Giotto: A Time-Triggered Language for Embedded Programming

Robert Fehrmann

robert.fehrmann@fu-berlin.de

1 Einleitung

Bei der Entwicklung von embedded control systems mit *hard real-time*-Anforderungen sind zwei Gruppen beteiligt: *control engineers* und *software developer*. Der *control engineer* entwickelt mit Hilfe von Werkzeugen wie *Matlab* oder *MatrixX* das *control design* und die *mode switch*-Logik. Die Implementierung für eine bestimmte Plattform wird im Anschluss daran durch *software developer* übernommen. Seine Aufgabe liegt darin, das *control design* geeignet in periodisch wiederkehrende Aufgaben (*tasks*) zu zerlegen. Diese *tasks* werden einer *CPU* zugewiesen und mit Prioritäten versehen, so dass alle *hard real-time*-Anforderungen abgedeckt werden. Dieses Vorgehen hat den Nachteil, dass der *software developer* Aktivitäten ausführen muss, die nichts mit der Realisierung der Funktionalität und der zeitlichen Anforderung zu tun haben.

Giotto's Ziel ist es, eine Abstraktion für *control engineers* und *software developer* zu schaffen, die es ihnen erlaubt, sich auf die Aspekte des *control systems* zu fokussieren und plattformbezogene Aspekte, wie die Hardware oder das Betriebssystem, zu ignorieren. Zu diesem Zweck werden Giotto Programme geschrieben. Sie beinhalten sowohl die Funktionalität, als auch die zeitlichen Anforderungen. Ob dieses Programm auf einer Plattform lauffähig ist, d.h. mit einer bestimmten Performanz oder eines bestimmten Scheduler verträglich ist, wird durch den *Giotto Compiler* überprüft. Trifft dies zu, so wird durch den Compiler der *timing code* für die Plattform erzeugt. Dieser gibt an, zu welchem Zeitpunkt ein *sensor* gelesen oder ein *actuator* aktualisiert wird. Der *functionality code* kann in einer Programmiersprache wie C oder Oberon implementiert werden. Dieser implementiert das Lesen der *sensors*, das Aktualisieren der *actuators* und die Ausführung der Steuerungsbefehle. *Timing* und *functionality code* werden zusammen auf einem *runtime system* ausgeführt, das aus einer *virtual machine*, der *Embedded Machine*, und einem *real-time*-Betriebssystem besteht.

Diese Arbeit stellt einen Überblick über das Giotto-Konzept dar. Dabei wurde mit vier Quellen gearbeitet. Als Leitfaden für diese Arbeit wurde [HHK03] genutzt. Für das ergänzendes Beispiel wurde [KSHP02] herangezogen. Die Informationen über die *Embedded Machine* und *E-Code*, sowie compilerspezifische Informationen wurden dem Artikel [HK07] entnommen. Für einen Überblick über das *time-triggered real-time*-Konzept wurde [Chr02] genutzt. Im weiteren Verlauf dieser Arbeit wird nur noch auf die Quellen ver-

wiesen, wenn sich weitere Informationen für den Leser durch das Lesen der Originalquelle ergeben.

2 Hauptteil

2.1 Vorgehen und Werkzeugeinsatz beim *Giotto-based control system* Entwicklungsansatz

Der *Giotto-based* Entwicklungsansatz wird durch zwei Arten von Werkzeugen unterstützt: *modeling tools* und *implementatation tools*. Mit *modeling tools*, wie *MathWork's Simulink* wird das Erstellen eines *controller models* erleichtert. Das Modell kann im Anschluss simuliert und auf die Anforderungen geprüft werden. Durch eine automatische Codeerzeugung kann darauf folgend *functionality code* des *controller program* erstellt werden. Der *timing code* wird im Gegensatz dazu per Hand geschrieben. Sind beide Teile, *functionality code* und *timing code* vorhanden, so wird unter zu Hilfenahme von Compiler und Debugger die Implementierung und die Optimierung des Steuerungsprogramms unter plattformspezifischen Aspekten umgesetzt. Zum Schluss wird durch den Giotto Compiler ein *timing code* erstellt, der plattformabhängig ist. Die beschriebenen Schritte werden in Abb. 1 gezeigt.

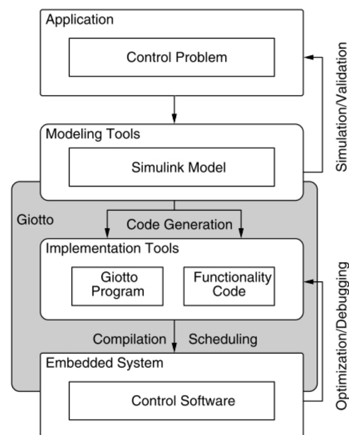

Abbildung 1: Die *Giotto* Werkzeugkette

2.2 Informelle Beschreibung der Giotto Programmiersprache

Im folgenden werden wichtige Elemente eines Giotto Programms informell erläutert. Eine formale Beschreibung ist unter [HHK03] zu finden.

Ports Ports bilden in Giotto das Medium, über das alle Daten ausgetauscht werden. Es werden drei Arten von Ports unterschieden: *sensor ports*, *actuator ports* und *task ports*. *Sensor ports* erhalten Daten (Updates) durch die Umgebung. *Actuator* und *task ports* bekommen Aktualisierungen durch das Giotto Programm. *Task ports* sind für den Austausch von Daten zwischen zwei gleichzeitig aktiven *tasks* oder zwischen Modi verantwortlich.

Tasks Ein Giotto *task* besitzt *input* und *output ports*. Über diese Schnittstellen können Informationen zu und von einer Funktion, die durch einen *task* implementiert wird, gelangen. Die Funktion eines *task* ist ein sequentielles Programm und kann in einer beliebigen Programmiersprache geschrieben werden. Hierbei ist zu beachten, dass die Ausführung der Funktion keine internen Synchronisationspunkte besitzt und auch nicht frühzeitig beendet werden kann.

Notation für die graphische Darstellung von Giotto *tasks*

- *task*: Der *task* wird durch einen Rahmen gekennzeichnet. Innerhalb des Rahmens steht der Bezeichner des *tasks* (t), die Frequenz (ω), der Bezeichner für die Funktion (f) und Modus. Auf dem Rahmen des *tasks* werden die *input-* und *outputports* dargestellt.
- *input-/outputports*: Werden durch ausgefüllte Kreise gekennzeichnet.

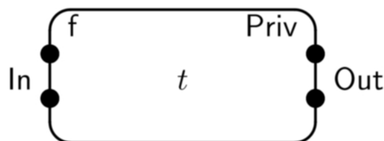

Abbildung 2: Ein Giotto *task*

Tasks invocation *Giotto tasks* sind periodisch. Das heißt, sie werden zu festgelegten Zeitpunkten ausgeführt. Die Ausführung eines *tasks* erfolgt innerhalb eines *modes*. Dies hat zur Folge, dass alle *task ports* zu den *mode ports* hinzugefügt werden. Die *real-time frequency* wird durch die für einen *mode* zur Verfügung stehende Zeit und die Frequenz bestimmt: *real-time frequency / task frequency* t_ω. Für jede Ausführung eines *tasks* werden zuvor *driver* spezifiziert, welche Messwerte der *sensor ports* oder *mode ports* geeignet umwandeln, so dass sie von einem *input port* eines *tasks* genutzt werden können um Werte an

einen *input port* zu liefern. Auch können *input ports* durch eine *driver* mit Konstanten versorgt werden. Die Ausführung eines *tasks* kann verhindert werden, indem ein *guard* beim *driver* gesetzt wird. Wertet dieses nach *true* aus, so wird der *task* ausgeführt, anderenfalls nicht.

Notation für die graphische Darstellung von Giotto *task invocations*

- *driver*: Darstellung durch einen Pfeil in Richtung eines *task input ports*. Kennzeichnung mit kleinen Buchstaben.

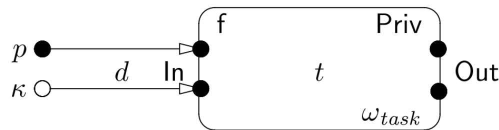

Abbildung 3: Die Ausführung eines Giotto *task*

Wird ein *task* ausgeführt, werden zwei Phasen durchlaufen. In der ersten Phase, der *instantaneous communication phase*, wird der *guard* des *drivers* überprüft und gegebenenfalls Werte in die *input ports* des *tasks* geschrieben. Innerhalb der Giotto Semantik wird für diese Phase keine Zeit benötigt. Im Anschluss an die *communication phase* folgt eine *scheduled computation phase*. In Abbildung 4 ist die Ausführung eines *tasks*, das Zusammenwirken der beiden Phasen und der Zeitverlauf schematisch dargestellt. Die Semantik von Giotto schreibt vor, dass zum Zeitpunkt t_{stop} sowohl die *state* und *output ports* aktualisiert werden. Die (deterministischen) Werte werden durch die Funktion des *tasks* erzeugt. Da die Zeit, wie lange ein *task* benötigt, um seine Aufgabe zu erfüllen, vorgegeben ist, existiert für jeden *task* ein Zeitpunkt, an dem seine *output ports* aktualisiert werden. Dadurch ist Kommunikation zwischen *tasks* für jede beliege Eingabe an *sensor ports* deterministisch. Dadurch entsteht sowohl ein Determinismus für die Zeit (*time-determinism*), als auch für die Werte (*value-determinism*). Diese beiden Eigenschaften und die *instantaneous communication phase* bilden wesentliche Elemente der Giotto Abstraktion.

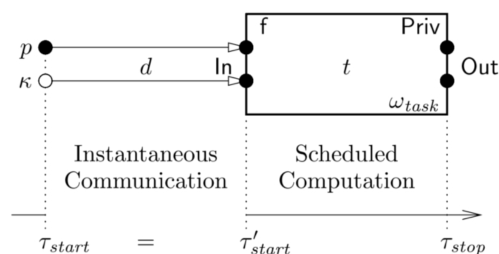

Abbildung 4: Die Ausführung eines Giotto *task*

Modes Ein *mode* beschreibt eine Menge von *tasks*. Ein Giotto Programm kann aus einem oder mehreren *modes* bestehen. Durch einen *mode* wird die wiederholte Ausführung von gleichen *tasks* ermöglicht. Zu jedem Zeitpunkt kann die Ausführung eines Giotto Programms nur in einem *mode* sein. Von diesem *mode*, kann jedoch in beliebige andere definierte *modes* gewechselt werden (*mode switch*). Durch einen *mode switch* ist es dem Programm möglich andere *tasks* auszuführen. Außer den *tasks*, hat jeder *mode*: eine Periode (*period*), eine Menge von *mode ports*, eine Menge von *task invocations*, eine Menge von *actuator ports* und eine Menge von möglichen *mode switches*.

Notation für die graphische Darstellung von Giotto *modes*

- *mode* Werden als Rahmen und die Menge von *task* dargestellt.
- **periode** Wird in der unteren rechten Ecke mit π angegeben.
- *mode ports* Werden auf dem Rahmen des *modes* durch ausgefüllte Kreise dargestellt. Mit gestrichelten Linien werden die *mode ports* mit *input ports* (über *driver*) verbunden.
- *task communication* Werden durch Linien von einem *output port* eines *tasks* zu einem *input port* eines anderen *task* dargestellt.

Ein Beispiel für einen *mode* mit zwei *tasks* ist in Abbildung 5 zu sehen. Die Frequenz

Abbildung 5: Ein Mode mit zwei *tasks* und der Periode 10ms

des ersten *task* ist $\omega_1 = 1$. Somit wird der erste *task* ein Mal alle $10ms$ (Periode des *modes*) ausgeführt. Der zweite *task* hat eine Frequenz von $\omega_2 = 2$, was eine zweifache Ausführung des *task* innerhalb der Periode zur Auswirkung hat. Der *task* t_1 besitzt einen *driver* d_1, der zum einen den Wert vom *mode port* o_1 in den *input port* von t_1 kopiert und zum anderen den Wert des *output ports* o_4 von t_2 in den *input port* von t_1. Außerdem wird durch ihn der *actuator port* a aktualisiert. Der *task* t_2 besitzt den *driver* d_2, welcher den

Wert von *output port* o_3 von *task* t_1 in den *input port* i_3 von t_2 kopiert und den Wert des eigenen *output ports* o_5 in den *input port* i_5 kopiert. Eine Ausführung über die Periode $10ms$ für diesen *mode* stellt eine Runde dar. In Abbildung 6 ist eine Runde des oben beschriebenen *modes* zu sehen. Solange es nicht zu einem Wechsel des *modes* kommt

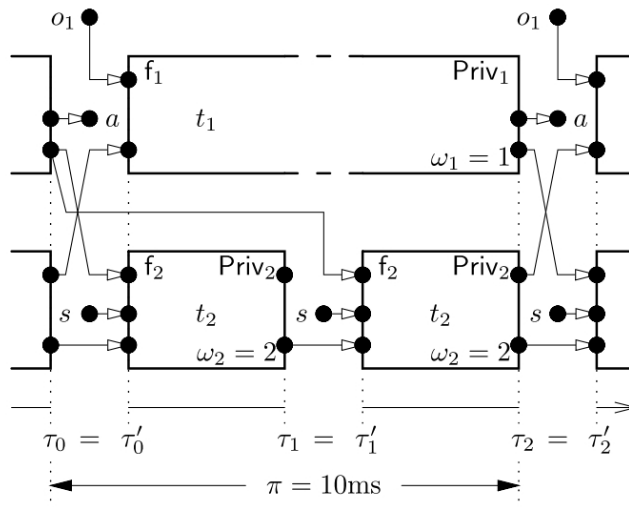

Abbildung 6: Ausführung des *modes* aus Abb. 5

(siehe Abb 8), wird diese Runde wiederholt ausgeführt. Jede Runde beginnt mit einer *instantaneous communication phase*. Für den dargestellten *mode* bedeutet dies, dass die Driver d_1 des *tasks* t_1 und der *driver* d_2 des *tasks* t_2 ausgeführt werden, welche die Werte der *input ports* der *tasks* t_1 und t_2 setzen. Im Anschluss werden die Funktionen f_1 und f_2 ausgeführt. Wie dies physikalisch funktioniert, also in welcher Reihenfolge oder auf wievielen CPU's ist für die Semantik von Giotto nicht von Bedeutung. Von Bedeutung ist, dass die Werte der Berechnung der Funktion t_2 zum Zeitpunkt τ_1 an den *ouput ports* von t_2 vorliegen. Denn zu diesem Zeitpunkt, nach $5ms$ endet die erste Ausführung von t_2. Im Anschluss daran kommt es erneut zu einer *instantaneous communication phase*, aber ausschließlich für den *tasks* t_2. Hervorzuheben ist, dass die Funktion von *tasks* t_2 zwei Mal mit dem gleichen Wert am *input port* i_3 gestartet wird. Dies ergibt sich daraus, dass ein neuer Wert am *output port* o_3 nur einmal alle $10ms$ aktualisiert wird.

Mode switsches Ein *mode switch* beschreibt den Wechsel eines *modes* zu Gunsten eines anderen *modes*. Im Folgenden wird der Wechsel von *mode* m nach *mode* m' beschrieben. Zunächst wird daher der *mode* m' definiert (vgl. Abb. **??**). Der *mode* m' unterscheidet sich von *mode* m, indem *task* t_3 an Stelle von *task* t_2 ausgeführt wird. *Task* t_3 besitzt eine Frequenz von vier und wird somit vier Mal pro Runde ausgeführt. *Task* t_3 besitzt *output port* o_6 und nutzt weiterhin o_4. Die Beschreibung des *driver* d_3 ist analog zu der des *driver* d_2.

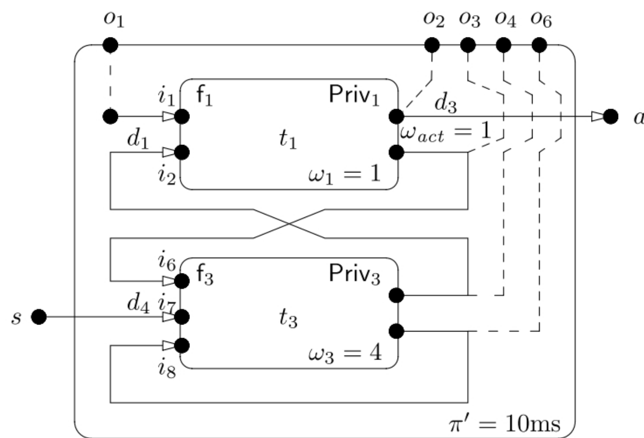

Abbildung 7: Ein zweiter *mode* m', mit *output port* o_6 und *task frequence* $\omega_3 = 4$

Für einen *mode switch* sind drei Informationen nötig: ein *target mode*, eine *switch frequency* und ein *driver*. Durch den *guard* eines *drivers* wird entschieden, ob der *mode* gewechselt wird oder nicht. Der *guard* stellt die *exit condition* für den *mode switch* dar. Wie oft der *guard* geprüft wird, wird durch die *switch frequency* festgelegt. Der Vorgang des Prüfens des *guards* wiederholt sich kontinuierlich. Der zu betrachtende *mode switch* besitzt eine *frequency* von zwei und einen *driver* d_5. Durch diese Frequenz und die Periode von $10ms$ des *modes* m', von dem der *mode switch* ausgeht, wird die *exit condition* zwei Mal in der Periode, also alle $5ms$ geprüft. Abbildung 8 zeigt die Aktionen, die ausgeführt werden, wenn der beschriebene *mode switch* ausgeführt wird. Dabei ist zu beachten, dass

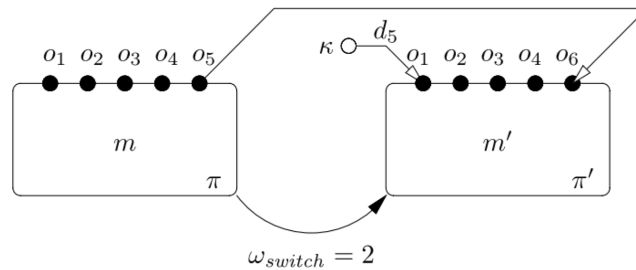

Abbildung 8: Ein *mode switch* zwischen *mode* m und *mode* m'

der *driver* $d5$ (*mode switch driver*) die Werte für die *ports* o_3,o_4,o_5 von *mode* m nach m' kopiert. Diese Anweisung ist nötig, da physikalisch bei einem *mode switch* auch ein Wechsel der CPU einher gehen kann. Außerdem läd d_5 den *mode port* o_1 mit der Konstanten κ und o_6 mit dem Wert von o_5. Abbildung 9 zeigt den Verlauf eines *mode switch* zum Zeitpunkt τ_1. Bis zu diesem Zeitpunkt ist der Ausführungsverlauf identisch zu dem

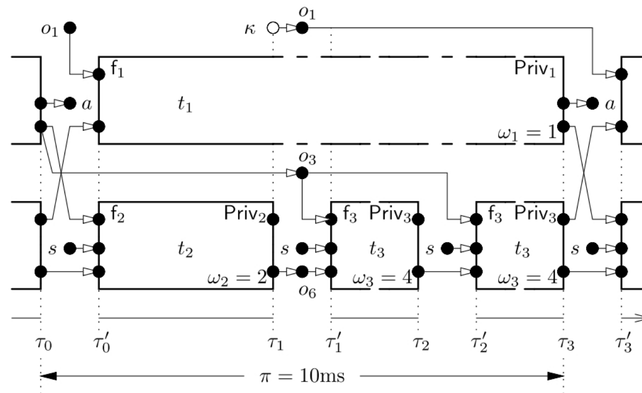

Abbildung 9: Ein *mode switch* zwischen *mode* m und *mode* m'

Verlauf in Abb. 8. Zum Zeitpunkt τ_1 ist die Ausführung von *task* t_2 beendet und der *driver* d_5 kann ausgeführt werden. Damit ist der *mode switch* vollendet. Alle Aktionen, die von diesem Zeitpunkt ausgeführt werden, egal ob die durch einen gerade vollzogenen *mode switch* geschehen ist, oder ob sie bereits zum wiederholten Mal in einer Runde im *mode* m' ausgeführt werden, werden durch die Semantik von m' bestimmt. Die Ausführung des *task* t_3 kann beginnen. Zum Zeitpunkt τ_2 wird der *task* t_3 ein weiteres Mal ausgeführt. Bei τ_3 ist die Runde beendet, da zu diesem Zeitpunkt der *task* t_1 das erste Mal fertig wird. Von diesem Zeitpunkt an, seit Beginn des *mode switch*, starten komplette Runden im *mode* m'. Nun können die Konstanten geladen werden, im Beispiel ausschließlich $\kappa \to o_1$.

Damit ein *mode switch* legitim ist, wird die folgende Bedingung an den *target modet* gestellt: Alle Ausführungen von *tasks*, die durch einen *mode switch* logisch unterbrochen werden, müssen auch im *target mode* logisch wieder aufgenommen werden können. Im Beispiel kommt es zu einem *mode switch* zum Zeitpunkt τ_1. Zu diesem Zeitpunkt läuft der *task* t_1 logisch noch. Daraus folgt, dass *mode* m' ebenfalls den *task* t_1 ausführen muss. Außerdem muss auf die Periode des *target modes* und die *frequence* der noch laufenden *tasks* geachtet werden. Im Beispiel ändert sich nichts, da sowohl *mode* m, als auch m' die Periode von $\pi = \pi' = 10ms$ haben. Der *task* t_1 wird in beiden *modes*, in einer Runde ein Mal ausgeführt. Hätte *mode* m' die Periode $20ms$, so müsste die *frequence* von *task* t_1 auf zwei angehoben werden.

2.2.1 Operationelle Semantik der Programmiersprache Giotto

Die Beschreibung der Operationellen Semantik von Giotto baut auf der Notation, die in [HHK03] definiert wurde auf.

mode frequency. Die *mode frequency* eines *modes* m beinhaltet drei Faktoren: die *task frequency* ω_{task} für alle *task invocations* $(\omega_{task}, ., .) \in Invokes[m]$, die *actuator fre-*

quency ω_{act} für alle *actuator updates* $(\omega_{act}, .) \in Updates[m]$ und die *switch frequency* ω_{switch} für alle *mode switches* $(\omega_{switch}, ., .) \in Switches[m]$. Der kleinste gemeinsame Multiplikator der drei Frequenzen beschreibt die *number of units* $(\omega_{max}[m])$.

program configuration. Eine *program configuration* $C = (\tau, m, u, v, \sigma_{active})$ besteht aus einem *time stamp* $\tau \in \mathbb{Q}$, einen *mode* $m \in Modes$, einen *unit counter* $u \in 0, ..., \omega_{max[m]-1}$, eine *valuation* $v \in Vals[Ports]$ für alle *ports* und einer Menge von *acive tasks* $\sigma_{active} \subseteq Tasks$.

program configuration update. Eine Aktualisierung einer *configuration* C läuft im wesentlichen folgendermaßen ab: zunächst enden *tasks*, als nächstes werden einige *actuators* aktualisiert, ein *mode switch* kann eintreten und anschließend werden neue *tasks* aktiviert.

Für eine saubere Definition der operationellen Semantik werden zunächst die vier folgenden Definitionen benötigt:

- Eine *task invocation* $(\omega_{task}, t, .) \in Invokes[m]$ ist *completed* für eine *configuration* C, wenn $t \in \sigma_{active}$ und $u * \omega_{task}/\omega_{max}[m] \in \mathbb{N}$
- Ein *actuator update* $(\omega_{act}, d) \in Updates[m]$ ist *enabled* für eine *configuration* C, wenn $u * \omega_{act}/\omega_{max}[m] \in \mathbb{N}$ und $g[d](v) = true$.
- Ein *mode switch* $(\omega_{switch}, ., d) \in Switches[m]$ ist *enabled* für eine *configuration* C, wenn $u * \omega_{switch}/\omega_{max}[m] \in \mathbb{N}$
- Eine *task invocation* $(\omega_{task}, ., d) \in Invokes[m]$ ist *enabled* für eine *configuration* C, wenn $u * \omega_{task}/\omega_{max}[m] \in \mathbb{N}$ und $g[d](v) = true$.

Der exakte Ablauf eines *configuration updates* verläuft in neun Schritten:

- ***Task output* und *private ports*.** Sei $\sigma_{complete}$ die Menge von *tasks* t die bereits ausgeführt wurden und abschlossen (*completed*) sind für eine *configuration* C. Für einen *port* $p \in Out[t] \cup Priv[t]$,, mit *task* $t \in \sigma_{complete}$, gilt: $v_{task}(p) = f[t](C[In[t] \cup Priv[t]])(p)$, anderenfalls gilt: $v_{task}(p) = v(p)$. Hiermit werden neue Werte für alle *output* und *priv ports* p gesetzt, deren *tasks* abgeschlossen sind. Sei C_{task} die *configuration*, die v_{task} als Werte für $OutPorts \cup PrivPorts$ enthält und sonst mit C übereinstimmt.
- **Actuator ports.** Sei $p \in ActPorts$ und $p \in Dst[d]$ für *actuator ports* $(., d) \in Updates[m]$, der für eine *configuration* C aktiviert (*enabled*) ist, dann gilt: $v_{act}(p) = h[d](C_{task}[Src[d]])(p)$, anderenfalls gilt: $v_{act}(p) = v(p)$. Durch diese Definition werden alle *actuator ports* aktualisiert. Sei C_{act} die *configuration*, die v_{act} als Werte für $ActPorts$ enthält und sonst mit C_{task} übereinstimmt.
- ***Sensor ports*.** Sei $p \in SensePort$, dann ist $v_{sense}(p)$ ein Wert für den der Typ $Type[p]$ gilt. Sei C_{sense} die *configuration*, die v_{sense} als Werte für $SensePorts$ hat und sonst mit C_{act} übereinstimmt.

- ***Target mode.*** Wird ein *mode switch* $(., m_{target}, .)$ auf eine *configuration* C aktiviert, so ist der nachfolgende *mode* wie folgt definiert: $m' = m_{target}$, anderenfalls kommt es zu keinem *mode switch* und es gilt: $m' = m$. Sei C_{target} die *configuration* mit *mode* m' und sonst mit C_{sense} übereinstimmt.

- ***Mode ports.*** Sei $p \in Dst[d]$ für einen *mode switch* $(., m_{target}, .) \in Switches[m]$, der in der *configuration* C_{sense} aktiviert (enabled) wurde, dann ist $v_{mode}(p) = h[d](C_{target}[Src[d]])(p)$, anderenfalls ist $v_{mode}(p) = C_{target}[OutPorts](p)$. Hierdurch werden die Werte für alle *mode ports* für den *target mode* gesetzt. Sei C_{mode} die *configuration*, die v_{mode} als Werte für $OutPorts$ enthält und sonst mit C_target übereinstimmt.

- ***Unit counter.*** Wurde in C_{sense} kein *mode switch* aktiviert, so sei $u' = (u+1)\ mod\ \omega_{max}[m]$. Wurde ein *mode switch* nach m' aktiviert, so sei $\sigma_{running} = sigma_{active} \setminus sigma_{complete}$. Ist $sigma_{running}$ leer, so sei $u' = 1$ und anderenfalls sei $u_{complete}$ das kleinste gemeinsame Vielfache von $\{\omega_{max}[m]/\omega_{task} | (\omega_{task}, t, .) \in Invokes[m]\ fr\ ein\ t \in \sigma_{running}\}$. Dieser Zeitpunkt bildet den frühsten gemeinsamen Zeitpunkt, an dem alle laufenden *task* gleichzeitig enden. Sei u_{actual} das kleinste gemeinsame Vielfache von $u_{complete}$ mit $u_{actual} \geq u$. Dies ist der früheste Zeitpunkt nach u, an dem alle *task* gleichzeitig enden. Die Dauer bis zum nächsten Endzeitpunkt ist demzufolge mit $\delta = (\pi[m]/\omega_{max}[m]) * (u_{actual} - u)$ definiert. Im *mode* m' werden $u_{togo} = (\omega_{max}[m']/\pi[m']) * \delta)$ Einheiten benötigt, bis es zu einem gemeinsamen Abschließen aller *task* kommt. Schließlich sei $u' = (1 - u_{togo})\ mod\ \omega_{max}[m']$. Damit wird springt ein *mode switch* immer zum nächstmöglichen Ende einer Runde für den *target mode* m'. Sei C_{unit} die *configuration* mit u', die sonst mit C_{mode} übereinstimmt.

- ***Task input ports.*** Sei p ein *port* $\in InPorts$ und $p \in Dest[d]$ für mindestens eine *task invocation* $(., ., d) \in Invokes[m']$, die aktiviert wurde in *configuration* C_{unit}, so ist $v_{input}(p) = h[d](C_{unit}[Src[d]])(p)$, anderenfalls ist $v_{input}(p) = v(p)$. Hiermit werden neue Werte für alle *task input ports* gesetzt. Sei C_{input} die *configuration*, die v_{input} als Werte für $InPorts$ enthält und sonst mit C_{unit} übereinstimmt.

- ***Active tasks.*** Sei $\sigma_enabled$ die Menge von *task* t, die durch eine *task invocation* $(., t, .) \in Invokes[m']$ innerhalb der *configuration* C_{input} aktiviert wurde. Die neue Menge von *active tasks* ist $\sigma'_{active} = (\sigma_{active} \setminus \sigma_{completed}) \cup \sigma_{enabled}$. Sei C_{active} die *configuration* die σ'_{active} enthält und sonst mit C_{input} übereinstimmt.

- ***Time stamp.*** Sei $\tau' = \tau + \pi[m']/\omega_{max}[m']$ der neue Zeitpunkt, an dem das Giotto Programm, das nächste Mal ausgeführt wird. Sei C_{succ} die *configuration* mit dem *timestamp* τ', und sonst mit C_{sense} übereinstimmt.

2.3 Ein Giotto Programm - Ausschnitt für einen Autopilot des OLGA Helikopter System

Mit dem Giotto Ansatz wurde 2002 der Autopilot des Helikoptersystem OLGA der ETH Zürich [Kot99] neu implementiert. Die Abb. 10 zeigt das System und Abb. 11 dessen strukturellen Aufbau.

Abbildung 10:

2.3.1 Die Software für den Autopiloten

Zur Realisierung der Software für den Autopiloten werden sechs *modes* benötigt. In Abb. 12 sind *modes* und ihre jeweiligen *tasks* dargestellt. Dabei werden *Init*, *Idle* und *Motor* zur korrekten Initialisierung des Systems benötigt. Durch *Motor* wird der Rotor in einem Bereich zwischen 0 rpm und 300 rpm gehalten, ein Bereich, in dem der Helikopter nicht abheben kann. Kommt ein Kommando der Bodenstation zum Abheben, wird im *mode* TakeOff so lange der Rotor beschleunigt, bis der Helikopter abhebt. Sobald das Abheben abgeschlossen ist, wird der *mode* gewechselt nach *ControlOff* und die Steuerung dem Piloten überlassen. In diesem *mode* kann der Pilot jederzeit die Kontrolle an den Autopiloten abgeben. Dies wird mit einem *mode switch* von *ControlOff* nach *ControlOn* realisiert.

2.3.2 Simulink Spezifikation des Giotto Modells

Simulink wurde zur Realisierung von Giotto Modells erweitert. Giotto Modelle spezifizieren die *real-time*-Interaktion der Systemkomponenten mit der Außenwelt. Die Ausführung der Komponenten eines Giotto Modells erfolgt periodisch. Jedes Giotto Modell spezifiziert eine Periode wie in Abb. 5 beschrieben. Das Modell für die Autopiloten enthält zwei *modes*: *ControlOff* und *ControlOn*. Die Periode ist 25ms. Die *task* werden durch *task blocks*

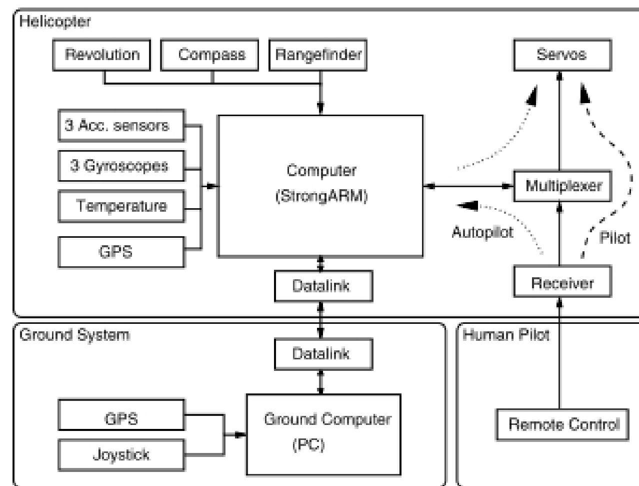

Abbildung 11:

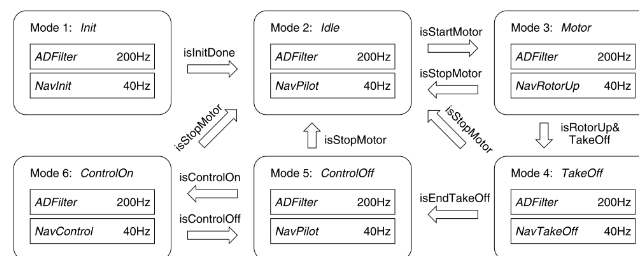

Abbildung 12:

modelliert, ebenfalls eine Erweiterung für Simulink. Sollen mehrere *task* gebündelt werden, so werden *case blocks* genutzt. Im Beispiel werden diese genutzt um *NavPilot* und *NavControl* zu bündeln (Abb. 13).

Dabei ist zu beachten, dass jeder *case block* eine Frequenz besitzt, die relativ zur Frequenz des gesamten Modells ist. Ist es nötig, die Ausführung eines *blocks* an eine Bedingung zu knüpfen, so kann der *Giotto switch block* genutzt werden. Dieser wird im Beispiel dafür genutzt, um zu unterscheiden, ob der Autopilot an- oder ausgeschaltet ist (isControl-lOn/Off). In Abb. 14 ist die Modellierung dieses Verhaltens zu sehen.

Der *Giotto switch block* wird einmal je *task invocation* des äußeren *case blocks* evaluiert, im Beispielfall demzufolge alle 25ms.

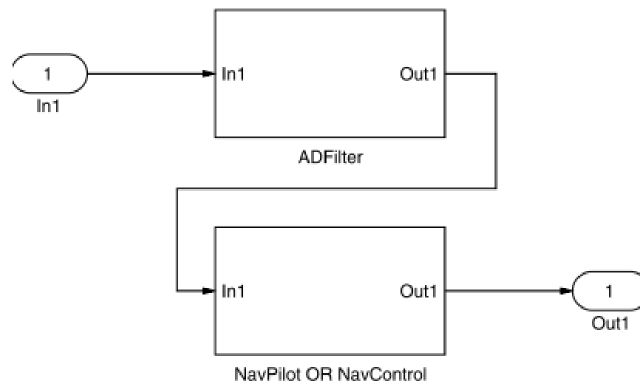

Abbildung 13:

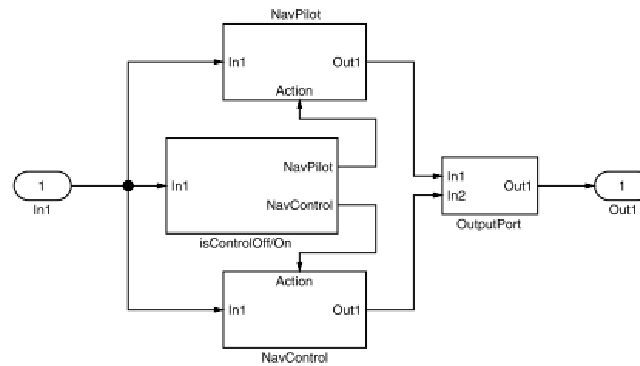

Abbildung 14:

2.3.3 Das Giotto Programm

Das Giotto Programm spezifiziert explizit semantische Details, die durch das Giotto Modell nur implizit spezifiziert wurden oder nicht modellierbar werden konnten. Um Daten zwischen den *ports* von *tasks* und *modes* zu transportieren, wird das *driver*-Konzept genutzt (2.2). Hierbei ist zwischen *Giotto task*, *mode*, *actuator driver* und *device driver* zu unterscheiden. Die ersten drei *driver* arbeiten wie in 2.2 beschrieben. Der *device driver* transportiert Werte von der Hardware oder einem *non-Giotto-task* zu einem *port* und umgekehrt. Das *Giotto Programm* für das Beispiel besteht aus einer Menge von *Giotto modes*. Die Frequenz für die jeweiligen *modes* wird beibehalten. Zusätzlich können nun Angaben gemacht werden, wie oft die *task* und *driver* ausgeführt werden sollen. Wie in Abb. 15 zu sehen ist, wird im *mode ControlOff* fünf Mal pro Periode des *modes* der *task ADFilter* ausgeführt, demzufolge alle 5ms.

```
mode ControlOff() period 25 {
  actfreq  1 do ServoUpdate;
  actfreq  1 do DataPoolUpdate;
  exitfreq 1 do ControlOn;
  taskfreq 5 do ADFilter;
  taskfreq 1 do NavPilot;}
mode ControlOn() period 25 {
  actfreq  1 do ServoUpdate;
  actfreq  1 do DataPoolUpdate;
  exitfreq 1 do ControlOff;
  taskfreq 5 do ADFilter;
  taskfreq 1 do NavControl;}
```

Abbildung 15:

Hingegen wird der *task NavControl* nur ein Mal pro Periode ausgeführt, also alle 25ms. Das Gleiche gilt für die Servos (*ServoUpdate*) und den Datenpool (*DataPoolUpdate*), die Informationen enthalten, die an die Basisstation gesendet werden. Zu beachten ist, dass die Ausführung von *Giotto task* logisch Zeit verbraucht, währenddessen *Giotto driver* logisch keine Zeit verbrauchen, da sie synchron ablaufen. Die *sensor ports* werden im *Giotto Programm* global wie folgt definiert:

```
sensor
  GPSPort     gps uses            GPSGet;
  LaserPort   laser uses          LaserGet;
  CompassPort compass uses        CompassGet;
  RPMPort     rpm uses            RotorGet;
  ServoPort   pilot uses          ServoGet;
  AnalogPort  accelerometers uses AccGet;
  AnalogPort  gyroscopes uses     GyrosGet;
  AnalogPort  temperature uses    TempGet;
  BoolPort    startswitch uses    StartSwitchGet;
  BoolPort    stopswitch uses     StopSwitchGet;
```

Abbildung 16:

Dabei nutzen die *sensor ports* die *Giotto device driver* um ihre Werte zu aktualisieren. Typen und *device driver* werden außerhalb Giotto's implementiert. *Giotto driver* werden immer in einer *time-triggered* angesteuert. Jedoch ist dies nicht immer sinnvoll, da einige *devices*, wie bspw. das *GPS* sofortige Aufmerksamkeit benötigen. In diesen Fällen wird ein *event-triggered* (*interrupt-driver*) *driver* eingesetzt. Dabei greift der *driver* nicht direkt auf das *device* zu, sondern entnimmt Werte aus einem Buffer, der durch das *device* gefüllt wird. Nachdem die *device driver* ausgeführt wurden, wird der *mode driver* aufgerufen. Dieser hat für das Beispiel folgende Implementierung:

```
driver ControlOff(stopswitch) output () {
  switch isControlOff(stopswitch)}
```

Abbildung 17:

Der *driver* besitzt eine Eingabe am *port stopswitch*. Der *device driver* für diesen *port* wurde bereits aufgerufen. Auf Grundlage des Wertes von *stopswitch* wird nun entschieden,

ob ein *mode switch* getätigt wird. Ist der aktuelle *mode* nun *ControlOn*, so ist der nächste Schritt, die Sensoren mit aktuellen Werten zu versorgen. Bevor nun die *task* deklariert werden können, müssen Definition existieren, die alle *output port* mit Werten initialisiert:

```
output
  AnalogPort   filter  :=  FilterInit;
  ServoPort    control :=  ServoInit;
  DataPoolPort data    :=  DataPoolInit;
```

Abbildung 18:

Dabei stellen *Filter, Init, ServoInit* und *DataPoolInit* Implementierung bspw. in Oberon oder C dar. Die *task ADFilter* und *NavControl* werden wie folgt definiert:

```
task ADFilter(accelerometers, gyroscopes, temperature, filter)
  output (filter) {
  schedule ADFilterImplementation(accelerometers, gyroscopes,
                                  temperature, filter)}
task NavControl(gps, laser, compass, filter, rpm, pilot, data)
  output (control, data) {
  schedule NavControlImplementation(gps, laser, compass, filter,
                                    rpm, pilot, control, data)}
```

Abbildung 19:

ADFilterImplementation und *NavControlImplementation* stellen hierbei den funktionalen Teil dar. Nach 25ms stehen in den *control* und *data ports* neue Werte bereit. Diese werden mit Hilfe der *actuator driver ServoUpdate* und *DataPoolUpdate* in die *servos* und *datapool ports* geladen. Die *actuator ports* werden analog zu den *sensor ports* definiert.

```
actuator
  ServoPort    servos   uses ServoPut;
  DataPoolPort datapool uses DataPoolPut;
```

Abbildung 20:

Die Definition der *actuator driver* sieht folgendermaßen aus:

Somit geschieht nach 25ms, direkt nach der Ausführung des *NavControl tasks*, folgendes: Zunächst werden *servos* und *datapool ports* geupdatet. Anschließen wird der *ServoPut device driver* ausgeführt, der die Werte von *servors port* nimmt und damit das *servo device* aktualisiert. Der *DataPoolPut device driver* hingegen fügt die aktuellen Werte von *datapool* in einen Buffer, die sobald das System *idle* wird, zur Bodenstation übertragen werden.

2.4 Der Giotto Compiler und dessen Aufgabe

Ein *Giotto Programm* spezifiziert nicht, wo, wann und wie es ausgeführt wird. Dieses Aufgabe, das Übertragen von einen *Giotto Programm* zu ausführbaren Code, übernimmt der

```
driver ControlOff(stopswitch) output () {
  switch isControlOff(stopswitch)}
```

Abbildung 21:

Giotto Compiler. Seine Aufgabe ist es, die Funktionalität und das Timing des Programms aufrecht zu halten. Zu diesem Zweck generiert der *Giotto Compiler* einen E-Code. Dieser wird auf dem Zielsystem von der *Embedded Machine* [HK07] interpretiert. Dabei bildet der *E-Code* ein portables API zum RTOS des Zielsystems. Für Operationen wie das Aufrufen oder schedulen der nativen Implementierung von *task* und *driver*, sowie Befehle zum kontrollieren der *Embedded Machine* zu spezifischen Zeitpunkten oder beim Auftreten von Ereignissen existieren E-Code Anweisungen. Ist der *E-Code time-safe* [HK07], so erfüllt er die Semantik des *Giotto Programms*. Ist *time-safety* gegeben, so verpasst kein *task* seine *deadline*. Geprüft wird *time-safety* durch den Compiler mit Hilfe einer Analyse der Schedulbarkeit. Hierfür ist es notwendig, dass der Compiler für jeden *task* und jeden *driver* eine obere Schranke für die Laufzeit kennt. Ist *time-safety* für *E-Code* erfüllt, so besitzt das Programm ebenfalls die *environment-determined* [HK07] Eigenschaft. Diese besagt, dass für jedes Verhalten der physischen Welt, das durch einen Sensor gemessen werden kann, der *E-Code* einen deterministischen Pfad von *actuator* Werten zu deterministischen Zeitpunkten durchläuft. Dadurch existieren innerhalb eines Giotto Systems keine *race conditions*, was das System vorhersagbar und beweisbar macht.

2.5 Embedded Machine und E-Code

3 Zusammenfassung und Fazit

Der Giotto Ansatz ermöglicht die Implementierung von *embedded control systems* mit harten Echtzeitanforderungen. Der Ansatz zielt vor allem auf die Trennung vom Schreiben eines Programmes, welches die Funktionalität und das Timing enthält und dem Anpassen des Programms an eine spezifische Zielplattform ab. Durch diese Trennung entstehen mehrere Vorteile. Zum einen können die Funktionalität und das Timing Gegenstand von formalen Beweisen gegen das mathematische Modell des Steuerungsdesign sein. Zum anderen wird ein großer Teil der Anpassung des Programms an die Zielplattform durch den *Giotto Compiler* übernommen. Ist der Compiler nicht in der Lage, aus einem *Giotto Programm* sofort *timing-code* zu erzeugen, so kann dieser durch den Entwickler mit zusätzlichen Annotationen [Chr02] unterstützt werden. Ebenfalls hervorzuheben ist, dass der Giotto Ansatz nicht nur ein theoretisches Konstrukt ist, sondern auch erfolgreich praktisch eingesetzt wurde. Dies konnte innerhalb dieser Arbeit in Ausschnitten nachvollzogen werden und in vollem Umfang in der Nachimplementierung des OLGA Systems [KSHP02]. Nachteilig zu sehen ist die Notwendigkeit der *Embedded Machine* auf dem Zielsystem. Für unterschiedliche Systeme muss diese jeweils neu implementiert werden. Dabei ist positiv anzurechnen, dass dies einen geringen Aufwand benötigt. Dies kann den Äußerungen

der Autoren entnommen werden, die für die Implementierung der *Embedded Machine* für das Helikopter Projekt nur eine Woche benötigten. Der Speicher war dabei mit sechs KB sehr gering. Ebenfalls positiv ist, dass die Performanz keine wesentlichen Einbußen durch zusätzliche Überprüfungen von Bedingungen, Aufrufe von Wrapperfunktionen oder das Kopieren der Werte von *ports* hat und für eine 25ms Periode auf weniger als 2 Prozent angegeben wird.

Literatur

[Chr02] Christoph M. Kirsch. Principles of Real-Time Programming. In *in Proc. Second International Workshop on Embedded Software (EMSOFT), LNCS 2491*, Seiten 61–75. Springer Verlag, 2002.

[HHK03] T.A. Henzinger, B. Horowitz und C.M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84 – 99, jan 2003.

[HK07] Thomas A. Henzinger und Christoph M. Kirsch. The embedded machine: Predictable, portable real-time code. *ACM Trans. Program. Lang. Syst.*, 29(6), Oktober 2007.

[Kot99] Markus Kottmann. Software for Model Helicopter Flight Control, 1999.

[KSHP02] Christoph Meyer Kirsch, Marco A. A. Sanvido, Thomas A. Henzinger und Wolfgang Pree. A Giotto-Based Helicopter Control System. In *Proceedings of the Second International Conference on Embedded Software*, EMSOFT '02, Seiten 46–60, London, UK, UK, 2002. Springer-Verlag.