T U M

I N S T I T U T F Ü R I N F O R M A T I K

FlexRay und FTCom: Formale Spezifikation in FOCUS

Christian Kühnel, Maria Spichkova

TUM-I0601
Februar 06

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-02-I0601-0/2.-FI

Druck: Institut für Informatik der
Technischen Universität München

FlexRay und FTCom: Formale Spezifikation in Focus *

Christian Kühnel and Maria Spichkova

24. März 2006

Zusammenfassung

Im Projekt Verisoft (Teilprojekt Automotive [2]) wurde das Kommunikationsprotokoll FlexRay abstrakt formalisiert und ausgewählte Eigenschaften verifiziert. Die auf OSEKtime FTCom basierten Kommunikationskonzepte wurden ebenfalls formalisiert, in Hinblick auf die Schnittstelle zum FlexRay-Bussystem. In diesem Technischen Bericht werden nun die verwendeten Formalisierungen beschrieben. Da die hier beschriebene Formalisierung in einer denotationellen Semantik vorliegt, kann sie als Grundlage für die Korrektheitsbeweise einer Implementierung verwendet werden.

*Diese Arbeit wurde von vom Bundesministerium für Bildung und Forschung (bmb+f) innerhalb des Projekts Verisoft [8] gefördert.

Inhaltsverzeichnis

1 Einleitung

Dieser Technischer Bericht ist auf die Formalisierung von FlexRay [1] und OSEKtime FTCom [6] ausgerichtet und soll als Ausgangspunkt für die Implementierung von FlexRay und OSEKtime FTCom dienen. Die Formalisierung von FTCom wurde in Hinblick auf die Schnittstelle zum FlexRay-Bussystem vorgenommen.

1.1 Focus

Als Beschreibungssprache für die Formalisierung wurde die Spezifikationssprache FOCUS gewählt, die am Lehrstuhl von Prof. Broy an der TU München entwickelt wurde. FOCUS eignet sich im vorliegenden Fall besonders, da es eine einfache Modellierung von zeitsynchronen, nebenläufigen Komponenten ermöglicht. Im folgenden wird kurz die hier verwendete Teilmenge von FOCUS eingeführt. Eine ausführliche Beschreibung findet man in [3].

FOCUS erlaubt die Spezifikation von nebenläufigen Komponenten die über getypte Nachrichten auf gezeiteten Ströme kommunizieren. Der hier gewählte Dialekt verwendet eine denotationelle Semantik, da diese der Beschreibung von zu verifizierenden Eigenschaften sehr nahe kommt. Die Zeit ist in diesem Falle global synchron und wird in "Ticks" angegeben.

Die syntaktischen Schnittstellen zwischen Komponenten werden in *glass-box* Diagrammen und den jeweiligen Komponenten beschrieben. Das Verhalten in der jeweiligen Komponente selbst.

Die Spezifikation einer Komponente zerfällt in drei Teile: die Ein- und Ausgabekanäle, die Annahmen *asm* (assumption) und die Zusicherungen *gar* (guarantee). Bei den Ein- und Ausgabekanälen werden deren Namen und Typen aufgelistet. Die Annahmen beschreiben Eigenschaften, die die Eingaben der Komponente erfüllen müssen, damit diese korrekt funktioniert. Die Zusicherungen beschreiben, welche Zusammenhänge zwischen Eingängen und Ausgängen bestehen und welche Bedingungen auf an Ausgängen gelten. Die Annahmen und Zusicherungen sind prädikatenlogische Ausdrücke.

Verwendete Formeln

Die in dieser Spezifikation verwendeten Formeln, sind größtenteils aus der Logik bekannt. Jedoch gibt es einige zusätzliche Symbole, die mit der Verarbeitung von Strömen zusammenhängen:

- Ein leerer Strom wird mit $\langle\rangle$ dargestellt, ein einelementiger mit $\langle x\rangle$.
- $\#x$ ist die Länge des Stroms x.
- Der Filteroperator $x \circledS s$ liefert den Strom aller Nachrichten aus s, die x erfüllen.
- $s_1 \frown s_2$ ist die Konkatenierung der Ströme s_1 und s_2.
- $m \,\&\, s$ fügt das Element m am Anfang des Stromes s ein, $m \,\&\, s = \langle m\rangle \frown s$.
- $\overline{s}$ ist der Strom s, aus dem alle Ticks entfernt wurden.
- $\mathsf{ft}.s$ liefert das erste Element des Stromes s.
- $\mathsf{rt}.s$ liefert den Stromes s ohne das ertste Element.
- $s.n$ liefert das n-te Element des Stromes s.
- $\mathsf{dom}.s$ liefert die Liste $[1...\#s]$.

- rng.s wandelt den Strom S in die Menge seiner Elemente um, also $\{s.j \mid j \in \mathsf{dom}.s\}$.

Mit der Formel ti(s, n) werden wir die Nachrichten bezeichnen, die auf dem Kanal s zwischen den Ticks $n-1$ und n vorhanden sind. Oder mathematisch ausgedrückt:

$$\mathsf{ti}(s,n) \stackrel{def}{=} \begin{cases} \langle\rangle & \textit{if} \quad n = 0 \\ \overline{s\downarrow_1} & \textit{if} \quad n = 1 \\ \overline{r} & \textit{otherwise}, \quad \textit{where} \quad s\downarrow_n = r \frown s\downarrow_{n-1} \end{cases}$$

Wir definieren einen neuen Operator $\mathsf{disjunct}(s_1, \ldots, s_n)$ als ein Prädikat, das für eine Menge von Ströme $s_1 \in M_1^{\underline{\omega}}, \ldots, s_n \in M_n^{\underline{\omega}}$ gilt, wenn zu jedem Zeitpunkt nur in einem der Ströme Nachrichten übertragen werden:

$$\mathsf{disjunct} \in M_1^{\underline{\omega}} \times \cdots \times M_n^{\underline{\omega}} \to \mathbb{B}$$
$$\mathsf{disjunct}(s_1, \ldots, s_n) \stackrel{def}{=} \forall\, t \in \mathbb{N} : \mid \{k : k \in [1...n] \wedge \mathsf{ti}(s_k, t) \neq \langle\rangle\} \mid \leq 1$$

Wir definieren einen Operator $\mathsf{maxmsg}_n(s)$ als ein Prädikat, das für einen Strom s gilt, wenn der Strom in jedem Zeitintervall höchstens n Nachrichten enthält:

$$\mathsf{maxmsg} \in \mathbb{N} \times M^{\underline{\omega}} \to \mathbb{B}$$
$$\mathsf{maxmsg}_n(s) \stackrel{def}{=} \forall\, t \in \mathbb{N}.\ \#\mathsf{ti}(s,t) \leq n$$

Wir führen hier eine neue Variante der FOCUS-Tabellen – tiTable – ein um die Zusicherungen für einige Komponenten besser darstellen zu können. tiTable unterscheidet sich von der klassichen FOCUS-Tabelle wie folgt:

- In einer klassichen FOCUS-Tabelle betrachten wir die Ströme *elementweise*.
- In einer tiTable betrachten wir die Ströme *intervalweise* – für die Kanäle alle mögliche Kombinationen von Nachrichten, die zwischen Ticks $t-1$ und t (im Zeitinterval t) vorkommen. Optional können wir auch eine *Assumption*-Spalte benutzen, wenn wir zusätzliche Annahmen für die Beschreibung der Kombinationen brauchen (vgl. Abschnitt 3.2). Für bessere Lesbarkeit, werden wir in der tiTable zwischen folgenden Gruppen „Doppellinientrennung“ benutzen: Eingabeströme, Ausgabeströme, lokale Variablen und zusätzliche Annahmen.

1.2 Architektur des Gesamtsystems

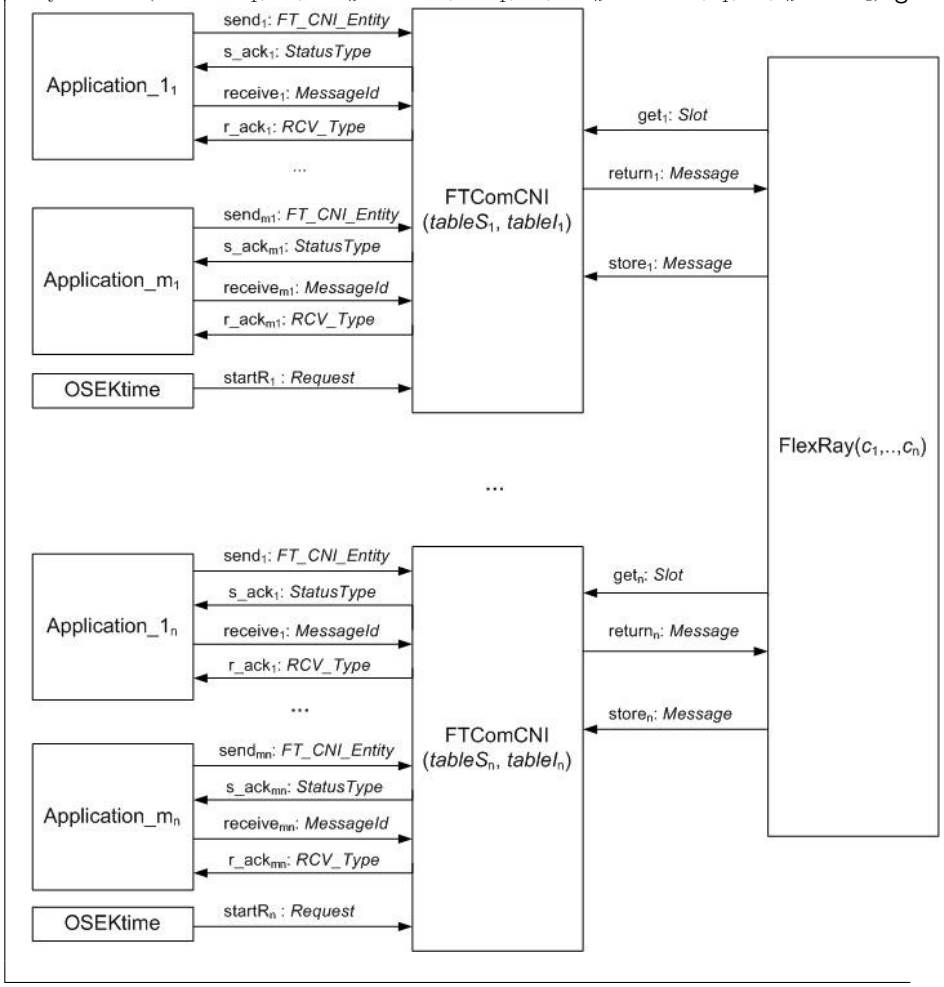
Die Architektur des Gesamtsystems konnen wir wie in der FOCUS-Spezifikation *SystemArch* darstellen.

Wir abstrahieren hier von der vollen Darstellung des Betriebssystems (OSEKtime OS) und der Anwendung-Komponenten, als auch von der Kommunikation zwischen diesen Komponenten, um die Aspekte des Nachrichtenaustauschs im System darzustellen.

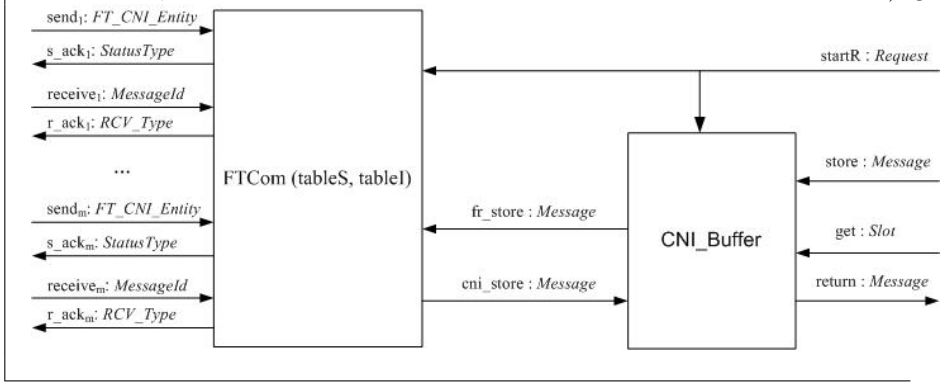
Die Komponente *FTComCNI* besteht aus einer FTCom–Komponente und einem CNI–Puffer. CNI ist eine Abkürzung von „Communication Network Interface“. CNI ist kein Teil von FTCom, gehört aber zum Gesamtsystem (vgl. Abb. 1 und [6]).

Im CNI–Puffer werden alle Nachrichten gespeichert, die redundant durch FlexRay verschickt wurden. Der Task, der diese Nachrichten mit einem RDA–Algorithmus (vgl. Abschnitt 1.2) bearbeitet und im FTCom–Puffer speichert wird in unserem Modell ein Mal pro FlexRay Runde aufgerufen (n Mal pro OSEKtime Dispatcher Runde, $1 \leq n$).

SystemArch(constant $tableS_1, \ldots, tableS_n \in Frames^*$; $tableI_1, \ldots, tableI_n \in Id2Slots^*$; $c_1, \ldots, c_n \in$ Config): glass-box



FTComCNI(constant $tableS_1, \ldots, tableS_n \in Frames^*$; $tableI_1, \ldots, tableI_n \in Id2Slots^*$): glass-box



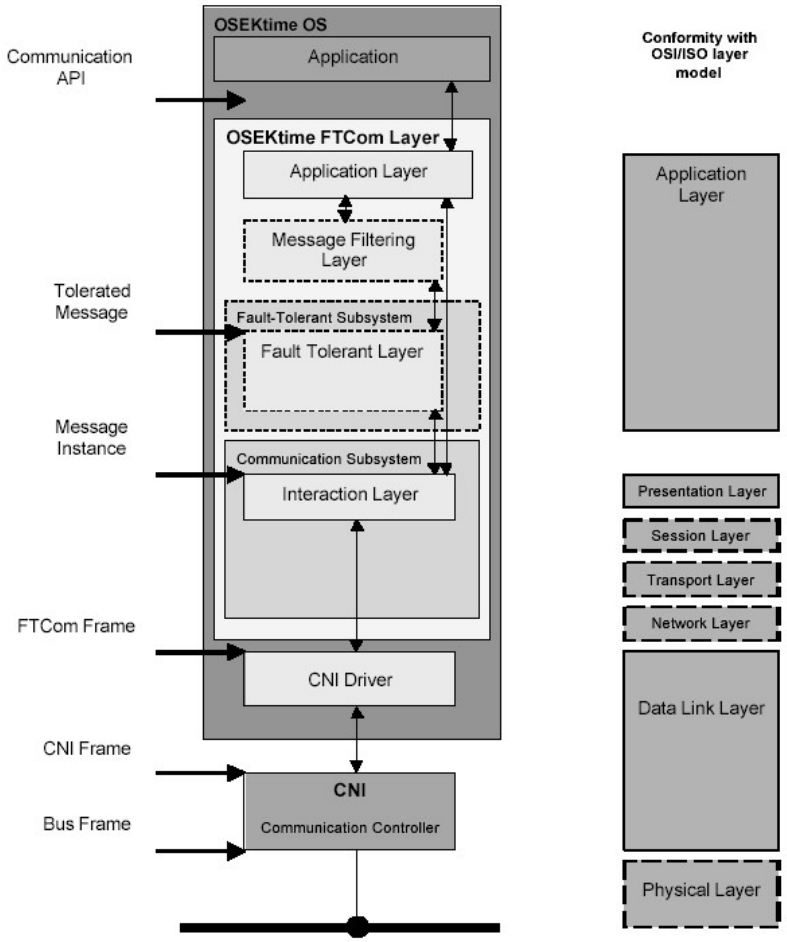


Abbildung 1: Model von OSEKtime FTCom Architektur (vom [6])

1.3 FlexRay

FlexRay ist ein zeitgesteuertes Kommunikationsprotokoll, das vom FlexRay-Konsortium [4] entwickelt wird. Die hier beschriebene Formaliserung basiert auf der "Protocol Specification 2.0" [1]. Diese Spezifikation liegt in der Sprache SDL vor und ist relativ implementierungsnah. Für die Verifikation im Teilprojekt Automotive mussen einige Teile abstrahiert werden, um den Rahmen des Projektes nicht zu sprengen. Folgende Abstraktionen wurden getroffen:

- Es findet keine Zeit-Synchronisation statt, synchrone Uhren werden angenommen.
- Das Start-up Verhalten wurde nicht modelliert, da die Uhren als synchron angenommen werden.
- Von der Bit-Codierung der Nachrichten und deren Abbildung auf Frames wurde abstrahiert.
- Es wurde kein Bus Guardian modelliert.

- Da in diesem Zusammenhang die Echtzeitfähigkeiten von FlexRay im Fordergrund stehen, wurde nur das "static segment" modelliert, jedoch nicht das "dynamic segment".
- Als Zeitbasis wurde 1 Slot = 1 Tick gewählt.
- Es wurde kein Fehlerverhalten modelliert. Entsprechend kommen alle verschickten Nachrichten ohne Fehler bei allen Empfängern an.
- Es wurde nur ein FlexRay Kanal modelliert.

1.4 FTCom

Die fehlertolerante Kommunikationssicht FTCom des Echtzeit-Betriebsystems OSEKtime [7] stellt sowohl die Funktionen bereit, die für die Kommunikation zwischen den Anwendungen (auf einem Knoten) benutzt werden, als auch die Funktionen für Zeitsynchronisation im Gesamtsystem und auch die Funktionen für die Fehlertolerz der Kommunikation zwischen den Knoten dienen. Die hier beschriebene Formaliserung basiert auf der natürlichsprachilichen Spezifikation „Fault-Tolerant Communication - Specification 1.0“ [6].

Da FlexRay mit einer hohen Wahrscheinlichkeit[1] den Fall entdeckt, dass die Nachrichten (FlexRay-Frames) wärend FlexRay-Sendens/Empfangens verfälscht wurden, wird jede verfälschte Nachricht mit der hohen Wahrscheinlichkeit gelöscht – wenn der Frame mit dem Slot-Identifikator i wärend FlexRay-Sendens/Empfangens verfälscht wurde, wird im CNI $msg(i, \langle\rangle)$ gespeichert.

Wenn diese Fehlerwahrscheinlichkeit für unseres System klein genug ist, nehmen wir einfach einfach die erste entsprechende Nachricht (mit entsprechenden Identifikator) von der Nachrichten, die durch FlexRay übertragen wurden.

Wenn wir aber in unserem System noch geringere Fehlerwahrscheinlichkeit brauchen, sollen wir den „Mehrheitsbeschluss“ zu benutzen (entsprechende Funktion *RDA_MV*, vgl. 3.1), da nicht alle Nachrichten in unserer Spezifikation numerische Werte haben, sondern die Nachrichten sind von unterschiedlichen Typen, die unterschiedliche Struktur haben.

Für die Formalisierung und Implementierung wurden einige Konzepte von FTCom abstrahiert, da diese für das gegenwärtige Gesamtsystemmodell nicht benötigt werden. Es wurden folgende Abstraktionen getroffen:

- Die Konstante *TT_E_FTCOM_FAILED* ist eine Abstraktion von mehreren Fehlersituationen, die FTCom behandeln kann: Wir vereinfachen den Typ *ttStatusType* bis auf drei Konstanten, da wir in unserem Gesamtsystemmodell nicht alle mögliche Konstanten von der FTCom-Spezifikation brauchen. Wir benutzen die Konstante *FTCOM_OK* und *FTCOM_FAILED* um den Status von FTCom zu beschreiben. Die Konstante *OS_DEADLINE* beschreibt den OSEKtime OS-Zustand, in dem eine Deadline-Überschreitung aufgetreten ist und wird bei FTCom nicht benutzt.

$$\begin{aligned} \mathsf{type}\ \mathit{StatusType} \quad &= \quad \mathit{FTCOM\_OK} \\ &\mid \quad \mathit{FTCOM\_FAILED} \\ &\mid \quad \mathit{OS\_DEADLINE}; \end{aligned}$$

[1] Die FlexRay-Frames enthalten außer den Daten vom Host auch den CRC (Cyclic Redundant Check).Stimmt der empfangene CRC mit dem im Empfänger berechneten überein, ist die Wahrscheinlichkeit der fehlerfreien Frame-Übertragung sehr hoch. In FlexRay wird jede beliebige Kombination der Veränderung von bis zu sechs Bits je Frame erkannt. Auch viele Fälle, bei denen sieben oder mehr Bits verändert wurden, werden erkannt, doch einige wenige Kombinationen können ab sieben geänderten Bits nicht mehr auf Protokollebene erkannt werden.

- *ttSyncTimes* bezeichnet in FTCom eine Funktion, die die lokale Zeit mit der globalen synchronisiert und die aktuellen Werte der lokalen und globalen Zeit, sowie die Bestätigung, ob die Synchronisation erfolgreich war, zurückgibt.

  Da wir in unserem Gesamtsystemmodell die synchrone Zeit von FlexRay zur Verfügung gestellt wird, benutzen wir die FTCom-Funktion *ttSyncTimes* nicht. Diese wurde gegenwärtig wie folgt modelliert: der Wert der lokalen Zeit auf den Knoten wird durch den Wert der globalen Zeit (synchrone Zeit, die von FlexRay zur Verfügung gestellt wird) ersetzt.

- Da wir auf die Benutzung globaler Variablen verzichten[2], müssen die Parameter der Funktionen *ttSendMessage* und *ttReceiveMessage* (gegenüber der FTCom-Spezifikation) in der Isabelle-Darstellung um den Typ *FT_CNI_Entity_List* und die Parameter der Funktion *ttSyncTimes* um den Typ *Time* (die Werte der lokalen und globalen Zeit) erweitert werden.

- Wir benützen eine vereinfachte Version der Signaturen der Funktionen *ttSendMessage* und *ttReceiveMessage*. Die Original-Signaturen sehen wie folgt aus (für entsprechende Modell in Focus vgl. 3.6):

  ttSendMessage (<Message>: ttMsgIdType,
  <Data>: ttAccessNameRefType): ttStatusType

  ttReceiveMessage (<Message>: ttMsgIdType,
  <Data>: ttAccessNameRefType): ttStatusType

  wobei *ttMsgType* der Typ der Nachrichtenidentifikatoren ist. Der Typ *ttAccessNameRefType* definiert die Referenz auf eine Variable vom Typ für Referenzen auf den Nachrichten-Inhalt.

2 Formalisierung von FlexRay

2.1 Typen

Der Typ *Slot* beschreibt einen Zeitschlitz innerhalb eines Zyklusses. Der Nachrichtentyp *Message* besteht aus einem Slot-Identifikator (*slot*) und den Nutzdaten (*data*). Der Datentyp *Data* wurde hier absichlich nicht spezifiziert, dieser kann zu einem späteren Zeitpunkt festegelegt werden, da er für diese Darstellung von FlexRay unerheblich ist. Die Konfiguration des Busses von Typ *Config* enthält die Schedulingtabelle *schedule* eines Knotens sowie die Länge des Kommunikationszyklusses *cycleLength*.

$$
\begin{array}{lcl}
\mathsf{type}\ \mathit{Slot} & = & \mathbb{N} \\
\mathsf{type}\ \mathit{Message} & = & \mathit{msg}(\mathit{slot} : \mathit{Slot}, \mathit{data} : \mathit{Data}) \\
\mathsf{type}\ \mathit{Config} & = & \mathit{conf}(\mathit{schedule} : \mathit{Slot}^*, \mathit{cycleLength} : \mathbb{N})
\end{array}
$$

2.2 FlexRayArchitecture

Die *FlexRayArch* ist eine Verfeinerung von *FlexRay* (siehe unten). Die darin enthaltene *FlexRayArchitecture* besteht aus mehreren *FlexRay-Controllern* und einem

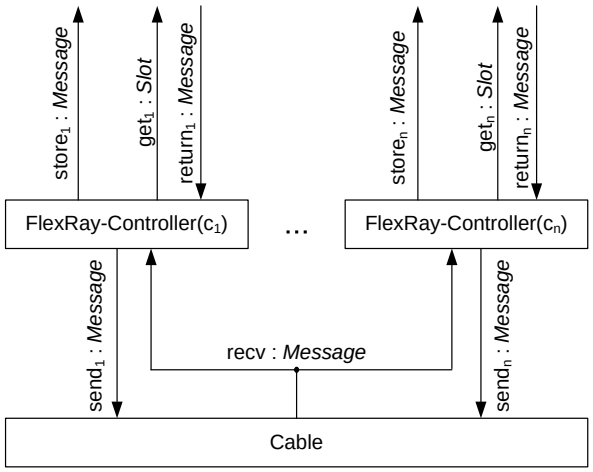
[2]Das Gesamtsystem wurde in Isabelle/HOL modelliert und System - und Komponenteneigenschaften wurden verifiziert. Isabelle [9] ist ein generisches Spezifikations- und Verifikationssystem, das keine globale Variablen unterstützt.

Netzwerkkabel *Cable*. Die nicht verbundenen Kanäle *get*, *return* und *store* der FlexRay-Controller werden mit der darüberliegenden Schicht (z.B. FTCom) des jeweiligen Knotens verbunden und ermöglichen die Kommunikation mit dem *FlexRay-Controller*.

FlexRayArch (constant $c_1, ..., c_n \in$ Config) — timed

in	$return_1, ..., return_n$: *Message*
out	$store_1, ..., store_n$: *Message*; $get_1, ..., get_n$: *Slot*
asm	$\forall\, i \in [1..n]$: $\mathsf{maxmsg}_1(return_i)$
	$DisjointSchedules(c_1, ..., c_n)$
	$IdenticCycleLength(c_1, ..., c_n)$
gar	*FlexRayArchitecture* (constant $c_1, ..., c_n \in$ *Config*)
	$(return_1, ..., return_n, store_1, ..., store_n, get_1, ..., get_n)$

FlexRayArchitecture (constant $c_1, ..., c_n \in$ Config) — glass-box



2.3 FlexRay

Um die allgemeinen Prädikate für FlexRay formulieren zu können ist hier die Komponente FlexRay textuell definiert. Die Spezifikation *FlexRayArchitecture* ist eine Verfeinerung vom Zusicherungsteil der Spezifikation FlexRay.

FlexRay (constant $c_1, ..., c_n \in$ Config) — timed

in	$return_1, ..., return_n : Message$
out	$store_1, ..., store_n : Message;\ get_1, ..., get_n : Slot$

asm	$\forall\, i \in [1..n] : \mathsf{maxmsg}_1(return_i)$
	$DisjointSchedules(c_1, ..., c_n)$
	$IdenticCycleLength(c_1, ..., c_n)$
gar	$MessageTransmission(return_1, ..., return_n, store_1, ..., store_n, get_1, ..., get_n,$ $c_1..., c_n)$
	$\forall\, i \in [1..n] : \mathsf{maxmsg}_1(get_i) \wedge \mathsf{maxmsg}_1(store_i)$

Um zu verhindern, dass zwei Controller zum gleichen Zeitpunkt senden, muss mit der Annahme *disjointSchedules* sichergestellt werden, dass jeder *Slot* nur einmal vergeben ist. Damit kann das Prädikat disjunct zugesichtert werden, das später in der Komponente *Cable* benötigt wird.

DisjointSchedules

$c_1, ..., c_n \in Config$

$\forall\, i, j \in [1..n], j \neq i :$

$\forall\, x \in \mathsf{rng}.schedule(c_i), y \in \mathsf{rng}.schedule(c_j) :$

$x \neq y$

Außerdem müssen die Zykluslängen auf allen Knoten identisch sein:

IdenticCycleLength

$c_1, ..., c_n \in Config$

$\forall\, i, j \in [1..n] :$

$cycleLength(c_i) = cycleLength(c_j)$

Die Garantie *MessageTransmission* stellt die Nachrichtenübertragung sicher. Wenn zum Zeitpunkt t der Knoten k laut dessen Schedule senden sollten, dann holt er über die Kanäle get_k und $return_k$ die zu versendende Nachricht von der übergeordneten Schicht und verschickt die Nachricht. Die restlichen Knoten im Netzwerk empfangen diese und leiten sie über die Kanäle $store_j$ lokal weiter.

MessageTransmission

$store_1, ..., store_n, return_1, ..., return_n \in Message^{\underline{\omega}}$
$get_1, ..., get_n \in Slot^{\underline{\omega}}$
$c_1, ..., c_n \in Config$

$\forall\, t \in \mathbb{N}, k \in [1..n]$:

$s \in schedule(c_k) : s = t \bmod cycleLength(c_k) \rightarrow$

$\mathsf{ti}(get_k, t) = \langle s \rangle \wedge$

$\forall\, j \in [1..n], j \neq k : \mathsf{ti}(store_j, t) = \mathsf{ti}(return_k, t)$

2.4 Cable

Das *Cable* simuliert die Broadcast-Eigentschaften des physikalsichen Netzwerkkabels. Dazu wird jede eintreffende Nachricht an alle angeschlossenen Teilnehmer weitergeleitet.

Cable — timed

in $send_1, ..., send_n : Message$

out $recv : Message$

asm $\mathsf{disjunct}(send_1, ..., send_n)$

gar $\mathrm{Broadcast}(send_1, ..., send_n, recv)$

Als Voraussetzung für die korrekte Funktion muss zu jedem Zeitpunkt die Anzahl der sendenden Knoten kleiner gleich 1 sein. Dies ist gleichbedeutend mit der Annahme dass die Schedules der verschiedenen Knoten disjunkt sind (siehe oben).

Wenn ein *FlexRay-Controller* eine Nachricht verschickt, wird diese an alle anderen Knoten weitergeleitet.

Broadcast

$send_1, ..., send_n, recv \in Message^{\underline{\omega}}$

$\forall\, t \in \mathbb{N}$:

if $\exists\, k \in [1...n] : \mathsf{ti}(send_k, t) \neq \langle\rangle$

then $\mathsf{ti}(recv, t) = \mathsf{ti}(send_k, t)$

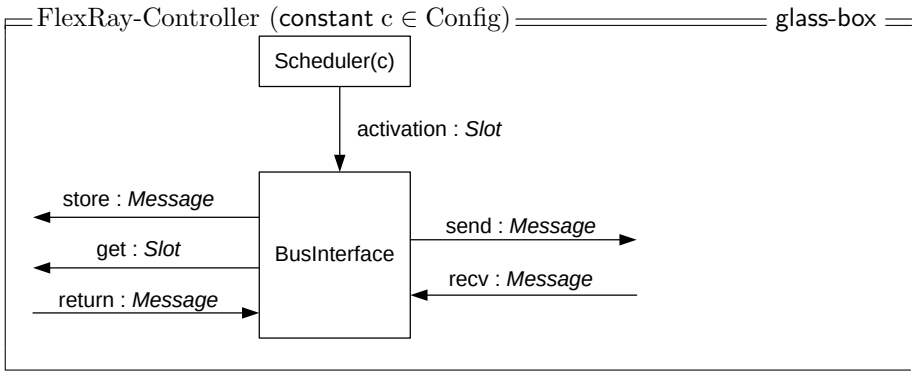
else $\mathsf{ti}(recv, t) = \langle\rangle$

fi

2.5 FlexRay-Controller

Ein *FlexRay-Controller* besteht aus einem *BusInterface* und einem *Scheduler*. Der *Scheduler* signalisiert dem *BusInterface* zu welchen Zeitpunkten eine bestimmte

Nachricht verschickt werden soll. Das *BusInterface* übernimmt die Interaktion mit den anderen Knoten.



2.5.1 Scheduler

Der *Scheduler* übernimmt die zeitliche Steuerung des Nachrichtenversands. Er informiert das *BusInterface* über den Typ einer Nachricht, die verschickt werden soll. Dazu benötigt der *Scheduler* die Konfiguration c als Konstante, sie enthält den *schedule* und die Länge des Zyklusses *cycleLength*. Der *schedule* gibt an zu welchem Zeitpunkt innerhalb des Zyklusses eine Nachricht verschickt werden soll.

Scheduler (constant c ∈ Config) — timed

out $activation : Slot$

univ $slot \in \mathbb{N}$

$\forall\, t \in \mathbb{N}$:

$slot = t \bmod cycleLength(c)$

if $slot \in schedule(c)$

then $\mathsf{ti}(activation, t) = \langle slot \rangle$

else $\mathsf{ti}(activation, t) = \langle\rangle$

fi

2.5.2 BusInterface

Das *BusInterface* kümmert sich um den tatsächlichen Empfang und Versand der Nachrichten. Dazu wird mit *Receive* jede empfangene Nachricht an die übergeordnete Schicht (wie z.B. FTCom) weitergeleitet. Beim Versand wird mit *Send* nach der Aktivierung durch den *Scheduler* die aktuelle Nachricht von der übergeordneten Schicht geholt und danach auf den Bus verschickt.

BusInterface — timed

in $activation : Slot;\ recv, return : Message$

out $get : Slot;\ send, store : Message$

Receive($recv, store, activation$)

Send($activation, get, return, send$)

Receive

$recv, store \in Message^{\underline{\omega}};\ activation \in Slot^{\underline{\omega}}$

$\forall\, t \in \mathbb{N}$:

if $\mathsf{ti}(activation, t) = \langle\rangle$

then $\mathsf{ti}(store, t) = \mathsf{ti}(recv, t)$

else $\mathsf{ti}(store, t) = \langle\rangle$

fi

Send

$return, send \in Message^{\underline{\omega}};\ get, activation \in Slot^{\underline{\omega}}$

$\forall\, t \in \mathbb{N}$:

if $\mathsf{ti}(activation, t) = \langle\rangle$

then $\mathsf{ti}(get, t) = \langle\rangle \wedge \mathsf{ti}(send, t) = \langle\rangle$

else $\mathsf{ti}(get, t) = \mathsf{ti}(activation, t) \wedge \mathsf{ti}(send, t) = \mathsf{ti}(return, t)$

fi

---

3 Formalisierung von FTCom

Den FT-CNI-Speicher modellieren wir als eine Liste vom Typ *FT_CNI_Entity*. Den Typ *FT_CNI_Entity* stellten wir als ein Paar von Nachrichten-Identifikator und entsprechender Nachricht dar (der Identifikator beschreibt die Nachricht eindeutig). *MessageId* gibt den Typ der Nachrichtenidentifikatoren an. Für den Typ *MessageId* verwenden wir in unserem Modell die natürlichen Zahlen:

type $MessageId = \mathbb{N}$

Der FTCom–Datentyp *DataType* der Nachrichten wurde hier absichtlich nicht spezifiziert, dieser kann zu einem späteren Zeitpunkt festegelegt werden. Der Datentyp *RCV_Type* entspricht dem Ergebnistyp der Funktion *ttReceiveMessage* und der Datentyp *SND_Type* entspricht dem Ergebnistyp der Funktion *ttSendMessage*. Der Datentyp *Request* wird hier für die Anfragen der Ausführung des RDA– und Replica–Tasks benutzt.

$$\mathsf{type}\ D = some(d \in DataType) \mid none$$

$$\mathsf{type}\ FT\_CNI\_Entity = entity(message\_id \in MessageId, ftcdata \in DataType)$$

$$\mathsf{type}\ RCV\_Type = rcv(status\_R \in StatusType, d \in D)$$

$$\mathsf{type}\ SND\_Type = snd(status\_S \in StatusType, newftcom \in FT\_CNI\_Entity^*)$$

$$\mathsf{type}\ Request = \mathbb{B}\mathrm{it}$$

Eine FTCom-Nachricht besteht aus einem Nechrichten-Identifikator (*message_id*) und den Nutzdaten (*ftcdata*). Ein Slot ist ein Zeitschlitz innerhalb eines FlexRay-Zyklusses. Der Typ *Slot* (vgl. die FlexRay-Spezifikation 2.5.2) ist vom Typ der natürlichen Zahlen. Der Type *Message* wurde wie folgt definiert:

$$\mathsf{type}\ Message = msg(slot \in Slot, data \in Data)$$

Der Datentyp *Message* (vgl. Abschnitt 2.5.2) entspricht einem Frame. Dieser einhält einen endlichen Strom von Nachrichten, die während eines FlexRay-Slots übertragen werden:

$$\mathsf{type}\ Data = FT\_CNI\_Entity^*$$

Wenn wir das Modell „One_message_per_frame“ betrachten, nehmen wir an, dass die Typen *Data* und *FT_CNI_Entity* gleich sind. Im einfachsten Fall, wenn eine FTCom-Nachricht nur einmal pro FlexRay-Runde geschickt werden soll, können wir sogar nehmen, dass $Data = DataType$.

Wir stellen eine Frame-Tabelle als einen endlichen Strom (eine endliche Liste) vom Typ *Frames* dar, wobei der Typ *Frames* wie folgt definiert ist:

$$\mathsf{type}\ Frames = frame(sl \in Slot, msl \in MessageId^*)$$

Auf diese Weise übernimmt die Frame-Tabelle die Replikation-Aufgabe: eine Nachricht kann in mehreren Frames (während mehrerer Slots) übertragen werden.

Im Allgemeinen entsprechen einem Slot k n Nachrichten (mit entsprechenden Identifikatoren $x_1, \ldots, x_n$), die als ein Frame während dieses Slots übertragen werden.

Für das Modell „One_message_per_frame“ gilt für die Frame-Tabelle $tableS \in Frames^*$ folgendes:

$$\forall\, n \in \mathsf{dom}.tableS :\ \#msl(table.n) = 1$$

Die Frame-Tabelle eines Knotens ist korrekt, wenn sie nicht leer ist und jeder Slot-Identifikator in der Tabelle höchstens einmal vorkommt. *ProducedMessages* bezeichnet hier die Menge der Indentifikatoren der Nachrichten, die auf dem Knoten erzeugt wurden.

CorrectTableS

$tableS \in Frames^*$

$tableS \neq \langle\rangle$

$\forall\, i, j \in \mathsf{dom}.tableS :\ i \neq j \Rightarrow sl(tableS.i) \neq sl(tableS.j)$

$\forall\, f \in \mathsf{rng}.tableS :$
$\quad x \in msl(f) \rightarrow x \in ProducedMessages$

Wir definieren für jeden Knoten auch eine Liste $tableI \in MessageId^*$ der Nachrichten, die vom CNI-Puffer angefragt werden. Diese Liste ist korrekt, wenn jeder FTCom-Nachrichten-Identifikator in der Tabelle höchstens ein Mal vorkommt.

CorrectTableI

$tableI \in MessageId^*$

$tableI \neq \langle\rangle$

$\forall\, i, j \in \mathsf{dom}.tableI : i \neq j \rightarrow tableI.i \neq tableI.j$

$\forall\, f \in \mathsf{rng}.tableI :\ f \notin ProducedMessages$

Für das Gesamtsystem *SystemArch* brauchen wir auch folgende Eigenschaft:

> Die Tabelle *GtableS* und die Liste *GtableI* sollen korrekt und „invers“ sein, wobei *GtableS* eine Vereinigung aller Tabellen $tableS_i$ und *GtableI* eine Vereinigung aller Listen $tableI_i$ im Gesamtsystem ist.

Korrektheit bedeutet hier, dass das Prädikat *CorrectTableS* für die Tabelle *GtableS* gelten soll, wobei die Menge *ProducedMessages* in diesem Fall die Menge aller Identifikatoren der Nachrichten, die im Gesamtsystem erzeugt werden, bezeichnen. Die Eigenschaft der Tabellen, dass diese invers sind, wurde als Prädikat *InverseSI* definiert.

InverseSI

$gS \in Frames^*;\ \ gI \in MessageId^*$

$\forall\, i \in \mathsf{rng}.gI :\ \exists\, s \in \mathsf{rng}.gS :\ i \in \mathsf{rng}.msl(s)$

$\forall\, frame(s,\ id\_list) \in \mathsf{rng}.gS :\ \forall\, i \in \mathsf{rng}.id\_list :\ i \in \mathsf{rng}.gI$

$\forall\, i, j \in \mathsf{dom}.gS :\ i \neq j \Rightarrow sl(gS.i) \neq sl(gS.j)$

Hinweis: Laut der Annahme *CorrectTableS(tableS)* gilt für alle $sl \in Slot$ folgendes: Da in der Tabelle *tableS* alle $sl : Slot$ nur einmal vorkommen können, kann auch die entsprechende Liste *idlist* in der Tabelle *tableS* nur einmal vorkommen.

$$\exists\, idlist \in MessageId^* :\ frame(sl, idlist) \in \mathsf{rng}.tableS$$
$$\Longrightarrow$$
$$!\,\exists\, idlist \in MessageId^* :\ frame(sl, idlist) \in \mathsf{rng}.tableS$$

3.1 Allgemeine Hilfsfunktionen

In diesem Abschnitt besprechen wir alle Hilfsfunktionen, die wir für unsere FTCom-Spezifikation definiert haben.

3.1.1 DelMessage

Die Hilfsfunktion *DelMessage(i, s)* löscht alle Nachrichten mit dem Identifikator i aus dem Strom s.

DelMessage

$$MessageId \times FT\_CNI\_Entity^* \rightarrow FT\_CNI\_Entity^*$$

$$DelMessage(i,\ \langle\rangle) = \langle\rangle$$
$$DelMessage(i,\ x \,\&\, xs) =$$
$$\quad \mathsf{if}\ \ i = message\_id(x)$$
$$\quad \mathsf{then}\ \ DelMessage(i,\ xs)$$
$$\quad \mathsf{else}\ \ x \,\&\, DelMessage(i,\ xs)\ \ \mathsf{fi}$$

3.1.2 DelFrame

Die Hilfsfunktion *DelFrame(i, s)* löscht alleFrames mit dem Identifikator i aus dem Strom s.

DelFrame

$$Slot \times Message^* \rightarrow Message^*$$

$$DelFrame(i,\ \langle\rangle) = \langle\rangle$$
$$DelFrame(i,\ x \,\&\, xs) =$$
$$\quad \mathsf{if}\ \ i = slot(x)$$
$$\quad \mathsf{then}\ \ DelFrame(i,\ xs)$$
$$\quad \mathsf{else}\ \ x \,\&\, DelFrame(i,\ xs)\ \ \mathsf{fi}$$

3.1.3 ReceiveMessage

Die Funktion *ReceiveMessage* sucht die von der Anwendung angefragte Nachricht im FT-CNI-Speicher (FTCom–Puffer). Diese Funktion gibt die Nachricht mit dem entsprechenden Identifikator aus dem lokalen Speicher (oder *none*, wenn im lokalen Speicher keine Nachricht mit diesem Identifikator gibt), sowie eine Bestätigung, ob diese Operation erfolgreich war, zurück.

ReceiveMessage

$$MessageId \times FT\_CNI\_Entity^* \rightarrow RCV\_Type$$

$$ReceiveMessage(i,\ \langle\rangle) = rcv(FTCOM\_FAILED, none)$$

$$ReceiveMessage(i,\ x \,\&\, xs) =$$
$$\quad \textsf{if}\ \ i = message\_id(x)$$
$$\quad \textsf{then}\ \ rcv(FTCOM\_OK, some(x))$$
$$\quad \textsf{else}\ \ ReceiveMessage(i,\ xs))\ \ \textsf{fi}$$

3.1.4 Message2Entities

Die Funktion *Message2Entities* konvertiert einen Strom vom Typ *Message* zum entsprechenden Strom von Typ *FT_CNI_Entity*:

Message2Entities

$$Message^* \rightarrow FT\_CNI\_Entity^*$$

$$Message2Entities(\langle\rangle) = \langle\rangle$$
$$Message2Entities(x \,\&\, xs) = data(x) \frown Message2Entities(xs)$$

3.1.5 Entities2Data

Die Funktion *Entities2Data* konvertiert einen Strom vom Typ *FT_CNI_Entity* zum entsprechenden Strom von Typ *DataType* und filtert diesen Strom nach einem Nachrichten-Identifikator:

Entities2Data

$$FT\_CNI\_Entity^* \times MessageId \rightarrow DataType^*$$

$$Entities2Data(\langle\rangle,\ i) = \langle\rangle$$
$$Entities2Data(x \,\&\, xs,\ i) =$$
$$\quad \textsf{if}\ \ message\_id(x) = i$$
$$\quad \textsf{then}\ \ ftcdata(x) \,\&\, Entities2Data(xs,\ i)$$
$$\quad \textsf{else}\ \ Entities2Data(xs,\ i)\ \ \textsf{fi}$$

3.1.6 Replicate und ReplicateMessages

Die Replikationsaufgabe kann die Frame-Tabelle übernehmen: eine Nachricht wird in mehreren Frames (während mehrerer Slots) übertragen.

Die Funktion *ReplicateMessages* nimmt einen endlichen ungezeiteten Strom (eine Liste) y vom Typ *MessageId*, einen endlichen ungezeiteten Strom b von FTCom-Nachtichten (eine Liste) und liefert einen endlichen ungezeiteten Strom (eine Liste) der entsprechenden aktuellen Nachrichten. Für jeden Nachrichtenidentifikator aus der Liste y wird die Nachricht mit diesem Identifikator im Strom b gesucht und ausgegeben.

ReplicateMessages

$$MessageId^* \times FT\_CNI\_Entity^* \rightarrow FT\_CNI\_Entity^*$$

$$ReplicateMessages(\langle\rangle,\ b) = \langle\rangle$$

$$ReplicateMessages(y \,\&\, ys,\ b) =$$
$$\quad \mathsf{if}\ \ \exists\, a \in DataType :\ entity(y, a) \in \mathsf{rng}.b$$
$$\quad \mathsf{then}\ \ entity(y, a) \,\&\, ReplicateMessages(ys,\ b)$$
$$\quad \mathsf{else}\ \ ReplicateMessages(ys,\ b)\ \ \mathsf{fi}$$

Die Funktion *Replicate* nimmt einen endlichen ungezeiteten Strom (eine Liste) x vom Typ *Frames*, einen endlichen gezeiteten Strom b von FTCom-Nachtichten und liefert einen endlichen ungezeiteten Strom (eine Liste) vom Typ *Message* der entsprechenden aktuellen FlexRay-Frames. Für jedes Element $frame(x_s, x_m)$ der Liste x wird einen Frame vom Form $msg(x_s, y)$ erzeugt, wobei y ist den Ergebnis, der die Funktion $ReplicateMessages(x_m, b)$ liefert.

Replicate

$$Frames^* \times FT\_CNI\_Entity^* \rightarrow Message^*$$

$$Replicate(\langle\rangle,\ b) = \langle\rangle$$

$$Replicate(frame(x_s, x_m) \,\&\, xs,\ b) =$$
$$\quad msg(x_s, ReplicateMessages(x_m, b)) \,\&\, Replicate(xs,\ t, send_1, \ldots, send_m)$$

3.1.7 StreamRepl und StreamReplMessages

Die Funktion *StreamReplMessages* nimmt einen endlichen ungezeiteten Strom (eine Liste) x vom Typ *MessageId*, eine natürliche Zahl (aktuelle Zeit) und m gezeitete Ströme $send_1, \ldots, send_m$ vom Typ *FT_CNI_Entity* und liefert einen endlichen ungezeiteten Strom (eine Liste) der entsprechenden aktuellen Nachrichten. Für jeden Nachrichtenidentifikator von der Liste x wird die letzte Nachricht mit diesem Identifikator in der Strömen $send_1, \ldots, send_m$ gesucht und ausgegeben.

StreamReplMessages

$$MessageId^* \times \mathbb{N} \times FT\_CNI\_Entity^{\underline{\omega}} \times \cdots \times FT\_CNI\_Entity^{\underline{\omega}}$$
$$\rightarrow FT\_CNI\_Entity^*$$

$$StreamReplMessages(\langle\rangle,\ t, send_1, \ldots, send_m) = \langle\rangle$$

$$StreamReplMessages(x \,\&\, xs,\ t, send_1, \ldots, send_m) =$$
$$\quad \mathsf{if}\ \ (\exists\, i \in [1...m] : \exists\, a \in FT\_CNI\_Entity :$$
$$\qquad \exists\, t_1 : t_1 < t\ \wedge\ \mathsf{ti}(send_i, t_1) = \langle entity(x, a)\rangle$$
$$\quad \wedge\ \forall\, t_2 : t_1 < t_2 < t :\ \neg \exists\, j \in [1...m] :$$
$$\qquad \mathsf{ti}(send_j, t_2) = \langle entity(x, b)\rangle\ \wedge\ a \neq b)$$
$$\quad \mathsf{then}\ \ entity(x, a) \,\&\, StreamReplMessages(xs,\ t, send_1, \ldots, send_m)$$
$$\quad \mathsf{else}\ \ StreamReplMessages(xs,\ t, send_1, \ldots, send_m)\ \ \mathsf{fi}$$

Die Funktion *StreamRepl* nimmt einen endlichen ungezeiteten Strom (eine Liste) vom Typ *Frames* (eine Frame-Tabelle), eine natürliche Zahl (aktuelle Zeit) und m

gezeitete Ströme $send_1, \ldots, send_m$ vom Typ *FT_CNI_Entity* und liefert einen endlichen ungezeiteten Strom (eine Liste) der entsprechenden aktuellen FlexRay-Frames. Für jedes Element y der Frame-Tabelle wird die einen Frame mit dem Frame-Identifikator $sl(y)$ erzeugt, wenn die Nachrichten mit Identifikatoren aus der Liste $msl(y)$ im FTCom-Puffer vorkommen ($StreamReplMessages(msl(y), t, send_1, \ldots, send_m) \neq \langle\rangle$).

StreamRepl

$$Frame^* \times \mathbb{N} \times FT\_CNI\_Entity^{\underline{\omega}} \times \cdots \times FT\_CNI\_Entity^{\underline{\omega}} \rightarrow Message^*$$

$$StreamRepl(\langle\rangle,\ t, send_1, \ldots, send_m) = \langle\rangle$$

$$StreamRepl(y \,\&\, ys,\ t, send_1, \ldots, send_m) =$$
$$\quad \textsf{if}\ \ mlist \neq \langle\rangle$$
$$\quad \textsf{then}\ \ msg(sl(y), mlist) \,\&\, StreamRepl(ys,\ t, send_1, \ldots, send_m)$$
$$\quad \textsf{else}\ \ StreamRepl(ys,\ t, send_1, \ldots, send_m)\ \ \textsf{fi}$$

$$\textsf{where}\ mlist = StreamReplMessages(msl(y), t, send_1, \ldots, send_m)$$

3.1.8 RDA_MV

Die Funktion *RDA_MV* sucht ein Element, das im endlichen ungezeiteten Strom (Liste) die Mehrheit hat. Der Strom darf nicht leer sein. Wir definieren diese Funktion für einen beliebigen Datentyp M.

RDA_MV

$$M^* \rightarrow M$$

$$RDA\_MV(s) = majority(\textsf{ft}.s, 1, \textsf{rt}.s)$$

$$\quad \textsf{where}\ majority\ \ \textsf{so that}$$

$$\quad\ \forall x,\ y \in M;\ n \in \mathbb{N};\ xs \in M^* :$$
$$\quad\ majority(y,\ n,\ \langle\rangle) = y$$
$$\quad\ majority(y,\ n,\ x \,\&\, xs) =$$
$$\quad\quad \textsf{if}\ \ x = y$$
$$\quad\quad \textsf{then}\ \ majority(y,\ n+1,\ xs)$$
$$\quad\quad \textsf{else}\ \ \textsf{if}\ \ n = 0$$
$$\quad\quad\quad \textsf{then}\ \ majority(x,\ 1,\ xs)$$
$$\quad\quad\quad \textsf{else}\ \ majority(y,\ n-1,\ xs)\ \ \textsf{fi}\ \ \textsf{fi}$$

3.1.9 DoRDA

Die Funktion *DoRDA(tableI, y)* konvertiert eine Liste y vom Typ *Message* in eine Liste y' vom Typ *FT_CNI_Entity* mit der Funktion *Message2Entities*. Für jeden Nachrichten-Identifikator i aus der Liste *tableI* werden aus der Liste y' die *ftcdata*-Teile aller Nachrichten mit dem Indentifikator i genommen und darauf die Funktion *RDA_F* angewendet, sofern es entsprechende Nachrichten gibt.

Als die Funktion *RDA_F* benutzt man hier eine Funktion, die dem ausgewählten RDA-Algorithmus entspricht (vgl. Abschnitt 1.2). Wenn wir uns zum „Majority

Vote“-Algorithmus entscheiden, benutzen wir die Funktion *RDA_MV*. Wenn aber wir einfach die erste entsprechende Nachricht nehmen wollen, benutzen wir den FOCUS-Operator ft.

DoRDA

$$MessageId^* \times Message^* \rightarrow FT\_CNI\_Entity^*$$

$$DoRDA(tableI, y) = rda(tableI, Message2Entities(y))$$

where rda so that

$$rda(\langle\rangle,\ y) = \langle\rangle$$
$$rda(x \,\&\, xs,\ y) =$$
if $Entities2Data(y,\ x) = \langle\rangle$
then $rda(xs,\ y)$
else $entity(x,\ RDA\_F(Entities2Data(y,\ x)))\ \&\ rda(xs,\ y)$ fi

3.1.10 UpdateFTC

Die Funktion *UpdateFTC(new, old_ftcom)* aktualisiert die Nachrichten im FTCom-Puffer *old_ftcom* durch die aktuellen Nachrichten von der Liste *new*.

UpdateFTC

$$FT\_CNI\_Entity^* \times FT\_CNI\_Entity^* \rightarrow FT\_CNI\_Entity^*$$

$$UpdateFTC(\langle\rangle,\ old\_ftcom) = old\_ftcom$$

$$UpdateFTCom(x \,\&\, xs,\ old\_ftcom =$$
$$x \,\&\, DelMessage(message\_id(x),\ old\_ftcom)$$

3.1.11 UpdateCNI

Die Funktion *UpdateCNI(new, old_cni)* aktualisiert die Frames im CNI-Puffer *old_cni* durch die aktuellen Frames von der Liste *new*.

UpdateCNI

$$Message^* \times Message^* \rightarrow Message^*$$

$$UpdateCNI(\langle\rangle,\ old\_cni) = old\_cni$$

$$UpdateCNI(x \,\&\, xs,\ old\_cni =$$
$$x \,\&\, DelFrame(slot(x),\ old\_cni)$$

3.2 FTCom

Angenommen, wir haben m Anwendungen. Die Komponente FTCom hat dann $2 * m + 2$ Eingabe- und $2 * m + 2$ Ausgabekanäle:

- Über die Kanäle $send_1, \ldots, send_m : FT\_CNI\_Entity$ wurden die Nachrichten von den Anwendungen verschickt. Dies entspricht dem Aufruf der FTCom-Funktion *ttSendMessage* durch eine Anwendung.

- Über die Kanäle $s\_ack_1, \ldots, s\_ack_m$: *StatusType* erhalten die Anwendungen die Bestätigungen, dass FTCom die Nachrichten erhalten hat. Dies entspricht der Rückgabe der Funktion *ttSendMessage*.
- Über die Kanäle $receive_1, \ldots, receive_m$: *MessageId* fragen die Anwendungen die aktuelle Nachrichten von FTCom an. Dies entspricht dem Aufruf der FTCom-Funktion *ttReceiveMessage* durch eine Anwendung.
- Über die Kanäle $r\_ack_1, \ldots, r\_ack_m$: RCV_Type erhalten die Anwendungen die angefragten Nachrichten. Dies entspricht der Rückgabe der Funktion *ttReceiveMessage*.
- Über den Kanal *fr_store* : *Message* wurden die aktuellen Frames vom , CNI–Puffer empfangen.
- Über den Kanal *start_rda* : *Request* kommt ein Signal von OSEKtime OS, dass man die Nachrichten im FTCom–Puffer durch die Werte, die man durch FlexRay empfangen hat, aktualisieren soll (eine RDA-Anfrage, 1 : *Request*) oder dass man die Frames im CNI–Puffer aktualisieren soll (eine Replikation-Anfrage, 0 : *Request*).
- Über den Kanal *cni_store* : *Message* wurden die aktuellen Frames an den CNI–Puffer geschickt.

Wir verwenden für unsere FTCom-Komponente die folgenden Annahmen:

- Die Frame-Tabellen *tableS* und *tableI*, die die FTCom-Komponente als Parameter bekommet, sind korrekt.
- Zu jedem Zeitpunkt werden nur in einem der Ströme $send_1, \ldots, send_m$, $receive_1, \ldots, receive_m$, *startR*, *fr_store* Nachrichten übertragen. Dies bedeutet, dass die Anwendungen sequentiell laufen und dass die RDA- und Replica-Funktionen auch in der OSEKtime Dispatcher Tabelle eingetragen wurden.
- Jeder der Ströme $send_1, \ldots, send_m, receive_1, \ldots, receive_m$, *startR* darf in jedem Zeitinterval höchstens eine Nachricht haben.

Die folgende Zusicherungen spezifizieren wir als Tabelle *FTCOM_GAR*.

1. Wenn im Zeitintervall t durch den Kanal $send_i$ ($i \in [1...m]$) eine Nachricht x gekommen ist, wird in diesem Zeitintervall durch den Kanal s_ack_i die Bestätigung $FTCOM\_OK$ geschickt.
2. Wenn im Zeitinterval t eine RDA–Anfrage kommt 1 : *Request*, erfolgt keine Ausgabe in diesem Zeitintervall.
3. Wenn im Zeitinterval t eine Replikation–Anfrage kommt 0 : *Request*, werden über den Kanal *cni_store* die aktuelle Frames an den CNI–Puffer geschickt, die durch die Funktion *StreamRepl* erzeugt werden.
4. Wenn im Zeitintervall t über keinen der Eingabe-Kanälen eine Nachricht kommt oder wenn durch den Kanal *fr_store* der aktuelle Inhalt y des CNI-Puffers kommt, erfolgt keine Ausgabe in diesem Zeitintervall.
5. Wenn im Zeitinterval t durch den Kanal $receive_j$ eine aktuelle Nachricht mit dem Indetifikator a angefragt wird, wird in diesem Zeitintervall durch den Kanal r_ack_j die Nachricht $\langle rcv(FTCOM\_OK, none)\rangle$ geschickt, falls
   - Die Nachricht $entity(a,\ x)$ noch von keiner der Anwendungen im FTCom-Puffer gespeichert wurde;

- Die Nachricht $entity(a,\ x)$ wurde noch nicht vom CNI-Puffer gespeichert. Entweder soll die Nachricht auf diesem Knoten „lokal“ erzeugt werden (Identifikator a gehört nicht zur Identifikatoren-Tabelle $tableI$), oder wurde diese Nachricht vom anderen Knoten nach dem CNI-Puffer noch nicht geschickt.

6. Wenn im Zeitinterval t durch den Kanal $receive_j$ eine aktuelle Nachricht mit dem Indetifikator a angefragt wird, wird in diesem Zeitintervall durch den Kanal r_ack_j die Nachricht $\langle rcv(FTCOM\_OK, Some\ x)\rangle$ geschickt, falls folgendes gilt:

   - Identifikator a gehört zur Identifikatoren-Tabelle $tableI$;
   - Die Nachricht $entity(a,\ x)$ wurde noch vom CNI-Puffer gespeichert und wurde weder von einer Anwendung noch durch RDA-Funktion übergeschrieben wurde.

7. Wenn im Zeitinterval t durch den Kanal $receive_j$ eine aktuelle Nachricht mit dem Indetifikator a angefragt wird, wird in diesem Zeitintervall durch den Kanal r_ack_j die Nachricht $\langle rcv(FTCOM\_OK, Some\ x)\rangle$ geschickt, falls folgendes gilt:

   Die Nachricht $entity(a,\ x)$ wurde von einem der Anwendungen im FTCom-Puffer gespeichert und weder von einer Anwendung noch durch RDA-Funktion übergeschrieben wurde.

FTCom (constant $tableS \in Frames^*$; $tableI \in Message^*$) black-box

in $send_1, \ldots, send_m : FT\_CNI\_Entity$; $startR : Request$;
$receive_1, \ldots, receive_m : MessageId$; $fr\_store : Message$;

out $s\_ack_1, \ldots, s\_ack_m : StatusType$;
$r\_ack_1, \ldots, r\_ack_m : RCV\_Type$; $cni\_store : Message$

local $b \in FT\_CNI\_Entity^*$

init $b = \langle\rangle$

asm
$CorrectTableS(tableS)$
$CorrectTableI(tableI)$
$\mathsf{disjunct}(send_1, \ldots, send_m,\ receive_1, \ldots, receive_m,\ fr\_store, startR)$
$\mathsf{maxmsg}_1(startR)$
$\mathsf{maxmsg}_1(send_1)\ \wedge\ \ldots\ \wedge\ \mathsf{maxmsg}_1(send_m)$
$\mathsf{maxmsg}_1(receive_1)\ \wedge\ \ldots\ \wedge\ \mathsf{maxmsg}_1(receive_m)$

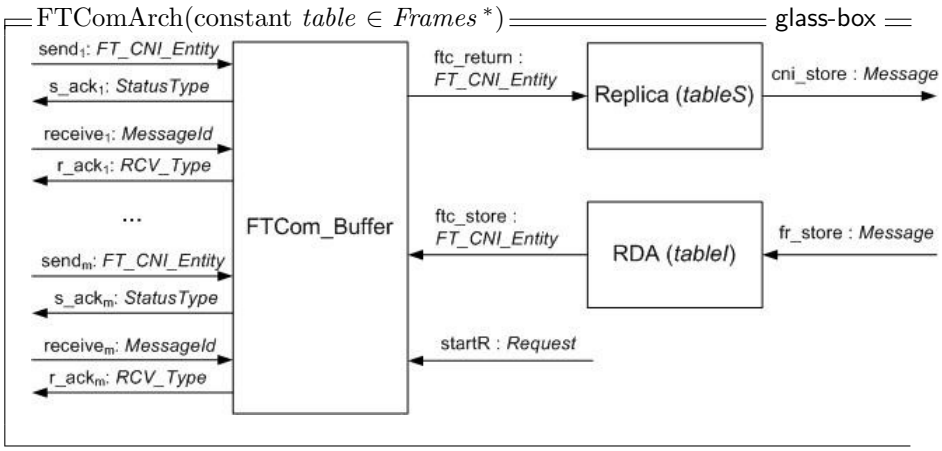
gar
tiTable $FTCOM\_GAR$

Als die Funktion RDA_F benutzt man in der Tabelle $FTCOM\_GAR$ eine Funktion, die dem ausgewählten RDA-Algorithmus entspricht (vgl. Abschnitt 1.2). Wenn wir uns zum „Majority Vote“-Algorithmus entscheiden, benutzen wir die Funktion

RDA_MV. Wenn aber wir einfach die erste entsprechende Nachricht nehmen wollen, benutzen wir den FOCUS-Operator ft.

Die Spezifikation *FTComArch* ist eine Verfeinerung von der Spezifikation *FTCom*. Wir stellen die *FTComArch* Komponente als drei kommunizierende Teilkomponenten dar: die Pufferkomponente *FTCom_Buffer*, die Replikationskomponente *Replica* und die RDA-Komponente *RDA*.

Die Pufferkomponente *FTCom_Buffer* ist mit der Komponenten *Replica* und *RDA* durch die lokalen Kanäle *fr_store* : *FT_CNI_Entity* und *fr_return* : *FT_CNI_Entity* verbunden.



tiTable FTCOM_GAR: $\forall\, t \in \mathbb{N};\ i, j \in [1...m]$:

	$send_i$	$startR$	fr_store	$receive_j$	s_ack_i	r_ack_j	cni_store	$Assumption$
1	$\langle x \rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle FTCOM\_OK \rangle$	$\langle\rangle$	$\langle\rangle$	true
2	$\langle\rangle$	$\langle 1 \rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	true
3	$\langle\rangle$	$\langle 0 \rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$StreamRepl(tableS, t, send_1, \ldots, send_m)$	true
4	$\langle\rangle$	$\langle\rangle$	y	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	true
5	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle a \rangle$	$\langle\rangle$	$\langle rcv(FTCOM\_OK, none) \rangle$	$\langle\rangle$	$\forall\, t_1 < t : (\forall\, i \in [1...m] :\ \neg\, \exists\, x.\ \mathsf{ti}(send_i,\ t_1) = \langle entity(a,\ x) \rangle)$ $\wedge\ entity(a,\ x) \notin \mathsf{rng}.Messages2Entities(\mathsf{ti}(fr\_store,\ t_1))$ $\vee\ a \notin \mathsf{rng}.tableI$
6	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle a \rangle$	$\langle\rangle$	$\langle rcv(FTCOM\_OK, z) \rangle$ $z = Some\ x$	$\langle\rangle$	$a \in \mathsf{rng}.tableI\ \wedge$ $\exists\, t_1 < t : \mathsf{ti}(fr\_store,\ t_1) = z\ \wedge\ z \neq \langle\rangle\ \wedge$ $RDA\_F(Entities2Data(Messages2Entities(z), a)) = x$ $\wedge\ \forall\, t2 :\ t_1 < t_2 < t :$ $(\mathsf{ti}(fr\_store,\ t_2) = q\ \wedge\ q \neq \langle\rangle\ \wedge$ $RDA\_F(Entities2Data(Messages2Entities(q), a)) = y$ $\vee\ \mathsf{ti}(send_k,\ t_2) = \langle entity(a,\ y) \rangle)$ $\Rightarrow x = y$
7	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle a \rangle$	$\langle\rangle$	$\langle rcv(FTCOM\_OK, z) \rangle$ $z = Some\ x$	$\langle\rangle$	$\exists\, t_1 < t : \mathsf{ti}(send_k,\ t_1) = \langle entity(a,\ x) \rangle\ \wedge$ $\wedge\ \forall\, t2 :\ t_1 < t_2 < t :$ $(\mathsf{ti}(fr\_store,\ t_2) = q\ \wedge\ q \neq \langle\rangle\ \wedge a \in \mathsf{rng}.tableI\ \wedge$ $RDA\_F(Entities2Data(Messages2Entities(q), a)) = y$ $\vee\ \mathsf{ti}(send_k,\ t_2) = \langle entity(a,\ y) \rangle)$ $\Rightarrow x = y$

3.3 FTCom–Puffer

Hier werden wir eine lokale Variable $b \in FT\_CNI\_Entity^*$ benutzen, um den Zustand des Puffers darzustellen. Am Anfang soll dieser leer sein: $b = \langle\rangle$.

Wir übernehmen die Annahmen, die wir für die Pufferkomponente *FTCom_Buffer* brauchen, von der Komponente *FTCom* und definieren folgende Zusicherungen in der Tabelle *Buffer_Gar*:

1. Wenn im Zeitintervall t über keinen der Eingabe-Kanälen eine Nachricht kommt, erfolgt auch keine Ausgabe in diesem Zeitintervall durch den keinen der Kanälen.

2. Wenn im Zeitintervall t durch den Kanal $send_i$ $(i \in [1...m])$ eine Nachricht x gekommen ist, wird in diesem Zeitintervall durch den Kanal s_ack_i die Bestätigung $FTCOM\_OK$ geschickt und die lokale Variable b wird aktualisiert – die Nachricht mit dem Identifikator $message\_id(x)$ wird vom Puffer gelöscht (wenn sie dort gibt) und die Nachricht x wird im Puffer eingefügt.

3. Wenn im Zeitinterval t durch den Kanal $receive_j$ eine aktuelle Nachricht mit dem Indetifikator a angefragt wird, wird die Funktion *ReceiveMessage* auf diesen Identifikator a und die lokale Variable b angewendet. Die lokale Variable b bleibt unverändert.

4. Wenn im Zeitinterval t durch den Kanal *ftc_store* die aktuelle Nachrichten von *RDA*-Komponente kommen, wird keine Ausgabe (durch den keinen der Kanälen) in diesem Zeitintervall, aber die lokale Variable b wird die Funktion *UpdateFTC* aktualisiert.

5. Wenn im Zeitinterval t eine RDA–Anfrage kommt $1 : Request$, erfolgt keine Ausgabe in diesem Zeitintervall durch den keinen der Kanälen.

6. Wenn im Zeitinterval t eine Replikation–Anfrage kommt $0 : Request$, wird den Kanal *ftc_return* die aktuelle Puffersinhalt nach *Replica*-Komponente geschickt.

Die gesamte FOCUS-Spezifikation der Pufferkomponente stellen wir wie folgt dar:

FTCom_Buffer — timed

in $send_1, \ldots, send_m : FT\_CNI\_Entity$;
$receive_1, \ldots, receive_m : MessageId$;
$ftc\_store : FT\_CNI\_Entity$
$startR : Request$

out $s\_ack_1, \ldots, s\_ack_m : StatusType$;
$r\_ack_1, \ldots, r\_ack_m : RCV\_Type$;
$ftc\_return : FT\_CNI\_Entity$

---

local $b \in FT\_CNI\_Entity^*$

---

init $b = \langle\rangle$

---

asm
disjunct$(send_1, \ldots, send_m,\ receive_1, \ldots, receive_m,\ cni_store,\ startR)$
maxmsg$_1(startR)$
maxmsg$_1(send_1)\ \wedge\ \ldots\ \wedge$ maxmsg$_1(send_m)$
maxmsg$_1(receive_1)\ \wedge\ \ldots\ \wedge$ maxmsg$_1(receive_m)$

---

gar
tiTable *Buffer_Gar*

tiTable Buffer_GAR: $\forall\, t \in \mathbb{N};\ i, j \in [1...m]$:

	$send_i$	$receive_j$	ftc_store	$startR$	s_ack_i	r_ack_j	ftc_return	b'
1	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	b
2	$\langle x\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle FTCOM\_OK\rangle$	$\langle\rangle$	$\langle\rangle$	$x\&DelMessage(message\_id(x), b)$
3	$\langle\rangle$	$\langle a\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle ReceiveMessage(a, b)\rangle$	$\langle\rangle$	b
4	$\langle\rangle$	$\langle\rangle$	y	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$UpdateFTC(y, b)$
5	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle 1\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	b
6	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle 0\rangle$	$\langle\rangle$	$\langle\rangle$	b	b

3.4 Replikation

Wir übernehmen die Annahme über die Frame-Tabelle *tableS* von der Komponente *FTCom* und definieren folgende Zusicherung:

Wenn im Zeitintervall t durch den Kanal *ftc_return* Nachrichten vom Typ *FT_CNI_Entity* kommen, werden diese durch die Funktion *Replicate* laut der Frame-Tabelle *tableS* zum Typ *Message* konvertiert und das Ergebnis wird im gleichen Zeitinterval t durch den Kanal *cni_store* ausgegeben.

Die FOCUS-Spezifikation der *Replica*-Komponente stellen wir wie folgt dar:

Replica(constant tableS ∈ $Frames^*$) — timed

in $ftc\_return : FT\_CNI\_Entity$

out $cni\_store : Message$

asm
$CorrectTableI(tableS)$
gar
$\forall\, t \in \mathbb{N}$:
ti($cni\_store,\ t$) =
if ti($ftc\_return,\ t$) = $\langle\rangle$
then $\langle\rangle$
else $Replicate(tableS,\ \mathsf{ti}(ftc\_return,\ t))$ fi

3.5 RDA

Wir übernehmen die Annahme über die Frame-Tabelle *tableI* von der Komponente *FTCom* und definieren folgende Zusicherung:

Wenn im Zeitintervall t durch den Kanal *fr_store* Nachrichten vom Typ *Message* kommen, werden diese durch die Funktion *DoRDA* laut der Frame-Tabelle *tableI* zum Typ *FT_CNI_Entity* konvertiert und das Ergebnis wird im gleichen Zeitinterval t durch den Kanal *ftc_store* ausgegeben.

Die FOCUS-Spezifikation der *RDA*-Komponente stellen wir wie folgt dar:

RDA(constant tableI ∈ $Message^*$) — timed

in $fr\_store : Message$

out $ftc\_store : FT\_CNI\_Entity$

asm
$CorrectTableI(tableI)$
gar
$\forall\, t \in \mathbb{N}$:
ti($ftc\_store,\ t$) =
if ti($fr\_store,\ t$) = $\langle\rangle$
then $\langle\rangle$
else $DoRDA(tableI,\ \mathsf{ti}(fr\_store,\ t))$ fi

3.6 CNI_Buffer

Der Puffer *CNI_Buffer* ist verantwortlich für die Zwischenspeicherung der Frames, die FTCom von FlexRay bekommt und umgekehrt.

Wir verwenden für unsere *CNI*-Komponente die folgenden Annahme: die Ströme *cni_store*, *store*, *get* und *startR* sind disjunkt.

Hier werden wir eine lokale Variable $c \in Message^*$ benutzen, um den Zustand des Puffers darzustellen. Am Anfang soll der leer sein: $c = \langle\rangle$.

Die folgenden Zusicherungen spezifizieren wir als Tabelle *CNI_GAR*:

1. Wenn im Zeitintervall t über keinen der Eingabe-Kanälen eine Nachricht kommt, erfolgt durch den keinen der Kanälen auch keine Ausgabe in diesem Zeitintervall.

2. Wenn im Zeitintervall t durch den Kanal *get* ein nichtleerer Strom s von Slotidentifikatoren gekommen ist, wird die Funktion *takeMessages* auf diesen Strom s und die lokale Variable c angewendet und das Ergebnis wird in diesem Zeitintervall durch den Kanal *return* ausgegeben. Die lokale Variable c bleibt unverändert.

3. Wenn im Zeitintervall t durch den Kanal *store* ein nichtleerer Strom m von FlexRay-Frames (von FlexRay) gekommen ist, wird der CNI-Puffer (die lokele Variable c) durch die Funktion *UpdateCNI* (vgl. Abschnitt 3.1.11) mit diesem Strom m aktualisiert.

4. Wenn im Zeitintervall t durch den Kanal *cni_store* ein nichtleerer Strom m von FlexRay-Frames (von FTCom) gekommen ist, wird der CNI-Puffer (die lokele Variable c) durch die Funktion *UpdateCNI* (vgl. Abschnitt 3.1.11) mit diesem Strom m aktualisiert.

   Wenn OSEKtime OS auf dem Knoten immer erst *RDA* aufruft und nur dann *Replication* (die ungezeitete Version des Stromes *stratR* ist eine Folge von $1 : Request$ und $0 : Request$, die mit $1 : Request$ anfängt), könnten wir für diese Zeile in der Tabelle auch $c' = m$ statt $UpdateCNI(m, c)$ definieren.

5. Wenn im Zeitinterval t eine RDA–Anfrage kommt $1 : Request$, wird durch den Kanal *fr_store* der Inhalt des CNI-Puffers an die Komponente *FTCom_Buffer* geschickt.

6. Wenn im Zeitinterval t eine Replikation–Anfrage kommt $0 : Request$, erfolgt keine Ausgabe in diesem Zeitintervall durch den keinen der Kanälen.

CNI_Buffer — timed

in *cni_store* : *Message*; *store* : *Message*; *get* : *Slot*; *startR* : *Request*

out *return* : *Message*; *fr_store* : *Message*

local $c \in Message^*$

init $c = \langle\rangle$

asm
  disjunct(*get*, *store*, *startR*, *cni_store*)

gar
  tiTable *CNI_GAR*

tiTable CNI_GAR: $\forall\, t \in \mathbb{N}$:

	get	*store*	*cni_store*	*startR*	*return*	*fr_store*	c'
1	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	c
2	$s,\ s \neq \langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$takeMessages(s, c)$	$\langle\rangle$	c
3	$\langle\rangle$	$m,\ m \neq \langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$UpdateCNI(m, c)$
4	$\langle\rangle$	$\langle\rangle$	$m,\ m \neq \langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$UpdateCNI(m, c)$
5	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle 1\rangle$	$\langle\rangle$	c	c
6	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle 0\rangle$	$\langle\rangle$	$\langle\rangle$	c

4 Zusammenfassung und Ausblick

In diesem Bericht wurde eine Abstraktion des FlexRay in der formalen Sprache Focus dargestellt. Die vorliegende Formalisierung von FlexRay hat gezeigt, dass sich Focus sehr gut für die Spezifikation von verteilten, zeitgesteuerten Systemen eignet. In diesem Bericht wurde ebenfalls eine Abstraktion des FTCom in Focus vorgestellt, diese erfolgte unter Berücksichtigung der formalen Spezifikation von FlexRay.

In Teilprojekt Automotive wurde für die nächste Ausbaustufe des Systems die bestehende FlexRay–Modellierung durch zwei Bus Guardians [5], einen zweiten Kommunikationskanal und die Modellierung von Übertragungsfehlern erweitert. Der Replikation/RDA–Teil wurde auch entsprechend der Focus-Spezifation in Isabelle implementiert.

Literatur

[1] FlexRay Consortium . *FlexRay Communication System - Protocol Specification - Version 2.0*, 2004.

[2] Verisoft–Automotive Project.
`http://www4.in.tum.de/~verisoft/automotive`.

[3] Broy, M., and Stoelen, K. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[4] FlexRay Consortium. `http://www.flexray.com`.

[5] FlexRay Consortium. *FlexRay Communication System - Bus Guardian Specification - Version 2.0*, 2004.

[6] OSEK/VDX. *Fault-Tolerant Communication - Spezification 1.0*, 2001.

[7] OSEK/VDX. *Time-Triggered Operating System - Spezification 1.0*, 2001.

[8] Verisoft Project. `http://www.verisoft.de`.

[9] Wenzel, M. *The Isabelle/Isar Reference Manual*. TU München, 2004.