**UNIVERSITÄT ESSEN**
INSTITUT FÜR INFORMATIK
Arbeitsgruppe Systemmodellierung/Verlässlichkeit von Rechnernetzen
http://www.informatik.uni-essen.de

# Entwurf eines TTP/C-Feldbusmodells mit der Spezifikationssprache SDL

Jens Lisner, Patrik Kessler

Technischer Bericht Nr.5
(August 2001)

email: kessler@informatik.uni-essen.de
Telefon: (+49) (+201) 183-4251
email: lisner@informatik.uni-essen.de
Telefon: (+49) (+201) 183-3186

Fax: (+49) (+201) 183-2419

# Inhaltsverzeichnis

# 1. Motivation und Ziele

Im Auto werden immer mehr Funktionalitäten nicht mehr mechanisch, sondern rein elektronisch gelöst. Man spricht von sogenannten „X-By-Wire“-Systemen. Eine mechanische Rückfallebene wird es zukünftig nicht mehr geben. Dabei werden Steuerungssysteme unter Verwendung von automobilinternen Bussystemen eingesetzt. Beispielsweise werden Antiblockiersysteme, Antischlupfregelungen, Fahrdynamik-regelungen oder auch Lenksysteme zukünftig mit diesen Systemen ausgestattet sein. Die Anforderungen an solche Systeme sind bezüglich Verfügbarkeit und Fehlertoleranz sehr hoch. Ein Einsatz im Auto erfordert auch nach der Entwicklung einen erheblichen Validations- und Verifikationsaufwand.
Das Ziel dieser Arbeit ist es ein Modell zu erstellen, das die grundlegenden Eigenschaften des TTP/C-Felbusprotokolls (Time-Trigger-Protokoll) beschreibt.
Dabei soll das Buszugriffsverfahren TDMA (time division multiple access) grundlegend beschrieben werden. Das TTP/C-Modell soll ein erweiterbares Grundkonzept für weitere Untersuchungen im Bereich der Feldbustechnik im Automobilbereich darstellen.

# 2. TTP/C

## 2.1 Überblick über das Protokoll und die Komponenten

Das TTP/C-Protokoll ist ein Feldbusprotokoll, das von der UNI-Wien für den Automobilbereich entwickelt wurde.
Beim TTP/C handelt es sich um ein zeitgesteuertes Protokoll. Es ist durch eine exakte zeitliche Reihenfolge der Buszugriffe der Busteilnehmer (Busknoten) geprägt. Dieses Verfahren wird TDMA genannt. Alle Knoten müssen sich an zeitliche Vorgaben halten. Dieses Verfahren erfordert eine Zeitsynchronisation aller Knoten und die Einführung einer globalen Zeit. Das Konzept des Buszugriffs ersetzt im Normalbetrieb ein Kollisionsauflösungsverfahren.
Das Kernstück eines Busknotens ist der TTP/C-Controller. Des Weiteren besteht ein Knoten aus Bustreiber, Host und I/O Interfaces. Die Schnittstelle zwischen TTP/C-Controller und Host bildet die CNI (Communication Netzwork Interface). Der TTP/C-Controller besteht aus Protokoll-Prozessor und TTP/C-Control-Data (MEDL). Der

Busguardian ist meistens nah an der physikalischen Schicht zu finden. Oft wird er sogar im Bustreiber implementiert.

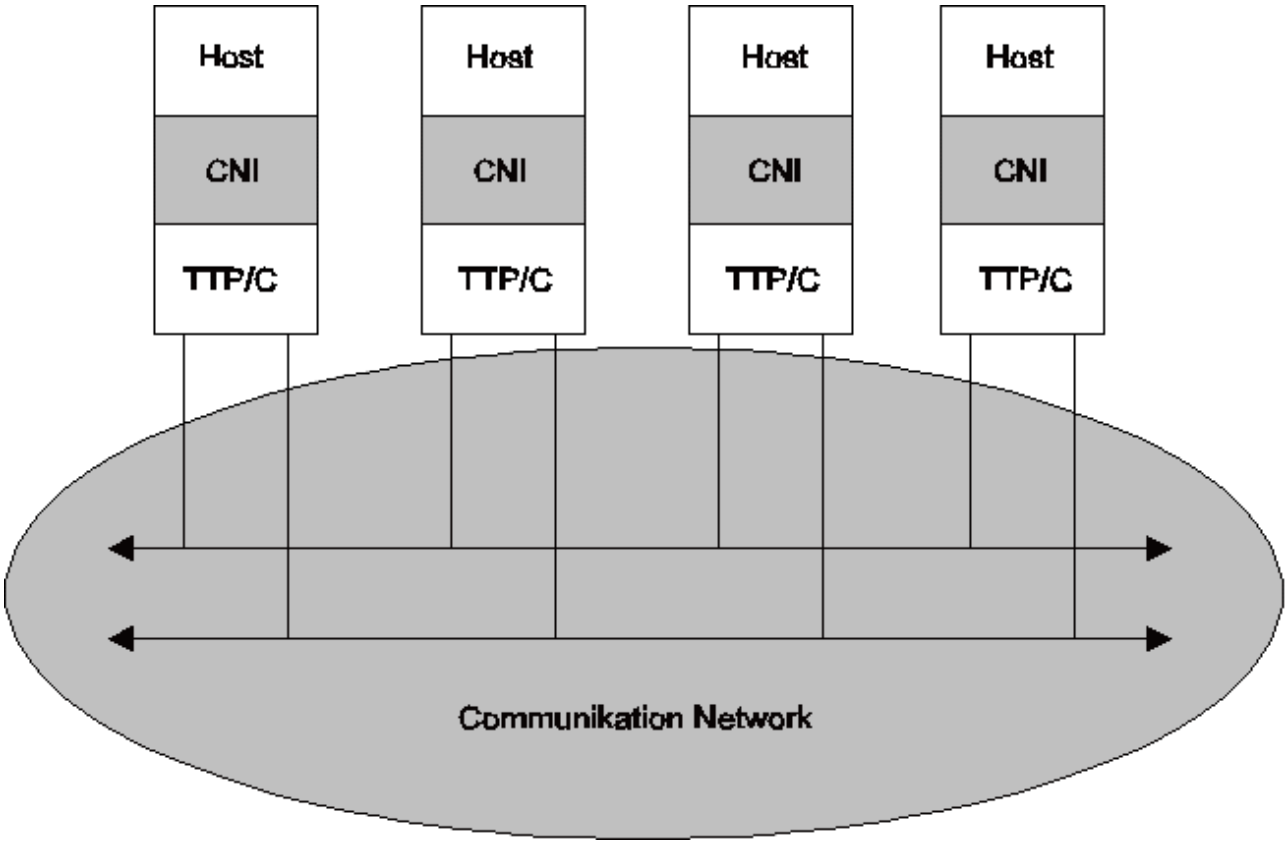


**Abbildung 1: Bustopologie in einem TTP/C-Netzwerk**

## 2.2 Begriffe

**Knoten:** Endpunkte des Netzwerks, also SRU und Lastprozess

**CNI:** Schnittstelle zum Datenaustausch zwischen Controller und Host

**Timer:** Taktgeber für Prozesse innerhalb des Knotens

**SRU:** kompletter Knotenendpunkt bestehend aus Busguardian, Controller- und Lastprozess (Applikation). Das TTP/C Protokoll unterscheidet zwischen echten Teilnehmern (real member SRU) und virtuellen Teilnehmern (virtual member SRU)

**Host:** gehört zur Applikationsschicht; tauscht mit dem Knoten über die CNI Daten aus; Teil des Lastprozesses

**Applikation:** entspricht dem Lastmodell

**Frame:** über den Bus verschickte Daten (Framesignal = Datensignal)

**MEDL:** Message Descriptor List: enthält die Konfigurations-, Laufzeit- und CNI-Daten des Knotens (z.Z. durch den CNI-Prozess teilweise realisiert.)

**Slot:** periodischer Sendezeitschlitz für eine bestimmte SRU

**TDMA-Zyklus:** Zeit, in der alle Slots einmal durchlaufen werden

**Cluster-Zyklus:** Zusammenfassung mehrerer TDMA-Zyklen

**PSP:** Pre-Send-Phase

**PRP:** Post-Receive-Phase

**IFG:** Inter-Frame-Gap

**NBW:** Non-Blocking-Write Protocol (wird zur Kommunikation zwischen Host und CNI eingesetzt)

## 2.3 Das Zeitkonzept von TTP

Beim TTP/C-Protokoll ist jede Aktion vom Erreichen bestimmter Zeitpunkte abhängig. Das bedeutet, dass alle Knoten über die gleichen Zeitinformationen verfügen müssen. Die Grundlage der Zeitinformationen bilden die in jedem Knoten erzeugten Microticks, die auf den Schwingungen eines eingebauten Oszillators basieren.

Eine gewisse Anzahl von Microticks ergeben einen Macrotick. Die Slotlänge ist in Einheiten von Macroticks angegeben. Die Dauer aller Slots zusammengenommen ergeben einen TDMA-Zyklus.

Jeder Netzknoten kennt seinen eigenen Zeitplan, der in der MEDL (Message-Descriptor-List) hinterlegt ist. Der Zeitplan bestimmt, welche Handlungen ausgeführt werden. Insbesondere beinhaltet er die Informationen, wann Nachrichten versendet werden müssen. Die MEDL enthält die Informationen über Anfang und Ende eines Sendeschlitzes sowie globale Informationen zu allen Netzknoten.

Alle Netzknoten kennen somit das beabsichtigte (zukünftige) Systemverhalten, wenn sie eine Nachricht erhalten. Eine Bestätigung (acknowledgement) braucht daher nicht versendet werden, da ein Netzknoten sofort erkennt, wenn eine Nachricht nach der erwarteten Ankunftszeit beispielsweise nicht oder korrekt empfangen wird.

Die Uhren der Netzknoten müssen daher genau genug aufeinander abgestimmt sein, damit sich die Knoten auf einen aktuellen Slot einigen können. Jeder Sendeknoten besitzt bestimmte Sendeslots, in denen er Nachrichten über den Bus schicken kann.

Jeder Slot kann in zwei Phasen aufgeteilt werden:

Kommunikationsphase:

In der Kommunikationsphase sendet der Absender seine Nachricht über den Bus. Die Empfänger können diese dann unmittelbar auslesen.

Berechnungsphase:

In der zweiten Phase verändert jeder Knoten seinen inneren Zustand. Die Zustandsänderung ist von der empfangenen Nachricht abhängig.

Die Zeit in TTP/C ist zyklisch. Das bedeutet: Nach jeder TDMA-Runde wird die gemeinsame Zeitbasis wieder auf Null gesetzt. Um Fehler zu vermeiden, muss eine Applikation berücksichtigen, dass anwendungsabhängige Ereignisse im schlechtesten Fall erst nach einer kompletten TDMA-Runde verarbeitet werden können.

Damit gilt:

Wenn i Slots / 1 TDMA-Zyklus und

j Macroticks / 1 Slot und

k Microticks / 1 Macrotick

dann gilt:

1 TDMA-Zyklus = i * j * k Microticks

1 Slot = j * k Microticks

1 Macrotick = k Microticks

## 2.4 Uhrensynchronisation

Bei verteilten verlässlichen Echtzeitsystemen ist die fehlertolerante Uhrenabstimmung ausschlaggebend. Der Zeitsynchronisierungsdienst ist ein wesentlicher Bestandteil des TTPs und die Basis für das Erreichen der erforderlichen Echtzeitfähigkeit.

Um ein beliebiges Auseinanderdriften der Uhren auszugleichen, müssen sie in regelmäßigen Abständen synchronisiert werden. Dabei wird der Ankunftspunkt ankommender Nachrichten benutzt, um den Wert der Uhr des Absenders zu schätzen. Es gibt keinen besonderen Abstimmungsmechanismus, der die Information der Uhr eines anderen Netzknoten bereitstellt. Die Timing-Information von ausgewählten Netzknoten werden dafür gesammelt und ausgewertet.

Die Uhr eines Netzknotens wird durch einen Zähler realisiert, der über einen Oszillator periodisch inkrementiert wird. Der Uhrensynchronisationsalgorithmus besitzt die Aufgabe, die Änderung der Uhr in Bezug zu den Abweichungen der Uhren der anderen Knoten neu zu berechnen.
Die Methode der Uhrensynchronisation beruht darauf, Schätzungen für den Erhalt von Nachrichten anderer Netzknoten zu sammeln und hieraus die Änderung für die eigene Uhr zu berechnen. Als Grundlage dazu wird die Differenz der erwarteten zur tatsächlichen Ankunftszeit der Nachricht benutzt. Durch diese Methode ist kein zusätzlicher Nachrichtenaufwand für die Uhrensynchronisation erforderlich.
Die Zeitwerte werden in einem push-down-Stack der Größe 4 gespeichert. Ältere Werte werden nach einer Weile verworfen. Netzknoten, von denen bekannt ist, daß der Quarzoszillator zu ungenau ist für die Korrekturberechnung, sind von der Berechnung ausgeschlossen. Dies wird durch ein in der MEDL gesetztes Flag erreicht.
TTP benutzt den Fault-Tolerant-Average-Algorithmus (FTA), um die Korrektur zu berechnen. Von den Werten werden der größte und der kleinste verworfen. Aus den beiden verbleibenden wird der Durchschnitt gebildet und zur Änderung benutzt. Zur Berechnung der Korrektur (clock state correction term) ist mit diesem Algorithmus eine TDMA-Runde erforderlich.
In jedem Zeitslot werden folgende Schritte ausgeführt:

1. Bestimmung des Unterschiedes zwischen erwarteter und tatsächlicher Ankunftszeit der einzutreffenden Nachricht.

2. Wenn eine gültige Nachricht empfangen wurde, wird ein Flag für den aktuellen Slot in der MEDL gesetzt. Der gemessene Zeitunterschiedswert wird dann in den Stack geschoben, wenn die Nachricht gültig war, ansonsten wird er verworfen.

3. Wenn das CS-Flag in den MEDL gesetzt ist, wird ein Uhrenkorrekturwert aus den letzten beiden Werten berechnet. Anhand dieses Korrekturwertes wird die lokale Uhr entsprechend eingestellt.
Am Ende des TDMA-Zyklus` wird die Uhrenkorrektur vorgenommen.

Die Uhrensynchronisation ist im vorliegenden Modell noch nicht spezifiziert.

## 2.5 Controller

Der Controller ist das Kernstück des TTP/C-Protokolls. Er hat die Aufgabe, Frames zu versenden und zu empfangen. Der Busguardian leitet gesendete Nachrichten zum Bus weiter. Diese werden vom Controller direkt empfangen. Der Controller arbeitet komplett selbstständig. Die Aktivitäten werden komplett von den in der MEDL hinterlegten Informationen gesteuert. Der Host nimmt keinen Einfluss auf die Kommunikation. Nachdem die globale Zeit den Sendezeitpunkt des Controllers erreicht hat, wird aus dem Speicher der CNI die Nachricht entnommen und versendet. Empfangene Nachrichteninhalte werden ebenfalls wieder in einer Speicherstelle der CNI hinterlegt.
Der Controller kann an einem oder an zwei Kanälen angeschlossen sein. Über beide Kanäle können sowohl die gleichen als auch verschiedene Nachrichten versendet werden. Der Controller führt auch die Uhrensynchronisation durch. Werden zwei Kanäle benutzt, wird der Mittelwert der Abweichungen auf beiden Kanälen zur Berechnung der Korrektur herangezogen.

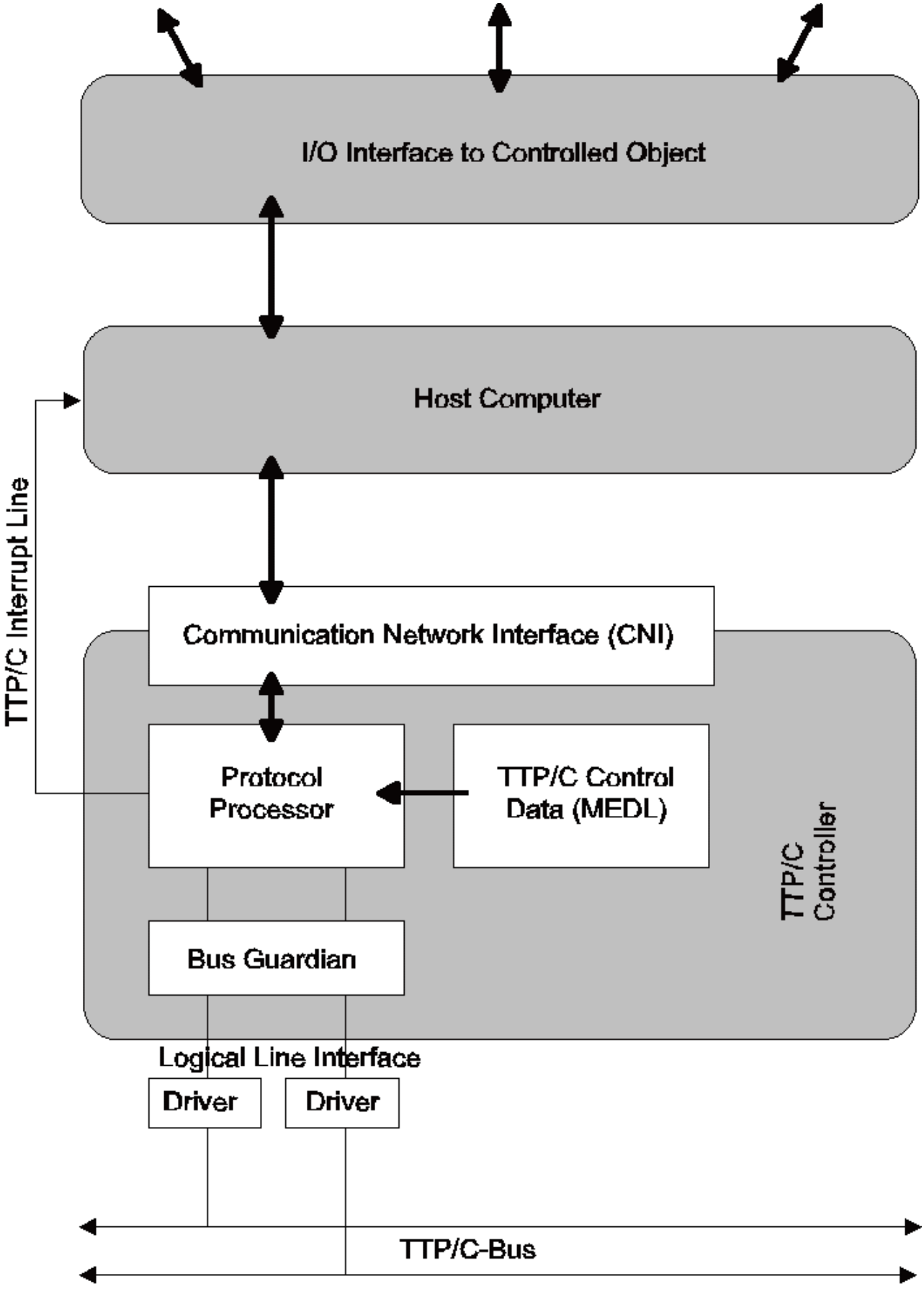


**Abbildung 2:SRU in einem TTP/C-Netzwerk**

## 2.6 Der Busguardian

Der Buswächter ist ein autonomes Teilsystem im Netzknoten, der die Kommunikationskanäle vor zeitlichen Übermittlungsfehlern schützt.
Physikalisch gesehen agiert der Busguardian auf der gleichen Schicht wie der TTP/C-Controller; er kann auch als ein unabhängiges Gerät ausgeführt werden.

Aufgaben des Busguardian
Der Busguardian muss den Bus vor Monopolisierung durch einen Knoten schützen, der ständig unkontrolliert Frames zum Bus sendet (babbling idiot).
Darüber hinaus muss der Busguardian den Bus vor einer Übermittlung des Controllers außerhalb seines Sendeschlitzes schützen, darf jedoch den Controller nicht hindern, innerhalb seiner zugewiesenen (korrekten) Zeit zu senden. Sendet der Controller außerhalb seines ihm zugewiesenen Slots, wird der Zugriff auf den Bus durch den Busguardian gesperrt. Eine Wiederaufnahme der Kommunikation ist dann nur noch durch einen Neustart des Knotens möglich.
Der Busguardian muss eine Uhr benutzen, die vom Oszillator des Controllers unabhängig ist. Dadurch wird eine zeitliche Kopplung der beiden Teilsysteme verhindert und somit vorsorglich eine Fehlerquelle vermieden.
Ein periodisches Synchronisationssignal vom Controller zum Busguardian (ARM-Signal) ist erforderlich, um das Busguardian-Fenster bei Modewechsel bzw. nach einer Uhrenabstimmung zu synchronisieren. Es zeigt die Lage des Sendeschlitzes an und kann statisch (im Controller-Design) oder konfigurierbar (im MEDL) sein.

Um eine hohe Fehlererkennung zu erreichen, sollte der Busguardian möglichst physikalisch in der Nähe der Kanäle angesiedelt sein. Im besten Fall wird das Ausgabesignal sogar vom Bustreiber kontrolliert.
Der Busguardian muss das Übermittlungsfenster für den Controller zur richtigen Zeit öffnen und wieder schließen. Für die vollständige Frame-Übermittlung öffnet er das Übertragungsfenster. Nach dem Sendeslot (Schlitzdauer) muss er es wieder schließen und während der gesamten Übertragungsrunde geschlossen halten.
Um Uhrenabweichungen zwischen dem Busguardian-Timing und dem Controller-Timing zu berücksichtigen, sollte der Busguardian das Fenster ein wenig ($\varepsilon$) früher als die

erwartete Sendezeit öffnen und das Fenster ein wenig später (ε) als das erwartete Ende von Übermittlungen schließen.

## 2.7 Die MEDL (Message-Descriptor-List)

Die MEDL enthält keine Knoteninformationen der CNI-Schicht, sondern nur netzwerkspezifische Daten. Jeder Knoten enthält seine eigenen individuellen MEDL-Informationen. Außerdem enthält er eventuell Daten, die der Controller für bestimmte eigene Zwecke benötigt.
Eine MEDL ist immer abhängig von der Controllerimplementierung. Es gibt keine allgemein gültige Struktur.
Die grundlegende Funktionalität muss jedoch von allen Implementierungen bereitgestellt werden.
Die MEDL enthält Konfigurationsparameter wie SRU-Name, Schedule Identification, Application Identification, Busguardianparameter usw.
Wenn der Host eine Modeänderung erreichen will, fragt er seinen Controller, ob dies momentan möglich ist.
Wenn die Modeänderung genehmigt wird, generiert der Controller ein mode-change-request, anderenfalls einen mode-violation-host-error.

## 2.8 CNI (Communikations Network Interface)

Durch die CNI wird eine Schnittstelle vom Controller zum Systemprogrammierer des Hostcomputers bereitgestellt.
Die CNI ist in einem Speicherbereich beinhaltet, der gleichzeitig Zugriff von der Host-CPU und vom Controller erlaubt. Dieser Speicherbereich wird als DPRAM (dual-ported access memory) bezeichnet, auf dem mit dem NBW-Protokoll zugegriffen wird. Aus der Sicht des Host-Rechners beginnt die CNI bei einer bestimmten Adresse, die als CNIBASE in der MEDL zu finden ist.
Konzeptionell kann die CNI in zwei bedeutende Bereiche unterteilt werden:
Status/Control Area:

Der Status/Control-Bereich wird zum Austausch von Kontrollinformationen zwischen dem Controller und dem Host-Rechner genutzt.
Er erhält die CNI-Statusfelder, die vom Controller aktualisiert und vom Host-Rechner gelesen werden. Außerdem enthält er die CNI-Control-Felder, die vom Host-Rechner aktualisiert und vom Controller gelesen werden.
CNI Message Area:
Der CNI-Message-Bereich enthält alle TTP/C-Nachrichten, die versendet oder von der SRU empfangen werden. Das message-data-Feld enthält die Anwendungsdaten, die vom Controller versendet werden.
Die Aktualisierung der CNI-Felder, die sich im Laufe der TDMA-Runde verändern, findet am Ende des Slots jeder SRU während der post-receive-Phase (PRP) statt. Sobald die Aktualisierung ausgeführt wurde, werden die Werte des neuen SRU-Slots für weitere Operationen genutzt.

# 3. Anforderungsdefinition

## 3.1 Kontext der Analyse

Das Modell soll als Gerüst für Leistungsbewertungen des TTP/C-Feldbusprotokolls dienen. Dabei soll eine Leistungsbewertung sowohl durch Simulation als auch durch Validation ermöglicht werden.

## 3.2 Anforderungen

Das Modell soll folgende Anforderungen erfüllen:

Strukturen der Bustopologie
Das Modell soll den grundlegenden Aufbau bezüglich Knoten und physikalischem Busmedium nachbilden.

Busfunktionalität
Die Busfunktionalität muss insofern nachgebildet werden, dass Nachrichten wie auf einem realen Übertragungsmedium an alle Busteilnehmer versendet werden können. In einem

realen Bussystem nimmt der Bus den Signalpegel an, der von einem sendenden Knoten auf das Übertragungsmedium gelegt wird. Im Modell soll diese Eigenschaft durch Versenden der Nachrichten an alle Busknoten nachgebildet werden.

Busguardian

Der Busguardian muss die ihm übertragenen Aufgaben modellhaft erfüllen. Dazu gehört, das Bussystem vor einem fehlerhaften Controller zu schützen, indem ein Zeitfenster die Übertragung des Controllers freigibt.

Strukturen der TDMA

Der Controller muss die Zugriffsstrategie des TTP/C-Protokolls realisieren. Dazu gehört das Zuteilen von Sendeslots an die jeweiligen Busknoten.

MEDL

Die Message-Descriptor-Liste beinhaltet Informationen, die den Kommunikationsablauf betreffen. Im Modell müssen diese Informationen ansatzweise im Controller zur Zuweisung von Sendeslots enthalten sein.

Übertragung von Frames

Das Modell muss fähig sein, Frames über den Bus von einem Knoten zu senden und von den anderen Knoten zu empfangen.

Zeitkonzept

Über zwei separate Prozesse muss das Zeitkonzept vom Controller und vom Busguardian modelliert werden. Dabei muss die Zeitbasis aus Microticks, Macroticks und Zeitslots beruhen.

Die Uhrensynchronisation wird in einer späteren Entwurfsphase ins Modell integriert.

Struktur von Lastmodell und Fehlermodell

Das Modell muss konzeptionell die Möglichkeit offen halten, Lastmodelle und Fehlermodell in Form von SDL-Prozessen zu integrieren.

CNI

Ein Datenaustausch zwischen Controller- und Applikationsprozess muss durch ein CNI-Prozess erreicht werden.

# 4. Modellierung

## 4.0 Annahmen

Bei Leistungsanalysen am Modell sollte das Hauptaugenmerk auf der Spezifikation von grundlegenden funktionalen und leistungsrelevanten Aspekten liegen. Eine bis ins Detail exakte Nachbildung des Protokolls ist nicht erforderlich.

Das TTP/C-Feldbusmodell beschreibt in diesem Entwurfsstadium nur einen Teil der kompletten TTP/C-Funktionalität.

## 4.1 Systemstruktur

Das TTP/C-System wird in einen Bus- (Last- und Fehlerinjektionsblock) und in mehrere Knotenblöcke aufgeteilt. Zwischen den Knotenblöcken und dem Busblock sind jeweils bidirektionale Signal-Kanäle vorhanden. Zwischen diesen Blöcken werden die normalen Kommunikationssignale ausgetauscht.

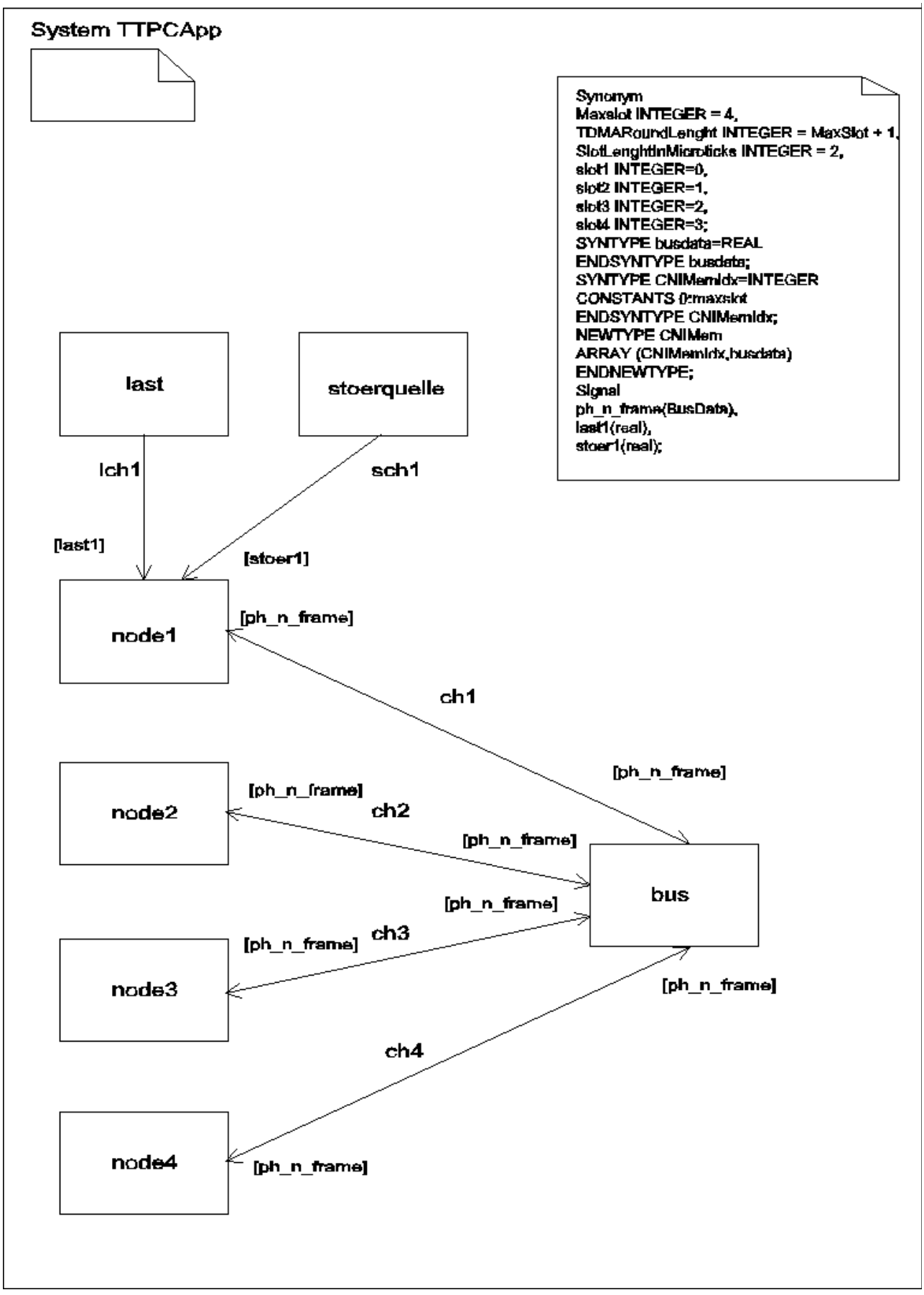


**Abbildung 3: Systemstruktur des Modells**

## 4.2 Blockstruktur

### 4.2.1 Knoten

Im Knotenblock ist die gesamte Funktionalität der TTP/C-SRU enthalten. Die Prozesse *timer* und *BGtimer* erzeugen die Zeitbasis für den Controller und den Busguardian. Die Signalrouten sind unidirektional vom Timerprozess zum Controller bzw. zum Busguardian. Ein Knoten besteht aus jeweils einem Controller und einem Busguardian. Signale zum Bus müssen den Busguardian passieren. Signale vom Bus werden direkt an den Controller gesendet.

Der Prozess *busguardian* stellt ein Bindeglied zwischen Controller und Environment bezüglich zu sendender Nachrichten dar. Es besteht eine unidirektionale Signalroute sowohl vom Controller zum Busguardian als auch vom Busguardian zur Blockumgebung. Der Applikationsprozess stellt das Bindeglied zwischen Lastprozessen und Controllerprozess her.

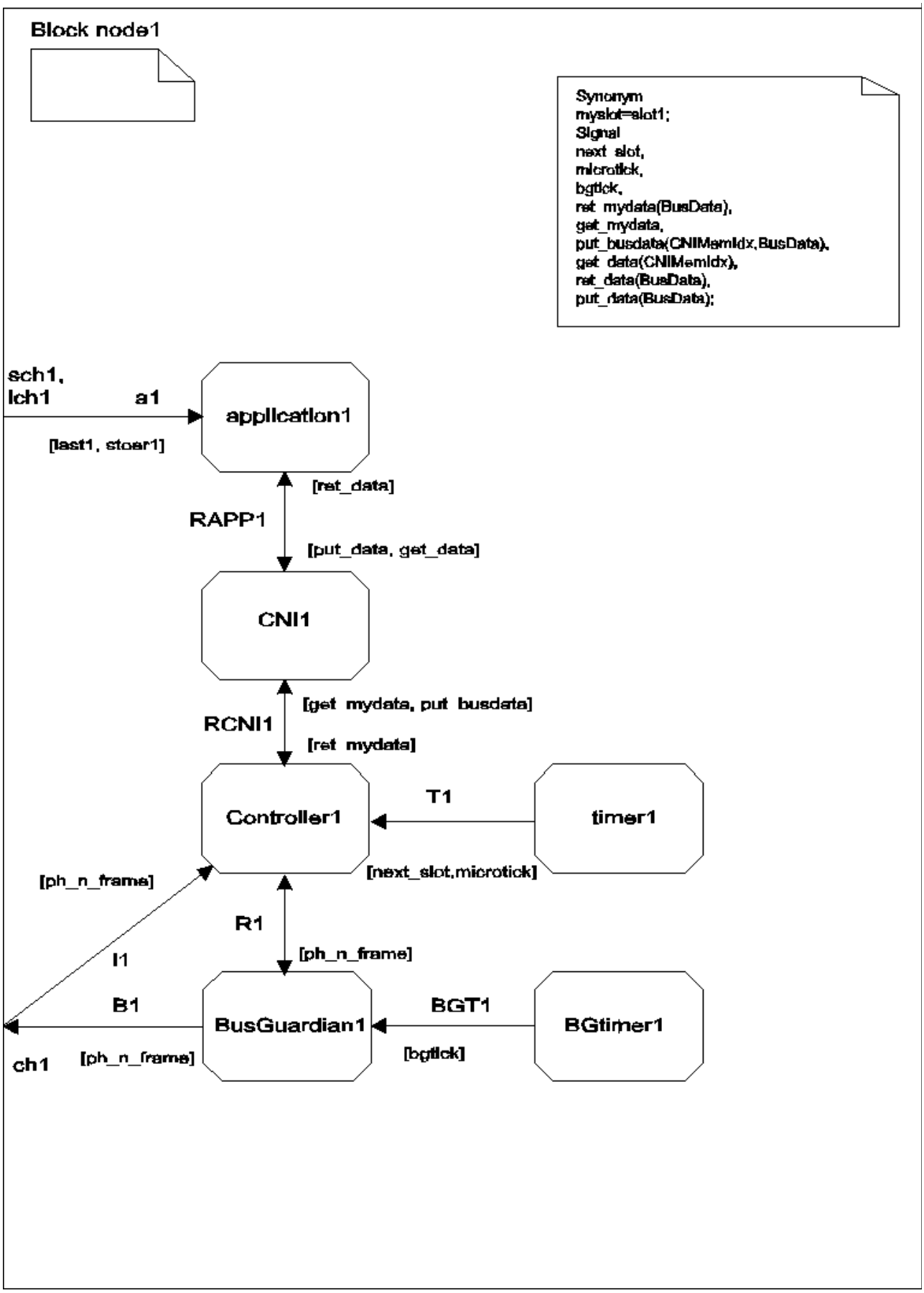


**Abbildung 4: Knoten mit allen Prozessen**

### 4.2.2 Bus

Der Block *bus* modelliert das physikalische Verhalten des Schnittstellenwandlers, indem er die gesendeten Frames empfängt und an alle SRUs weiterleitet.

## 4.3 Prozessstruktur

### 4.3.1 Controller

Der Prozess *controller* ist ein Zweizustandsprozess. Aus dem Zustand *wait_for_slot* kann der Prozess zum einen das Timersignal von dem Timerprozess und zum anderen das Signal *ph_n_frame* von der Applikationsschicht empfangen. Das Signal *next_slot* wird in der Signaleingangsqueue priorisiert verarbeitet. Dadurch wird ein signalunabhängiger Zeittakt erreicht. Nach jedem Empfang eines *next_slot*-Signals wird der Slotzähler um eins erhöht. Ist die TDMA-Runde abgeschlossen, wird der Slotzähler wieder auf Null gesetzt.
Hat der Slotzähler die Slotnummer des Knotens erreicht, besitzt der Knoten die Senderechte und übermittelt seine Nachricht über den Buswächter an den Bus.
Der Prozess *Controlle*r kann das Signal *ph_n_frame* vom Bus, also von den anderen sendenden Knoten, empfangen.

```
start
   slot := 0;                    /* Initialisierung: erster Slot ist
                                 Slot 0. */

Zustand: warten
  falls signal next_slot empfangen:     /* Ein neuer Slot beginnt. */
    falls slot = myslot:              /* Der für diesen Controller
                                 reservierte Slot ist erreicht. Die
                                 von der Applikation in der CNI
                                 hinterlegten Nutzdaten müssen auf
                                 den Bus gelegt werden. */
      hole daten von der CNI
      sende daten zum bus

    falls slot = MaxSlot;             /* Sicherstellen, das die Zeit
                                 zyklisch von 0..MaxSlot verläuft. */
      slot := 0;
    sonst:
      slot := slot + 1;

    nächster Zustand: warten

  falls signal ph_n_frame(data) empfangen:
                               /* Daten wurden vom Bus empfangen. */
    sende (data, slot) zur CNI        /* Die Daten werden für die
                                 Applikation in der CNI hinterlegt.
                                 Die Slotnummer dient der Zuordnung
                                 zum sendenden Knoten. */
    nächster Zustand: warten
```

### 4.3.2 Timer

Der Prozess *Timer* sendet dem Controller ein Signal pro Macrotick. Das Signal wird in diesem Entwurf noch nicht verwendet. Es wird vom Controllerprozess verworfen. Wenn ein neuer Slot beginnt, wird stattdessen ein *next_slot*- Signal gesendet.

Der Prozess *BGTimer* sendet je Macrotick ein Signal an den Busguardian.

### 4.3.3 CNI

Der CNI-Prozess stellt die Schnittstelle für den Datenaustausch zwischen Host und Kommunikationscontroller her. Die lokal vorgehaltenen Daten werden auf Anforderung an den Sender zurückgegeben.

### 4.3.4 Busguardian

Der Buswächter erhält vom Prozess *BG_Timer* ständig Signale, mit denen die fortschreitende Zeit (also im Prinzip das Quarz) modelliert wird. Damit wird die zeitliche Unabhängigkeit des Buswächters gegenüber dem Controller erreicht. Der Prozess *guardian* ist ein Einzustandsprozess. Nach Empfang des Timersignals wird geprüft, ob der Sendeslot erreicht ist. Dabei wird der Zähler um eins erhöht. Die Variable *w_open* gibt den Sendekanal frei. Das empfangene Signal *ph_n_frame* vom Prozess *controller* wird versendet.

Das Signal *bg_tick* wird in der Prozesseingangsqueue priorisiert verarbeitet. Dadurch wird ein signalunabhängiger Zeittakt erreicht.

Nach Beendigung der TDMA-Runde wird der Tickzähler wieder auf Null gesetzt.

Beispiel:

Eine TDMA-Runde beträgt 15 Slots.

Der betrachtete Knoten erhält die Busrechte im 10. Slot.

Die Slotlänge beträgt 200 Bit.

starttick = slot * slotlenght = 10 *200 = 2000

endtick = starttick + slotlenght -1= 2000 + 200 -1 = 2199

maxtick = slots * slothlenght -1 = 15 * 200 -1 = 2999

Bei diesem Beispiel öffnet der betrachtete Busguardian den Sendeslot von tick-Nummer 2000 bis 2199. In der TDMA-Runde befindet er sich an 10. Stelle. Die Öffnungsdauer beträgt 200 Ticks.

```
start
   tick := 0;                    /* Initialisierung: Zeitpunkt 0 in der
                                 ersten TDMA-Runde. */
   window_open := false;         /* Das Sendefenster ist nicht
                                 offen: keine Sendeberechtigung */
   window_locked := false;       /* Das Sendefenster ist nicht
                                 verschlossen: Sobald der
                                 Sendebereich erreicht wird, darf
                                 gesendet werden. */

Zustand: warten
  falls signal bgmacrotick empfangen:   /* Uhrensignal trifft ein: Macrotick.
                                 Die Uhr hat Vorrang vor den Daten. */
    falls StartTick <= tick <= EndTick:
     window_open := true;        /* Sobald das Sendefenster des
                                 Controllers erreicht ist, wird die
                                 Sendeberechtigung erteilt. */
    sonst:
     window_open := false;

    tick := tick + 1;
    falls tick = MaxTick + 1:
       tick := 0;                /* Setze Zähler zurück, falls Ende
                                 der TDMA-Runde erreicht. */
    nächster Zustand: warten

  falls signal ph_n_frame vom controller empfangen
    falls window_locked = false:      /* Gesendet werden darf nur, falls
     falls  window_open:              das Fenster nicht verschlossen
                                 und der Sendebereich erreicht ist. */
       sende signal ph_n_frame zum bus;
     sonst:
       window_locked := true;    /* Es wurde versucht, außerhalb
                                 des Sendebereichs zu senden. Das
                                 Fenster wird dauerhaft
                                 verschlossen. Der Knoten darf
                                 keine Frames mehr auf den Bus
                                 legen. */
    nächster Zustand: warten
```

# 5. Funktionale Überprüfung mit Hilfe von MSC's

Der Entwurf wird anhand des automatisch erzeugten MSCs (Message-Sequence-Chart) überprüft.

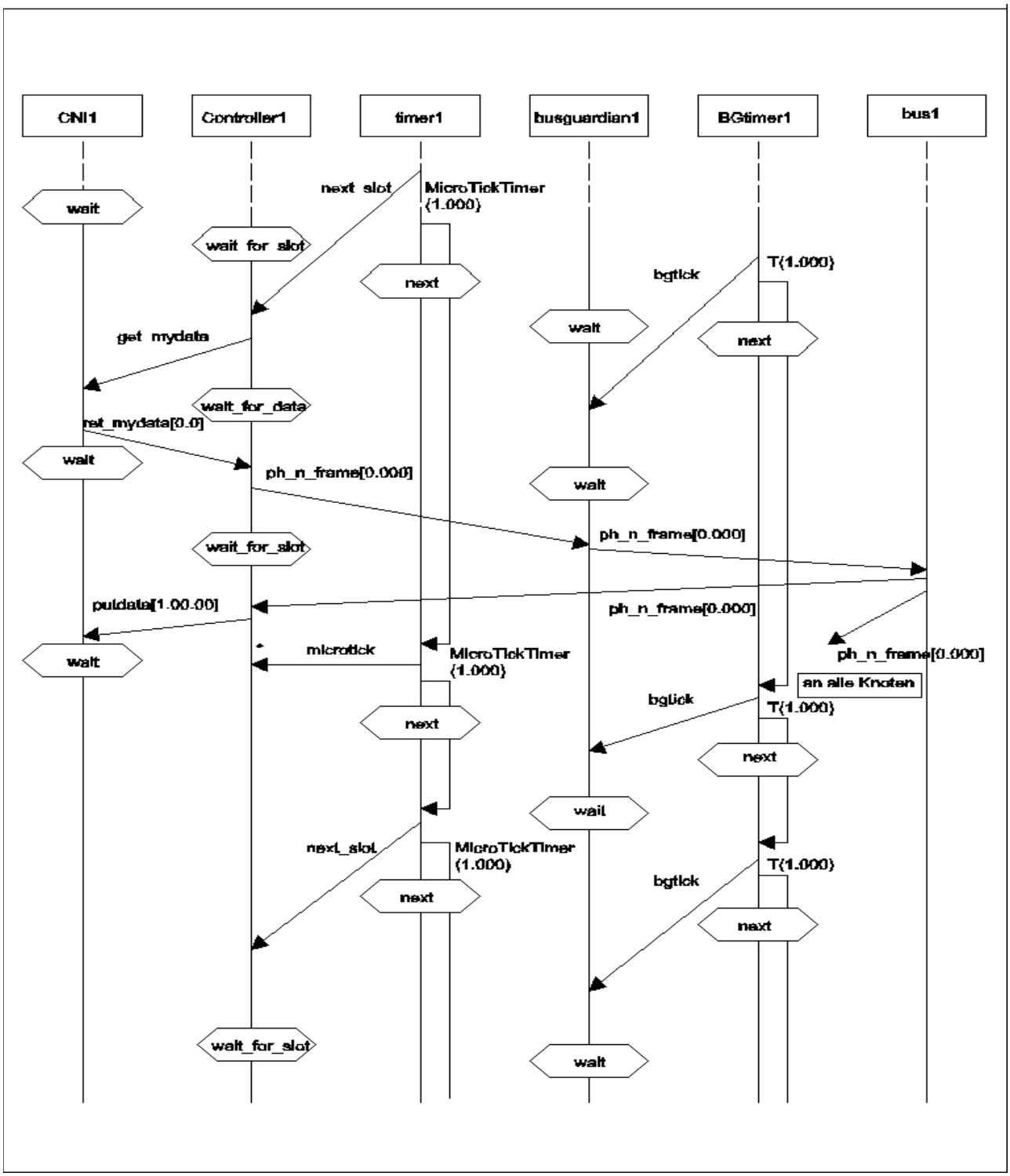


**Abbildung 5: MSC eines SRU-Knotens**

Abbildung 5 zeigt das Message Sequence Chart (MSC) eines SRU-Knotens. In dieser Darstellung wurden nur die Signale eines Knotens aufgezeigt. Auf Signale von Last- und Störprozessen wurde verzichtet.
Dieses MSC zeigt die Vorgänge innerhalb eines Zeitslots.
Der Prozess *timer1* sendet zur Initialisierung ein *next_slot*-Signal zum Prozess *Controller1.* Außerdem wird der Timer *MicroTickTimer* gesetzt. Nach der Initialisierungphase befindet sich der Prozess *Controller1* im Zustand *wait_for_slot*, der Prozess *CNI1* im Zustand *wait* und der Prozess *busguardian1* im Zustand *wait*. Der Prozess BG*timer1* sendet zur Initialisierung ein bgtick-Signal zum Prozess busguardian1. Außerdem wird der Timer *T1* gesetzt. Die Prozesse *timer1* und *BGtimer1* generieren zyklische Signale, die sie jeweils zu den Prozessen *Controller1* und *busguardian1* senden. Sie bilden die Zeitbasis des Systems und sind für die Slotverteilung verantwortlich.
Nachdem der Prozess *Controller1* das Signal *next_slot* empfangen hat, fordert es durch Senden des Signals g*et_mydata* die Daten an, die er in seinem Slot versenden soll. Der Controller erhält seine Sendedaten mit dem Signal *ret_mydata* vom Prozess *CNI1*. Danach wird ein *ph_n_frame*-Signal zum Prozess *busguardian1* gesendet. Dieser hat das Sendefenster dieses Knoten zuvor bei Erhalt des Signals *bgtick* geöffnet und kann somit das Nachrichtenframe zum Prozess *bus1* senden. Der Prozess bus1 verteilt das Signal *ph_n_frame* an alle Controllerprozesse der Knoten.
Der Prozess *Controller1* hat im ersten Slot der ersten TDMA seine Daten versendet. Alle Controllerprozesse der Knoten (auch der Prozess *Controller1* von Knoten1) empfangen das Signal *ph_n_frame*. Die Empfangsdaten werden zum Prozess *CNI1* mit dem Signal *putdata* gesendet. Die Signale *MicroTickTimer* und *bgtick* werden jeweils priorisiert empfangen. Dadurch ist das Einhalten der Slotgrößen gewährleistet. Prozess *timer1* sendet zusätzlich das Signal *microtick,* das von Prozess *Controller1* nicht empfangen werden kann. Dieses Signal wird bei späteren Modellierungsstufen benötigt.

Abbildung 6 zeigt die Slotverteilung innerhalb einer TDMA-Runde. Hierbei ist zu erkennen, dass die Frames von Prozess *bus* an alle Knoten verteilt werden.

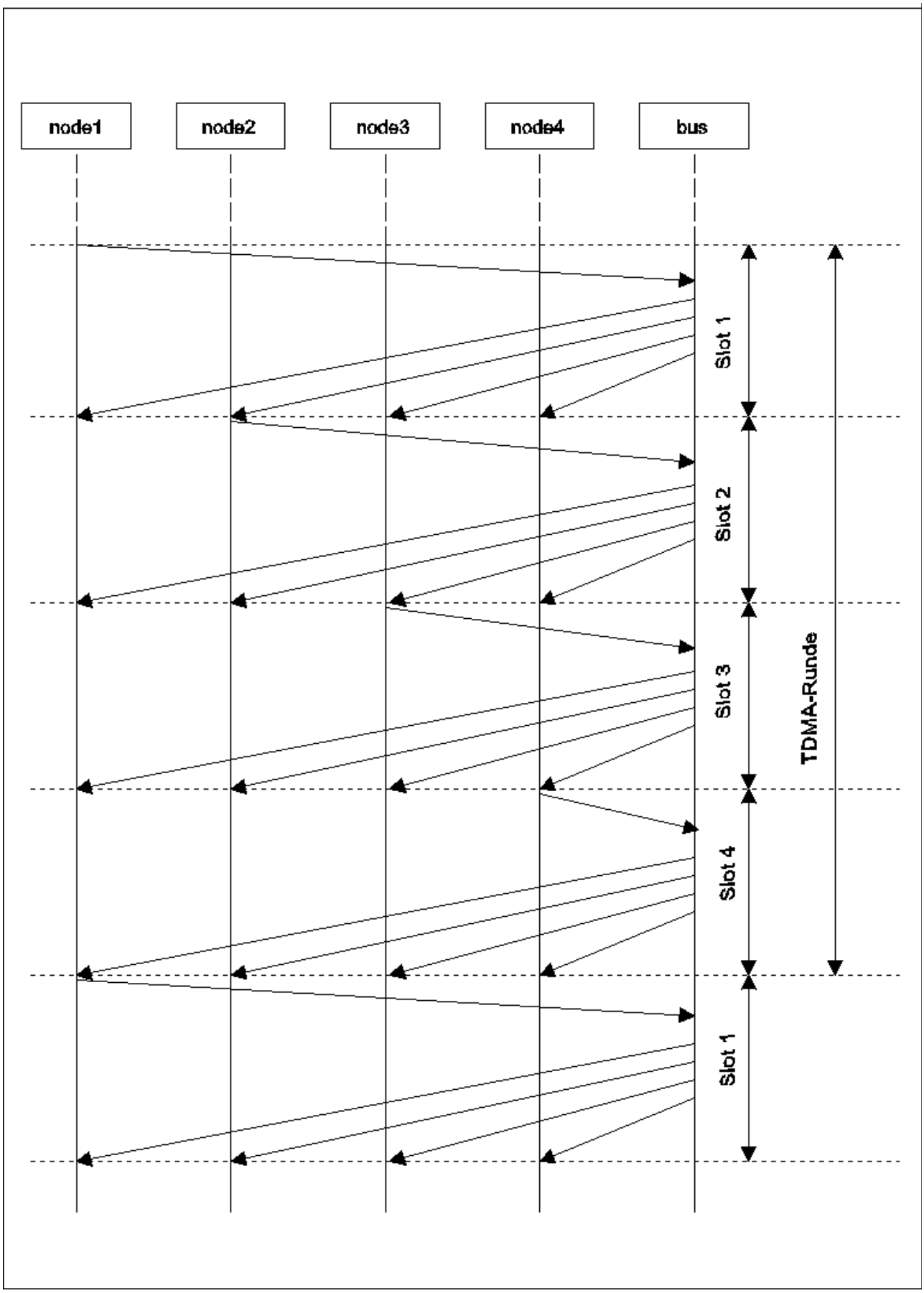


**Abbildung 6: MSC-Slotaufteilung/TDMA-Runde**

# 6. Zusammenfassung

Das Modell stellt einen Entwurf dar, der die grundlegenden Eigenschaften des TTP/C-Feldbusprotokolls beschreibt. Für das Protokoll wurde ein bis auf die Microtickebene detailliertes Modell entwickelt. Controller, Busguardian, Timerprozesse, CNI und Applikation bildeten die Komponenten der SRUs nach. Das physikalische Übertragungsmedium wurde durch den Busblock modelliert. Die Verifikation des Modells erfolgte mit Hilfe eines automatisch generierten Message-Sequence-Charts.

# 7. Ausblick

Das vorhandene Modell ist in seiner konzeptionellen Darstellung bereits für die Weiterentwicklung von Fehler- und Lastmodellen vorgesehen. Das Zeitkonzept ist noch durch die Uhrensynchronisation zu erweitern. Durch die Modellierung des Buszugriffsverfahrens TDMA (Time Divison Multiple Access) bildet dieses Modell auch die Grundlage zur Realisierung von verwandten Verfahren - beispielsweise das Zugriffsverfahren FTDMA (Flexible Time Divison Multiple Access).

# 8. Literatur

[1] Specification of the TTP/C Protocol; Specification version 0.5, Document edition 1.0 of 21-JUL-1999; TTTech Computertechnik AG Wien

[2] Ellsberger, Jan; Hogrefe, Dieter; Sarma, Amardeo: SDL; Telelogic

[3] Specification Byteflight; Specification Version 0.5; 29.10.1999; BMW AG

[4] Kopetz, Ochsenreiter; Clock Synchronisation in Distributed Real-Time Systems; 1987

[5] Lyndelius, Lynch; A New Fault Tolerant Algorithm for Clock Synchronisation; 1988

[6] Lonn; The Fault Tolerant Daisy Clock Synchronisation Algorithm; 1999

## 9. SDL-Code

```
system TTP;
  synonym
    Maxslot INTEGER = 3,                  /* Slots werden von 0 an gezählt.
                                          */
    TDMARoundLenght INTEGER = MaxSlot + 1,
    SlotLenghtInMicroticks INTEGER = 2,     /* In dieser Version gibt es noch
                                            keine Macroticks. */
    slot1 INTEGER = 0, slot2 INTEGER = 1,
    slot3 INTEGER = 2, slot4 INTEGER = 3;

    /* Datentypen
     * busdata : Ueber den Bus zu uebertragene Nutzdaten
     * CNIMemIdx : Indextyp fuer den Speicher in der CNI
     *             Fuer jeden Slot ist in der CNI ein Speicherplatz vorgesehen.
     * CNIMem : Nutzdatenspeicher in der CNI
     */
  syntype busdata = REAL
  endsyntype busdata;

  syntype CNIMemidx = INTEGER
    constants 0 : maxslot
  endsyntype CNIMemIdx;

  newtype CNIMem
    ARRAY( CNIMemIdx, busdata)
  endnewtype;

  /* Signale
   * ph_n_frame (BusData) : Frame mit Nutzdaten.
   * last1 (real) : ???
   * stoer1 (real) : ???
   */
```

```
signal
  ph_n_frame(BusData),
  last1(real),
  stoer1(real);

/* Bus mit 4 Knoten.
 * Lastquelle und Stoerquelle sind mit Knoten 1 verbunden.
 * Die Knoten tauschen untereinander ph_n_frame-Signale über den Bus aus.
 */
channel lch1
  from last to node1 with last1;
endchannel lch1;
channel sch1
  from stoerquelle to node1 with stoer1;
endchannel sch1;
channel ch1
  from bus to node1 with ph_n_frame;
  from node1 to bus with ph_n_frame;
endchannel ch1;
channel ch2
  from bus to node2 with ph_n_frame;
  from node2 to bus with ph_n_frame;
endchannel ch2;
channel ch3
  from bus to node3 with ph_n_frame;
  from node3 to bus with ph_n_frame;
endchannel ch3;
channel ch4
  from bus to node4 with ph_n_frame;
  from node4 to bus with ph_n_frame;
endchannel ch4;

block last referenced;
block stoerquelle referenced;
```

```
  block node1 referenced;
  block node2 referenced;
  block bus referenced;
  block node3 referenced;
  block node4 referenced;

endsystem TTP;



block last;
  signalroute l
    from last_node1 to env with last1;
  process last_node1 referenced;
  connect lch1 and l;
endblock last;


process  <<block last>> last_node1;
  timer
    lastabstand;
  dcl
    lastwert,
    handle real;

  start;
      grst0 :
      task handle := 5;
      task lastwert := - 0.02 + 0.04 * (handle / 32768);
      output last1(lastwert);
      set(now + 20, lastabstand);
      nextstate next;

  state next;
```

```
    input lastabstand;
      join grst0;
  endstate;
endprocess  <<block last>> last_node1;



block node1;
  synonym myslot = slot1;

  /* Signale (Block Node1)
   * next_slot : Timer meldet dem Controller den Beginn eines neuen Slots.
   * microtick : Timer meldet dem Controller einen Microtick.
   *           (Dieses Signal wird derzeit nicht verwendet und vom
   *           Controller verworfen.)
   * bgtick : Microtick des BusGuardianTimers (BGTimer) an den BusGuardian.
   * get_mydata, ret_mydata, put_busdata : sind Teil eines Protokolls zum
   *                              Datenaustausch zwischen Controller
   *                              und CNI
   * get_data, ret_data, put_data :       dito, fuer die Kommunikation
   *                              zwischen CNI und Applikation
   */

  signal
    next_slot,
    microtick,
    bgtick,
    ret_mydata(BusData),
    get_mydata,
    put_busdata(CNIMemIdx, BusData),
    get_data(CNIMemIdx),
    ret_data(BusData),
    put_data(BusData);
```

```
/* Kern der SRU sind die Prozesse Controller und BusGuardian, die jeweils
 * von eigenen Timern getaktet werden (Timer und BGTimer).
 * Daten vom Controller gehen ueber den BusGuardian zum Bus. Daten vom Bus
 * werden direkt an den Controller weitergeleitet. Der Controller legt
 * empfangene Daten in der CNI ab und holt zu sendende Daten aus der CNI.
 * Die Applikation liest Daten aus der CNI und legt zu sendende Daten dort
 * ab. Die Applikation ist mit Last- und Stoerquelle verbunden.
 */
signalroute a1
  from env to application1 with last1, stoer1;
signalroute RAPP1
  from application1 to CNI1 with put_data, get_data;
  from CNI1 to application1 with ret_data;
signalroute RCNI1
  from CNI1 to Controller1 with ret_mydata;
  from Controller1 to CNI1 with get_mydata, put_busdata;
signalroute T1
  from timer1 to Controller1 with next_slot, microtick;
signalroute R1
  from Controller1 to BusGuardian1 with ph_n_frame;
signalroute I1
  from env to Controller1 with ph_n_frame;
signalroute B1
  from BusGuardian1 to env with ph_n_frame;
signalroute BGT1
  from BGtimer1 to BusGuardian1 with bgtick;

process application1 referenced;
process CNI1 referenced;
process Controller1 referenced;
process timer1 referenced;
process BusGuardian1 referenced;
process BGtimer1 referenced;
connect sch1, lch1 and a1;
```

```
  connect ch1 and I1, B1;

endblock node1;

/*
 * Dummy Prozess, der von der CNI kommende Signale abfängt
 * Dies triit niemals auf, da die Applikation keine Anfrage an die CNI stellt, und
 * somit keine Kommunikation stattfindet. Auch die Signale von Last-
 * und Stoerquelle werden verworfen.
 */
process application1;
  dcl
    data busdata;

  start;
      nextstate wait;

  state wait;
    input ret_data(data);
      nextstate -;
  endstate;
endprocess application1;

/*
 * CNI
 * Die CNI bildet die Schnittstelle zwischen Applikation und SRU. Sie ist hier
 * nur fuer den Datenaustausch verantwortlich. Um Daten aus der CNI auszulesen,
 * muss der jeweilige Prozess (Controller oder Applikation) eine Anfrage
 * (get_mydata oder get_data (slotnr)) stellen, die mit dem geforderten Datum
 * (ret_mydata (data) oder ret_data (data)) beantwortet wird. Müssen Daten
 * geschrieben werden, werden diese mit put_mydata (data) oder put_busdata
 * (slot, data) an die CNI uebergeben.
 * Dieser Vorgang ist nicht Teil des TTP/C-Protokolls!
 * Er wurde gewaehlt, da SDL keine ausreichende Möglichkeit bietet,
```

```
 * "shared memory" zwischen Prozessen zu verwalten. Die Möglichkeit remote auf
 * Daten anderer Prozesse zuzugreifen, hat sich als problematisch erwiesen.
 */
process CNI1;
  dcl
    CNIData CNIMem,
    slot_nr CNIMemidx,
    data BusData;

  start;
      task CNIData := (. 0 .);
      nextstate wait;

  state wait;
    input get_data(slot_nr);
      output ret_data(CNIData(slot_nr));
      nextstate -;
    input get_mydata;
      output ret_mydata(CNIData(mySlot));
      nextstate -;
    input put_data(data);
      task CNIData(MySlot) := data;
      nextstate -;
    input put_busdata(slot_nr, data);
    task CNIData(slot_nr) := data;
    nextstate -;
  endstate;
endprocess CNI1;


/*
 * Der Controller nimmt vom Bus gesendete Daten und sendet sie mit der
 * Nummer des aktuellen Slots an die CNI weiter. Falls der eigene Slot
 * erreicht ist, müssen die eigenen Daten aus der CNI geholt und an den
```

```
 * BusGuardian gesendet werden.
 * Um die Daten dem richtigen Slot zuzuordnen, muessen die Slots mitgezaehlt
 * werden. Der Beginn eines Slots wird vom Timer mit dem Signal next_slot
 * signalisiert.
 */
process Controller1;
  dcl
    data BusData,
    Slot Integer;

  start;
      task slot := 0;
      nextstate wait_for_slot;

  /* Warte auf Timersignal (next_slot) oder Daten vom Bus. */
  state wait_for_slot;
    /* das Signal des timers hat Vorrang vor allen anderen, um eine bestimmte
     * Reihenfolge zu gewaehrleisten.
     */
    priority input next_slot;
      decision slot = myslot;          /* Daten von der CNI holen, wenn
                                       eigener Slot beginnt */
      (true) :
        output get_mydata;
        nextstate wait_for_data;
      (false) :
      enddecision;
      grst1 :
      /* Slots mitzaehlen. */
      decision slot = TDMARoundLenght - 1;
      (false) :
        task slot := slot + 1;
      (true) :
        task slot := 0;
```

```
      enddecision;
      grst2 :
      nextstate wait_for_slot;
    input ph_n_frame(data);
    /* Daten vom Bus eingetroffen: Speichern in CNI */
    /* Achtung: Da der Bus die Daten an alle Knoten verschickt, werden die vom
     * Knoten gesendeten Daten auch an diesen wieder zurückgeschickt und in
     * die CNI geschrieben.
     */
      output put_busdata(Slot, data);
      join grst2;
  endstate;

/* Warten auf die Daten von der CNI. Diese werden dann an den BusGuardian
 * gesendet.
 */
  state wait_for_data;
    input ret_mydata(data);
      output ph_n_frame(data) via R1;
      join grst1;
  endstate;
endprocess Controller1;



/* BusGuardian
 * Dieser Prozess wird von einem eigenen Timer (bgtick) getaktet. Er verwaltet
 * drei Zustaende: Fenster geschlossen, geoeffnet und verschlossen. Innerhalb
 * des Zeitraumes des eigenen Slots werden ph_n_frame Signale vom Controller
 * an den Bus gesendet. Wird ein ph_n_frame-Signal im Zustand w_closed
 * empfangen, wechselt der Prozess zu w_locked, in dem kein Datum mehr
 * versendet wird.
 * Achtung: Die Zustaende werden durch boolsche Variablen repraesentiert! (to
 * be fixed).
 */
```

```
process BusGuardian1;
 synonym StartTick INTEGER = MySlot * SlotLenghtInMicroticks,
  EndTick INTEGER = StartTick + SlotLenghtInMicroticks,
  MaxTick INTEGER = TDMARoundLenght * SlotLenghtInMicroticks - 1;
 dcl
  ticks INTEGER,
  w_open BOOLEAN,
  w_locked BOOLEAN,
  data busdata;

 start;
    task ticks := 0,
      w_open := false,
      w_locked := false;
    nextstate wait;

 state wait;
  priority input bgtick;
    decision ticks;
    (StartTick : EndTick) :
      task w_open := true;
    else :
      task w_open := false;
    enddecision;
    task ticks := ticks + 1;
    decision ticks = MaxTick + 1;
    (true) :
      task ticks := 0;
    (false) :
    enddecision;
    nextstate -;
  input ph_n_frame(data);
    decision w_locked;
    (false) :
```

```
        decision w_open;
        (true) :
          output ph_n_frame(data) via B1;
        (false) :
          task w_locked := true;
        enddecision;
      (true) :
      enddecision;
      nextstate -;
  endstate;
endprocess BusGuardian1;


/*
 * Timer verschickt die Signale Microtick und next_slot. Microtick ist in
 * dieser Implementation eine SDL-Zeiteinheit. next_slot ist eine Slotlaenge,
 * gemessen in Microticks.
 * Empfaenger ist der Controller.
 */
process timer1;
  dcl
    Ticks INTEGER;
  timer
    MicroTickTimer;

  start;
      output next_slot;
      task ticks := 0;
      set(now + 1, MicroTickTimer);
      nextstate next;

  state next;
    input MicroTickTimer;
      task Ticks := Ticks + 1;
```

```
      decision Ticks = SlotLenghtInMicroticks;
      (true) :
        task ticks := 0;
        output next_slot;
      (false) :
        output microtick;
    enddecision;
    set(now + 1, microticktimer);
    nextstate -;
  endstate;
endprocess timer1;



/*
 * BGtimer erzeugt die Zeitbasis fuer den Busguardian in Microticks. Die
 * Einheit ist eine SDL-Zeiteinheit, weshalb es zwischen Controller und
 * BusGuardian in dieser Implementierung niemals zu Konflikten kommt.
 */
process BGtimer1;
  timer
    T;

  start;
      output bgtick;
      set(now + 1, T);
      nextstate next;

  state next;
    input T;
    output bgtick;
    set(now + 1, T);
    nextstate -;
  endstate;
endprocess BGtimer1;
```

```
block stoerquelle;
signalroute S1
  from stoerung_node1 to env with stoer1;
process stoerung_node1 referenced;
connect sch1 and S1;
endblock stoerquelle;
```

```
process stoerung_node1;
timer
  stoerabstand;
dcl
  stoerwert,
  handle real;

start;
    grst3 :
    task handle := 6;
    task stoerwert := - 0.02 + 0.04 * (handle / 32768);
    output stoer1(stoerwert);
    set(now + 20, stoerabstand);
    nextstate next;

state next;
  input stoerabstand;
    join grst3;
endstate;
endprocess stoerung_node1;
```

```
block node2;
synonym myslot = slot2;
signal
  next_slot,
  microtick,
  bgtick,
  ret_mydata(BusData),
  get_mydata,
  put_busdata(CNIMemIdx, BusData),
  get_data(CNIMemIdx),
  ret_data(BusData),
  put_data(BusData);
signalroute RAPP2
  from application2 to CNI2 with put_data, get_data;
  from CNI2 to application2 with ret_data;
signalroute RCNI2
  from CNI2 to controller2 with ret_mydata;
  from controller2 to CNI2 with get_mydata, put_busdata;
signalroute T2
  from Timer2 to controller2 with next_slot, microtick;
signalroute R2
  from controller2 to BusGuardian2 with ph_n_frame;
signalroute I2
  from env to controller2 with ph_n_frame;
signalroute B2
  from BusGuardian2 to env with ph_n_frame;
signalroute BGT2
  from BGTimer2 to BusGuardian2 with bgtick;
process application2 referenced;
process CNI2 referenced;
process controller2 referenced;
process Timer2 referenced;
process BusGuardian2 referenced;
```

```
process BGTimer2 referenced;
connect ch2 and I2, B2;
endblock node2;



process controller2;
dcl
  data BusData,
  Slot Integer;

start;
    task slot := 0;
    nextstate wait_for_slot;

state wait_for_slot;
  priority input next_slot;
    decision slot = myslot;
    (true) :
      output get_mydata;
      nextstate wait_for_data;
    (false) :
    enddecision;
    grst4 :
    decision slot = TDMARoundLenght - 1;
    (false) :
      task slot := slot + 1;
    (true) :
      task slot := 0;
    enddecision;
    grst5 :
    nextstate wait_for_slot;
  input ph_n_frame(data);
    output put_busdata(Slot, data);
```

```
    join grst5;
endstate;

state wait_for_data;
  input ret_mydata(data);
    output ph_n_frame(data) via R2;
    join grst4;
endstate;
endprocess controller2;



process BusGuardian2;
dcl
  ticks INTEGER,
  w_open BOOLEAN,
  w_locked BOOLEAN,
  data busdata;
synonym StartTick INTEGER = MySlot * SlotLenghtInMicroticks,
  EndTick INTEGER = StartTick + SlotLenghtInMicroticks,
  MaxTick INTEGER = TDMARoundLenght * SlotLenghtInMicroticks - 1;

start;
    task ticks := 0,
      w_open := false,
      w_locked := false;
    nextstate wait;

state wait;
  priority input bgtick;
    decision ticks;
    (StartTick : EndTick) :
      task w_open := true;
    else :
```

```
      task w_open := false;
    enddecision;
    task ticks := ticks + 1;
    decision ticks = maxtick + 1;
    (true) :
      task ticks := 0;
    (false) :
    enddecision;
    nextstate -;
  input ph_n_frame(data);
    decision w_locked;
    (false) :
      decision w_open;
      (true) :
        output ph_n_frame(data) via B2;
      (false) :
        task w_locked := true;
      enddecision;
    (true) :
    enddecision;
    nextstate -;
endstate;
endprocess BusGuardian2;



process application2;
dcl
  data busdata;

start;
    nextstate wait;

state wait;
```

```
  input ret_data(data);
    nextstate -;
endstate;
endprocess application2;



process CNI2;
dcl
  CNIData CNIMem,
  slot_nr CNIMemidx,
  data BusData;

start;
    task CNIData := (. 0 .);
    nextstate wait;

state wait;
  input get_data(slot_nr);
    output ret_data(CNIData(slot_nr));
    nextstate -;
  input get_mydata;
    output ret_mydata(CNIData(mySlot));
    nextstate -;
  input put_data(data);
    task CNIData(MySlot) := data;
    nextstate -;
  input put_busdata(slot_nr, data);
    task CNIData(slot_nr) := data;
    nextstate -;
endstate;
endprocess CNI2;
```

```
process Timer2;
dcl
  Ticks INTEGER;
timer
  MicroTickTimer;

start;
    output next_slot;
    task ticks := 0;
    set(now + 1, MicroTickTimer);
    nextstate next;

state next;
  input MicroTickTimer;
    task Ticks := Ticks + 1;
    decision Ticks = SlotLenghtInMicroticks;
    (true) :
      task ticks := 0;
      output next_slot;
    (false) :
      output microtick;
    enddecision;
    set(now + 1, microticktimer);
    nextstate -;
endstate;
endprocess Timer2;



process BGTimer2;
timer
  T;
```

```
start;
    output bgtick;
    set(now + 1, T);
    nextstate next;

state next;
  input T;
    output bgtick;
    set(now + 1, T);
    nextstate -;
endstate;
endprocess BGTimer2;



block bus;
signalroute RI
  from env to bus1 with ph_n_frame;
  from bus1 to env with ph_n_frame;
process bus1 referenced;
connect ch1, ch2, ch3, ch4, ch5 and RI;
endblock bus;



process bus1;
dcl
  data busdata;

start;
    nextstate wait;

state wait;
  input ph_n_frame(data);
```

```
    output ph_n_frame(data) via ch1;
    output ph_n_frame(data) via ch2;
    output ph_n_frame(data) via ch3;
    output ph_n_frame(data) via ch4;
    output ph_n_frame(data) via ch5;
    nextstate -;
endstate;
endprocess bus1;



block node3;
synonym myslot = slot3;
signal
  next_slot,
  microtick,
  bgtick,
  ret_mydata(BusData),
  get_mydata,
  put_busdata(CNIMemIdx, BusData),
  get_data(CNIMemIdx),
  ret_data(BusData),
  put_data(BusData);
signalroute RAPP3
  from application3 to CNI3 with put_data, get_data;
  from CNI3 to application3 with ret_data;
signalroute RCNI3
  from CNI3 to controller3 with ret_mydata;
  from controller3 to CNI3 with get_mydata, put_busdata;
signalroute T3
  from Timer3 to controller3 with next_slot, microtick;
signalroute R3
  from controller3 to BusGuardian3 with ph_n_frame;
signalroute I3
```

```
  from env to controller3 with ph_n_frame;
signalroute B3
  from BusGuardian3 to env with ph_n_frame;
signalroute BGT3
  from BGTimer3 to BusGuardian3 with bgtick;
process application3 referenced;
process CNI3 referenced;
process controller3 referenced;
process Timer3 referenced;
process BusGuardian3 referenced;
process BGTimer3 referenced;
connect ch3 and I3, B3;
endblock node3;



process controller3;
dcl
  data BusData,
  Slot Integer;

start;
    task slot := 0;
    nextstate wait_for_slot;

state wait_for_slot;
  priority input next_slot;
    decision slot = myslot;
    (true) :
      output get_mydata;
      nextstate wait_for_data;
    (false) :
    enddecision;
    grst6 :
```

```
    decision slot = TDMARoundLenght - 1;
    (false) :
      task slot := slot + 1;
    (true) :
      task slot := 0;
    enddecision;
    grst7 :
    nextstate wait_for_slot;
  input ph_n_frame(data);
    output put_busdata(Slot, data);
    join grst7;
endstate;

state wait_for_data;
  input ret_mydata(data);
    output ph_n_frame(data) via R3;
    join grst6;
endstate;
endprocess controller3;



process BusGuardian3;
dcl
  ticks INTEGER,
  w_open BOOLEAN,
  w_locked BOOLEAN,
  data busdata;
synonym StartTick INTEGER = MySlot * SlotLenghtInMicroticks,
  EndTick INTEGER = StartTick + SlotLenghtInMicroticks,
  MaxTick INTEGER = TDMARoundLenght * SlotLenghtInMicroticks - 1;

start;
    task ticks := 0,
```

```
      w_open := false,
      w_locked := false;
    nextstate wait;

state wait;
  priority input bgtick;
    decision ticks;
    (StartTick : EndTick) :
      task w_open := true;
    else :
      task w_open := false;
    enddecision;
    task ticks := ticks + 1;
    decision ticks = maxtick + 1;
    (true) :
      task ticks := 0;
    (false) :
    enddecision;
    nextstate -;
  input ph_n_frame(data);
    decision w_locked;
    (false) :
      decision w_open;
      (true) :
        output ph_n_frame(data) via B3;
      (false) :
        task w_locked := true;
      enddecision;
    (true) :
    enddecision;
    nextstate -;
endstate;
endprocess BusGuardian3;
```

```
process application3;
dcl
  data busdata;

start;
    nextstate wait;

state wait;
  input ret_data(data);
    nextstate -;
endstate;
endprocess application3;



process CNI3;
dcl
  CNIData CNIMem,
  slot_nr CNIMemidx,
  data BusData;

start;
    task CNIData := (. 0 .);
    nextstate wait;

state wait;
  input get_data(slot_nr);
    output ret_data(CNIData(slot_nr));
    nextstate -;
  input get_mydata;
    output ret_mydata(CNIData(mySlot));
    nextstate -;
```

```
  input put_data(data);
    task CNIData(MySlot) := data;
    nextstate -;
  input put_busdata(slot_nr, data);
    task CNIData(slot_nr) := data;
    nextstate -;
endstate;
endprocess CNI3;



process Timer3;
dcl
  Ticks INTEGER;
timer
  MicroTickTimer;

start;
    output next_slot;
    task ticks := 0;
    set(now + 1, MicroTickTimer);
    nextstate next;

state next;
  input MicroTickTimer;
    task Ticks := Ticks + 1;
    decision Ticks = SlotLenghtInMicroticks;
    (true) :
      task ticks := 0;
      output next_slot;
    (false) :
      output microtick;
    enddecision;
    set(now + 1, microticktimer);
```

```
    nextstate -;
endstate;
endprocess Timer3;



process BGTimer3;
timer
  T;

start;
    output bgtick;
    set(now + 1, T);
    nextstate next;

state next;
  input T;
    output bgtick;
    set(now + 1, T);
    nextstate -;
endstate;
endprocess BGTimer3;



block node4;
synonym myslot = slot4;
signal
  next_slot,
  microtick,
  bgtick,
  ret_mydata(BusData),
  get_mydata,
  put_busdata(CNIMemIdx, BusData),
```

```
  get_data(CNIMemIdx),
  ret_data(BusData),
  put_data(BusData);
signalroute RAPP4
  from application4 to CNI4 with put_data, get_data;
  from CNI4 to application4 with ret_data;
signalroute RCNI4
  from CNI4 to controller4 with ret_mydata;
  from controller4 to CNI4 with get_mydata, put_busdata;
signalroute T4
  from Timer4 to controller4 with next_slot, microtick;
signalroute R4
  from controller4 to BusGuardian4 with ph_n_frame;
signalroute I4
  from env to controller4 with ph_n_frame;
signalroute B4
  from BusGuardian4 to env with ph_n_frame;
signalroute BGT4
  from BGTimer4 to BusGuardian4 with bgtick;
process application4 referenced;
process CNI4 referenced;
process controller4 referenced;
process Timer4 referenced;
process BusGuardian4 referenced;
process BGTimer4 referenced;
connect ch4 and I4, B4;
endblock node4;



process controller4;
dcl
  data BusData,
  Slot Integer;
```

```
start;
    task slot := 0;
    nextstate wait_for_slot;

state wait_for_slot;
  priority input next_slot;
    decision slot = myslot;
    (true) :
      output get_mydata;
      nextstate wait_for_data;
    (false) :
    enddecision;
    grst8 :
    decision slot = TDMARoundLenght - 1;
    (false) :
      task slot := slot + 1;
    (true) :
      task slot := 0;
    enddecision;
    grst9 :
    nextstate wait_for_slot;
  input ph_n_frame(data);
    output put_busdata(Slot, data);
    join grst9;
endstate;

state wait_for_data;
  input ret_mydata(data);
    output ph_n_frame(data) via R4;
    join grst8;
endstate;
endprocess controller4;
```

```
process BusGuardian4;
dcl
  ticks INTEGER,
  w_open BOOLEAN,
  w_locked BOOLEAN,
  data busdata;
synonym StartTick INTEGER = MySlot * SlotLenghtInMicroticks,
  EndTick INTEGER = StartTick + SlotLenghtInMicroticks,
  MaxTick INTEGER = TDMARoundLenght * SlotLenghtInMicroticks - 1;

start;
    task ticks := 0,
      w_open := false,
      w_locked := false;
    nextstate wait;

state wait;
  priority input bgtick;
    decision ticks;
    (StartTick : EndTick) :
      task w_open := true;
    else :
      task w_open := false;
    enddecision;
    task ticks := ticks + 1;
    decision ticks = maxtick + 1;
    (true) :
      task ticks := 0;
    (false) :
    enddecision;
    nextstate -;
  input ph_n_frame(data);
```

```
    decision w_locked;
    (false) :
      decision w_open;
      (true) :
        output ph_n_frame(data) via B4;
      (false) :
        task w_locked := true;
      enddecision;
    (true) :
    enddecision;
    nextstate -;
endstate;
endprocess BusGuardian4;



process application4;
dcl
  data busdata;

start;
    nextstate wait;

state wait;
  input ret_data(data);
    nextstate -;
endstate;
endprocess application4;



process CNI4;
dcl
  CNIData CNIMem,
```

```
  slot_nr CNIMemidx,
  data BusData;

start;
    task CNIData := (. 0 .);
    nextstate wait;

state wait;
  input get_data(slot_nr);
    output ret_data(CNIData(slot_nr));
    nextstate -;
  input get_mydata;
    output ret_mydata(CNIData(mySlot));
    nextstate -;
  input put_data(data);
    task CNIData(MySlot) := data;
    nextstate -;
  input put_busdata(slot_nr, data);
    task CNIData(slot_nr) := data;
    nextstate -;
endstate;
endprocess CNI4;



process Timer4;
dcl
  Ticks INTEGER;
timer
  MicroTickTimer;

start;
    output next_slot;
    task ticks := 0;
```

```
    set(now + 1, MicroTickTimer);
    nextstate next;

state next;
  input MicroTickTimer;
    task Ticks := Ticks + 1;
    decision Ticks = SlotLenghtInMicroticks;
    (true) :
      task ticks := 0;
      output next_slot;
    (false) :
      output microtick;
    enddecision;
    set(now + 1, microticktimer);
    nextstate -;
endstate;
endprocess Timer4;



process BGTimer4;
timer
  T;

start;
    output bgtick;
    set(now + 1, T);
    nextstate next;

state next;
  input T;
    output bgtick;
    set(now + 1, T);
    nextstate -;
```

```
endstate;
endprocess BGTimer4;
```