)
- Ptolemy unterstützt
  - Modellierung
  - Simulation
  - Codegenerierung
  - Formale Verifikation (teilweise)
- Weitere Informationen unter: http://ptolemy.eecs.berkeley.edu/





***Claudius Ptolemaeus**, (* um 100, vermutlich in Ptolemais Hermii, Ägypten; † um 175, vermutlich in Alexandria), war ein griechischer Mathematiker, Geograph, Astronom, Astrologe, Musiktheoretiker und Philosoph. Ptolemäus schrieb die *Mathematike Syntaxis* („mathematische Zusammen-stellung“), später *Megiste Syntaxis* („größte Zusammenstellung“), heute *Almagest* (abgeleitet vom Arabischen *al-Majisṭī*) genannte Abhandlung zur Mathematik und Astronomie in 13 Büchern. Sie war bis zum Ende des Mittelalters ein Standardwerk der Astronomie und enthielt neben einem ausführlichen Sternenkatalog eine Verfeinerung des von Hipparchos von Nicäa vorgeschlagenen geozentrischen Weltbildes, das später nach ihm *Ptolemäisches Weltbild* genannt wurde. (Wikipedia)

Ptolemy: Aktororientiertes Design



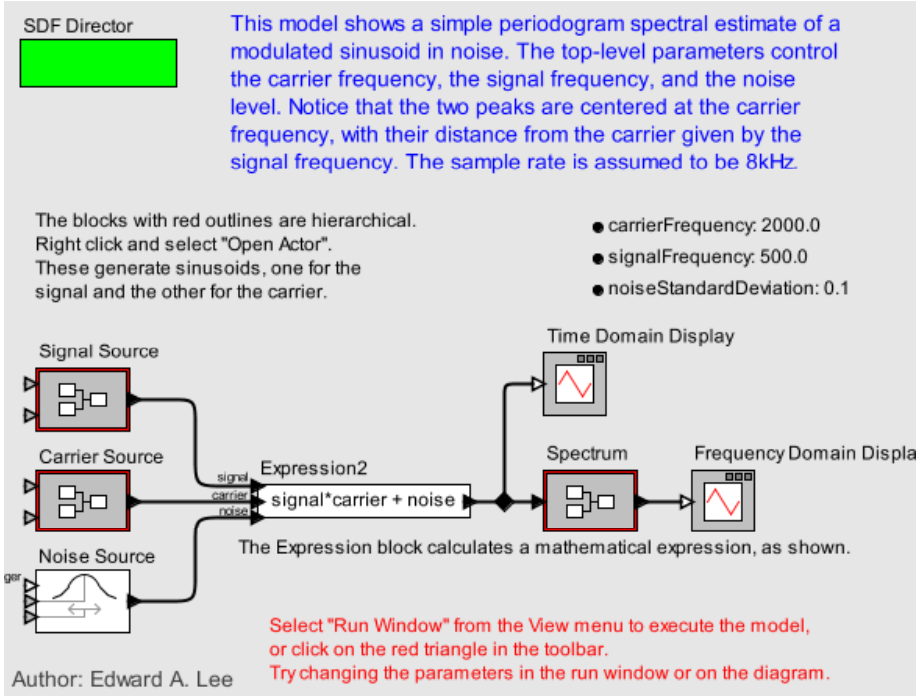
- Ptolemy-Modelle basieren auf Aktoren anstelle von Objekten
- Objekte:
  - Fokus liegt auf Kontrollfluss
  - Objekte werden manipuliert
- Aktoren
  - Fokus liegt auf Datenfluss
  - Aktoren manipulieren das System
- Vorteil beider Ansätze: erhöhte Wiederverwendbarkeit
- Vorteil von Aktoren: leichtere Darstellung von Parallelität

Ptolemy: Aktororientiertes Design

- Ausführungsmodelle (models of computation) bestimmen die Interaktion von Komponenten/Aktoren
- Die Eignung eines Ausführungsmodells hängt von der Anwendungs-domäne, aber auch der verwendeten Hardware, ab
- In Ptolemy wird durch die Einführung von „Dirigenten“ (director) die funktionale Ausführung (Verschaltung der Aktoren) von der zeitlichen Ausführung (Abbildung im Direktor) getrennt.
- Aktoren können unter verschiedenen Ausführungsmodellen verwendet werden (z.B. synchron, asynchron)
- Verschiedene Ausführungsmodelle können hierarchisch geschachtelt werden (modal models).
  - Typisches Beispiel: Synchroner Datenfluss und Zustandsautomaten

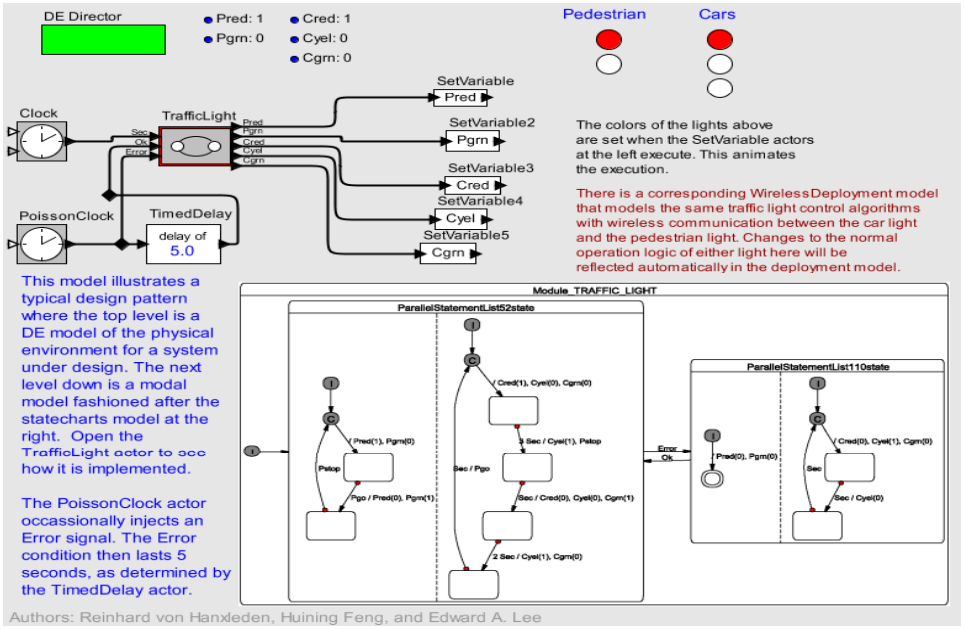
Example Ptolemy Model of Computation: Synchronuous Dataflow

- Prinzip:
  - Annahme: unendlich schnelle Maschine
  - Daten werden zyklisch verarbeitet
  - Pro Runde wird genau einmal der Datenfluss ausgeführt
- Vorteile:
  - Statische Speicherallokation
  - Statischer Schedule berechenbar
  - Verklemmungen detektierbar
  - Laufzeit kann bestimmt werden
- Werkzeuge:
  - Matlab
  - Labview
  - **EasyLab**



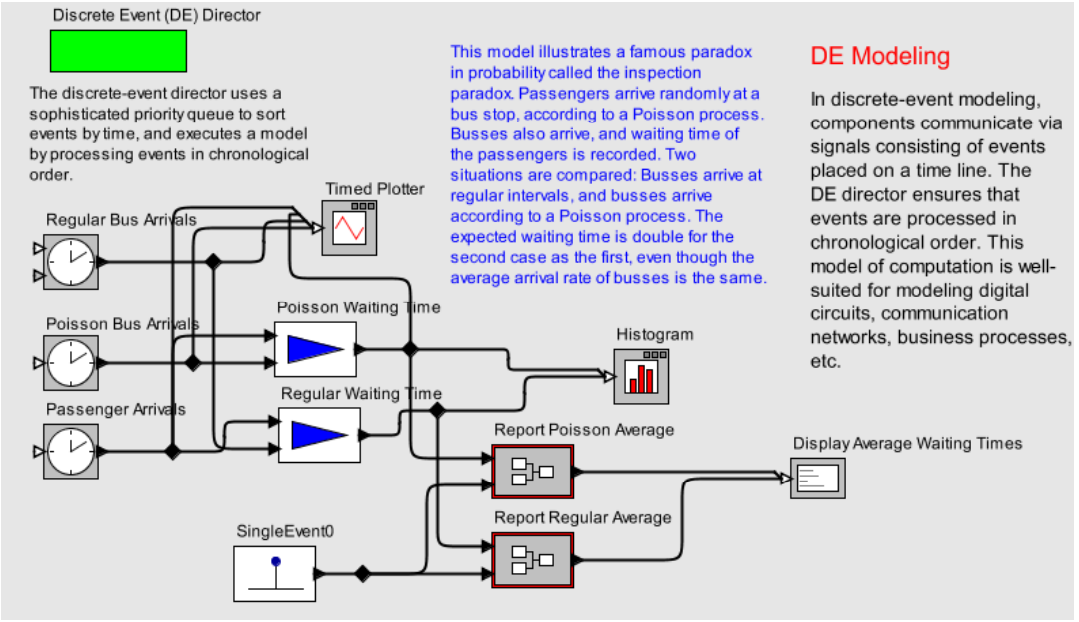
Example Ptolemy Model of Computation: Synchronous Reactive

- Prinzip:
  - Annahme: unendlich schnelle Maschine
  - Diskrete Ereignisse (DE) werden zyklisch verarbeitet (Ereignisse müssen nicht jede Runde eintreffen)
  - Pro Runde wird genau eine Reaktion berechnet
  - Häufig verwendet in Zusammenhang mit Finite State Machines
- Vorteile:
  - einfache formale Verifikation
- Werkzeuge:
  - Esterel Studio
  - Scade



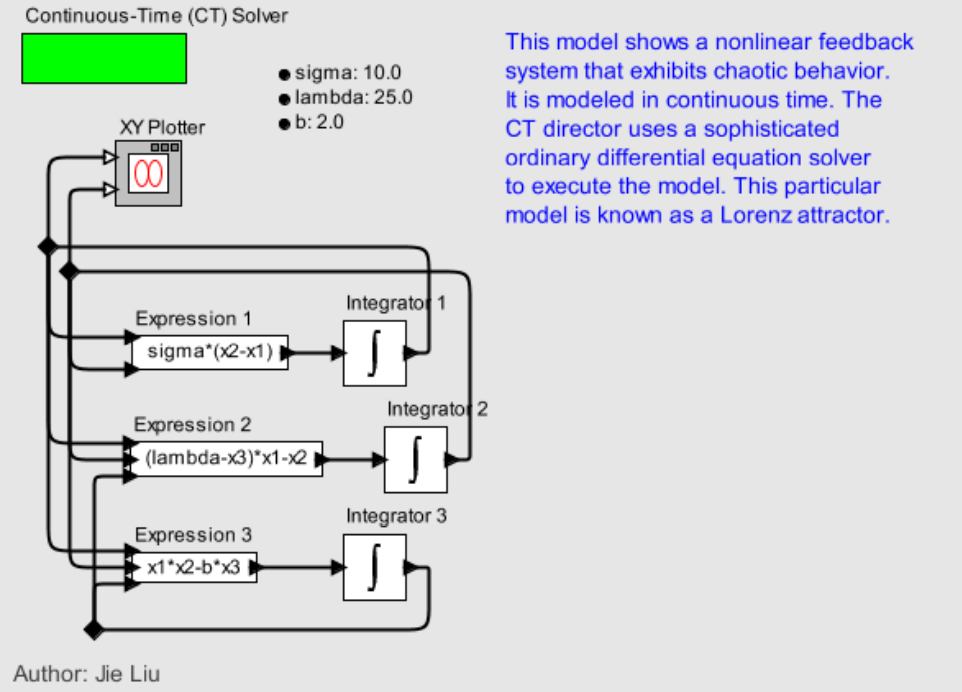
Example Ptolemy Model of Computation: Discrete Event

- Prinzip:
  - Kommunikation über Ereignisse
  - Jedes Ereignis trägt einen Wert und einen Zeitstempel
- Anwendungsgebiet:
  - Digitale Hardware
  - Telekommunikation
- Werkzeuge:
  - VHDL
  - Verilog
- Varianten:
  - Distributed Discrete Events



Example Ptolemy Model of Computation: Continuous Time

- Prinzip:
  - Verwendung kontinuierlicher Signale (bestimmt gemäß Differential-gleichungen)
- Anwendungsgebiet:
  - Simulation
- Werkzeuge:
  - Simulink
  - Labview



Weitere Models of Computation

- Component Interaction:
  - Mischung von daten- und anfragegetriebener Ausführung
  - Beispiel: Web Server
- Discrete Time:
  - Erweiterung des synchronen Datenflussmodels um Zeit zwischen Ausführungen zur Unterstützung von Multiratensystemen
- Time-Triggered Execution
  - Die Ausführung wird zeitlich geplant
  - Anwendungsgebiet: kritische Regelungssysteme
  - Werkzeug: Giotto, FTOS
- Process Networks
  - Prozess senden zur Kommunikation Nachrichten über Kanäle
  - Kanäle können Nachrichten speichern: asynchrone Nachrichten
  - Anwendungsgebiet: verteilte Systeme
- Rendezvous
  - synchrone Kommunikation verteilter Prozesse (Prozesse warten am Kommunikationspunkt, bis Sender und Empfänger bereit sind)
  - Beispiele: CSP, CCS, Ada

Modellierung von Echtzeitsystemen

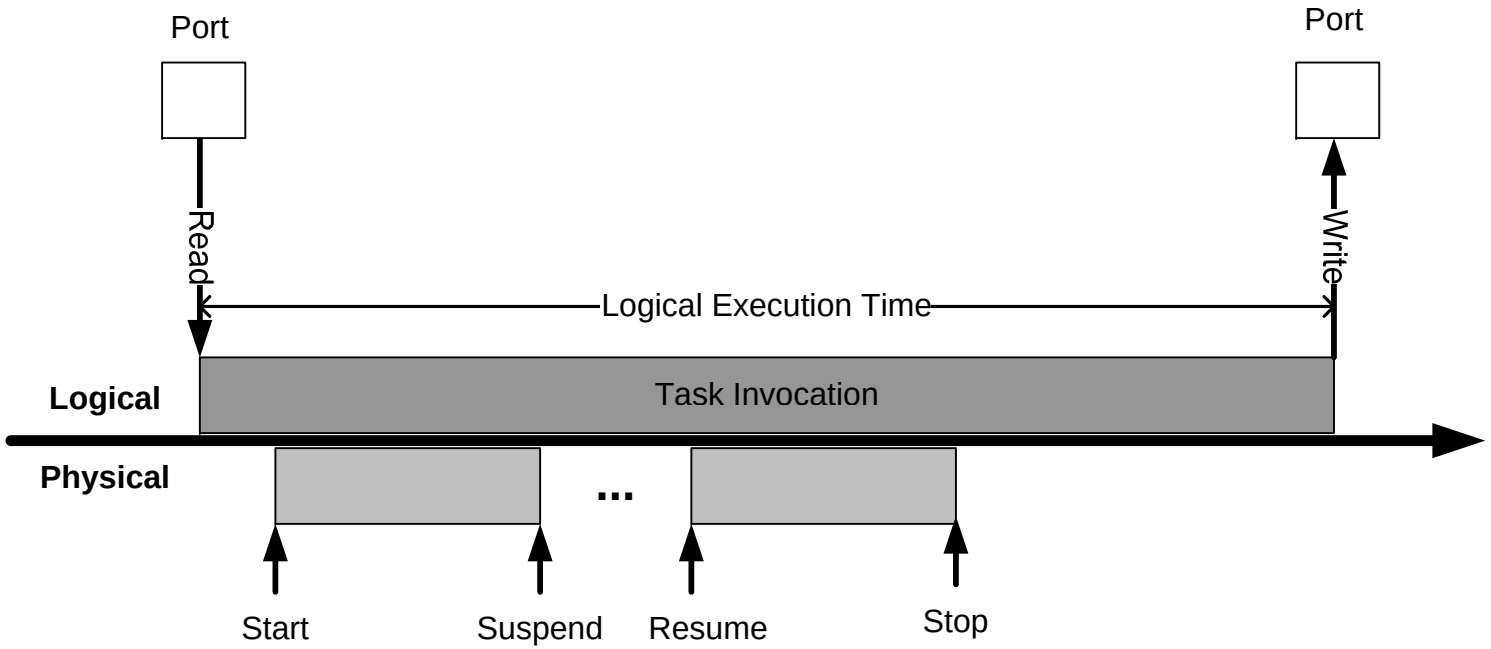
Zeitgesteuerte Systeme

Werkzeug: Giotto

Giotto: Hintergrund

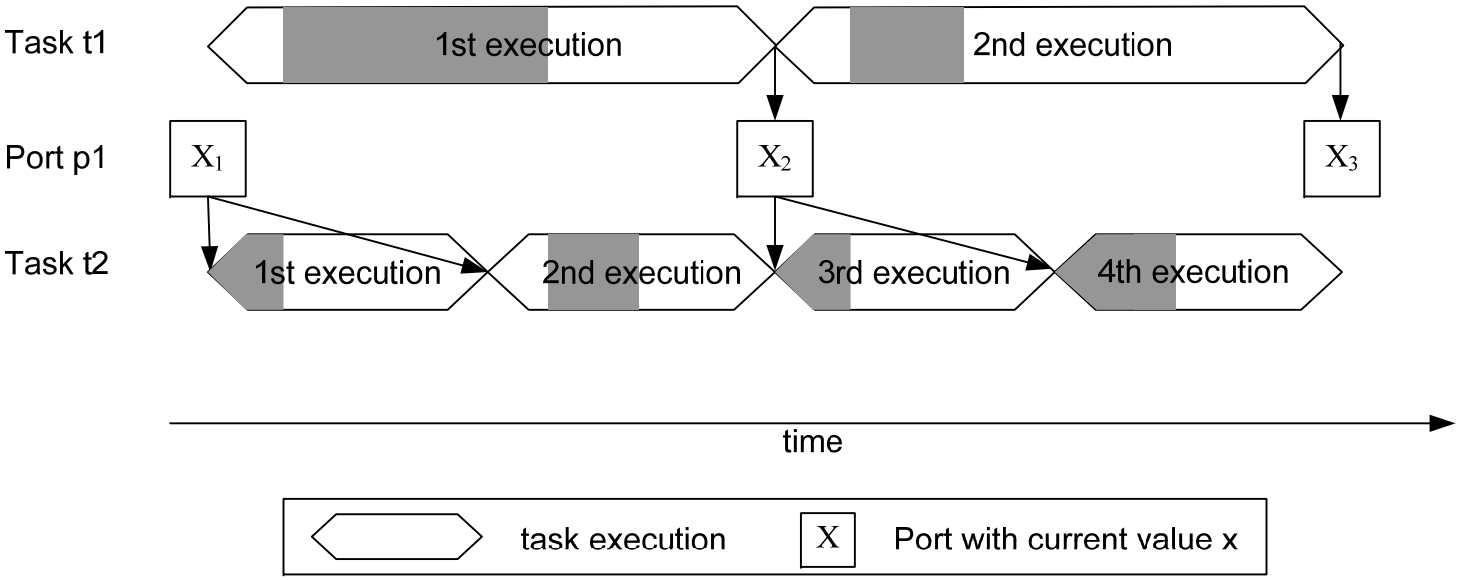
- Programmierumgebung für eingebettete Systeme (evtl. ausgeführt im verteilten System)
- Ziel:
  - strikte Trennung von plattformunabhängiger Funktionalität und plattformabhängigen Scheduling und Kommunikation
  - temporaler Determinismus
- Hauptkonzept: Logische Ausführungszeiten
- Aktoren:
  - Tasks
    - Programmblock aus sequentiellen Code
    - keine Synchronisationspunkte, blockende Operationen erlaubt
    - Schnittstellen: Ports
  - Drivers: realisieren die Kommunikation zwischen Ports
  - Flexibilität durch Modes/Guards
- Ausführung durch virtuelle Maschinen:
  - Embedded Machine: Reaktion der Tasks auf physikalische Ereignisse
  - Scheduling Machine: physikalisches Scheduling
- http://embedded.eecs.berkeley.edu/giotto/

Logische Ausführungszeit



Motivation siehe http://www.cs.uic.edu/~shatz/SEES/henzinger.slides.ppt

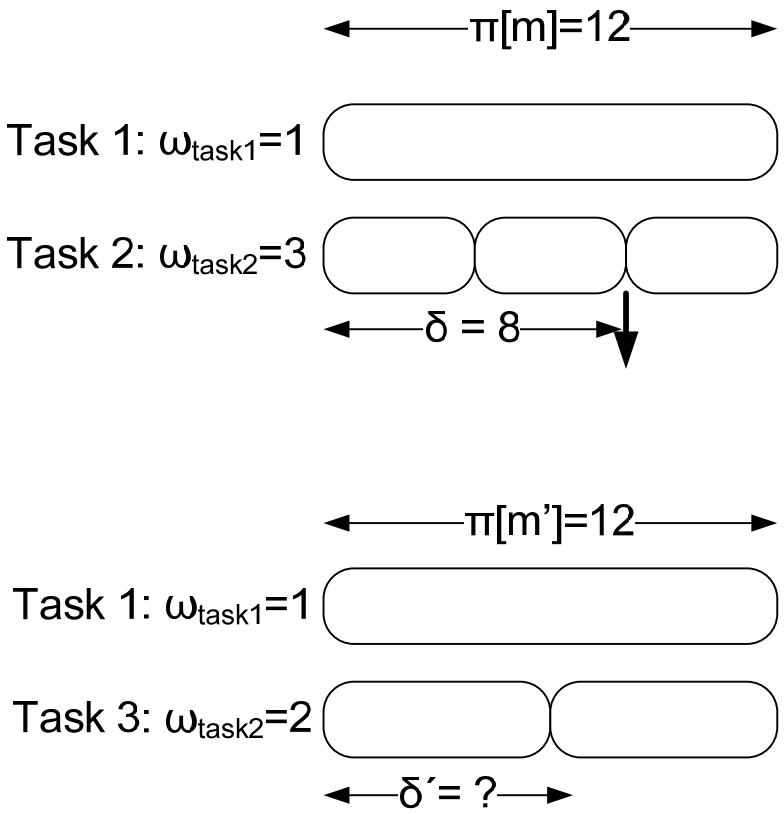
Kommunikation zwischen Tasks



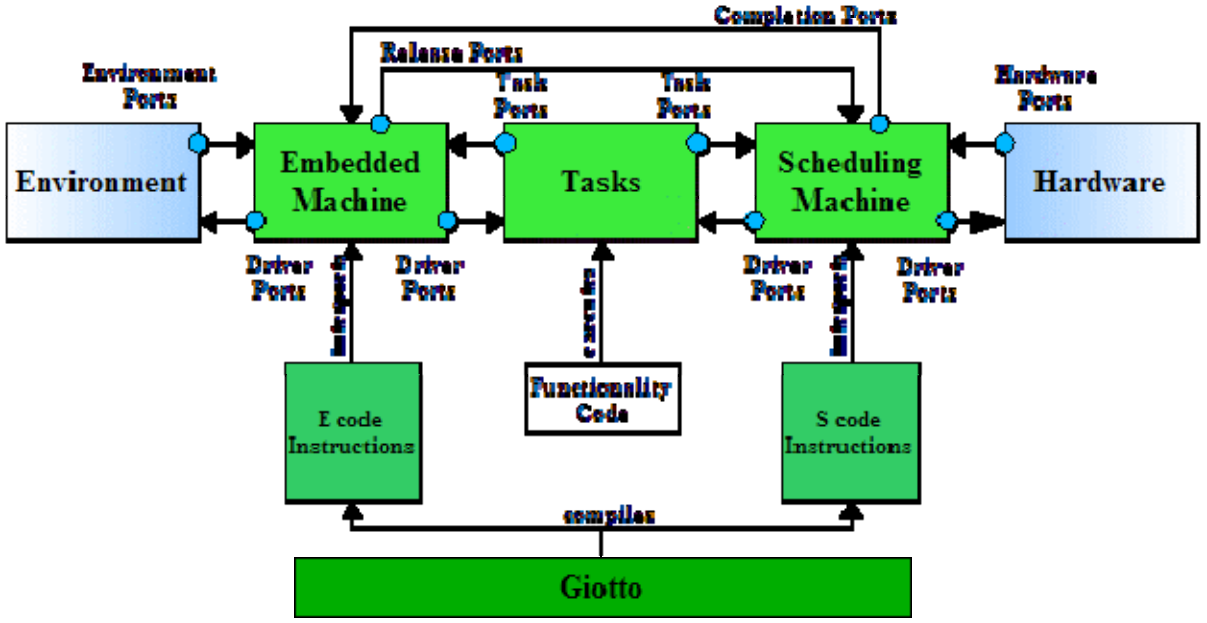
Modes / Guards

- Um die Ausführung flexibel zu gestalten, bietet Giotto Modes und Guards an
  - Guards: Boolsche Funktion, die über die Ausführung eines Tasks entscheidet (wird vor Start des Tasks aufgerufen)
  - Mode: Menge von Tasks und Drivers die zeitgleich ausgeführ werden, es kann immer nur ein Mode aktiv sein.
- Nicht-harmonischer Moduswechsel (Unterbrechung eines laufenden Modes):
  - Voraussetzung: $\pi[m]/\omega_{task} = \pi[m']/\omega'_{task}$
    m: Quellmodus, m': Zielmodus, $\pi[m]$: Modusdauer m, ω_{task}: Taskfrequenz $\Rightarrow$ Logische Ausführungszeit muss gleich sein
  - Wechselmechanismus:

    γ = LCM $\{\pi[m]/\omega_{task} | \{\omega_{task}, t,) \in$ Invokes[m],
    $\delta' = \pi[m']-(\varepsilon-\delta)$ mit $\varepsilon = n^* \gamma \geq \delta$
    LCM: least common multiple, δ: aktuelle Rundenzeit, δ': neue Rundenzeit in m', $\varepsilon - \delta$: Zeit bis zum nächsten gleichzeitigen Beendigungspunkt



Ausführungsumgebung



Zusammenfassung

- Das Konzept der logischen Ausführungszeiten erlaubt eine Abstrahierung von der physikalischen Ausführungszeit und somit die Trennung von plattformunabhängigem Verhalten (Funktionalität und zeitl. Verhalten) und plattformabhängiger Realisierung (Scheduling, Kommunikation)
- Die Ausführung erfolgt über zwei virtuelle Maschinen:
  - E-Machine: Interaktion mit der Umgebung (reaktiv)
  - S-Machine: Interaktion mit der ausführenden Plattform (proaktiv), Vorteil: Schedule kann vorab berechnet werden
- Weitere Literaturhinweise:
  - Henzinger et al.: Giotto: A time-triggered language für embedded programming, Proceedings of the IEEE, vol.91, no.1, pp. 84-99, Jan 2003
  - Henzinger et al.: Schedule-Carrying Code, Proceedings of the Third International Conference on Embedded Software (EMSOFT), 2003