Stream processing components: Isabelle/HOL formalisation and case studies

Maria Spichkova

October 13, 2025

Abstract

This set of theories presents an Isabelle/HOL formalisation of stream processing components introduced in Focus, a framework for formal specification and development of interactive systems. This is an extended and updated version of the formalisation, which was elaborated within the methodology "Focus on Isabelle" [6]. In addition, we also applied the formalisation on three case studies that cover different application areas: process control (Steam Boiler System), data transmission (FlexRay communication protocol), memory and processing components (Automotive-Gateway System).

Contents

1 Introduction

The set of theories presented in this paper is an extended and updated Isabelle/HOL[5] formalisation of stream processing components elaborated within the methodology "Focus on Isabelle" [6]. This paper is organised as follows: in the first section we give a general introduction to the Focus stream processing components [1] and briefly describe three case studies to show how the formalisation can be used for specification and verification of system properties. After that we present the Isabelle/HOL representation of these concepts and a number of auxiliary theories on lists and natural numbers useful for the proofs in the case studies. The last three sections introduce the case studies, where system properties are verified formally using the Isabelle theorem prover.

1.1 Stream processing components

The central concept in Focus is a *stream* representing a communication history of a *directed channel* between components. A system in Focus is specified by its components that are connected by channels, and are described in terms of its input/output behavior. The channels in this specification framework are *asynchronous communication links* without delays. They are *directed* and generally assumed to be *reliable*, and *order preserving*. Via these channels components exchange information in terms of *messages* of specified types. For any set of messages M, M^{∞} and M^{*} denote the sets of all infinite and all finite untimed streams respectively:

$$M^{\infty} \stackrel{\text{def}}{=} \mathbb{N}_{+} \rightarrow M \qquad M^{*} \stackrel{\text{def}}{=} \cup_{n\in\mathbb{N}}([1..n] \rightarrow M)$$

A *timed stream*, as suggested in our previous work [6], is represented by a sequence of *time intervals* counted from 0, each of them is a finite sequence of messages that are listed in their order of transmission:

$$M^{\underline{\infty}} \stackrel{\text{def}}{=} \mathbb{N}_{+} \rightarrow M^{*} \qquad M^{\underline{*}} \stackrel{\text{def}}{=} \cup_{n\in\mathbb{N}}([1..n] \rightarrow M^{*})$$

A specification can be elementary or composite – composite specifications are built hierarchically from the elementary ones. Any specification characterises the relation between the *communication histories* for the external *input* and *output channels*: the formal meaning of a specification is exactly the *input/output relation.* This is specified by the lists of input and output channel identifiers, I and O, while the syntactic interface of the specification S is denoted by $(I_S \triangleright O_S)$.

To specify the behaviour of a real-time system we use *infinite timed streams* to represent the input and the output streams. The type of *finite timed streams* will be used only if some argumentation about a timed stream that was truncated at some point of time is needed. The type of *finite* *untimed streams* will be used to argue about a sequence of messages that are transmitted during a time interval. The type of *infinite untimed streams* will be used in the case of timed specifications only to represent local variables of FOCUS specification. Our definition in Isabelle/HOL of corresponding types is given below:

- Finite timed streams of type *'a* are represented by the type *'a fstream*, which is an abbreviation for the type *'a list list.*
- Finite untimed streams of type *'a* are represented by the list type: *'a list.*
- Infinite timed streams of type *'a* are represented by the type *'a istream*, which represents the functional type *nat* ⇒ *'a list.*
- Infinite untimed streams of type *'a* are represented by the functional type *nat* ⇒ *'a*.

1.2 Case Study 1: Steam Boiler System

A steam boiler control system can be represent as a distributed system consisting of a number of communicating components and must fulfil real time requirements. This case study shows how we can deal with local variables (system's states) and in which way we can represent mutually recursive functions to avoid problems in proofs. The main idea of the steam boiler specification was taken from [1]: The steam boiler has a water tank, which contains a number of gallons of water, and a pump, which adds 10 gallons of water per time unit to its water tank, if the pump is on. At most 10 gallons of water are consumed per time unit by the steam production, if the pump is off. The steam boiler has a sensor that measures the water level.

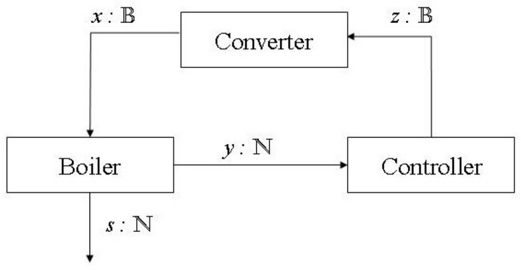
We specified the following components: *ControlSystem* (general requirements specification), *ControlSystemArch* (system architecture), *SteamBoiler*, *Converter*, and *Controller*. We present here the following Isabelle/HOL theories for this system:

- *SteamBoiler.thy* – specifications of the system components,
- *SteamBoiler_proof* – proof of refinement relation between the requirements and the architecture specifications.

The specification *ControlSystem* describes the requirements for the steam boiler system: in each time interval the system outputs it current water level in gallons and this level should always be between 200 and 800 gallons (the system works in the time-synchronous manner).

The specification *ControlSystemArch* describes a general architecture of the steam boiler system. The system consists of three components: a steam boiler, a converter, and a controller.

ControlSystemArch ═══ glass-box



The *SteamBoiler* component works in time-synchronous manner: the current water level is controlled every time interval. The boiler has two output channels with equal streams ($y = s$) and it fixes the initial water level to be 500 gallons. For every point of time the following must be true: if the pump is off, the boiler consumes at most 10 gallons of water, otherwise (the pump is on) at most 10 gallons of water will be added to its water tank.

The *Converter* component converts the asynchronous output produced by the controller to time-synchronous input for the steam boiler. Initially the pump is off, and at every later point of time (from receiving the first instruction from the controller) the output will be the last input from the controller.

The *Controller* component, contrary to the steam boiler component, behaves in a purely asynchronous manner to keep the number of control signals small, it means it might not be desirable to switch the pump on and off more often than necessary. The controller is responsible for switching the steam boiler pump on and off. If the pump is off: if the current water level is above 300 gallons the pump stays off, otherwise the pump is started and will run until the water level reaches 700 gallons. If the pump is on: if the current water level is below 700 gallons the pump stays on, otherwise the pump is turned off and will be off until the water level reaches 300 gallons.

To show that the specified system fulfills the requirements we need to show that the specification *ControlSystemArch* is a refinement of the specification *ControlSystem*. It follows from the definition of behavioral refinement that in order to verify that *ControlSystem* $\rightsquigarrow$ *ControlSystemArch* it is enough to prove that

$$[\![ControlSystemArch]\!] \Rightarrow [\![ControlSystem]\!]$$

Therefore, we have to prove a *lemma* that says the specification *ControlSystemArch* is a refinement of the specification *ControlSystem*:

lemma *L0-ControlSystem*: $[\![$ *ControlSystemArch s* $]\!] \Longrightarrow$ *ControlSystem s*

1.3 Case Study 2: FlexRay Communication Protocol

In this section we present a case study on FlexRay, communication protocol for safety-critical real-time applications. This protocol has been developed by the FlexRay Consortium [2] for embedded systems in vehicles, and its advantages are deterministic real-time message transmission, fault tolerance, integrated functionality for clock synchronisation and higher bandwidth.

FlexRay contains a set of complex algorithms to provide the communication services. From the view of the software layers above FlexRay only a few of these properties become visible. The most important ones are static cyclic communication schedules and system-wide synchronous clocks. These provide a suitable platform for distributed control algorithms as used e.g. in drive-by-wire applications. The formalization described here is based on the "Protocol Specification 2.0"[3].

The static message transmission model of FlexRay is based on *rounds*. FlexRay rounds consist of a constant number of time slices of the same length, so called *slots*. A node can broadcast its messages to other nodes at statically defined slots. At most one node can do it during any slot.

For the formalisation of FlexRay in Focus we would like to refer to [4] and [6]. To reduce the complexity of the system several aspects of FlexRay have been abstracted in this formalisation:

(1) There is no clock synchronization or start-up phase since clocks are assumed to be synchronous. This corresponds very well with the *time-synchronous* notion of Focus.

(2) The model does not contain bus guardians that protect channels on the physical layer from interference caused by communication that is not aligned with FlexRay schedules.

(3) Only the static segment of the communication cycle has been included not the dynamic, as we are mainly interested in time-triggered systems.

(4) The time-basis for the system is one slot i.e. one slot FlexRay corresponds to one tick in in the formalisation.

(5) The system contains only one FlexRay channel. Adding a second channel would mean simply doubling the FlexRay component with a different configuration and adding extra channels for the access to the *CNI_Buffer* component.

The system architecture consists of the following components, which describe the FlexRay components accordingly to the FlexRay standard [3]:

- *FlexRay* (general requirements specification),
- *FlexRayArch* (system architecture),
- *FlexRayArchitecture* (guarantee part of the system architecture),

- *Cable*,
- *Controller*,
- *Scheduler*, and
- *BusInterface*.

We present the following Isabelle/HOL theories in this case study:

- *FR_types.thy* – datatype definitions,
- *FR.thy* – specifications of the system components and auxiliary functions and predicates,
- *FR_proof* – proof of refinement relation between the requirements and the architecture specifications.

The type *Frame* that describes a FlexRay frame consists of a slot identifier of type $\mathbb{N}$ and the payload. The type of payload is defined as a finite list of type *Message*. The type *Config* represents the bus configuration and contains the scheduling table *schedule* of a node and the length of the communication round *cycleLength*. A scheduling table of a node consists of a number of slots in which this node should be sending a frame with the corresponding identifier (identifier that is equal to the slot).

$$\begin{aligned}
\textsf{type}\ \mathit{Message} &= \mathit{msg}\ (\mathit{message\_id} : \mathbb{N}, \mathit{ftcdata} : \mathit{Data}) \\
\textsf{type}\ \mathit{Frame} &= \mathit{frm}\ (\mathit{slot} : \mathbb{N}, \mathit{data} : \mathit{Data}) \\
\textsf{type}\ \mathit{Config} &= \mathit{conf}\ (\mathit{schedule} : \mathbb{N}^*, \mathit{cycleLength} : \mathbb{N})
\end{aligned}$$

We do not specify the type *Data* here to have a polymorphic specification of FlexRay (this type can be underspecified later to any datatype), therefore, in Isabelle/HOL it will be also defined as a polymorphic type $'a$. The types $'a$ *nFrame*, *nNat* and *nConfig* are used to represent sheaves of channels of types *Frame*, $\mathbb{N}$ and *Config* respectively. In the specification group will be used channels *recv* and *activations*, as well as sheaves of channels ($return_1$, $\dots$,$return_n$), ($c_1, \dots, c_n$), ($store_1$, $\dots$, $store_n$), (get_1, $\dots$, get_n), and ($send_1$, $\dots$, $send_n$). We also need to declare some constant, *sN*, for the number of specification replication and the corresponding number of channels in sheaves, as well as to define the list of sheaf upper bounds, *sheafNumbers*.

The architecture of the FlexRay communication protocol is specified as the FOCUS specification *FlexRayArch*. Its assumption-part consists of three constraints: (i) all bus configurations have disjoint scheduling tables, (ii) all bus configurations have the equal length of the communication round, (iii) each FlexRay controller can receive tab most one data frame each time interval from the environment' of the FlexRay system. The guarantee-part of *FlexRayArch* is represented by the specification *FlexRayArchitecture* (see below).

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

in $return_1, ..., return_n : Frame$

out $store_1, ..., store_n : Frame;\ get_1, ..., get_n : \mathbb{N}$

asm $\forall\, i \in [1..n] : \mathsf{msg}_1(return_i)$

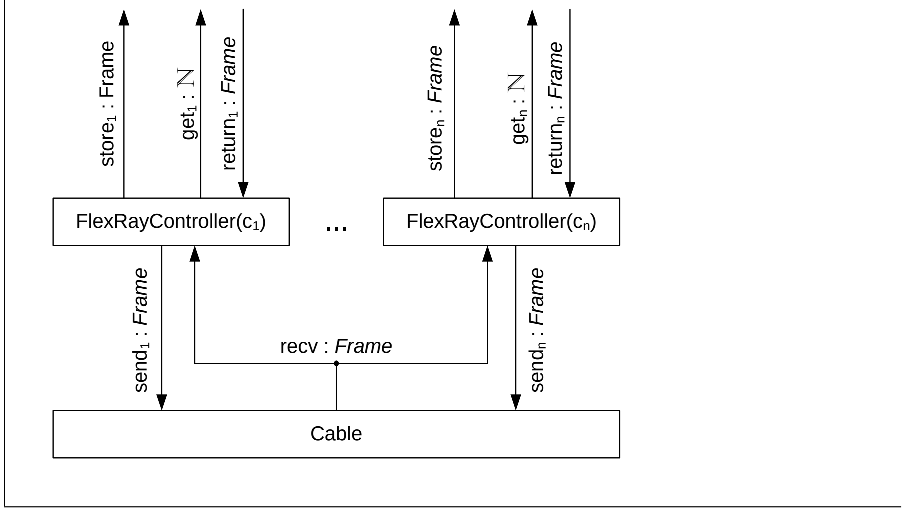
$DisjointSchedules(c_1, \dots, c_n)$

$IdenticCycleLength(c_1, \dots, c_n)$

gar $(store_1, \dots, store_n, get_1, \dots, get_n) :=$

$FlexRayArchitecture(c_1, \dots, c_n)(return_1, dots, return_n)$

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



The component *Cable* simulate the broadcast properties of the physical network cable – every received FlexRay frame is resent to all connected nodes. Thus, if one *FlexRayController* send some frame, this frame will be resent to all nodes (to all *FlexRayController*s of the system). The assumption is that all input streams of the component *Cable* are disjoint – this holds by the properties of the *FlexRayController* components and the overall system assumption that the scheduling tables of all nodes are disjoint. The guarantee is specified by the predicate *Broadcast*.

The Focus specification *FlexRayController* represent the controller component for a single node of the system. It consists of the components *Scheduler* and *BusInterface*. The *Scheduler* signals the *BusInterface*, that is responsible for the interaction with other nodes of the system (i.e. for the real send and receive of frames), on which time which FlexRay frames must be send from the node. The *Scheduler* describes the communication scheduler. It sends at every time t interval, which is equal modulo the length of the communication cycle to some FlexRay frame identifier (that corresponds to the number of the slot in the communication round) from the scheduler table, this frame identifier.

The specification *FlexRay* represents requirements on the protocol: If the scheduling tables are correct in terms of the predicates *DisjointSchedules* (all bus configurations have disjoint scheduling tables) and *IdenticCycleLength* (all bus configurations have the equal length of the communication round), and also the FlexRay component receives in every time interval at most one message from each node (via channels $return_i$, $1 \leq i \leq n$), then

- the frame transmission by FlexRay must be correct in terms of the predicate *FrameTransmission*: if the time t is equal modulo the length of the cycle (FlexRay communication round) to the element of the scheduler table of the node k, then this and only this node can send a data atn the tth time interval;
- FlexRay component sends in every time interval at most one message to each node via channels get_i and $store_i$, $1 \leq i \leq n$).

To show that the specified system fulfill the requirements we need to show that the specification *FlexRayArch* is a refinement of the specification *FlexRay*. It follows from the definition of behavioral refinement that in order to verify that $FlexRay \rightsquigarrow FlexRayArch$ it is enough to prove that

$$[\![FlexRayArch]\!] \Rightarrow [\![FlexRay]\!]$$

Therefore, we have to define and to prove a lemma, that says the specification *FlexRayArch* is a refinement of the specification *FlexRay*:

lemma *main-fr-refinement*:
FlexRayArch n nReturn nC nStore nGet $\Longrightarrow$ *FlexRay n nReturn nC nStore nGet*

1.4 Case Study 3: Automotive-Gateway

This section introduces the case study on telematics (electronic data transmission) gateway that was done for the Verisoft project[1]. If the gateway receives from a ECall application of a vehicle a signal about crash (more precise, the command to initiate the call to the Emergency Service Center, ESC), and after the establishing the connection it receives the command to send the crash data, received from sensors. These data are restored in the internal buffer of the gateway and should be resent to the ESC and the voice communication will be established, assuming that there is no connection fails. The system description consists of the following specifications:

[1]http://www.verisoft.de

- *GatewaySystem* (gateway system architecture),
- *GatewaySystemReq* (gateway system requirements),
- *ServiceCenter* (Emergency Service Center),
- *Gateway* (gateway architecture),
- *GatewayReq* (gateway requirements),
- *Sample* (the main component describing its logic),
- *Delay* (the component modelling the communication delay), and
- *Loss* (the component modelling the communication loss).

We present the following Isabelle/HOL theories in this case study:

- *Gateway_types.thy* – datatype definitions,
- *Gateway.thy* – specifications of the system components,
- *Gateway_proof* – proofs of refinement relations between the requirements and the architecture specifications (for the components *Gateway* and *GatewaySystem*).

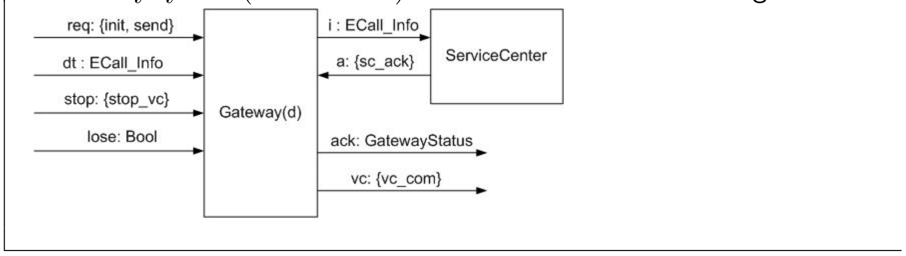
The datatype *ECall_Info* represents a tuple, consisting of the data that the Emergency Service Center needs – here we specify these data to contain the vehicle coordinates and the collision speed, they can also extend by some other information. The datatype *GatewayStatus* represents the status (internal state) of the gateway.

$$\begin{aligned}
&\textsf{type } \mathit{Coordinates} && = \mathbb{N} \times \mathbb{N} \\
&\textsf{type } \mathit{CollisionSpeed} && = \mathbb{N} \\
&\textsf{type } \mathit{ECall\_Info} && = \mathit{ecall}(\mathit{coord} \in \mathit{Coordinates}, \mathit{speed} \in \mathit{CollisionSpeed}) \\
&\textsf{type } \mathit{GatewayStatus} && = \{\ \mathit{init\_state},\ \mathit{call},\ \mathit{connection\_ok}, \\
& && \qquad \mathit{sending\_data},\ \mathit{voice\_com}\ \}
\end{aligned}$$

To specify the automotive gateway we will use a number of datatypes consisting of one or two elements: $\{init, send\}$, $\{stop\_vc\}$, $\{vc\_com\}$ and $\{sc\_ack\}$. We name these types *reqType*, *stopType*, *vcType* and *aType* correspondingly.

The Focus specification of the general gateway system architecture is presented below:

GatewaySystem(const d ∈ ℕ) glass-box



The stream *loss* is specified to be a time-synchronous one (exactly one message each time interval). It represents the connection status: the message true at the time interval t corresponds to the connection failure at this time interval, the message false at the time interval t means that at this time interval no data loss on the gateway connection.

The specification *GatewaySystemReq* specifies the requirements for the component *GatewaySystem*: Assuming that the input streams *req* and *stop* can contain at every time interval at most one message, and assuming that the stream *lose* contains at every time interval exactly one message. If

- at any time interval t the gateway system is in the initial state,
- at time interval $t+1$ the signal about crash comes at first time (more precise, the command to initiate the call to the ESC,
- after $3+m$ time intervals the command to send the crash data comes at first time,
- the gateway system has received until the time interval $t+2$ the crash data,
- there is no connection fails from the time interval t until the time interval $t+4+k+2d$,

then at time interval $t+4+k+2d$ the voice communication is established.

The component *ServiceCenter* represents the interface behaviour of the ESC (wrt. connection to the gateway): if at time t a message about a vehicle crash comes, it acknowledges this event by sending the at time $t+1$ message *sc_ack* that represents the attempt to establish the voice communication with the driver or a passenger of the vehicle. if there is no connection failure, after d time intervals the voice communication will be started.

We specify the gateway requirements (*GatewayReq*) as follows:

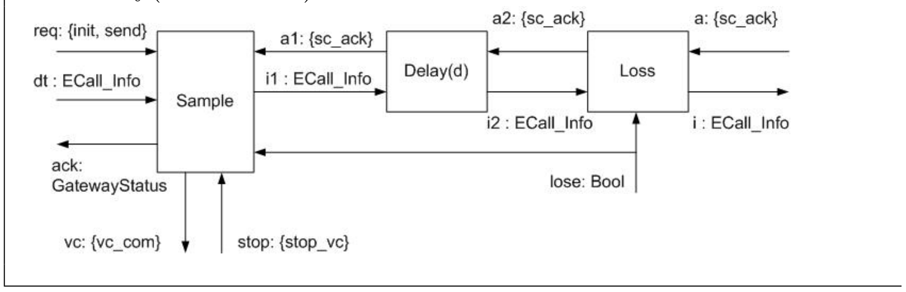
1. If at time t the gateway is in the initial state *init_state*, and it gets the command to establish the connection with the central station, and also there is no environment connection problems during the next 2 time intervals, it establishes the connection at the time interval $t+2$.

2. If at time t the gateway has establish the connection, and it gets the command to send the ECall data to the central station, and also there is no environment connection problems during the next $d+1$ time intervals, then it sends the last corresponding data. The central station becomes these date at the time $t+d$.

3. If the gateway becomes the acknowledgment from the central station that it has receives the sent ECall data, and also there is no environment connection problems, then the voice communication is started.

The specification of the gateway architecture, *Gateway*, is parameterised one: the parameter $d \in \mathbb{N}$ denotes the communication delay between the central station and a vehicle. This component consists of three subcomponents: *Sample*, *Delay*, and *Loss*:

Gateway(const d $\in \mathbb{N}$) timed



The component *Delay* models the communication delay. Its specification is parameterised one: it inherits the parameter of the component *Gateway*. This component simply delays all input messages on d time intervals. During the first d time intervals no output message will be produced.

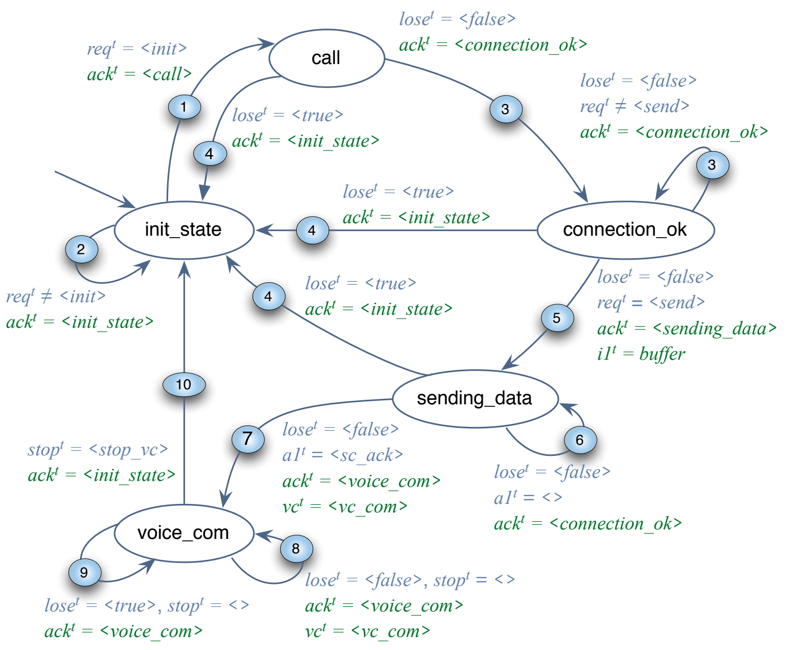
The component *Loss* models the communication loss between the central station and the vehicle gateway: if during time interval t from the component *Loss* no message about a lost connection comes, the messages come during time interval t via the input channels a and $i2$ will be forwarded without any delay via channels $a2$ and i respectively. Otherwise all messages come during time interval t will be lost.

The component *Sample* represents the logic of the gateway component. If it receives from a ECall application of a vehicle the command to initiate the call to the ESC it tries to establish the connection. If the connection is established, and the component *Sample* receives from a ECall application of a vehicle the command to send the crash data, which were already received and stored in the internal buffer of the gateway, these data will be resent to the ESC. After that this component waits to the acknowledgment from the ESC. If the acknowledgment is received, the voice communication will be established, assuming that there is no connection fails.

For the component *Sample* we have the assumption, that the streams *req*, $a1$, and *stop* can contain at every time interval at most one message, and also that the stream *loss* must contain at every time interval exactly one message. This component uses local variables *st* and *buffer* (more precisely, a local variable *buffer* and a state variable *st*). The guarantee part of the component *Sample* can be specified as a timed state transition diagram (TSTS) and an expression which says how the local variable *buffer* is computed, or using the corresponding table representation, which is semantically equivalent to the TSTD.

To show that the specified gateway architecture fulfils the requirements we need to show that the specification *Gateway* is a refinement of the specification *GatewayReq*. Therefore, we need to define and to prove the following lemma:

lemma *Gateway-L0*:



call
init_state
connection_ok
sending_data
voice_com
1
$req^t = \langle init \rangle$
$ack^t = \langle call \rangle$
2
$req^t \neq \langle init \rangle$
$ack^t = \langle init\_state \rangle$
3
$lose^t = \langle false \rangle$
$ack^t = \langle connection\_ok \rangle$
3
$lose^t = \langle false \rangle$
$req^t \neq \langle send \rangle$
$ack^t = \langle connection\_ok \rangle$
4
$lose^t = \langle true \rangle$
$ack^t = \langle init\_state \rangle$
4
$lose^t = \langle true \rangle$
$ack^t = \langle init\_state \rangle$
4
$lose^t = \langle true \rangle$
$ack^t = \langle init\_state \rangle$
5
$lose^t = \langle false \rangle$
$req^t = \langle send \rangle$
$ack^t = \langle sending\_data \rangle$
$i1^t = buffer$
6
$lose^t = \langle false \rangle$
$a1^t = \langle\rangle$
$ack^t = \langle connection\_ok \rangle$
7
$lose^t = \langle false \rangle$
$a1^t = \langle sc\_ack \rangle$
$ack^t = \langle voice\_com \rangle$
$vc^t = \langle vc\_com \rangle$
8
$lose^t = \langle false \rangle, stop^t = \langle\rangle$
$ack^t = \langle voice\_com \rangle$
$vc^t = \langle vc\_com \rangle$
9
$lose^t = \langle true \rangle, stop^t = \langle\rangle$
$ack^t = \langle voice\_com \rangle$
10
$stop^t = \langle stop\_vc \rangle$
$ack^t = \langle init\_state \rangle$

Figure 1: Timed state transition diagram for the component Sample

Gateway req dt a stop lose d ack i vc
$\Longrightarrow$ *GatewayReq req dt a stop lose d ack i vc*

To show that the specified gateway architecture fulfills the requirements we need to show that the specification *GatewaySystem* is a refinement of the specification *GatewaySystemReq*. Therefore, we need to define and to prove the following lemma:

lemma *GatewaySystem-L0*:
GatewaySystem req dt stop lose d ack vc
$\Longrightarrow$ *GatewaySystemReq req dt stop lose d ack vc*

2 Theory ArithExtras.thy

theory *ArithExtras*
imports *Main*
begin

datatype *natInf* = *Fin nat*
                | *Infty*   (‹∞›)
primrec
nat2inat :: *nat list* ⇒ *natInf list*
where
  nat2inat [] = [] |
  nat2inat (*x*#*xs*) = (*Fin x*) # (*nat2inat xs*)

end

3 Auxiliary Theory ListExtras.thy

theory *ListExtras*
imports *Main*
begin

definition
  disjoint :: *'a list* ⇒ *'a list* ⇒ *bool*
where
 disjoint x y ≡ (*set x*) ∩ (*set y*) = {}

primrec
  mem :: *'a* ⇒ *'a list* ⇒ *bool* (**infixr** ‹*mem*› *65*)
where
  x mem [] = *False* |
  x mem (*y* # *l*) = ((*x* = *y*) ∨ (*x mem l*))

definition
  memS :: *'a* ⇒ *'a list* ⇒ *bool*
where
 memS x l ≡ *x* ∈ (*set l*)

lemma *mem-memS-eq*: *x mem l* ≡ *memS x l*
proof (*induct l*)
  case *Nil*
  from *this* **show** *?case* **by** (*simp add*: *memS-def*)
next
    fix *a la* **case** (*Cons a la*)
    from *Cons* **show** *?case* **by** (*simp add*: *memS-def*)
 qed

lemma *mem-set-1*:
assumes *a mem l*

```
shows a ∈ set l
using assms by (metis memS-def mem-memS-eq)

lemma mem-set-2:
assumes a ∈ set l
shows a mem l
using assms by (metis (full-types) memS-def mem-memS-eq)

lemma set-inter-mem:
assumes x mem l1
    and x mem l2
shows set l1 ∩ set l2 ≠ {}
using assms  by (metis IntI empty-iff mem-set-1)

lemma mem-notdisjoint:
assumes x mem l1
    and x mem l2
shows ¬ disjoint l1 l2
using assms by (metis disjoint-def set-inter-mem)

lemma mem-notdisjoint2:
assumes h1:disjoint (schedule A) (schedule B)
    and h2:x mem schedule A
shows ¬ x mem schedule B
proof −
 { assume  x mem schedule B
   from h2 and this have ¬  disjoint (schedule A) (schedule B)
    by (simp add: mem-notdisjoint)
  from h1 and this have False by simp
 } then have ¬ x mem schedule B by blast
 then show ?thesis by simp
qed

lemma Add-Less:
assumes 0 < b
shows   (Suc a − b < Suc a) = True
using assms by auto

lemma list-length-hint1:
assumes l ≠ []
shows   0 < length l
using assms by simp

lemma list-length-hint1a:
assumes l ≠ []
shows   0 < length l
using assms by simp

lemma list-length-hint2:
```

assumes *length x* = *Suc 0*
shows [*hd x*] = *x*
using *assms*
by (*metis Zero-neq-Suc list.sel*(*1*) *length-Suc-conv neq-Nil-conv*)

lemma *list-length-hint2a*:
assumes *length l* = *Suc 0*
shows *tl l* = []
using *assms*
by (*metis list-length-hint2 list.sel*(*3*))

lemma *list-length-hint3*:
assumes *length l* = *Suc 0*
shows $l \neq$ []
using *assms*
by (*metis Zero-neq-Suc list.size*(*3*))

lemma *list-length-hint4*:
assumes *length x* $\leq$ *Suc 0*
  and $x \neq$ []
shows *length x* = *Suc 0*
using *assms*
by (*metis le-0-eq le-Suc-eq length-greater-0-conv less-numeral-extra*(*3*))

lemma *length-nonempty*:
assumes $x \neq$ []
shows *Suc 0* $\leq$ *length x*
using *assms*
by (*metis length-greater-0-conv less-eq-Suc-le*)

lemma *last-nth-length*:
assumes $x \neq$ []
shows *x* ! ((*length x*) − *Suc 0*) = *last x*
using *assms*
by (*metis One-nat-def last-conv-nth*)

lemma *list-nth-append0*:
assumes *i* < *length x*
shows *x* ! *i* = (*x* • *z*) ! *i*
using *assms*
by (*metis nth-append*)

lemma *list-nth-append1*:
assumes *i* < *length x*
shows (*b* # *x*) ! *i* = (*b* # *x* • *y*) ! *i*
proof −
 from *assms* **have** *i* < *length* (*b* # *x*) **by** *auto*
 then have *sg2*: (*b* # *x*) ! *i* = ((*b* # *x*) • *y*) ! *i*
  by (*rule list-nth-append0*)

```
  then show ?thesis by simp
qed

lemma list-nth-append2:
assumes i < Suc (length x)
shows   (b # x) ! i = (b # x • a # y) ! i
using assms
by (metis Cons-eq-appendI length-Suc-conv list-nth-append0)

lemma list-nth-append3:
assumes h1:¬ i < Suc (length x)
       and i − Suc (length x) < Suc (length y)
shows (a # y) ! (i − Suc (length x)) = (b # x • a # y) ! i
proof (cases i)
  assume i=0
  with h1 show ?thesis by (simp add: nth-append)
next
  fix ii assume i = Suc ii
  with h1 show ?thesis by (simp add: nth-append)
qed

lemma list-nth-append4:
assumes i < Suc (length x + length y)
       and ¬ i − Suc (length x) < Suc (length y)
shows False
using assms by arith

lemma list-nth-append5:
assumes i − length x < Suc (length y)
       and ¬ i − Suc (length x) < Suc (length y)
shows ¬ i < Suc (length x + length y)
using assms by arith

lemma list-nth-append6:
assumes ¬ i − length x < Suc (length y)
       and ¬ i − Suc (length x) < Suc (length y)
shows ¬ i < Suc (length x + length y)
using assms by arith

lemma list-nth-append6a:
assumes i < Suc (length x + length y)
       and ¬ i − length x < Suc (length y)
shows False
using assms by arith

lemma list-nth-append7:
assumes i − length x < Suc (length y)
       and i − Suc (length x) < Suc (length y)
shows   i < Suc (Suc (length x + length y))
```

using *assms* **by** *arith*

lemma *list-nth-append8*:
assumes $\neg$ *i* < *Suc* (*length x* + *length y*)
        and *i* < *Suc* (*Suc* (*length x* + *length y*))
shows   *i* = *Suc* (*length x* + *length y*)
using *assms* **by** *arith*

lemma *list-nth-append9*:
assumes *i* − *Suc* (*length x*) < *Suc* (*length y*)
shows   *i* < *Suc* (*Suc* (*length x* + *length y*))
using *assms* **by** *arith*

lemma *list-nth-append10*:
assumes $\neg$ *i* < *Suc* (*length x*)
        and $\neg$ *i* − *Suc* (*length x*) < *Suc* (*length y*)
shows $\neg$ *i* < *Suc* (*Suc* (*length x* + *length y*))
using *assms* **by** *arith*

end

4 Auxiliary arithmetic lemmas

theory *arith-hints*
imports *Main*
begin

lemma *arith-mod-neq*:
  assumes *a mod n* ≠ *b mod n*
  shows *a* ≠ *b*
  using *assms* **by** *blast*

lemma *arith-mod-nzero*:
  fixes *i* :: *nat*
  assumes *i* < *n* **and** *0* < *i*
  shows *0* < (*n* ∗ *t* + *i*) *mod n*
  using *assms* **by** *simp*

lemma *arith-mult-neq-nzero1*:
  fixes *i*::*nat*
  assumes *i* < *n*
        and *0* < *i*
  shows *i* + *n* ∗ *t* ≠ *n* ∗ *q*
proof −
  from *assms* **have** *sg1*:(*i* + *n* ∗ *t*) *mod n* = *i* **by** *auto*
  also have *sg2*:(*n* ∗ *q*) *mod n* = *0* **by** *simp*
  from *this* **and** *assms* **have** (*i* + *n* ∗ *t*) *mod n* ≠ (*n* ∗ *q*) *mod n*
    by *simp*
  from *this* **show** *?thesis* **by** (*rule arith-mod-neq*)

```
qed

lemma arith-mult-neq-nzero2:
  fixes i::nat
  assumes i < n
        and 0 < i
  shows n ∗ t + i ≠ n ∗ q
using assms
by (metis arith-mult-neq-nzero1 add.commute)

lemma arith-mult-neq-nzero3:
  fixes i::nat
  assumes i < n
        and 0 < i
  shows n + n ∗ t + i ≠ n ∗ qc
proof −
   from assms have sg1: n ∗(Suc t) + i  ≠ n ∗ qc
     by (rule arith-mult-neq-nzero2)
   have sg2: n + n ∗ t + i = n ∗(Suc t) + i by simp
   from sg1 and sg2  show ?thesis  by arith
qed

lemma arith-modZero1:
  (t + n ∗ t) mod Suc n = 0
by (metis mod-mult-self1-is-0 mult-Suc)

lemma arith-modZero2:
  Suc (n + (t + n ∗ t)) mod Suc n = 0
by (metis add-Suc-right add-Suc-shift mod-mult-self1-is-0 mult-Suc mult.commute)

lemma arith1:
  assumes h1:Suc n ∗ t = Suc n ∗ q
  shows t = q
using assms
by (metis mult-cancel2 mult.commute neq0-conv zero-less-Suc)

lemma arith2:
  fixes t n q :: nat
  assumes h1:t + n ∗ t = q + n ∗ q
  shows t = q
using assms
by (metis arith1 mult-Suc)

end
```

5 FOCUS streams: operators and lemmas

```
theory stream
  imports ListExtras ArithExtras
```

begin

5.1 Definition of the FOCUS stream types

type-synonym *'a fstream* = *'a list list*

— Infinite timed FOCUS stream
type-synonym *'a istream* = *nat* ⇒ *'a list*

— Infinite untimed FOCUS stream
type-synonym *'a iustream* = *nat* ⇒ *'a*

— FOCUS stream (general)
datatype *'a stream* =

```
        FinT 'a fstream — finite timed streams
      | FinU 'a list — finite untimed streams
      | InfT 'a istream — infinite timed streams
      | InfU 'a iustream — infinite untimed streams
```

5.2 Definitions of operators

definition
  infU-dom :: *natInf set*
 where
  infU-dom ≡ $\{x.\ \exists\ i.\ x = (Fin\ i)\} \cup \{\infty\}$

— domain of a finite untimed stream (using natural numbers enriched by Infinity)
definition
  finU-dom-natInf :: *'a list* ⇒ *natInf set*
  where
  finU-dom-natInf s ≡ $\{x.\ \exists\ i.\ x = (Fin\ i) \wedge i < (length\ s)\}$

— domain of a finite untimed stream
primrec
finU-dom :: *'a list* ⇒ *nat set*
where
  finU-dom [] = {} |
  finU-dom (*x#xs*) = {*length xs*} ∪ (*finU-dom xs*)

— range of a finite timed stream
primrec
  finT-range :: *'a fstream* ⇒ *'a set*
where
  finT-range [] = {} |
  finT-range (*x#xs*) = (*set x*) ∪ *finT-range xs*

— range of a finite untimed stream
definition
  finU-range :: *'a list* ⇒ *'a set*
where

finU-range $x \equiv$ *set* x

— range of an infinite timed stream
definition
 infT-range :: $'a$ *istream* $\Rightarrow$ $'a$ *set*
where
 infT-range $s \equiv \{y.\ \exists\ i{::}nat.\ y\ mem\ (s\ i)\}$

— range of a finite untimed stream
definition
 infU-range :: $(nat \Rightarrow 'a) \Rightarrow 'a\ set$
where
 infU-range $s \equiv \{\ y.\ \exists\ i{::}nat.\ y = (s\ i)\ \}$

— range of a (general) stream
definition
 stream-range :: $'a$ *stream* $\Rightarrow$ $'a$ *set*
where

```
stream-range s ≡ case s of
      FinT x ⇒ finT-range x
    | FinU x ⇒ finU-range x
    | InfT x ⇒ infT-range x
    | InfU x ⇒ infU-range x
```

— finite timed stream that consists of n empty time intervals
primrec
 nticks :: *nat* $\Rightarrow$ $'a$ *fstream*
where
 nticks 0 = [] |
 nticks (*Suc i*) = [] # (*nticks i*)

— removing the first element from an infinite stream
— in the case of an untimed stream: removing the first data element
— in the case of a timed stream: removing the first time interval
definition
 inf-tl :: $(nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a)$
where
 inf-tl $s \equiv (\lambda\ i.\ s\ (Suc\ i))$

— removing i first elements from an infinite stream s
— in the case of an untimed stream: removing i first data elements
— in the case of a timed stream: removing i first time intervals
definition
 inf-drop :: $nat \Rightarrow (nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a)$
where
 inf-drop $i\ s \equiv\ \lambda\ j.\ s\ (i{+}j)$

— finding the first nonempty time interval in a finite timed stream
primrec

```
 fin-find1nonemp :: 'a fstream ⇒ 'a list
where
 fin-find1nonemp [] = [] |
 fin-find1nonemp (x#xs) =
    ( if x = []
      then fin-find1nonemp xs
      else x )
```

— finding the first nonempty time interval in an infinite timed stream

```
definition
  inf-find1nonemp :: 'a istream ⇒ 'a list
where
 inf-find1nonemp s
  ≡
  ( if (∃ i. s i ≠ [])
    then s (LEAST i. s i ≠ [])
    else [] )
```

— finding the index of the first nonempty time interval in a finite timed stream

```
primrec
 fin-find1nonemp-index :: 'a fstream ⇒ nat
where
 fin-find1nonemp-index [] = 0 |
 fin-find1nonemp-index (x#xs) =
    ( if x = []
      then Suc (fin-find1nonemp-index xs)
      else 0 )
```

— finding the index of the first nonempty time interval in an infinite timed stream

```
definition
  inf-find1nonemp-index :: 'a istream ⇒ nat
where
 inf-find1nonemp-index s
  ≡
  ( if (∃ i. s i ≠ [])
    then (LEAST i. s i ≠ [])
    else 0 )
```

— length of a finite timed stream: number of data elements in this stream

```
primrec
  fin-length :: 'a fstream ⇒ nat
where
  fin-length [] = 0 |
  fin-length (x#xs) = (length x) + (fin-length xs)
```

— length of a (general) stream

```
definition
   stream-length :: 'a stream ⇒ natInf
where
```

```
  stream-length s ≡
      case s of
                (FinT x) ⇒ Fin (fin-length x)
              | (FinU x) ⇒ Fin (length x)
              | (InfT x) ⇒ ∞
              | (InfU x) ⇒ ∞
```

— removing the first k elements from a finite (nonempty) timed stream

```
axiomatization
  fin-nth :: 'a fstream ⇒ nat ⇒ 'a
where
   fin-nth-Cons:
  fin-nth (hds # tls) k =
      ( if hds = []
        then fin-nth tls k
        else ( if (k < (length hds))
               then nth hds k
               else fin-nth tls (k − length hds) ))
```

— removing i first data elements from an infinite timed stream s

```
primrec
  inf-nth :: 'a istream ⇒ nat ⇒ 'a
where
 inf-nth s 0 =  hd (s (LEAST i.(s i) ≠ []))  |
 inf-nth s (Suc k) =
      ( if ((Suc k) < (length (s 0)))
        then (nth (s 0) (Suc k))
        else ( if (s 0) = []
               then (inf-nth (inf-tl (inf-drop
                    (LEAST i. (s i) ≠ []) s)) k )
               else inf-nth (inf-tl s) k ))
```

— removing the first k data elements from a (general) stream

```
definition
    stream-nth :: 'a stream ⇒ nat ⇒ 'a
where
   stream-nth s k ≡
         case s of (FinT x) ⇒ fin-nth x k
                 | (FinU x) ⇒ nth x k
                 | (InfT x) ⇒ inf-nth x k
                 | (InfU x) ⇒ x k
```

— prefix of an infinite stream

```
primrec
  inf-prefix :: 'a list ⇒ (nat ⇒ 'a) ⇒ nat ⇒ bool
where
  inf-prefix [] s k = True |
  inf-prefix (x#xs) s k = ( (x = (s k)) ∧ (inf-prefix xs s (Suc k)) )
```

— prefix of a finite stream

```
primrec
  fin-prefix :: 'a list ⇒ 'a list ⇒ bool
where
  fin-prefix [] s = True |
  fin-prefix (x#xs) s =
     (if (s = [])
      then False
      else (x = (hd s)) ∧ (fin-prefix xs s) )
```

— prefix of a (general) stream

```
definition
   stream-prefix :: 'a stream ⇒ 'a stream ⇒ bool
where
  stream-prefix p s ≡
   (case p of
       (FinT x) ⇒
       (case s of (FinT y) ⇒  (fin-prefix x y)
               | (FinU y) ⇒ False
               | (InfT y) ⇒ inf-prefix x y 0
               | (InfU y) ⇒ False )
     | (FinU x) ⇒
       (case s of (FinT y) ⇒ False
               | (FinU y) ⇒  (fin-prefix x y)
               | (InfT y) ⇒ False
               | (InfU y) ⇒  inf-prefix x y 0 )
     | (InfT x) ⇒
       (case s of (FinT y) ⇒ False
               | (FinU y) ⇒ False
               | (InfT y) ⇒ (∀ i. x i = y i)
               | (InfU y) ⇒ False )
     | (InfU x) ⇒
       (case s of (FinT y) ⇒ False
               | (FinU y) ⇒ False
               | (InfT y) ⇒ False
               | (InfU y) ⇒ (∀ i. x i = y i) )  )
```

— truncating a finite stream after the n-th element

```
primrec
fin-truncate :: 'a list ⇒ nat ⇒ 'a list
where
  fin-truncate [] n = [] |
  fin-truncate (x#xs) i =
      (case i of 0 ⇒ []
         | (Suc n) ⇒ x # (fin-truncate xs n))
```

— truncating a finite stream after the n-th element
— n is of type of natural numbers enriched by Infinity

```
definition
```

```
 fin-truncate-plus :: 'a list ⇒ natInf ⇒ 'a list
where
fin-truncate-plus s n
 ≡
 case n of (Fin i) ⇒ fin-truncate s i
         | ∞      ⇒ s
```

— truncating an infinite stream after the n-th element
primrec

```
  inf-truncate :: (nat ⇒ 'a) ⇒ nat ⇒ 'a list
where
   inf-truncate s 0 = [ s 0 ] |
   inf-truncate s (Suc k) = (inf-truncate s k) • [s (Suc k)]
```

— truncating an infinite stream after the n-th element
— n is of type of natural numbers enriched by Infinity
definition

```
  inf-truncate-plus :: 'a istream ⇒ natInf ⇒ 'a stream
 where
 inf-truncate-plus s n
  ≡
  case n of (Fin i) ⇒ FinT (inf-truncate s i)
          | ∞      ⇒ InfT s
```

— concatanation of a finite and an infinite stream
definition

```
    fin-inf-append ::
        'a list ⇒ (nat ⇒ 'a) ⇒ (nat ⇒ 'a)
where
fin-inf-append us s ≡
 (λ i. ( if (i < (length us))
         then (nth us i)
         else s (i − (length us)) ))
```

— insuring that the infinite timed stream is time-synchronous
definition

```
   ts :: 'a istream ⇒ bool
where
ts s ≡ ∀ i. (length (s i) = 1)
```

— insuring that each time interval of an infinite timed stream contains at most n data elements
definition

```
  msg :: nat ⇒ 'a istream ⇒ bool
where
 msg n s ≡ ∀ t. length (s t) ≤ n
```

— insuring that each time interval of a finite timed stream contains at most n data elements

primrec
  fin-msg :: *nat* ⇒ *'a list list* ⇒ *bool*
where
 fin-msg n [] = *True* |
 fin-msg n (*x*#*xs*) = (((*length x*) ≤ *n*) ∧ (*fin-msg n xs*))

— making a finite timed stream to a finite untimed stream
definition
   fin-make-untimed :: *'a fstream* ⇒ *'a list*
where
  fin-make-untimed x ≡ *concat x*

— making an infinite timed stream to an infinite untimed stream
— (auxiliary function)
primrec
  inf-make-untimed1 :: *'a istream* ⇒ *nat* ⇒ *'a*
where
inf-make-untimed1-0:
  inf-make-untimed1 s 0 = *hd* (*s* (*LEAST i*.(*s i*) ≠ [])) |
inf-make-untimed1-Suc:
  inf-make-untimed1 s (*Suc k*) =
    (*if* ((*Suc k*) < *length* (*s 0*))
      then nth (*s 0*) (*Suc k*)
      else (*if* (*s 0*) = []
            then (*inf-make-untimed1* (*inf-tl* (*inf-drop*
                    (*LEAST i*. ∀ *j*. *j* < *i* ⟶ (*s j*) = [])
                     s)) *k*)
            else inf-make-untimed1 (*inf-tl s*) *k*))

— making an infinite timed stream to an infinite untimed stream
— (main function)
definition
   inf-make-untimed :: *'a istream* ⇒ (*nat* ⇒ *'a*)
where
  inf-make-untimed s
  ≡
  λ *i*. *inf-make-untimed1 s i*

— making a (general) stream untimed
definition
    make-untimed :: *'a stream* ⇒ *'a stream*
where
   make-untimed s ≡
      case s of (*FinT x*) ⇒ *FinU* (*fin-make-untimed x*)
             | (*FinU x*) ⇒ *FinU x*
             | (*InfT x*) ⇒
               (*if* (∃ *i*.∀ *j*. *i* < *j* ⟶ (*x j*) = [])
                then FinU (*fin-make-untimed* (*inf-truncate x*
                      (*LEAST i*.∀ *j*. *i* < *j* ⟶ (*x j*) = [])))

```
                else InfU (inf-make-untimed x))
            | (InfU x) ⇒ InfU x


— finding the index of the time interval that contains the k-th data element
— defined over a finite timed stream
primrec
  fin-tm :: 'a fstream ⇒ nat ⇒ nat
where
  fin-tm [] k = k |
  fin-tm (x#xs) k =
    (if k = 0
     then 0
     else (if (k ≤ length x)
           then (Suc 0)
           else Suc(fin-tm xs (k − length x))))

— auxiliary lemma for the definition of the truncate operator
lemma inf-tm-hint1:
  assumes i2 = Suc i − length a
      and ¬ Suc i ≤ length a
      and a ≠ []
  shows i2 < Suc i
using assms
by auto


— filtering a finite timed stream
definition
   finT-filter :: 'a set => 'a fstream => 'a fstream
where
  finT-filter m s ≡  map (λ s. filter (λ y. y ∈ m) s) s

— filtering an infinite timed stream
definition
   infT-filter :: 'a set => 'a istream => 'a istream
where
  infT-filter m s ≡  (λi.( filter (λ x. x ∈ m) (s i)))

— removing duplications from a finite timed stream
definition
   finT-remdups :: 'a fstream => 'a fstream
where
  finT-remdups s ≡  map (λ s. remdups s) s

— removing duplications from an infinite timed stream
definition
   infT-remdups :: 'a istream => 'a istream
where
```

infT-remdups s ≡ (λ*i*.(*remdups* (*s i*)))

— removing duplications from a time interval of a stream
primrec
  fst-remdups :: *'a list* ⇒ *'a list*
where
 fst-remdups [] = [] |
 fst-remdups (*x#xs*) =
    (*if xs* = []
     then [*x*]
     else (*if x* = (*hd xs*)
           then fst-remdups xs
           else (*x#xs*)))

— time interval operator
definition
  ti :: *'a fstream* ⇒ *nat* ⇒ *'a list*
where
 ti s i ≡
    (*if s* = []
     then []
     else (*nth s i*))

— insuring that a sheaf of channels is correctly defined
definition
  CorrectSheaf :: *nat* ⇒ *bool*
where
  CorrectSheaf n ≡ *0* < *n*

— insuring that all channels in a sheaf are disjunct
— indices in the sheaf are represented using an extra specified set
definition
  inf-disjS :: *'b set* ⇒ (*'b* ⇒ *'a istream*) ⇒ *bool*
where
  inf-disjS IdSet nS
  ≡
  ∀ (*t*::*nat*) *i j*. (*i*:*IdSet*) ∧ (*j*:*IdSet*) ∧
  ((*nS i*) *t*) ≠ [] ⟶ ((*nS j*) *t*) = []

— insuring that all channels in a sheaf are disjunct
— indices in the sheaf are represented using natural numbers
definition
  inf-disj :: *nat* ⇒ (*nat* ⇒ *'a istream*) ⇒ *bool*
where
  inf-disj n nS
  ≡
  ∀ (*t*::*nat*) (*i*::*nat*) (*j*::*nat*).
  i < *n* ∧ *j* < *n* ∧ *i* ≠ *j* ∧ ((*nS i*) *t*) ≠ [] ⟶
  ((*nS j*) *t*) = []

```
— taking the prefix of n data elements from a finite timed stream
— (defined over natural numbers)
fun fin-get-prefix  :: ('a fstream × nat) ⇒ 'a fstream
where
  fin-get-prefix([], n) = [] |
  fin-get-prefix(x#xs, i) =
     ( if (length x) < i
       then x # fin-get-prefix(xs, (i − (length x)))
       else [take i x] )

— taking the prefix of n data elements from a finite timed stream
— (defined over natural numbers enriched by Infinity)
definition
  fin-get-prefix-plus :: 'a fstream ⇒ natInf ⇒ 'a fstream
where
 fin-get-prefix-plus s n
  ≡
  case n of (Fin i) ⇒ fin-get-prefix(s, i)
          | ∞       ⇒ s

— auxiliary lemmas
lemma length-inf-drop-hint1:
  assumes s k ≠ []
  shows    length (inf-drop k s 0) ≠ 0
using assms
by (auto simp: inf-drop-def)

lemma length-inf-drop-hint2:
(s 0 ≠ [] ⟶ length (inf-drop 0 s 0) < Suc i
  ⟶ Suc i − length (inf-drop 0 s 0) < Suc i)
  by (simp add: inf-drop-def list-length-hint1)

— taking the prefix of n data elements from an infinite timed stream
— (defined over natural numbers)
fun infT-get-prefix  :: ('a istream × nat) ⇒ 'a fstream
where
  infT-get-prefix(s, 0) = []
|
  infT-get-prefix(s, Suc i) =
    ( if (s 0) = []
      then ( if (∀ i. s i = [])
             then []
             else (let
                      k = (LEAST k. s k ≠ [] ∧ (∀ i. i < k ⟶ s i = []));
                      s2 = inf-drop (k+1) s
                   in (if (length (s k)=0)
                       then []
                       else (if (length (s k) < (Suc i))
                        then s k # infT-get-prefix (s2,Suc i − length (s k))
                        else [take (Suc i) (s k)] )))
     )
   else
     (if ((length (s 0)) < (Suc i))
      then (s 0) # infT-get-prefix( inf-drop 1 s, (Suc i) − (length (s 0)))
      else [take (Suc i) (s 0)]
      )
  )
```

— taking the prefix of n data elements from an infinite untimed stream
— (defined over natural numbers)

```
primrec
  infU-get-prefix :: (nat ⇒ 'a) ⇒ nat ⇒ 'a list
where
  infU-get-prefix s 0 = [] |
  infU-get-prefix s (Suc i)
       = (infU-get-prefix s i) • [s i]
```

— taking the prefix of n data elements from an infinite timed stream
— (defined over natural numbers enriched by Infinity)

```
definition
  infT-get-prefix-plus :: 'a istream ⇒ natInf ⇒ 'a stream
where
infT-get-prefix-plus s n
  ≡
  case n of (Fin i) ⇒ FinT (infT-get-prefix(s, i))
          | ∞      ⇒ InfT s
```

— taking the prefix of n data elements from an infinite untimed stream
— (defined over natural numbers enriched by Infinity)

```
definition
  infU-get-prefix-plus :: (nat ⇒ 'a) ⇒ natInf ⇒ 'a stream
where
 infU-get-prefix-plus s n
  ≡
  case n of (Fin i) ⇒ FinU (infU-get-prefix s i)
          | ∞      ⇒ InfU s
```

— taking the prefix of n data elements from an infinite stream
— (defined over natural numbers enriched by Infinity)

```
definition
  take-plus :: natInf ⇒ 'a list ⇒ 'a list
where
 take-plus n s
  ≡
  case n of (Fin i) ⇒ (take i s)
          | ∞       ⇒ s
```

— taking the prefix of n data elements from a (general) stream
— (defined over natural numbers enriched by Infinity)
definition
```
  get-prefix :: 'a stream ⇒ natInf ⇒ 'a stream
```
where
```
  get-prefix s k ≡
       case s of  (FinT x) ⇒ FinT (fin-get-prefix-plus x k)
                | (FinU x) ⇒ FinU (take-plus k x)
                | (InfT x) ⇒ infT-get-prefix-plus x k
                | (InfU x) ⇒ infU-get-prefix-plus x k
```

— merging time intervals of two finite timed streams
primrec
```
  fin-merge-ti :: 'a fstream ⇒ 'a fstream ⇒ 'a fstream
```
where
```
 fin-merge-ti [] y = y |
 fin-merge-ti (x#xs) y =
   ( case y of [] ⇒ (x#xs)
        | (z#zs) ⇒ (x•z) # (fin-merge-ti xs zs))
```

— merging time intervals of two infinite timed streams
definition
```
 inf-merge-ti :: 'a istream ⇒ 'a istream ⇒ 'a istream
```
where
```
 inf-merge-ti x y
  ≡
  λ i. (x i)•(y i)
```

— the last time interval of a finite timed stream
primrec
```
  fin-last-ti :: ('a list) list ⇒ nat ⇒ 'a list
```
where
```
 fin-last-ti s 0 = hd s |
 fin-last-ti s (Suc i) =
    ( if s!(Suc i) ≠ []
      then s!(Suc i)
      else fin-last-ti s i)
```

— the last nonempty time interval of a finite timed stream
— (can be applied to the streams which time intervals are empty from some moment)
primrec
```
  inf-last-ti :: 'a istream ⇒ nat ⇒ 'a list
```
where
```
 inf-last-ti s 0 = s 0 |
 inf-last-ti s (Suc i) =
    ( if s (Suc i) ≠ []
      then s (Suc i)
      else inf-last-ti s i)
```

5.3 Properties of operators

lemma *inf-last-ti-nonempty-k*:
  assumes *inf-last-ti dt t* $\neq$ []
  shows *inf-last-ti dt* (*t* + *k*) $\neq$ []
using *assms*
by (*induct k*, *auto*)

lemma *inf-last-ti-nonempty*:
  assumes *s t* $\neq$ []
  shows *inf-last-ti s* (*t* + *k*) $\neq$ []
using *assms*
by (*induct k*, *auto*, *induct t*, *auto*)

lemma *arith-sum-t2k*:
t + *2* + *k* = (*Suc t*) + (*Suc k*)
by *arith*

lemma *inf-last-ti-Suc2*:
  assumes *dt* (*Suc t*) $\neq$ [] $\vee$ *dt* (*Suc* (*Suc t*)) $\neq$ []
  shows *inf-last-ti dt* (*t* + *2* + *k*) $\neq$ []
proof (*cases dt* (*Suc t*) $\neq$ [])
  assume *a1*:*dt* (*Suc t*) $\neq$ []
  from *a1* **have** *sg2*:*inf-last-ti dt* ((*Suc t*) + (*Suc k*)) $\neq$ []
    by (*rule inf-last-ti-nonempty*)
  from *sg2* **show** *?thesis* **by** (*simp add*: *arith-sum-t2k*)
next
  assume *a2*:$\neg$ *dt* (*Suc t*) $\neq$ []
  from *a2* **and** *assms* **have** *sg1*:*dt* (*Suc* (*Suc t*)) $\neq$ [] **by** *simp*
  from *sg1* **have** *sg2*:*inf-last-ti dt* (*Suc* (*Suc t*) + *k*) $\neq$ []
    by (*rule inf-last-ti-nonempty*)
  from *sg2* **show** *?thesis* **by** *auto*
qed

5.3.1 Lemmas for concatenation operator

lemma *fin-length-append*:
  fin-length (*x*•*y*) = (*fin-length x*) + (*fin-length y*)
by (*induct x*, *auto*)

lemma *fin-append-Nil*: *fin-inf-append* [] *z* = *z*
by (*simp add*: *fin-inf-append-def*)

lemma *correct-fin-inf-append1*:
  assumes *s1* = *fin-inf-append* [*x*] *s*
  shows *s1* (*Suc i*) = *s i*
using *assms*
by (*simp add*: *fin-inf-append-def*)

lemma *correct-fin-inf-append2*:

```
  fin-inf-append [x] s (Suc i) = s i
by (simp add: fin-inf-append-def)

lemma fin-append-com-Nil1:
  fin-inf-append [] (fin-inf-append y z)
   = fin-inf-append ([] • y) z
by (simp add: fin-append-Nil)

lemma fin-append-com-Nil2:
  fin-inf-append x (fin-inf-append [] z)
  = fin-inf-append (x • []) z
by (simp add: fin-append-Nil)

lemma fin-append-com-i:
  fin-inf-append x (fin-inf-append y z) i = fin-inf-append (x • y) z i
proof (cases x)
  assume Nil:x = []
  thus ?thesis by (simp add: fin-append-com-Nil1)
next
  fix a l assume Cons:x = a # l
  thus ?thesis
  proof (cases y)
    assume y = []
    thus ?thesis by (simp add: fin-append-com-Nil2)
  next
    fix aa la assume Cons2:y = aa # la
    show ?thesis
    apply (simp add: fin-inf-append-def, auto, simp add: list-nth-append0)
    by (simp add: nth-append)
  qed
qed
```

5.3.2 Lemmas for operators *ts* and *msg*

```
lemma ts-msg1:
  assumes ts p
  shows    msg 1 p
using assms
by (simp add: ts-def msg-def)

lemma ts-inf-tl:
  assumes ts x
  shows    ts (inf-tl x)
using assms
by (simp add: ts-def inf-tl-def)

lemma ts-length-hint1:
 assumes ts x
 shows    x i ≠ []
```

```
proof –
  from assms have sg1:length (x i) = Suc 0  by (simp add: ts-def)
  thus ?thesis by auto
qed

lemma ts-length-hint2:
 assumes ts x
 shows    length (x i) = Suc (0::nat)
using assms
by (simp add: ts-def)

lemma ts-Least-0:
  assumes ts x
  shows    (LEAST i. (x i) ≠ [] ) = (0::nat)
proof –
  from assms have sg1:x 0 ≠ [] by (rule ts-length-hint1)
  thus ?thesis
  apply (simp add: Least-def)
  by auto
qed

lemma inf-tl-Suc: inf-tl x i = x (Suc i)
by (simp add: inf-tl-def)

lemma ts-Least-Suc0:
  assumes ts x
  shows    (LEAST i. x (Suc i) ≠ []) = 0
proof –
  from assms have x (Suc 0) ≠ [] by (simp add: ts-length-hint1)
  thus ?thesis by (simp add: Least-def, auto)
qed

lemma ts-inf-make-untimed-inf-tl:
  assumes ts x
  shows     inf-make-untimed (inf-tl x) i = inf-make-untimed x (Suc i)
using assms
apply (simp add: inf-make-untimed-def)
by (metis Suc-less-eq gr-implies-not0 ts-length-hint1 ts-length-hint2)

lemma ts-inf-make-untimed1-inf-tl:
  assumes ts x
  shows    inf-make-untimed1 (inf-tl x) i = inf-make-untimed1 x (Suc i)
using assms
by (metis inf-make-untimed-def ts-inf-make-untimed-inf-tl)

lemma msg-nonempty1:
  assumes h1:msg (Suc 0) a
        and h2:a t = aa # l
  shows l = []
```

proof –
  from *h1* **have** *length* (*a t*) ≤ *Suc 0* **by** (*simp add*: *msg-def*)
  from *h2* **and** *this* **show** *?thesis* **by** *auto*
qed

lemma *msg-nonempty2*:
  assumes *h1*:*msg* (*Suc 0*) *a*
        and *h2*:*a t* ≠ []
  shows *length* (*a t*) = (*Suc 0*)
proof –
  from *h1* **have** *sg1*:*length* (*a t*) ≤ *Suc 0* **by** (*simp add*: *msg-def*)
  from *h2* **have** *sg2*:*length* (*a t*) ≠ *0* **by** *auto*
  from *sg1* **and** *sg2* **show** *?thesis* **by** *arith*
qed

5.3.3 Lemmas for *inf_truncate*

lemma *inf-truncate-nonempty*:
  assumes *z i* ≠ []
  shows   *inf-truncate z i* ≠ []
proof (*induct i*)
    case *0*
    from *assms* **show** *?case* **by** *auto*
  next
    case (*Suc i*)
     from *assms* **show** *?case* **by** *auto*
qed

lemma *concat-inf-truncate-nonempty*:
  assumes *z i* ≠ []
  shows   *concat* (*inf-truncate z i*) ≠ []
using *assms*
proof (*induct i*)
    case *0*
    thus *?case* **by** *auto*
  next
    case (*Suc i*)
    thus *?case* **by** *auto*
qed

lemma *concat-inf-truncate-nonempty-a*:
  assumes *z i* = [*a*]
  shows   *concat* (*inf-truncate z i*) ≠ []
using *assms*
by (*metis concat-inf-truncate-nonempty list.distinct*(*1*))

lemma *concat-inf-truncate-nonempty-el*:
  assumes *z i* ≠ []

  shows *concat* (*inf-truncate z i*) $\neq$ []
using *assms*
by (*metis concat-inf-truncate-nonempty*)

lemma *inf-truncate-append*:
  (*inf-truncate z i* • [*z* (*Suc i*)]) = *inf-truncate z* (*Suc i*)
by (*metis inf-truncate.simps*(*2*))

5.3.4 Lemmas for *fin_make_untimed*

```
lemma fin-make-untimed-append:
  assumes fin-make-untimed x ≠ []
  shows   fin-make-untimed (x • y) ≠ []
using assms by (simp add: fin-make-untimed-def)


lemma fin-make-untimed-inf-truncate-Nonempty:
  assumes z k ≠ []
       and k ≤ i
  shows fin-make-untimed (inf-truncate z i) ≠ []
using assms
  apply (simp add: fin-make-untimed-def)
  proof (induct i)
    case 0
    thus ?case by auto
  next
    case (Suc i)
    thus ?case
    proof cases
      assume k ≤ i
      from Suc and this show ∃ xs∈set (inf-truncate z (Suc i)). xs ≠ []
        by auto
    next
      assume ¬ k ≤ i
      from Suc and this have k = Suc i by arith
      from Suc and this show ∃ xs∈set (inf-truncate z (Suc i)). xs ≠ []
        by auto
     qed
qed

lemma last-fin-make-untimed-append:
  last (fin-make-untimed (z • [[a]])) = a
by (simp add: fin-make-untimed-def)

lemma last-fin-make-untimed-inf-truncate:
  assumes z i = [a]
  shows   last (fin-make-untimed (inf-truncate z i)) = a
using assms
proof (induction i)
```

  case *0* **thus** *?case* **by** (*simp add*: *fin-make-untimed-def*)
next
  case (*Suc i*) **thus** *?case* **by** (*simp add*: *fin-make-untimed-def*)
qed

lemma *fin-make-untimed-append-empty*:
 fin-make-untimed (*z* • [[]]) = *fin-make-untimed z*
by (*simp add*: *fin-make-untimed-def*)

lemma *fin-make-untimed-inf-truncate-append-a*:
 fin-make-untimed (*inf-truncate z i* • [[*a*]]) !
 (*length* (*fin-make-untimed* (*inf-truncate z i* • [[*a*]])) − *Suc 0*) = *a*
by (*simp add*: *fin-make-untimed-def*)

lemma *fin-make-untimed-inf-truncate-Nonempty-all*:
 assumes $z\ k \neq []$
 shows $\forall\ i.\ k \leq i \longrightarrow$ *fin-make-untimed* (*inf-truncate z i*) ≠ []
using *assms* **by** (*simp add*: *fin-make-untimed-inf-truncate-Nonempty*)

lemma *fin-make-untimed-inf-truncate-Nonempty-all0*:
 assumes $z\ 0 \neq []$
 shows ∀ *i*. *fin-make-untimed* (*inf-truncate z i*) ≠ []
using *assms* **by** (*simp add*: *fin-make-untimed-inf-truncate-Nonempty*)

lemma *fin-make-untimed-inf-truncate-Nonempty-all0a*:
 assumes $z\ 0 = [a]$
 shows ∀ *i*. *fin-make-untimed* (*inf-truncate z i*) ≠ []
using *assms* **by** (*simp add*: *fin-make-untimed-inf-truncate-Nonempty-all0*)

lemma *fin-make-untimed-inf-truncate-Nonempty-all-app*:
 assumes $z\ 0 = [a]$
 shows ∀ *i*. *fin-make-untimed* (*inf-truncate z i* • [*z* (*Suc i*)]) ≠ []
proof
 fix *i*
 from *assms* **have** *fin-make-untimed* (*inf-truncate z i*) ≠ []
  by (*simp add*: *fin-make-untimed-inf-truncate-Nonempty-all0a*)
 from *assms* **and** *this* **show**
  fin-make-untimed (*inf-truncate z i* • [*z* (*Suc i*)]) ≠ []
  by (*simp add*: *fin-make-untimed-append*)
qed

lemma *fin-make-untimed-nth-length*:
 assumes $z\ i = [a]$
 shows
 fin-make-untimed (*inf-truncate z i*) !
   (*length* (*fin-make-untimed* (*inf-truncate z i*)) − *Suc 0*)
  = *a*
proof −
from *assms* **have** *sg1*:*last* (*fin-make-untimed* (*inf-truncate z i*)) = *a*

```
  by (simp add: last-fin-make-untimed-inf-truncate)
from assms have sg2:concat (inf-truncate z i) ≠ []
  by (rule concat-inf-truncate-nonempty-a)
from assms and sg1 and sg2 show ?thesis
  by (simp add: fin-make-untimed-def last-nth-length)
qed
```

5.3.5 Lemmas for *inf_disj* and *inf_disjS*

```
lemma inf-disj-index:
  assumes h1:inf-disj n nS
       and nS k t ≠ []
       and k < n
  shows  (SOME i. i < n ∧  nS i t ≠ []) = k
proof –
  from h1 have ∀ j. k < n ∧ j < n ∧ k ≠ j ∧ nS k t ≠ [] ⟶ nS j t = []
    by (simp add: inf-disj-def, auto)
  from this and assms show ?thesis by auto
qed

lemma inf-disjS-index:
  assumes h1:inf-disjS IdSet nS
     and k:IdSet
     and nS k t ≠ []
  shows (SOME i. (i:IdSet) ∧ nSend i t ≠ []) = k
proof –
  from h1 have ∀ j. k ∈ IdSet ∧ j ∈ IdSet ∧ nS k t ≠ [] ⟶ nS j t = []
    by (simp add: inf-disjS-def, auto)
  from this and assms show ?thesis by auto
qed

end
```

6 Properties of time-synchronous streams of types bool and bit

```
theory BitBoolTS
imports Main stream
begin

datatype bit = Zero | One

primrec
   negation :: bit ⇒ bit
where
  negation Zero = One |
  negation One = Zero
```

```
lemma ts-bit-stream-One:
  assumes h1:ts x
      and h2:x i ≠ [Zero]
  shows x i = [One]
proof -
  from h1 have sg1:length (x i) = Suc 0
    by (simp add: ts-def)
  from this and h2 show ?thesis
  proof (cases x i)
    assume Nil:x i = []
    from this and sg1 show ?thesis by simp
  next
  fix a l assume Cons:x i = a # l
    from this and sg1 and h2 show ?thesis
    proof (cases a)
      assume a = Zero
      from this and sg1 and h2 and Cons show ?thesis by auto
    next
      assume a = One
      from this and sg1 and Cons show ?thesis by auto
    qed
  qed
qed

lemma ts-bit-stream-Zero:
  assumes h1:ts x
      and h2:x i ≠ [One]
  shows x i = [Zero]
proof -
  from h1 have sg1:length (x i) = Suc 0
    by (simp add: ts-def)
  from this and h2 show ?thesis
  proof (cases x i)
    assume Nil:x i = []
    from this and sg1 show ?thesis by simp
  next
  fix a l assume Cons:x i = a # l
    from this and sg1 and h2 show ?thesis
    proof (cases a)
      assume a = Zero
      from this and sg1 and Cons show ?thesis by auto
    next
      assume a = One
      from this and sg1 and h2 and Cons show ?thesis by auto
    qed
  qed
qed
```

```
lemma ts-bool-True:
  assumes h1:ts x
        and h2:x i ≠ [False]
  shows x i = [True]
proof −
  from h1 have sg1:length (x i) = Suc 0
    by (simp add: ts-def)
  from this and h2 show ?thesis
  proof (cases x i)
    assume Nil:x i = []
    from this and sg1 show ?thesis by simp
  next
  fix a l assume Cons:x i = a # l
    from this and sg1 have x i = [a] by simp
    from this and h2 show ?thesis by auto
  qed
qed

lemma ts-bool-False:
  assumes h1:ts x
        and h2:x i ≠ [True]
  shows x i = [False]
proof −
  from h1 have sg1:length (x i) = Suc 0
    by (simp add: ts-def)
  from this and h2 show ?thesis
  proof (cases x i)
    assume Nil:x i = []
    from this and sg1 show ?thesis by simp
  next
  fix a l assume Cons:x i = a # l
    from this and sg1 have x i = [a] by simp
    from this and h2 show ?thesis by auto
  qed
qed

lemma ts-bool-True-False:
  fixes x::bool istream
  assumes ts x
  shows x i = [True] ∨ x i = [False]
proof (cases x i = [True])
  assume x i = [True]
  from this and assms show ?thesis by simp
next
  assume x i ≠ [True]
  from this and assms show ?thesis by (simp add: ts-bool-False)
qed

end
```

7 Changing time granularity of the streams

theory *JoinSplitTime*
imports *stream arith-hints*
begin

7.1 Join time units

```
primrec
  join-ti ::'a istream ⇒ nat ⇒ nat ⇒ 'a list
where
join-ti-0:
 join-ti s x 0 = s x |
join-ti-Suc:
 join-ti s x (Suc i) = (join-ti s x i) • (s (x + (Suc i)))

primrec
  fin-join-ti ::'a fstream ⇒ nat ⇒ nat ⇒ 'a list
where
fin-join-ti-0:
 fin-join-ti s x 0 = nth s x |
fin-join-ti-Suc:
 fin-join-ti s x (Suc i) = (fin-join-ti s x i) • (nth s (x + (Suc i)))

definition
  join-time ::'a istream ⇒ nat ⇒ 'a istream
where
 join-time s n t ≡
  (case n of
       0 ⇒ []
  |(Suc i) ⇒ join-ti s (n∗t) i)

lemma join-ti-hint1:
  assumes join-ti s x (Suc i) = []
  shows   join-ti s x i = []
using assms by auto

lemma join-ti-hint2:
  assumes join-ti s x (Suc i) = []
  shows   s (x + (Suc i)) = []
using assms by auto

lemma join-ti-hint3:
  assumes join-ti s x (Suc i) = []
  shows   s (x + i) = []
using assms by (induct i, auto)

lemma join-ti-empty-join:
  assumes i ≤ n
      and join-ti s x n = []
  shows      s (x+i) = []
using assms
proof (induct n)
  case 0 then show ?case by auto
next
  case (Suc n) then show ?case
  by (metis join-ti-hint1 join-ti-hint2 le-SucE)
qed

lemma join-ti-empty-ti:
  assumes ∀ i ≤ n. s (x+i) = []
  shows    join-ti s x n = []
using assms by (induct n, auto)

lemma join-ti-1nempty:
  assumes ∀ i. 0 < i ∧ i < Suc n ⟶ s (x+i) = []
  shows    join-ti s x n = s x
using assms by (induct n, auto)

lemma join-time1t: ∀ t. join-time s (1::nat) t = s t
by (simp add: join-time-def)


lemma join-time1: join-time s 1 = s
by (simp add: fun-eq-iff join-time-def)

lemma join-time-empty1:
  assumes h1:i < n
         and h2:join-time s n t = []
  shows      s (n∗t + i) = []
proof (cases n)
  assume n = 0
  from assms and this show ?thesis by (simp add: join-time-def)
next
  fix x
  assume a2:n = Suc x
  from assms and a2 have sg1:join-ti s (t + x ∗ t) x = []
    by (simp add: join-time-def)
  from a2 and h1 have i ≤ x by simp
  from this and sg1 and a2 show ?thesis by (simp add: join-ti-empty-join)
qed

lemma fin-join-ti-hint1:
  assumes fin-join-ti s x (Suc i) = []
  shows    fin-join-ti s x i = []
using assms by auto

lemma fin-join-ti-hint2:
  assumes fin-join-ti s x (Suc i) = []
```

  shows   *nth s* (*x* + (*Suc i*)) = []
using *assms* **by** *auto*

lemma *fin-join-ti-hint3*:
  assumes *fin-join-ti s x* (*Suc i*) = []
  shows   *nth s* (*x* + *i*) = []
using *assms* **by** (*induct i*, *auto*)

lemma *fin-join-ti-empty-join*:
  assumes *i* ≤ *n*
       and *fin-join-ti s x n* = []
  shows   *nth s* (*x*+*i*) = []
using *assms*
proof (*induct n*)
  case *0* **then show** *?case* **by** *auto*
next
  case (*Suc n*) **then show** *?case*
  proof (*cases i* = *Suc n*)
    assume *i* = *Suc n*
    from *Suc* **and** *this* **show** *?thesis* **by** *simp*
  next
    assume *i* ≠ *Suc n*
    from *Suc* **and** *this* **show** *?thesis* **by** *simp*
  qed
qed

lemma *fin-join-ti-empty-ti*:
  assumes ∀ *i* ≤ *n*. *nth s* (*x*+*i*) = []
  shows   *fin-join-ti s x n* = []
using *assms* **by** (*induct n*, *auto*)

lemma *fin-join-ti-1nempty*:
  assumes ∀ *i*. *0* < *i* ∧ *i* < *Suc n* ⟶ *nth s* (*x*+*i*) = []
  shows   *fin-join-ti s x n* = *nth s x*
using *assms* **by** (*induct n*, *auto*)

7.2 Split time units

definition
  split-time ::′*a istream* ⇒ *nat* ⇒ ′*a istream*
where
 split-time s n t ≡
  (*if* (*t mod n* = *0*)
    then s (*t div n*)
    else [])

lemma *split-time1t*: ∀ *t*. *split-time s 1 t* = *s t*
by (*simp add*: *split-time-def*)

lemma *split-time1*: *split-time s 1* = *s*
by (*simp add*: *fun-eq-iff split-time-def*)

lemma *split-time-mod*:
  assumes *t mod n* ≠ *0*
  shows   *split-time s n t* = []
using *assms* **by** (*simp add*: *split-time-def*)

lemma *split-time-nempty*:
  assumes *0* < *n*
  shows   *split-time s n* (*n* ∗ *t*) = *s t*
using *assms* **by** (*simp add*: *split-time-def*)

lemma *split-time-nempty-Suc*:
  assumes *0* < *n*
  shows   *split-time s* (*Suc n*) ((*Suc n*) ∗ *t*) = *split-time s n* (*n* ∗ *t*)
proof −
  have *0* < *Suc n* **by** *simp*
  then have *sg1*:*split-time s* (*Suc n*) ((*Suc n*) ∗ *t*) = *s t*
    by (*rule split-time-nempty*)
  from *assms* **have** *sg2*:*split-time s n* (*n* ∗ *t*) = *s t*
    by (*rule split-time-nempty*)
  from *sg1* **and** *sg2* **show** *?thesis* **by** *simp*
qed

lemma *split-time-empty*:
  assumes *i* < *n* **and** *h2*:*0* < *i*
  shows   *split-time s n* (*n* ∗ *t* + *i*) = []
proof −
  from *assms* **have** *0* < (*n* ∗ *t* + *i*) *mod n* **by** (*simp add*: *arith-mod-nzero*)
  from *assms* **and** *this* **show** *?thesis* **by** (*simp add*: *split-time-def*)
qed

lemma *split-time-empty-Suc*:
  assumes *h1*:*i* < *n*
       and *h2*:*0* < *i*
  shows *split-time s* (*Suc n*) ((*Suc n*)∗ *t* + *i*) = *split-time s n* (*n* ∗ *t* + *i*)
proof −
  from *h1* **have** *i* < *Suc n* **by** *simp*
  from *this* **and** *h2* **have** *sg2*:*split-time s* (*Suc n*) (*Suc n* ∗ *t* + *i*) = []
    by (*rule split-time-empty*)
  from *assms* **have** *sg3*:*split-time s n* (*n* ∗ *t* + *i*) = []
    by (*rule split-time-empty*)
  from *sg3* **and** *sg2* **show** *?thesis* **by** *simp*
qed

lemma *split-time-hint1*:
  assumes *n* = *Suc m*
  shows   *split-time s* (*Suc n*) (*i* + *n* ∗ *i* + *n*) = []

```
proof -
  have sg1:i + n * i + n = (Suc n) * i + n by simp
  have sg2:n < Suc n by simp
  from assms have sg3:0 < n by simp
  from sg2 and sg3 have sg4:split-time s (Suc n) (Suc n * i + n) = []
     by (rule split-time-empty)
  from sg1 and sg4  show ?thesis by auto
qed
```

7.3 Duality of the split and the join operators

```
lemma join-split-i:
  assumes 0 < n
  shows   join-time (split-time s n) n i = s i
proof (cases n)
  assume n = 0
  from this and assms show ?thesis by simp
next
  fix k
  assume a2:n = Suc k
  have sg0:0 < Suc k by simp
  then have sg1:(split-time s (Suc k)) (Suc k * i) = s i
    by (rule split-time-nempty)
  have sg2:i + k * i = (Suc k) * i by simp
  have sg3:∀ j. 0 < j ∧ j < Suc k ⟶ split-time s (Suc k) (Suc k * i + j) = []
    by (clarify, rule split-time-empty, auto)
  from sg3 have sg4:join-ti (split-time s (Suc k)) ((Suc k) * i) k =
                    (split-time s (Suc k)) (Suc k * i)
    by (rule join-ti-1nempty)
  from a2 and sg4 and sg1 show ?thesis by (simp add: join-time-def)
qed

lemma join-split:
  assumes 0 < n
  shows join-time (split-time s n) n = s
using assms  by (simp add: fun-eq-iff join-split-i)

end
```

8 Steam Boiler System: Specification

```
theory  SteamBoiler
imports stream BitBoolTS
begin

definition
 ControlSystem :: nat istream ⇒ bool
where
 ControlSystem s ≡
 (ts s) ∧
  (∀ (j::nat). (200::nat) ≤ hd (s j) ∧ hd (s j) ≤ (800:: nat))
```

definition
```
  SteamBoiler :: bit istream ⇒ nat istream ⇒ nat istream ⇒ bool
```
where
```
 SteamBoiler x s y ≡
  ts x
  ⟶
  ((ts y) ∧ (ts s) ∧ (y = s) ∧
   ((s 0) = [500::nat]) ∧
   (∀ (j::nat). (∃ (r::nat).
                 (0::nat) < r ∧ r ≤ (10::nat) ∧
                 hd (s (Suc j)) =
                    (if hd (x j) = Zero
                     then (hd (s j)) − r
                     else (hd (s j)) + r)) ))
```

definition
```
  Converter :: bit istream ⇒ bit istream ⇒ bool
```
where
```
 Converter z x
  ≡
  (ts x)
  ∧
  (∀ (t::nat).
    hd (x t) =
        (if (fin-make-untimed (inf-truncate z t) = [])
         then
             Zero
         else
             (fin-make-untimed (inf-truncate z t)) !
                 ((length (fin-make-untimed (inf-truncate z t))) − (1::nat))
       ))
```

definition
```
  Controller-L ::
    nat istream ⇒ bit iustream ⇒ bit iustream ⇒ bit istream ⇒ bool
```
where
```
 Controller-L y lIn lOut z
  ≡
  (z 0 = [Zero])
  ∧
  (∀ (t::nat).
  ( if (lIn t) = Zero
    then ( if 300 < hd (y t)
           then  (z t) = []     ∧ (lOut t) = Zero
           else  (z t) = [One] ∧ (lOut t) = One
         )
```

```
   else ( if  hd (y t) < 700
          then  (z t) = []      ∧ (lOut t) = One
          else  (z t) = [Zero] ∧ (lOut t) = Zero ) ))

definition
  Controller :: nat istream ⇒ bit istream ⇒ bool
where
 Controller y z
  ≡
  (ts y)
  ⟶
  (∃ l. Controller-L y (fin-inf-append [Zero] l) l z)

definition
  ControlSystemArch :: nat istream ⇒ bool
where
 ControlSystemArch s
  ≡
  ∃ x z :: bit istream. ∃ y :: nat istream.
    ( SteamBoiler x s y ∧ Controller y z ∧ Converter z x )

end
```

9 Steam Boiler System: Verification

```
theory SteamBoiler-proof
imports SteamBoiler
begin
```

9.1 Properties of the Boiler Component

```
lemma L1-Boiler:
  assumes SteamBoiler x s y
       and ts x
  shows ts s
using assms by (simp add: SteamBoiler-def)

lemma L2-Boiler:
  assumes SteamBoiler x s y
       and ts x
  shows ts y
using assms by (simp add: SteamBoiler-def)

lemma L3-Boiler:
  assumes SteamBoiler x s y
       and ts x
  shows 200 ≤ hd (s 0)
using assms by (simp add: SteamBoiler-def)
```

```
lemma L4-Boiler:
  assumes SteamBoiler x s y
        and ts x
  shows hd (s 0) ≤ 800
using assms by (simp add: SteamBoiler-def)

lemma L5-Boiler:
  assumes h1:SteamBoiler x s y
        and h2:ts x
        and h3:hd (x j) = Zero
  shows (hd (s j)) ≤ hd (s (Suc j)) + (10::nat)
proof −
   from h1 and h2 obtain r where
     a1:r ≤ 10 and
     a2:hd (s (Suc j)) = (if hd (x j) = Zero then hd (s j) − r else hd (s j) + r)
     by (simp add: SteamBoiler-def, auto)
   from a2 and h3 have hd (s (Suc j)) = hd (s j) − r by simp
   from this and a1 show ?thesis by auto
qed

lemma L6-Boiler:
  assumes h1:SteamBoiler x s y
        and h2:ts x
        and h3:hd (x j) = Zero
  shows (hd (s j)) − (10::nat) ≤ hd (s (Suc j))
proof −
   from h1 and h2 obtain r where
     a1:r ≤ 10 and
     a2:hd (s (Suc j)) = (if hd (x j) = Zero then hd (s j) − r else hd (s j) + r)
     by (simp add: SteamBoiler-def, auto)
   from a2 and h3 have hd (s (Suc j)) = hd (s j) − r by simp
   from this and a1 show ?thesis by auto
qed

lemma L7-Boiler:
  assumes h1:SteamBoiler x s y
      and h2:ts x
      and h3:hd (x j) ≠ Zero
  shows (hd (s j)) ≥ hd (s (Suc j)) − (10::nat)
proof −
   from h1 and h2 obtain r where
     a1:r ≤ 10 and
     a2:hd (s (Suc j)) = (if hd (x j) = Zero then hd (s j) − r else hd (s j) + r)
     by (simp add: SteamBoiler-def, auto)
   from a2 and h3 have hd (s (Suc j)) = hd (s j) + r by simp
   from this and a1 show ?thesis by auto
qed

lemma L8-Boiler:
  assumes h1:SteamBoiler x s y
      and h2:ts x
      and h3:hd (x j) ≠ Zero
  shows (hd (s j)) + (10::nat) ≥ hd (s (Suc j))
proof −
  from h1 and h2 obtain r where
    a1:r ≤ 10 and
    a2:hd (s (Suc j)) = (if hd (x j) = Zero then hd (s j) − r else hd (s j) + r)
    by (simp add: SteamBoiler-def, auto)
  from a2 and h3 have hd (s (Suc j)) = hd (s j) + r by simp
  from this and a1 show ?thesis by auto
qed
```

9.2 Properties of the Controller Component

```
lemma L1-Controller:
  assumes Controller-L s (fin-inf-append [Zero] l) l z
  shows   fin-make-untimed (inf-truncate z i) ≠ []
using assms
by (metis Controller-L-def fin-make-untimed-inf-truncate-Nonempty-all0a)

lemma L2-Controller-Zero:
  assumes Controller-L y (fin-inf-append [Zero] l) l z
      and l t = Zero
      and 300 < hd (y (Suc t))
  shows   z (Suc t) = []
using assms
by (metis Controller-L-def correct-fin-inf-append1)

lemma L2-Controller-One:
  assumes Controller-L y (fin-inf-append [Zero] l) l z
       and l t = One
       and hd (y (Suc t)) < 700
  shows z (Suc t) = []
using assms
by (metis Controller-L-def bit.distinct(1) correct-fin-inf-append2)

lemma L3-Controller-Zero:
  assumes Controller-L y (fin-inf-append [Zero] l) l z
      and l t = Zero
      and ¬ 300 < hd (y (Suc t))
  shows z (Suc t) = [One]
using assms
by (metis Controller-L-def correct-fin-inf-append1)

lemma L3-Controller-One:
  assumes Controller-L y (fin-inf-append [Zero] l) l z
    and l t = One
    and ¬ hd (y (Suc t)) < 700
  shows      z (Suc t) = [Zero]
using assms
by (metis Controller-L-def bit.distinct(1) correct-fin-inf-append1)

lemma L4-Controller-Zero:
  assumes h1:Controller-L y (fin-inf-append [Zero] l) l z
       and h2:l (Suc t) = Zero
  shows      (z (Suc t) = [] ∧ l t = Zero) ∨ (z (Suc t) = [Zero] ∧ l t = One)
proof (cases l t)
  assume a1:l t = Zero
  from this and h1 and h2 show ?thesis
  proof −
    from a1 have sg1:fin-inf-append [Zero] l (Suc t) = Zero
      by (simp add: correct-fin-inf-append1)
    from h1 and sg1 have sg2:
      if 300 < hd (y (Suc t))
       then z (Suc t) = [] ∧ l (Suc t) = Zero
       else z (Suc t) = [One] ∧ l (Suc t) = One
       by (simp add: Controller-L-def)
    show ?thesis
    proof (cases 300 < hd (y (Suc t)))
      assume a11:300 < hd (y (Suc t))
      from a11 and sg2 have sg3:z (Suc t) = [] ∧ l (Suc t) = Zero by simp
      from this and a1 show ?thesis by simp
    next
      assume a12:¬ 300 < hd (y (Suc t))
      from a12 and sg2 have sg4:z (Suc t) = [One] ∧ l (Suc t) = One by simp
      from this and h2 show ?thesis by simp
    qed
  qed
next
  assume a2:l t = One
  from this and h1 and h2 show ?thesis
  proof −
    from a2 have sg5:fin-inf-append [Zero] l (Suc t) ≠ Zero
      by (simp add: correct-fin-inf-append1)
    from h1 and sg5 have sg6:
      if hd (y (Suc t)) < 700
       then z (Suc t) = [] ∧ l (Suc t) = One
       else z (Suc t) = [Zero] ∧ l (Suc t) = Zero
       by (simp add: Controller-L-def)
    show ?thesis
    proof (cases hd (y (Suc t)) < 700)
      assume a21:hd (y (Suc t)) < 700
      from a21 and sg6 have sg7:z (Suc t) = [] ∧ l (Suc t) = One by simp
      from this and h2 show ?thesis by simp
    next
      assume a22:¬ hd (y (Suc t)) < 700
      from a22 and sg6 have sg8:z (Suc t) = [Zero] ∧ l (Suc t) = Zero by simp
    from this and a2 show ?thesis by simp
  qed
 qed
qed


lemma L4-Controller-One:
 assumes h1:Controller-L y (fin-inf-append [Zero] l) l z
    and h2:l (Suc t) = One
 shows     (z (Suc t) = [] ∧ l t = One) ∨ (z (Suc t) = [One] ∧ l t = Zero)
proof (cases l t)
 assume a1:l t = Zero
 from this and h1 and h2 show ?thesis
 proof −
  from a1 have sg1:fin-inf-append [Zero] l (Suc t) = Zero
   by (simp add: correct-fin-inf-append1)
  from h1 and sg1 have sg2:
   if 300 < hd (y (Suc t))
    then z (Suc t) = [] ∧ l (Suc t) = Zero
    else z (Suc t) = [One] ∧ l (Suc t) = One
    by (simp add: Controller-L-def)
  show ?thesis
  proof (cases 300 < hd (y (Suc t)))
   assume a11:300 < hd (y (Suc t))
   from a11 and sg2 have sg3:z (Suc t) = [] ∧ l (Suc t) = Zero by simp
   from this and h2 show ?thesis by simp
  next
   assume a12:¬ 300 < hd (y (Suc t))
   from a12 and sg2 have sg4:z (Suc t) = [One] ∧ l (Suc t) = One by simp
   from this and a1 show ?thesis by simp
  qed
 qed
next
 assume a2:l t = One
 from this and h1 and h2 show ?thesis
 proof −
  from a2 have sg5:fin-inf-append [Zero] l (Suc t) ≠ Zero
   by (simp add: correct-fin-inf-append1)
  from h1 and sg5 have sg6:
   if hd (y (Suc t)) < 700
    then z (Suc t) = [] ∧ l (Suc t) = One
    else z (Suc t) = [Zero] ∧ l (Suc t) = Zero
    by (simp add: Controller-L-def)
  show ?thesis
  proof (cases hd (y (Suc t)) < 700)
   assume a21:hd (y (Suc t)) < 700
   from a21 and sg6 have sg7:z (Suc t) = [] ∧ l (Suc t) = One by simp
   from this and a2 show ?thesis by simp
  next
```

```
      assume a22:¬ hd (y (Suc t)) < 700
      from a22 and sg6 have sg8:z (Suc t) = [Zero] ∧ l (Suc t) = Zero by simp
      from this and h2 show ?thesis by simp
    qed
  qed
qed

lemma L5-Controller-Zero:
  assumes h1:Controller-L y lIn lOut z
      and h2:lOut t = Zero
      and h3:z t = []
  shows lIn t = Zero
proof (cases lIn t)
  assume a1:lIn t = Zero
  from this show ?thesis by simp
next
  assume a2:lIn t = One
  from a2 and h1 have sg1:
    if hd (y t) < 700
     then z t = [] ∧ lOut t = One
     else z t = [Zero] ∧ lOut t = Zero
     by (simp add: Controller-L-def)
  show ?thesis
  proof (cases hd (y t) < 700)
    assume a3:hd (y t) < 700
    from a3 and sg1 have z t = [] ∧ lOut t = One by simp
    from this and h2 show ?thesis by simp
  next
    assume a4:¬ hd (y t) < 700
    from a4 and sg1 have z t = [Zero] ∧ lOut t = Zero by simp
    from this and h3 show ?thesis by simp
  qed
qed

lemma L5-Controller-One:
  assumes h1:Controller-L y lIn lOut z
      and h2:lOut t = One
      and h3:z t = []
  shows lIn t = One
proof (cases lIn t)
  assume a1:lIn t = Zero
  from a1 and h1 have sg1:
    if 300 < hd (y t)
     then z t = [] ∧ lOut t = Zero
     else z t = [One] ∧ lOut t = One
     by (simp add: Controller-L-def)
  show ?thesis
  proof (cases 300 < hd (y t))
    assume a3:300 < hd (y t)
    from a3 and sg1 have sg2:z t = [] ∧ lOut t = Zero by simp
    from this and h2 show ?thesis by simp
  next
    assume a4:¬ 300 < hd (y t)
    from a4 and sg1 have sg3:z t = [One] ∧ lOut t = One by simp
    from this and h3 show ?thesis by simp
  qed
next
  assume lIn t = One
  then show ?thesis by simp
qed

lemma L5-Controller:
  assumes Controller-L y lIn lOut z
       and lOut t = a
       and z t = []
  shows lIn t = a
using assms
by (metis L5-Controller-One L5-Controller-Zero bit.exhaust)

lemma L6-Controller-Zero:
  assumes Controller-L y (fin-inf-append [Zero] l) l z
       and l (Suc t) = Zero
       and z (Suc t) = []
  shows l t = Zero
using assms
by (metis L4-Controller-Zero not-Cons-self2)

lemma L6-Controller-One:
  assumes Controller-L y (fin-inf-append [Zero] l) l z
       and l (Suc t) = One
       and z (Suc t) = []
  shows l t = One
using assms
by (metis L4-Controller-One list.distinct(1))

lemma L6-Controller:
  assumes Controller-L y (fin-inf-append [Zero] l) l z
       and l (Suc t) = a
       and z (Suc t) = []
  shows l t = a
using assms
by (metis L5-Controller correct-fin-inf-append2)

lemma L7-Controller-Zero:
  assumes h1:Controller-L y (fin-inf-append [Zero] l) l z
       and h2:l t = Zero
  shows    last (fin-make-untimed (inf-truncate z t)) = Zero
using assms
```

```
proof (induct t)
  case 0
  from h1 have z 0 = [Zero] by (simp add: Controller-L-def)
  from this show ?case by (simp add: fin-make-untimed-def)
next
   fix t
   case (Suc t)
   from this show ?case
   proof (cases l t)
     assume a1:l t = Zero
     from Suc have
       (z (Suc t) = [] ∧ l t = Zero) ∨ (z (Suc t) = [Zero] ∧ l t = One)
       by (simp add: L4-Controller-Zero)
     from this and a1 have z (Suc t) = []
       by simp
     from Suc and this and a1 show ?case
       by (simp add: fin-make-untimed-append-empty)
   next
     assume a1:l t = One
     from Suc have
       (z (Suc t) = [] ∧ l t = Zero) ∨ (z (Suc t) = [Zero] ∧ l t = One)
       by (simp add: L4-Controller-Zero)
     from this and a1 have z (Suc t) = [Zero]
       by simp
     from a1 and Suc and this show ?case
       by (simp add: fin-make-untimed-def)
   qed
qed

lemma L7-Controller-One-l0:
  assumes Controller-L y (fin-inf-append [Zero] l) l z
         and y 0 = [500::nat]
  shows    l 0 = Zero
proof (rule ccontr)
  assume a1: ¬ l 0 = Zero
  from assms have sg1:z 0 = [Zero] by (simp add: Controller-L-def)
 have sg2:fin-inf-append [Zero] l (0::nat) = Zero by (simp add: fin-inf-append-def)
  from assms and a1 and sg1 and sg2 show False
   by (simp add: Controller-L-def)
qed

lemma L7-Controller-One:
  assumes h1:Controller-L y (fin-inf-append [Zero] l) l z
      and h2:l t = One
      and h3:y 0 = [500::nat]
  shows last (fin-make-untimed (inf-truncate z t)) = One
using assms
proof (induct t)
  case 0
  from h1 and h3 have l 0 = Zero
    by (simp add: L7-Controller-One-l0)
  from this and 0 show ?case by simp
next
   fix t
   case (Suc t)
   from this show ?case
   proof (cases l t)
     assume a1:l t = Zero
     from Suc have
       (z (Suc t) = [] ∧ l t = One) ∨ (z (Suc t) = [One] ∧ l t = Zero)
       by (simp add: L4-Controller-One)
     from this and a1 have z (Suc t) = [One]
       by simp
     from Suc and this and a1 show ?case
       by (simp add: fin-make-untimed-def)
   next
     assume a1:l t = One
     from Suc have
       (z (Suc t) = [] ∧ l t = One) ∨ (z (Suc t) = [One] ∧ l t = Zero)
       by (simp add: L4-Controller-One)
     from this and a1 have z (Suc t) = []
       by simp
     from a1 and Suc and this show ?case
       by (simp add: fin-make-untimed-def)
   qed
qed

lemma L7-Controller:
  assumes Controller-L y (fin-inf-append [Zero] l) l z
         and y 0 = [500::nat]
  shows      last (fin-make-untimed (inf-truncate z t)) =  l t
using assms
by (metis (full-types) L7-Controller-One L7-Controller-Zero bit.exhaust)

lemma L8-Controller:
  assumes Controller-L y (fin-inf-append [Zero] l) l z
  shows     z t = [] ∨ z t = [Zero] ∨ z t = [One]
proof (cases fin-inf-append [Zero] l t = Zero)
  assume a1:fin-inf-append [Zero] l t = Zero
  from a1 and assms have sg1:
   if 300 < hd (y t)
    then z t = [] ∧ l t = Zero
    else z t = [One] ∧ l t = One
    by (simp add: Controller-L-def)
  show ?thesis
  proof (cases 300 < hd (y t))
    assume a11:300 < hd (y t)
    from a11 and sg1 show ?thesis by simp
  next
    assume a12:¬ 300 < hd (y t)
    from a12 and sg1 show ?thesis by simp
  qed
next
  assume a2:fin-inf-append [Zero] l t ≠ Zero
  from a2 and assms have sg2:
   if hd (y t) < 700
    then z t = [] ∧ l t = One
    else z t = [Zero] ∧ l t = Zero
    by (simp add: Controller-L-def)
  show ?thesis
  proof (cases hd (y t) < 700)
    assume a21:hd (y t) < 700
    from a21 and sg2 show ?thesis by simp
  next
    assume a22:¬ hd (y t) < 700
    from a22 and sg2 show ?thesis by simp
  qed
qed

lemma L9-Controller:
  assumes h1:Controller-L s (fin-inf-append [Zero] l) l z
       and h2:fin-make-untimed (inf-truncate z i) !
            (length (fin-make-untimed (inf-truncate z i)) − Suc 0) = Zero
       and h3:last (fin-make-untimed (inf-truncate z i)) = l i
       and h5:hd (s (Suc i)) = hd (s i) − r
       and h6:fin-make-untimed (inf-truncate z i) ≠ []
       and h8:r ≤ 10
  shows 200 ≤ hd (s (Suc i))
proof −
  from h6 and h2 and h3 have sg0:l i = Zero
    by (simp add: last-nth-length)
  show ?thesis
  proof (cases fin-inf-append [Zero] l i = Zero)
    assume a1:fin-inf-append [Zero] l i = Zero
    from a1 and h1 have sg1:
      if 300 < hd (s i)
       then z i = [] ∧ l i = Zero
       else z i = [One] ∧ l i = One
       by (simp add: Controller-L-def)
    show ?thesis
    proof (cases 300 < hd (s i))
      assume a11:300 < hd (s i)
      from a11 and h5 and h8 show ?thesis by simp
    next
      assume a12:¬ 300 < hd (s i)
      from a12 and sg1 and sg0 show ?thesis by simp
    qed
  next
    assume a2:fin-inf-append [Zero] l i ≠ Zero
    from a2 and h1 have sg2:
      if hd (s i) < 700
       then z i = [] ∧ l i = One
       else z i = [Zero] ∧ l i = Zero
       by (simp add: Controller-L-def)
    show ?thesis
    proof (cases hd (s i) < 700)
      assume a21:hd (s i) < 700
      from this and sg2 and sg0 show ?thesis by simp
    next
      assume a22:¬ hd (s i) < 700
      from this and h5 and h8  show ?thesis by simp
    qed
  qed
qed

lemma L10-Controller:
  assumes h1:Controller-L s (fin-inf-append [Zero] l) l z
      and h2:fin-make-untimed (inf-truncate z i) !
            (length (fin-make-untimed (inf-truncate z i)) − Suc 0) ≠  Zero
      and h3:last (fin-make-untimed (inf-truncate z i)) = l i
      and h5:hd (s (Suc i)) = hd (s i) + r
      and h6:fin-make-untimed (inf-truncate z i) ≠ []
      and h8:r ≤ 10
  shows hd (s (Suc i)) ≤ 800
proof −
  from h6 and h2 and h3 have sg0:l i ≠  Zero
    by (simp add: last-nth-length)
  show ?thesis
  proof (cases fin-inf-append [Zero] l i = Zero)
    assume a1:fin-inf-append [Zero] l i = Zero
    from a1 and h1 have sg1:
      if 300 < hd (s i)
       then z i = [] ∧ l i = Zero
       else z i = [One] ∧ l i = One
       by (simp add: Controller-L-def)
    show ?thesis
    proof (cases 300 < hd (s i))
      assume a11:300 < hd (s i)
      from a11 and sg1 and sg0 show ?thesis by simp
    next
      assume a12:¬ 300 < hd (s i)
      from h5 and a12 and h8 show ?thesis by simp
    qed
  next
    assume a2:fin-inf-append [Zero] l i ≠ Zero
    from a2 and h1 have sg2:
      if hd (s i) < 700
       then z i = [] ∧ l i = One
       else z i = [Zero] ∧ l i = Zero
       by (simp add: Controller-L-def)
    show ?thesis
    proof (cases hd (s i) < 700)
      assume a21:hd (s i) < 700
      from this and h5 and h8 show ?thesis by simp
    next
      assume a22:¬ hd (s i) < 700
      from this and sg2 and sg0 show ?thesis by simp
    qed
  qed
qed
```

9.3 Properties of the Converter Component

```
lemma L1-Converter:
  assumes Converter z x
       and fin-make-untimed (inf-truncate z t) ≠ []
  shows     hd (x t) = (fin-make-untimed (inf-truncate z t)) !
              ((length (fin-make-untimed (inf-truncate z t))) − (1::nat))
using assms
by (simp add: Converter-def)

lemma L1a-Converter:
  assumes Converter z x
       and fin-make-untimed (inf-truncate z t) ≠ []
       and hd (x t) = Zero
  shows     (fin-make-untimed (inf-truncate z t)) !
              ((length (fin-make-untimed (inf-truncate z t))) − (1::nat))
            = Zero
using assms
by (simp add: L1-Converter)
```

9.4 Properties of the System

```
lemma L1-ControlSystem:
  assumes ControlSystemArch s
  shows ts s
proof −
  from assms obtain x z y
    where a1:Converter z x and a2: SteamBoiler x s y
    by (simp only: ControlSystemArch-def, auto)
  from this have ts x
    by (simp add: Converter-def)
  from a2 and this show ?thesis by (rule L1-Boiler)
qed

lemma L2-ControlSystem:
  assumes ControlSystemArch s
  shows (200::nat) ≤ hd (s i)
proof –
  from assms obtain x z y
    where a1:Converter z x and a2: SteamBoiler x s y and a3:Controller y z
    by (simp only: ControlSystemArch-def, auto)
  from this have sg1:ts x  by (simp add: Converter-def)
  from sg1 and a2 have sg2:ts y  by (simp add: L2-Boiler)
  from sg1 and a2 have sg3:y = s by (simp add: SteamBoiler-def)
  from a1 and a2 and a3 and sg1 and sg2 and sg3 show 200 ≤ hd (s i)
  proof (induction i)
    case 0
    from this show ?case  by (simp add: L3-Boiler)
  next
    fix i
    case (Suc i)
    from this obtain l
      where a4: Controller-L s (fin-inf-append [Zero] l) l z
      by (simp add: Controller-def, atomize, auto)
    from Suc and a4 have sg4:fin-make-untimed (inf-truncate z i) ≠ []
      by (simp add:  L1-Controller)
   from a2 and sg1 have y0asm:y 0 = [500::nat]  by (simp add: SteamBoiler-def)
     from Suc and a4 and sg4 and y0asm have sg5: last (fin-make-untimed
(inf-truncate z i)) =  l i
      by (simp add: L7-Controller)
    from a2 and sg1 obtain r where
         aa0:0 < r and
         aa1:r ≤ 10 and
        aa2:hd (s (Suc i)) = (if hd (x i) = Zero then hd (s i) − r else hd (s i) + r)
         by (simp add: SteamBoiler-def, auto)
    from Suc and a4 and sg4 and sg5 show ?case
    proof (cases hd (x i) = Zero)
       assume aaZero:hd (x i) = Zero
       from a1 and sg4 and this have
         sg7:(fin-make-untimed (inf-truncate z i)) !
             ((length (fin-make-untimed (inf-truncate z i))) − Suc 0)  = Zero
         by (simp add: L1-Converter)
       from aa2 and aaZero have sg8:hd (s (Suc i)) = hd (s i) − r by simp
       from a4 and sg7 and sg5  and sg8 and sg4 and aa1 show ?thesis
          by (rule L9-Controller)
     next
       assume aaOne:hd (x i) ≠ Zero
       from a1 and sg4 and this have
         sg7:(fin-make-untimed (inf-truncate z i)) !
             ((length (fin-make-untimed (inf-truncate z i))) − Suc 0) ≠ Zero
         by (simp add: L1-Converter)
       from aa2 and aaOne have sg9:hd (s (Suc i)) = hd (s i) + r by simp
       from Suc and this show ?thesis by simp
     qed
  qed
qed

lemma L3-ControlSystem:
  assumes ControlSystemArch s
  shows hd (s i) ≤ (800:: nat)
proof −
  from assms obtain x z y
    where a1:Converter z x and a2: SteamBoiler x s y and a3:Controller y z
    by (simp only: ControlSystemArch-def, auto)
  from this have sg1:ts x  by (simp add: Converter-def)
  from sg1 and a2 have sg2:ts y  by (simp add: L2-Boiler)
  from sg1 and a2 have sg3:y = s by (simp add: SteamBoiler-def)
   from a1 and a2 and a3 and sg1 and sg2 and sg3 show hd (s i) ≤ (800::
nat)
  proof (induction i)
    case 0
    from this show ?case  by (simp add: L4-Boiler)
  next
    fix i
    case (Suc i)
    from this obtain l
      where a4: Controller-L s (fin-inf-append [Zero] l) l z
      by (simp add: Controller-def, atomize, auto)
    from a4 have sg4:fin-make-untimed (inf-truncate z i) ≠ []
      by (simp add:  L1-Controller)
   from a2 and sg1 have y0asm:y 0 = [500::nat]  by (simp add: SteamBoiler-def)
     from Suc and a4 and sg4 and y0asm have sg5: last (fin-make-untimed
(inf-truncate z i)) =  l i
      by (simp add: L7-Controller)
    from a2 and sg1 obtain r where
         aa0:0 < r and
         aa1:r ≤ 10 and
        aa2:hd (s (Suc i)) = (if hd (x i) = Zero then hd (s i) − r else hd (s i) + r)
         by (simp add: SteamBoiler-def, auto)
    from this and Suc and a4 and sg4 and sg5 show ?case
    proof (cases hd (x i) = Zero)
       assume aaZero:hd (x i) = Zero
       from a1 and sg4 and this have
         sg7:(fin-make-untimed (inf-truncate z i)) !
             ((length (fin-make-untimed (inf-truncate z i))) − Suc 0)  = Zero
         by (simp add: L1-Converter)
       from aa2 and aaZero have sg8:hd (s (Suc i)) = hd (s i) − r by simp
       from this and Suc show ?thesis by simp
      next
       assume aaOne:hd (x i) ≠ Zero
       from a1 and sg4 and this have
         sg7:(fin-make-untimed (inf-truncate z i)) !
             ((length (fin-make-untimed (inf-truncate z i))) − Suc 0) ≠ Zero
        by (simp add: L1-Converter)
      from aa2 and aaOne have sg9:hd (s (Suc i)) = hd (s i) + r by simp
      from a4 and sg7 and sg5 and sg9 and sg4 and aa1 show ?thesis
         by (rule L10-Controller)
    qed
  qed
qed
```

9.5 Proof of the Refinement Relation

```
lemma L0-ControlSystem:
  assumes h1:ControlSystemArch s
  shows   ControlSystem s
using assms
by (metis ControlSystem-def L1-ControlSystem L2-ControlSystem L3-ControlSystem)

end
```

10 FlexRay: Types

```
theory FR-types
imports stream
begin

record 'a Message =
   message-id :: nat
   ftcdata    :: 'a

record 'a Frame =
   slot :: nat
   dataF :: ('a Message) list

record Config =
   schedule    :: nat list
   cycleLength :: nat

type-synonym 'a nFrame = nat ⇒ ('a Frame) istream

type-synonym nNat = nat ⇒ nat istream

type-synonym nConfig = nat ⇒ Config

consts sN :: nat

definition
  sheafNumbers :: nat list
where
 sheafNumbers ≡ [sN]
```

end

11 FlexRay: Specification

theory *FR*
imports *FR-types*
begin

11.1 Auxiliary predicates

```
definition
  DisjointSchedules :: nat ⇒ nConfig ⇒ bool
where
 DisjointSchedules n nC
  ≡
  ∀ i j. i < n ∧ j < n ∧ i ≠ j ⟶
  disjoint (schedule (nC i)) (schedule (nC j))
```

— The predicate IdenticCycleLength is true for sheaf of channels of type Config,
— if all bus configurations have the equal length of the communication round.

```
definition
   IdenticCycleLength :: nat ⇒ nConfig ⇒ bool
where
  IdenticCycleLength n nC
   ≡
   ∀ i j. i < n ∧ j < n ⟶
   cycleLength (nC i) = cycleLength (nC j)
```

— The predicate FrameTransmission defines the correct message transmission:
— if the time t is equal modulo the length of the cycle (Flexray communication round)
— to the element of the scheduler table of the node k, then this and only this node
— can send a data atn the *t*th time interval.

```
definition
   FrameTransmission ::
     nat ⇒ 'a nFrame ⇒ 'a nFrame ⇒ nNat ⇒ nConfig ⇒ bool
where
  FrameTransmission n nStore nReturn nGet nC
   ≡
   ∀ (t::nat) (k::nat). k < n ⟶
   ( let s = t mod (cycleLength (nC k))
     in
     ( s mem (schedule (nC k))
       ⟶
       (nGet k t) = [s] ∧
       (∀ j. j < n ∧ j ≠ k ⟶
            ((nStore j) t) = ((nReturn k) t)) ))
```

— The predicate Broadcast describes properties of FlexRay broadcast.

definition
  Broadcast ::
    $nat \Rightarrow 'a\ nFrame \Rightarrow 'a\ Frame\ istream \Rightarrow bool$
where
  Broadcast n nSend recv
```
  ≡
  ∀ (t::nat).
   ( if ∃ k. k < n ∧ ((nSend k) t) ≠ []
     then (recv t) = ((nSend (SOME k. k < n ∧ ((nSend k) t) ≠ [])) t)
     else (recv t) = [] )
```

— The predicate Receive defines the relations on the streams to represent
— data receive by FlexRay controller.
definition
  Receive ::
  $'a\ Frame\ istream \Rightarrow 'a\ Frame\ istream \Rightarrow nat\ istream \Rightarrow bool$
where
  Receive recv store activation
```
  ≡
  ∀ (t::nat).
   ( if (activation t) = []
     then (store t) = (recv t)
     else (store t) = [])
```

— The predicate Send defines the relations on the streams to represent
— sending data by FlexRay controller.
definition
  Send ::
  $'a\ Frame\ istream \Rightarrow 'a\ Frame\ istream \Rightarrow nat\ istream \Rightarrow nat\ istream \Rightarrow bool$
where
  Send return send get activation
```
  ≡
  ∀ (t::nat).
  ( if (activation t) = []
    then (get t) = [] ∧ (send t) = []
    else (get t) = (activation t) ∧ (send t) = (return t) )
```

11.2 Specifications of the FlexRay components

definition
  FlexRay ::
  $nat \Rightarrow 'a\ nFrame \Rightarrow nConfig \Rightarrow 'a\ nFrame \Rightarrow nNat \Rightarrow bool$
where
  FlexRay n nReturn nC nStore nGet
```
  ≡
  (CorrectSheaf n) ∧
  ((∀ (i::nat). i < n ⟶ (msg 1 (nReturn i))) ∧
   (DisjointSchedules n nC) ∧ (IdenticCycleLength n nC)
  ⟶
   (FrameTransmission n nStore nReturn nGet nC) ∧
   (∀ (i::nat). i < n ⟶ (msg 1 (nGet i)) ∧ (msg 1 (nStore i))) )
```

definition

```
  Cable :: nat ⇒ 'a nFrame ⇒ 'a Frame istream ⇒ bool
```

where

```
 Cable n nSend recv
  ≡
  (CorrectSheaf n)
   ∧
  ((inf-disj n nSend) ⟶ (Broadcast n nSend recv))
```

definition

```
  Scheduler :: Config ⇒ nat istream ⇒ bool
```

where

```
 Scheduler c activation
  ≡
  ∀ (t::nat).
  ( let s = (t mod (cycleLength c))
     in
      ( if (s mem (schedule c))
         then (activation t) = [s]
         else (activation t) = []) )
```

definition

```
 BusInterface ::
   nat istream ⇒ 'a Frame istream ⇒ 'a Frame istream ⇒
    'a Frame istream ⇒ 'a Frame istream ⇒ nat istream ⇒ bool
```

where

```
 BusInterface activation return recv  store send get
  ≡
  (Receive recv store activation) ∧
  (Send return send get activation)
```

definition

```
  FlexRayController ::
    'a Frame istream ⇒ 'a Frame istream ⇒  Config ⇒
     'a Frame istream ⇒ 'a Frame istream ⇒ nat istream ⇒ bool
```

where

```
 FlexRayController return recv c store send get
  ≡
 (∃ activation.
    (Scheduler c activation) ∧
    (BusInterface activation return recv store send get))
```

definition

```
 FlexRayArchitecture ::
  nat ⇒ 'a nFrame ⇒ nConfig ⇒ 'a nFrame ⇒ nNat ⇒ bool
where
 FlexRayArchitecture n nReturn nC nStore nGet
  ≡
  (CorrectSheaf n) ∧
  (∃ nSend recv.
    (Cable n nSend recv) ∧
    (∀ (i::nat). i < n ⟶
        FlexRayController (nReturn i) recv (nC i)
                   (nStore i) (nSend i) (nGet i)))

definition
 FlexRayArch ::
  nat ⇒ 'a nFrame ⇒ nConfig ⇒ 'a nFrame ⇒ nNat ⇒ bool
where
 FlexRayArch n nReturn nC nStore nGet
  ≡
  (CorrectSheaf n)  ∧
  ((∀ (i::nat). i < n ⟶ msg 1 (nReturn i)) ∧
   (DisjointSchedules n nC) ∧ (IdenticCycleLength n nC)
   ⟶
   (FlexRayArchitecture n nReturn nC nStore nGet))

end
```

12 FlexRay: Verification

```
theory FR-proof
imports FR
begin
```

12.1 Properties of the function Send

```
lemma Send-L1:
assumes Send return send get activation
      and send t ≠ []
shows (activation t) ≠ []
using assms by (simp add: Send-def, auto)

lemma Send-L2:
assumes Send return send get activation
      and (activation t) ≠ []
      and return t ≠ []
shows (send t) ≠ []
using assms by (simp add: Send-def)
```

12.2 Properties of the component Scheduler

```
lemma Scheduler-L1:
```

```
assumes h1:Scheduler C activation
       and h2:(activation t) ≠ []
shows (t mod (cycleLength C)) mem (schedule C)
using assms
proof –
  { assume  a1:¬ t mod cycleLength C mem schedule C
     from h1 have
     if t mod cycleLength C mem schedule C
      then activation t = [t mod cycleLength C]
      else activation t = []
        by (simp add: Scheduler-def Let-def)
    from a1 and this have activation t = [] by simp
    from this and h2 have sg3:False by simp
  } from this have  sg4:(t mod (cycleLength C)) mem (schedule C) by blast
 from this show ?thesis by simp
qed

lemma Scheduler-L2:
assumes Scheduler C activation
      and ¬ (t mod cycleLength C) mem (schedule C)
shows activation t = []
using assms by (simp add: Scheduler-def Let-def)

lemma Scheduler-L3:
assumes Scheduler C activation
      and (t mod cycleLength C) mem (schedule C)
shows activation t ≠ []
using assms by (simp add: Scheduler-def Let-def)

lemma Scheduler-L4:
assumes Scheduler C activation
      and (t mod cycleLength C) mem (schedule C)
shows activation t = [t mod cycleLength C]
using assms by (simp add: Scheduler-def Let-def)

lemma correct-DisjointSchedules1:
assumes h1:DisjointSchedules n nC
      and h2:IdenticCycleLength n nC
      and h3:(t mod cycleLength (nC i)) mem schedule (nC i)
      and h4:i < n
      and h5:j < n
      and h6:i ≠ j
shows ¬ (t mod cycleLength (nC j) mem schedule (nC j))
proof –
 from h1 and h4 and h5 and h6 have sg1:disjoint (schedule (nC i)) (schedule
(nC j))
   by (simp add: DisjointSchedules-def)
 from h2 and h4 and h5 have sg2:cycleLength (nC i) = cycleLength (nC j)
   by (metis IdenticCycleLength-def)
```

from *sg1* **and** *h3* **have** *sg3*:$\neg$ (*t mod* (*cycleLength* (*nC i*))) *mem* (*schedule* (*nC j*))
  by (*simp add*: *mem-notdisjoint2*)
from *sg2* **and** *sg3* **show** *?thesis* **by** *simp*
qed

12.3 Disjoint Frames

lemma *disjointFrame-L1*:
assumes *h1*:*DisjointSchedules n nC*
  and *h2*:*IdenticCycleLength n nC*
  and *h3*:$\forall$ *i* $<$ *n*. *FlexRayController* (*nReturn i*) *rcv*
    (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
  and *h4*:*nSend i t* $\neq$ []
  and *h5*:*i* $<$ *n*
  and *h6*:*j* $<$ *n*
  and *h7*:*i* $\neq$ *j*
shows *nSend j t* = []
proof −
from *h3* **and** *h5* **have** *sg1*:
FlexRayController (*nReturn i*) *rcv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
  by *auto*
from *h3* **and** *h6* **have** *sg2*:
FlexRayController (*nReturn j*) *rcv* (*nC j*) (*nStore j*) (*nSend j*) (*nGet j*)
  by *auto*
from *sg1* **obtain** *activation1* **where**
  a1:*Scheduler* (*nC i*) *activation1* **and**
  a2:*BusInterface activation1* (*nReturn i*) *rcv* (*nStore i*) (*nSend i*) (*nGet i*)
  by (*simp add*: *FlexRayController-def*, *auto*)
from *sg2* **obtain** *activation2* **where**
  a3:*Scheduler* (*nC j*) *activation2* **and**
  a4:*BusInterface activation2* (*nReturn j*) *rcv* (*nStore j*) (*nSend j*) (*nGet j*)
  by (*simp add*: *FlexRayController-def*, *auto*)
from *h1* **and** *h5* **and** *h6* **and** *h7* **have** *sg3*:*disjoint* (*schedule* (*nC i*)) (*schedule* (*nC j*))
  by (*simp add*: *DisjointSchedules-def*)
from *a2* **have** *sg4a*:*Send* (*nReturn i*) (*nSend i*) (*nGet i*) *activation1*
  by (*simp add*: *BusInterface-def*)
from *sg4a* **and** *h4* **have** *sg5*:*activation1 t* $\neq$ [] **by** (*simp add*: *Send-L1*)
from *a1* **and** *sg5* **have** *sg6*:(*t mod* (*cycleLength* (*nC i*))) *mem* (*schedule* (*nC i*))

  by (*simp add*: *Scheduler-L1*)
from *h2* **and** *h5* **and** *h6* **have** *sg7*:*cycleLength* (*nC i*) = *cycleLength* (*nC j*)
  by (*metis IdenticCycleLength-def*)
from *sg3* **and** *sg6* **have** *sg8*:$\neg$ (*t mod* (*cycleLength* (*nC i*))) *mem* (*schedule* (*nC j*))
  by (*simp add*: *mem-notdisjoint2*)
from *sg8* **and** *sg7* **have** *sg9*:$\neg$ (*t mod* (*cycleLength* (*nC j*))) *mem* (*schedule* (*nC j*))

```
    by simp
  from sg9 and a3 have sg10:activation2 t = [] by (simp add: Scheduler-L2)
  from a4 have sg11:Send (nReturn j) (nSend j) (nGet j) activation2
     by (simp add:  BusInterface-def)
  from sg11 and sg10 show ?thesis by (simp add: Send-def)
qed

lemma disjointFrame-L2:
assumes DisjointSchedules n nC
       and IdenticCycleLength n nC
       and ∀  i < n. FlexRayController (nReturn i) rcv
                     (nC i) (nStore i) (nSend i) (nGet i)
shows inf-disj n nSend
using assms
  apply (simp add: inf-disj-def, clarify)
  by (rule disjointFrame-L1, auto)


lemma disjointFrame-L3:
assumes h1:DisjointSchedules n nC
     and h2:IdenticCycleLength n nC
     and h3:∀  i < n. FlexRayController (nReturn i) rcv
                     (nC i) (nStore i) (nSend i) (nGet i)
    and h4:t mod cycleLength (nC i) mem schedule (nC i)
    and h5:i < n
    and h6:j < n
    and h7:i ≠ j
shows nSend j t = []
proof −
  from h2 and h5 and h6 have sg1:cycleLength (nC i) = cycleLength (nC j)
    by (metis IdenticCycleLength-def)
  from h1 and h5 and h6 and h7 have sg2:disjoint (schedule (nC i)) (schedule
(nC j))
    by (simp add: DisjointSchedules-def)
  from sg2 and h4 have sg3:¬ (t mod (cycleLength (nC i))) mem (schedule (nC
j))
    by (simp add: mem-notdisjoint2)
  from sg1 and sg3 have sg4:¬ (t mod (cycleLength (nC j))) mem (schedule (nC
j))
    by simp
  from h3 and h6 have sg5:
   FlexRayController (nReturn j) rcv (nC j) (nStore j) (nSend j) (nGet j)
    by auto
  from sg5 obtain activation2 where
     a1:Scheduler (nC j) activation2 and
     a2:BusInterface activation2 (nReturn j) rcv (nStore j) (nSend j) (nGet j)
     by (simp add: FlexRayController-def, auto)
  from sg4 and a1 have sg6:activation2 t = [] by (simp add: Scheduler-L2)
  from a2 have sg7:Send (nReturn j) (nSend j) (nGet j) activation2
```

by (*simp add*: *BusInterface-def*)
from *sg7* **and** *sg6* **show** *?thesis* **by** (*simp add*: *Send-def*)
qed

12.4 Properties of the sheaf of channels nSend

lemma *fr-Send1*:
assumes *frc*:*FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
and *h1*:¬ (*t mod cycleLength* (*nC i*) *mem schedule* (*nC i*))
shows (*nSend i*) *t* = []
proof −
from *frc* **obtain** *activation* **where**
a1:*Scheduler* (*nC i*) *activation* **and**
a2:*BusInterface activation* (*nReturn i*) *recv* (*nStore i*) (*nSend i*) (*nGet i*)
by (*simp add*: *FlexRayController-def*, *auto*)
from *a1* **and** *h1* **have** *sg1*:*activation t* = [] **by** (*simp add*: *Scheduler-L2*)
from *a2* **have** *sg2*:*Send* (*nReturn i*) (*nSend i*) (*nGet i*) *activation*
by (*simp add*: *BusInterface-def*)
from *sg2* **and** *sg1* **show** *?thesis* **by** (*simp add*: *Send-def*)
qed

lemma *fr-Send2*:
assumes *h1*:∀ *i*<*n*. *FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
and *h2*:*DisjointSchedules n nC*
and *h3*:*IdenticCycleLength n nC*
and *h4*:*t mod cycleLength* (*nC k*) *mem schedule* (*nC k*)
and *h5*:*k* < *n*
shows *nSend k t* = *nReturn k t*
proof −
from *h1* **and** *h5* **have** *sg1*:
FlexRayController (*nReturn k*) *recv* (*nC k*) (*nStore k*) (*nSend k*) (*nGet k*)
by *auto*
from *sg1* **obtain** *activation* **where**
a1:*Scheduler* (*nC k*) *activation* **and**
a2:*BusInterface activation* (*nReturn k*) *recv* (*nStore k*) (*nSend k*) (*nGet k*)
by (*simp add*: *FlexRayController-def*, *auto*)
from *a1* **and** *h4* **have** *sg3*:*activation t* ≠ [] **by** (*simp add*: *Scheduler-L3*)
from *a2* **have** *sg4*:*Send* (*nReturn k*) (*nSend k*) (*nGet k*) *activation*
by (*simp add*: *BusInterface-def*)
from *sg4* **and** *sg3* **show** *?thesis* **by** (*simp add*: *Send-def*)
qed

lemma *fr-Send3*:
assumes ∀ *i*<*n*. *FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
and *DisjointSchedules n nC*
and *IdenticCycleLength n nC*

```
    and t mod cycleLength (nC k) mem schedule (nC k)
    and k < n
    and nReturn k t ≠ []
 shows nSend k t ≠ []
using assms by (simp add: fr-Send2)

lemma fr-Send4:
 assumes ∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i)
(nGet i)
    and DisjointSchedules n nC
    and IdenticCycleLength n nC
    and t mod cycleLength (nC k) mem schedule (nC k)
    and k < n
    and nReturn k t ≠ []
 shows ∃ k. k < n ⟶ nSend k t ≠ []
using assms
by (metis  fr-Send3)

lemma fr-Send5:
 assumes h1:∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend
i) (nGet i)
    and h2:DisjointSchedules n nC
    and h3:IdenticCycleLength n nC
    and h4:t mod cycleLength (nC k) mem schedule (nC k)
    and h5:k < n
    and h6:nReturn k t ≠ []
    and h7:∀ k<n. nSend k t = []
 shows False
proof −
  from h1 and h2 and h3 and h4 and h5 and h6 have sg1:nSend k t ≠ []
    by (simp add: fr-Send2)
  from h7 and h5 have sg2:nSend k t = [] by blast
  from sg1 and sg2 show ?thesis by simp
qed

lemma fr-Send6:
assumes ∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i)
(nGet i)
      and DisjointSchedules n nC
      and IdenticCycleLength n nC
      and t mod cycleLength (nC k) mem schedule (nC k)
      and k < n
      and nReturn k t ≠ []
shows ∃ k<n. nSend k t ≠ []
using assms
by (metis fr-Send3)

lemma fr-Send7:
assumes ∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i)
```

```
(nGet i)
       and DisjointSchedules n nC
       and IdenticCycleLength n nC
       and t mod cycleLength (nC k) mem schedule (nC k)
       and k < n
       and j < n
       and nReturn k t = []
shows nSend j t = []
using assms
by (metis (full-types) disjointFrame-L3 fr-Send2)

lemma fr-Send8:
assumes ∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i)
(nGet i)
       and DisjointSchedules n nC
       and IdenticCycleLength n nC
       and t mod cycleLength (nC k) mem schedule (nC k)
       and k < n
       and nReturn k t = []
shows ¬ (∃ k<n. nSend k t ≠ [])
using assms by (auto, simp add: fr-Send7)

lemma fr-nC-Send:
assumes ∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i)
(nGet i)
       and k < n
       and DisjointSchedules n nC
       and IdenticCycleLength n nC
       and t mod cycleLength (nC k) mem schedule (nC k)
shows ∀ j. j < n ∧ j ≠ k ⟶ (nSend j) t = []
using assms by (clarify, simp add: disjointFrame-L3)

lemma length-nSend:
assumes h1:BusInterface activation (nReturn i) recv (nStore i) (nSend i) (nGet
i)
       and h2:∀ t. length (nReturn i t) ≤ Suc 0
shows  length (nSend i t) ≤ Suc 0
proof −
  from h1 have sg1:Send (nReturn i) (nSend i) (nGet i) activation
    by (simp add: BusInterface-def)
  from sg1 have sg2:
   if activation t = [] then nGet i t = [] ∧ nSend i t = []
    else nGet i t = activation t ∧ nSend i t = nReturn i t
    by (simp add: Send-def)
  show ?thesis
  proof (cases activation t = [])
    assume a1:activation t = []
    from sg2 and a1 show ?thesis by simp
  next
```

```
    assume a2:activation t ≠ []
    from h2 have sg3:length (nReturn i t) ≤ Suc 0 by auto
    from sg2 and a2 and sg3 show ?thesis by simp
  qed
qed

lemma msg-nSend:
assumes BusInterface activation (nReturn i) recv (nStore i) (nSend i) (nGet i)
       and msg (Suc 0) (nReturn i)
shows msg (Suc 0) (nSend i)
using assms by (simp add: msg-def, clarify, simp add: length-nSend)

lemma Broadcast-nSend-empty1:
assumes h1:Broadcast n nSend recv
       and h2:∀ k<n. nSend k t = []
shows      recv t = []
using assms
by (metis Broadcast-def)
```

12.5 Properties of the sheaf of channels nGet

```
lemma fr-nGet1a:
assumes h1:FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k)
(nGet k)
       and h2:t mod cycleLength (nC k) mem schedule (nC k)
shows nGet k t = [t mod cycleLength (nC k)]
proof −
  from h1 obtain activation1 where
    a1:Scheduler (nC k) activation1 and
    a2:BusInterface activation1 (nReturn k) recv (nStore k) (nSend k) (nGet k)
    by (simp add: FlexRayController-def, auto)
   from a2 have sg1:Send (nReturn k) (nSend k) (nGet k) activation1
    by (simp add: BusInterface-def)
   from sg1 have sg2:
    if activation1 t = [] then nGet k t = [] ∧ nSend k t = []
     else nGet k t = activation1 t ∧ nSend k t = nReturn k t
    by (simp add: Send-def)
  from a1 and h2 have sg3:activation1 t = [t mod cycleLength (nC k)]
    by (simp add: Scheduler-L4)
  from sg2 and sg3 show ?thesis by simp
qed

lemma fr-nGet1:
assumes ∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i)
(nGet i)
       and t mod cycleLength (nC k) mem schedule (nC k)
       and k < n
shows nGet k t = [t mod cycleLength (nC k)]
using assms
```

```
by (metis  fr-nGet1a)

lemma fr-nGet2a:
assumes h1:FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k)
(nGet k)
       and h2:¬ (t mod cycleLength (nC k) mem schedule (nC k))
shows nGet k t = []
proof −
   from h1 obtain activation1 where
     a1:Scheduler (nC k) activation1 and
     a2:BusInterface activation1 (nReturn k) recv (nStore k) (nSend k) (nGet k)
     by (simp add: FlexRayController-def, auto)
   from a2 have sg2:Send (nReturn k) (nSend k) (nGet k) activation1
     by (simp add: BusInterface-def)
   from sg2 have sg3:
    if activation1 t = [] then nGet k t = [] ∧ nSend k t = []
     else nGet k t = activation1 t ∧ nSend k t = nReturn k t
     by (simp add: Send-def)
  from a1 and h2 have sg4:activation1 t = []
     by (simp add: Scheduler-L2)
  from sg3 and sg4 show ?thesis  by simp
qed

lemma fr-nGet2:
assumes h1:∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend
i) (nGet i)
     and h2:¬ (t mod cycleLength (nC k) mem schedule (nC k))
     and h3:k < n
 shows nGet k t = []
proof −
  from h1 and h3 have sg1:
    FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)
    by auto
  from sg1 and h2 show ?thesis by (rule fr-nGet2a)
qed

lemma length-nGet1:
 assumes FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet
k)
 shows    length (nGet k t) ≤ Suc 0
proof (cases t mod cycleLength (nC k) mem schedule (nC k))
  assume t mod cycleLength (nC k) mem schedule (nC k)
  from assms and this have nGet k t = [t mod cycleLength (nC k)]
    by (rule fr-nGet1a)
  then show ?thesis by auto
next
  assume ¬ (t mod cycleLength (nC k) mem schedule (nC k))
  from assms and this have nGet k t = [] by (rule fr-nGet2a)
  then show ?thesis by auto
```

qed

lemma *msg-nGet1*:
 assumes *FlexRayController* (*nReturn k*) *recv* (*nC k*) (*nStore k*) (*nSend k*) (*nGet k*)
 shows *msg* (*Suc 0*) (*nGet k*)
using *assms*
by (*simp add*: *msg-def*, *auto*, *rule length-nGet1*)

lemma *msg-nGet2*:
assumes ∀ *i*<*n*. *FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
 and *k* < *n*
shows *msg* (*Suc 0*) (*nGet k*)
using *assms*
by (*metis msg-nGet1*)

12.6 Properties of the sheaf of channels nStore

lemma *fr-nStore-nReturn1*:
 assumes *h0*:*Broadcast n nSend recv*
 and *h1*:*inf-disj n nSend*
 and *h2*:∀ *i*<*n*. *FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
 and *h3*:*DisjointSchedules n nC*
 and *h4*:*IdenticCycleLength n nC*
 and *h5*:*t mod cycleLength* (*nC k*) *mem schedule* (*nC k*)
 and *h6*:*k* < *n*
 and *h7*:*j* < *n*
 and *h8*:*j* ≠ *k*
 shows *nStore j t* = *nReturn k t*
proof −
 from *h2* **and** *h6* **have** *sg1*:
 FlexRayController (*nReturn k*) *recv* (*nC k*) (*nStore k*) (*nSend k*) (*nGet k*)
 by *auto*
 from *h2* **and** *h7* **have** *sg2*:
 FlexRayController (*nReturn j*) *recv* (*nC j*) (*nStore j*) (*nSend j*) (*nGet j*)
 by *auto*
 from *sg1* **obtain** *activation1* **where**
 a1:*Scheduler* (*nC k*) *activation1* **and**
 a2:*BusInterface activation1* (*nReturn k*) *recv* (*nStore k*) (*nSend k*) (*nGet k*)
 by (*simp add*: *FlexRayController-def*, *auto*)
 from *sg2* **obtain** *activation2* **where**
 a3:*Scheduler* (*nC j*) *activation2* **and**
 a4:*BusInterface activation2* (*nReturn j*) *recv* (*nStore j*) (*nSend j*) (*nGet j*)
 by (*simp add*: *FlexRayController-def*, *auto*)
 from *a4* **have** *sg3*:*Receive recv* (*nStore j*) *activation2*
 by (*simp add*: *BusInterface-def*)
 from *this* **have** *sg4*:

```
  if activation2 t = [] then nStore j t = recv t else nStore j t = []
   by (simp add: Receive-def)
  from a1 and h5 have sg5:activation1 t ≠ []
     by (simp add: Scheduler-L3)
 from h4 and h6 and h7 have sg6:cycleLength (nC k) = cycleLength (nC j)
   by (metis IdenticCycleLength-def)
  from h3 and h6 and h7 and h8 have sg7:disjoint (schedule (nC k)) (schedule
(nC j))
   by (simp add: DisjointSchedules-def)
  from sg7 and h5 have sg8:¬ (t mod (cycleLength (nC k))) mem (schedule (nC
j))
   by (simp add: mem-notdisjoint2)
  from sg6 and sg8 have sg9:¬ (t mod (cycleLength (nC j))) mem (schedule (nC
j))
   by simp
  from sg9 and a3 have sg10:activation2 t = [] by (simp add: Scheduler-L2)
  from sg10 and sg4 have sg11:nStore j t = recv t by simp
  from h0 have sg15:
  if ∃ k<n. nSend k t ≠ []
   then recv t = nSend (SOME k. k < n ∧ nSend k t ≠ []) t
   else recv t = []
   by (simp add: Broadcast-def)
  show ?thesis
  proof (cases nReturn k t = [])
   assume a5: nReturn k t = []
   from h2 and h3 and h4 and h5 and h6 and a5 have sg16:¬ (∃ k<n. nSend
k t ≠ [])
     by (simp add: fr-Send8)
   from sg16 and sg15 have sg17:recv t = [] by simp
   from sg11 and sg17 have sg18:nStore j t = [] by simp
   from this and a5 show ?thesis by simp
  next
   assume a6:nReturn k t ≠ []
   from h2 and h3 and h4 and h5 and h6 and a6 have sg19:∃ k<n. nSend k
t ≠ []
     by (simp add: fr-Send6)
   from h2 and h3 and h4 and h5 and h6 and a6 have sg20:nSend k t ≠ []
     by (simp add: fr-Send3)
   from h1 and sg20 and h6 have sg21:(SOME k. k < n ∧ nSend k t ≠ []) = k
     by (simp add: inf-disj-index)
   from sg15 and sg19 have sg22:recv t = nSend (SOME k. k < n ∧ nSend k t
≠ []) t
     by simp
   from sg22 and sg21 have sg23:recv t = nSend k t by simp
   from h2 and h3 and h4 and h5 and h6 have sg24:nSend k t = nReturn k t
     by (simp add: fr-Send2)
   from sg11 and sg23 and sg24 show ?thesis by simp
  qed
qed
```

lemma *fr-nStore-nReturn2*:
 assumes *h1*:*Cable n nSend recv*
  and *h2*:$\forall$ *i*<*n*. *FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
  and *h3*:*DisjointSchedules n nC*
  and *h4*:*IdenticCycleLength n nC*
  and *h5*:*t mod cycleLength* (*nC k*) *mem schedule* (*nC k*)
  and *h6*:*k* < *n*
  and *h7*:*j* < *n*
  and *h8*:*j* $\neq$ *k*
 shows *nStore j t* = *nReturn k t*
proof –
 from *h1* **have** *sg1*:*inf-disj n nSend* $\longrightarrow$ *Broadcast n nSend recv*
  by (*simp add*: *Cable-def*)
 from *h3* **and** *h4* **and** *h2* **have** *sg2*:*inf-disj n nSend*
  by (*simp add*: *disjointFrame-L2*)
 from *sg1* **and** *sg2* **have** *sg3*:*Broadcast n nSend recv* **by** *simp*
 from *sg3* **and** *sg2* **and** *assms* **show** *?thesis* **by** (*simp add*: *fr-nStore-nReturn1*)
qed

lemma *fr-nStore-empty1*:
 assumes *h1*:*Cable n nSend recv*
  and *h2*:$\forall$ *i*<*n*. *FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
  and *h3*:*DisjointSchedules n nC*
  and *h4*:*IdenticCycleLength n nC*
  and *h5*:(*t mod cycleLength* (*nC k*) *mem schedule* (*nC k*))
  and *h6*:*k* < *n*
 shows *nStore k t* = []
proof –
 from *h2* **and** *h6* **have** *sg1*:
  FlexRayController (*nReturn k*) *recv* (*nC k*) (*nStore k*) (*nSend k*) (*nGet k*)
  by *auto*
 from *sg1* **obtain** *activation1* **where**
  a1:*Scheduler* (*nC k*) *activation1* **and**
  a2:*BusInterface activation1* (*nReturn k*) *recv* (*nStore k*) (*nSend k*) (*nGet k*)
  by (*simp add*: *FlexRayController-def*, *auto*)
 from *a2* **have** *sg2*:*Receive recv* (*nStore k*) *activation1*
  by (*simp add*: *BusInterface-def*)
 from *this* **have** *sg3*:
  if activation1 t = [] *then nStore k t* = *recv t else nStore k t* = []
  by (*simp add*: *Receive-def*)
 from *a1* **and** *h5* **have** *sg4*:*activation1 t* $\neq$ []
  by (*simp add*: *Scheduler-L3*)
 from *sg3* **and** *sg4* **show** *?thesis* **by** *simp*
qed

```
lemma fr-nStore-nReturn3:
 assumes Cable n nSend recv
      and ∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i)
(nGet i)
     and DisjointSchedules n nC
     and IdenticCycleLength n nC
     and t mod cycleLength (nC k) mem schedule (nC k)
     and k < n
 shows  ∀ j. j < n ∧ j ≠ k ⟶ nStore j t = nReturn k t
using assms
by (clarify, simp add: fr-nStore-nReturn2)

lemma length-nStore:
 assumes h1:∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend
i) (nGet i)
     and h2:DisjointSchedules n nC
     and h3:IdenticCycleLength n nC
     and h4:inf-disj n nSend
     and h5:i < n
     and h6:∀  i<n. msg (Suc 0) (nReturn i)
     and h7:Broadcast n nSend recv
 shows length (nStore i t) ≤ Suc 0
proof −
  from h7 have sg1:
   if ∃ k<n. nSend k t ≠ []
    then recv t = nSend (SOME k. k < n ∧ nSend k t ≠ []) t
    else recv t = []
   by (simp add: Broadcast-def)
  show ?thesis
  proof (cases ∃ k<n. nSend k t ≠ [])
    assume ∃ k<n. nSend k t ≠ []
    from this obtain k where a2:k<n and a3:nSend k t ≠ []  by auto
    from h1 and a2 have
      FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)
      by auto
    then obtain activation1 where
      a4:Scheduler (nC k) activation1 and
      a5:BusInterface activation1 (nReturn k) recv (nStore k) (nSend k) (nGet k)
      by (simp add: FlexRayController-def, auto)
    from a5 have sg5:Send (nReturn k) (nSend k) (nGet k) activation1
      by (simp add:  BusInterface-def)
    from a5 have sg6:Receive recv (nStore k) activation1
      by (simp add: BusInterface-def)
    from sg5 and a3 have sg7:(activation1 t) ≠ [] by (simp add: Send-L1)
    from sg6 have sg8:
     if activation1 t = []
      then nStore k t = recv t else nStore k t = []
      by (simp add: Receive-def)
    from sg8 and sg7 have sg9:nStore k t = [] by simp
    from a4 and sg7 have sg10:(t mod (cycleLength (nC k))) mem (schedule (nC
k))
      by (simp add: Scheduler-L1)
    show ?thesis
    proof (cases i = k)
      assume  i = k
      from sg9 and this show ?thesis by simp
    next
      assume i ≠ k
       from h7 and h4 and h1 and h2 and h3 and sg10 and a2 and h5 and
this have sg11:
       nStore i t = nReturn k t
        by (simp add: fr-nStore-nReturn1)
      from h6 and a2 have sg12:msg (Suc 0) (nReturn k) by auto
      from a2 and h6 have sg13:length (nReturn k t) ≤ Suc 0
        by (simp add: msg-def)
      from sg11 and sg13  show ?thesis by simp
    qed
  next
    assume ¬ (∃ k<n. nSend k t ≠ [])
   from h7 and this have sg14:recv t = [] by (simp add: Broadcast-nSend-empty1)

    from h1 and h5 have
      FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
      by auto
   then obtain activation2 where
      a11:Scheduler (nC i) activation2 and
      a12:BusInterface activation2 (nReturn i) recv (nStore i) (nSend i) (nGet i)
      by (simp add: FlexRayController-def, auto)
    from a12 have Receive recv (nStore i) activation2
      by (simp add: BusInterface-def)
    then have sg17:
     if activation2 t = []
      then nStore i t = recv t else nStore i t = []
      by (simp add: Receive-def)
    show ?thesis
    proof (cases activation2 t = [])
      assume aa3:activation2 t = []
      from sg17 and aa3 and sg14 have nStore i t = [] by simp
      then show ?thesis by simp
    next
      assume aa4:activation2 t ≠ []
      from sg17 and aa4 have nStore i t = [] by simp
      then show ?thesis by simp
    qed
  qed
qed

lemma msg-nStore:
```

assumes ∀ *i*<*n*. *FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
    and *DisjointSchedules n nC*
    and *IdenticCycleLength n nC*
    and *inf-disj n nSend*
    and *i* < *n*
    and ∀ *i*<*n*. *msg* (*Suc 0*) (*nReturn i*)
    and *Cable n nSend recv*
shows *msg* (*Suc 0*) (*nStore i*)
using *assms*
  apply (*simp* (*no-asm*) *add*: *msg-def*, *simp add*: *Cable-def*, *clarify*)
  by (*simp add*: *length-nStore*)

12.7 Refinement Properties

lemma *fr-refinement-FrameTransmission*:
assumes *Cable n nSend recv*
    and ∀ *i*<*n*. *FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
    and *DisjointSchedules n nC*
    and *IdenticCycleLength n nC*
shows *FrameTransmission n nStore nReturn nGet nC*
using *assms*
  apply (*simp add*: *FrameTransmission-def Let-def*, *auto*)
  apply (*simp add*: *fr-nGet1*)
  by (*simp add*: *fr-nStore-nReturn3*)

lemma *FlexRayArch-CorrectSheaf*:
 assumes *FlexRayArch n nReturn nC nStore nGet*
 shows   *CorrectSheaf n*
using *assms* **by** (*simp add*: *FlexRayArch-def*)

lemma *FlexRayArch-FrameTransmission*:
 assumes *h1*:*FlexRayArch n nReturn nC nStore nGet*
   and *h2*:∀ *i*<*n*. *msg* (*Suc 0*) (*nReturn i*)
   and *h3*:*DisjointSchedules n nC*
   and *h4*:*IdenticCycleLength n nC*
 shows   *FrameTransmission n nStore nReturn nGet nC*
proof −
  from *assms* **obtain** *nSend recv* **where**
    a1:*Cable n nSend recv* **and**
   a2:∀ *i*<*n*. *FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
    by (*simp add*: *FlexRayArch-def FlexRayArchitecture-def*, *auto*)
  from *a1* **and** *a2* **and** *h3* **and** *h4* **show** *?thesis*
    by (*rule fr-refinement-FrameTransmission*)
qed

lemma *FlexRayArch-nGet*:

```
 assumes h1:FlexRayArch n nReturn nC nStore nGet
     and h2:∀ i<n. msg (Suc 0) (nReturn i)
     and h3:DisjointSchedules n nC
     and h4:IdenticCycleLength n nC
     and h5:i < n
 shows     msg (Suc 0) (nGet i)
proof −
 from assms obtain nSend recv where
    a1:Cable n nSend recv and
   a2:∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet
i)
    by (simp add: FlexRayArch-def  FlexRayArchitecture-def, auto)
  from a2 and h5 show ?thesis by (rule msg-nGet2)
qed

lemma FlexRayArch-nStore:
 assumes h1:FlexRayArch n nReturn nC nStore nGet
     and h2:∀ i<n. msg (Suc 0) (nReturn i)
     and h3:DisjointSchedules n nC
     and h4:IdenticCycleLength n nC
     and h5:i < n
 shows     msg (Suc 0) (nStore i)
proof −
 from assms obtain nSend recv where
    a1:Cable n nSend recv and
   a2:∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet
i)
    by (simp add: FlexRayArch-def  FlexRayArchitecture-def, auto)
  from h3 and h4 and a2 have sg1:inf-disj n nSend by (simp add: disjoint-
Frame-L2)
  from a2 and h3 and h4 and sg1 and h5 and h2 and a1  show ?thesis
    by (rule msg-nStore)
qed

theorem main-fr-refinement:
 assumes FlexRayArch n nReturn nC nStore nGet
 shows     FlexRay n nReturn nC nStore nGet
using assms
  by (simp add: FlexRay-def
                FlexRayArch-CorrectSheaf
                FlexRayArch-FrameTransmission
                FlexRayArch-nGet
                FlexRayArch-nStore)

end
```

13 Gateway: Types

```
theory Gateway-types
```

imports *stream*
begin

type-synonym
  Coordinates = *nat* × *nat*
type-synonym
  CollisionSpeed = *nat*

record *ECall-Info* =
  coord :: *Coordinates*
  speed :: *CollisionSpeed*

datatype *GatewayStatus* =
   init-state
  | *call*
  | *connection-ok*
  | *sending-data*
  | *voice-com*

datatype *reqType* = *init* | *send*

datatype *stopType* = *stop-vc*

datatype *vcType* = *vc-com*

datatype *aType* = *sc-ack*

end

14 Gateway: Specification

theory *Gateway*
imports *Gateway-types*
begin

definition
 ServiceCenter ::
  ECall-Info istream ⇒ *aType istream* ⇒ *bool*
where
 ServiceCenter i a
 ≡
 ∀ (*t*::*nat*).
 $a\ 0 = [] \wedge\ a\ (Suc\ t) = \ (if\ \ (i\ t) = []\ then\ []\ \ else\ [sc\text{-}ack])$

definition
 Loss ::
  bool istream ⇒ *aType istream* ⇒ *ECall-Info istream* ⇒
   aType istream ⇒ *ECall-Info istream* ⇒ *bool*
where

```
 Loss lose a i2 a2 i
  ≡
  ∀ (t::nat).
  ( if lose t = [False]
    then a2 t = a t ∧ i t = i2 t
    else a2 t = [] ∧ i t = []  )
```

definition

```
 Delay ::
  aType istream ⇒ ECall-Info istream ⇒ nat ⇒
   aType istream ⇒ ECall-Info istream ⇒ bool
```

where

```
 Delay a2 i1 d a1 i2
  ≡
  ∀ (t::nat).
   (t < d ⟶ a1 t = [] ∧ i2 t = []) ∧
   (t ≥ d ⟶ (a1 t = a2 (t−d)) ∧ (i2 t = i1 (t−d)))
```

definition

```
 tiTable-SampleT ::
 reqType istream  ⇒ aType istream ⇒
  stopType istream ⇒ bool istream ⇒
  (nat ⇒ GatewayStatus) ⇒ (nat ⇒ ECall-Info list) ⇒
  GatewayStatus istream ⇒ ECall-Info istream ⇒ vcType istream
  ⇒ (nat ⇒ GatewayStatus) ⇒ bool
```

where

```
 tiTable-SampleT req a1 stop lose st-in buffer-in
         ack i1 vc st-out
  ≡
  ∀ (t::nat)
     (r::reqType list) (x::aType list)
     (y::stopType list) (z::bool list).
   — 1:
   ( st-in t = init-state ∧ req t = [init]
     ⟶ ack t = [call] ∧ i1 t = [] ∧ vc t = []
         ∧ st-out t = call )
   ∧
   — 2:
   ( st-in t = init-state ∧ req t ≠ [init]
     ⟶  ack t = [init-state] ∧ i1 t = [] ∧ vc t = []
         ∧ st-out t = init-state )
   ∧
   — 3:
   ( (st-in t = call ∨ (st-in t = connection-ok ∧ r ≠ [send])) ∧
     req t = r ∧ lose t = [False]
     ⟶  ack t = [connection-ok] ∧ i1 t = [] ∧ vc t = []
         ∧ st-out t = connection-ok )
   ∧
   — 4:
```

((*st-in t* = *call* ∨ *st-in t* = *connection-ok* ∨ *st-in t* = *sending-data*)
  ∧ *lose t* = [*True*]
  ⟶ *ack t* = [*init-state*] ∧ *i1 t* = [] ∧ *vc t* = []
      ∧ *st-out t* = *init-state*)
∧
— 5:
(*st-in t* = *connection-ok* ∧ *req t* = [*send*] ∧ *lose t* = [*False*]
  ⟶ *ack t* = [*sending-data*] ∧ *i1 t* = *buffer-in t* ∧ *vc t* = []
      ∧ *st-out t* = *sending-data*)
∧
— 6:
(*st-in t* = *sending-data* ∧ *a1 t* = [] ∧ *lose t* = [*False*]
  ⟶ *ack t* = [*sending-data*] ∧ *i1 t* = [] ∧ *vc t* = []
      ∧ *st-out t* = *sending-data*)
∧
— 7:
(*st-in t* = *sending-data* ∧ *a1 t* = [*sc-ack*] ∧ *lose t* = [*False*]
  ⟶ *ack t* = [*voice-com*] ∧ *i1 t* = [] ∧ *vc t* = [*vc-com*]
      ∧ *st-out t* = *voice-com*)
∧
— 8:
(*st-in t* = *voice-com* ∧ *stop t* = [] ∧ *lose t* = [*False*]
  ⟶ *ack t* = [*voice-com*] ∧ *i1 t* = [] ∧ *vc t* = [*vc-com*]
      ∧ *st-out t* = *voice-com*)
∧
— 9:
(*st-in t* = *voice-com* ∧ *stop t* = [] ∧ *lose t* = [*True*]
  ⟶ *ack t* = [*voice-com*] ∧ *i1 t* = [] ∧ *vc t* = []
      ∧ *st-out t* = *voice-com*)
∧
— 10:
(*st-in t* = *voice-com* ∧ *stop t* = [*stop-vc*]
  ⟶ *ack t* = [*init-state*] ∧ *i1 t* = [] ∧ *vc t* = []
      ∧ *st-out t* = *init-state*)

definition
Sample-L ::
reqType istream ⇒ *ECall-Info istream* ⇒ *aType istream* ⇒
stopType istream ⇒ *bool istream* ⇒
(*nat* ⇒ *GatewayStatus*) ⇒ (*nat* ⇒ *ECall-Info list*) ⇒
GatewayStatus istream ⇒ *ECall-Info istream* ⇒ *vcType istream*
⇒ (*nat* ⇒ *GatewayStatus*) ⇒ (*nat* ⇒ *ECall-Info list*)
⇒ *bool*
where
Sample-L req dt a1 stop lose st-in buffer-in
      ack i1 vc st-out buffer-out
≡
(∀ (*t*::*nat*).
 buffer-out t =

```
   (if dt t = [] then buffer-in t  else dt t) )
  ∧
  (tiTable-SampleT req a1 stop lose st-in buffer-in
                 ack i1 vc st-out)

definition
 Sample ::
 reqType istream  ⇒ ECall-Info istream ⇒ aType istream ⇒
  stopType istream ⇒ bool istream ⇒
  GatewayStatus istream ⇒ ECall-Info istream ⇒ vcType istream
  ⇒ bool
where
 Sample req dt a1 stop lose  ack i1 vc
  ≡
  ((msg (1::nat) req) ∧
   (msg (1::nat) a1)  ∧
   (msg (1::nat) stop))
  ⟶
  (∃ st buffer.
  (Sample-L req dt a1 stop lose
            (fin-inf-append [init-state] st)
            (fin-inf-append [[]] buffer)
            ack i1 vc st buffer) )

definition
 Gateway ::
   reqType istream ⇒ ECall-Info istream ⇒ aType istream ⇒
    stopType istream ⇒ bool istream ⇒ nat ⇒
    GatewayStatus istream ⇒ ECall-Info istream ⇒ vcType istream
    ⇒ bool
where
 Gateway req dt a stop lose d ack i vc
  ≡ ∃ i1 i2 x y.
    (Sample req dt x stop lose ack i1 vc) ∧
    (Delay y i1 d x i2) ∧
    (Loss lose a i2 y i)


definition
  GatewaySystem ::
   reqType istream ⇒ ECall-Info istream ⇒
    stopType istream ⇒ bool istream ⇒ nat ⇒
    GatewayStatus istream ⇒ vcType istream
    ⇒ bool
where
 GatewaySystem req dt stop lose d ack vc
  ≡
  ∃ a i.
 (Gateway req dt a stop lose d ack i vc) ∧
 (ServiceCenter i a)

definition
 GatewayReq ::
   reqType istream ⇒ ECall-Info istream ⇒ aType istream ⇒
    stopType istream ⇒ bool istream ⇒ nat ⇒
    GatewayStatus istream ⇒ ECall-Info istream ⇒ vcType istream
    ⇒ bool
where
 GatewayReq req dt a stop lose d ack i vc
  ≡
 ((msg (1::nat) req) ∧  (msg (1::nat) a)  ∧
  (msg (1::nat) stop) ∧  (ts lose))
   ⟶
  (∀ (t::nat).
  ( ack t = [init-state] ∧ req (Suc t) = [init] ∧
    lose (t+1) = [False] ∧ lose (t+2) = [False]
    ⟶ ack (t+2) = [connection-ok])
  ∧
  ( ack t = [connection-ok] ∧ req (Suc t)  = [send] ∧
    (∀ (k::nat). k ≤ (d+1) ⟶ lose (t+k) = [False])
    ⟶ i ((Suc t) + d) = inf-last-ti dt t
        ∧  ack (Suc t) = [sending-data])
  ∧
  ( ack (t+d) = [sending-data] ∧ a (Suc t) = [sc-ack] ∧
    (∀ (k::nat). k ≤ (d+1) ⟶ lose (t+k) = [False])
    ⟶ vc ((Suc t) + d) = [vc-com]) )

definition
  GatewaySystemReq ::
   reqType istream ⇒ ECall-Info istream ⇒
    stopType istream ⇒ bool istream ⇒ nat ⇒
    GatewayStatus istream ⇒ vcType istream
    ⇒ bool
where
 GatewaySystemReq req dt  stop lose d ack vc
  ≡
 ((msg (1::nat) req) ∧ (msg (1::nat) stop) ∧ (ts lose))
   ⟶
  (∀ (t::nat) (k::nat).
  ( ack t = [init-state] ∧ req (Suc t) = [init]
  ∧ (∀ t1. t1 ≤ t ⟶ req t1 = [])
  ∧ req (t+2) = []
  ∧ (∀ m. m < k + 3 ⟶ req (t + m) ≠ [send])
  ∧  req (t+3+k)  = [send] ∧  inf-last-ti dt (t+2) ≠ []
  ∧ (∀ (j::nat).
     j ≤ (4 + k + d + d) ⟶ lose (t+j) = [False])
  ⟶ vc (t + 4 + k + d + d) = [vc-com]) )
```

```
end
```

15 Gateway: Verification

```
theory Gateway-proof-aux
imports Gateway BitBoolTS
begin
```

15.1 Properties of the defined data types

```
lemma aType-empty:
  assumes h1:msg (Suc 0) a
        and h2: a t ≠ [sc-ack]
  shows       a t = []
proof (cases a t)
  assume a1:a t = []
  from this show ?thesis by simp
next
  fix aa l
  assume a2:a t = aa # l
  show ?thesis
    proof (cases aa)
      assume a3:aa = sc-ack
      from h1 have sg1:length (a t) ≤ Suc 0 by (simp add: msg-def)
      from this and assms and a2 and a3 show ?thesis by auto
    qed
qed

lemma aType-nonempty:
  assumes h1:msg (Suc 0) a
      and h2: a t ≠ []
  shows       a t = [sc-ack]
proof (cases a t)
  assume a1:a t = []
  from this and h2 show ?thesis by simp
next
  fix aa l
  assume a2:a t = aa # l
  from a2 and h1 have sg1: l = [] by (simp add: msg-nonempty1)
  from a2 and h1 and sg1 show ?thesis
    proof (cases aa)
      assume a3:aa = sc-ack
      from this and sg1 and h2 and a2 show ?thesis  by simp
    qed
qed

lemma aType-lemma:
  assumes msg (Suc 0) a
  shows     a t = [] ∨ a t = [sc-ack]
```

using *assms*
by (*metis aType-nonempty*)

lemma *stopType-empty*:
  assumes *msg* (*Suc 0*) *a*
        and $a\ t \neq$ [*stop-vc*]
  shows $a\ t =$ []
using *assms*
by (*metis* (*full-types*) *list-length-hint2 msg-nonempty2 stopType.exhaust*)

lemma *stopType-nonempty*:
  assumes *msg* (*Suc 0*) *a*
        and $a\ t \neq$ []
  shows $a\ t =$ [*stop-vc*]
using *assms*
by (*metis stopType-empty*)

lemma *stopType-lemma*:
  assumes *msg* (*Suc 0*) *a*
  shows $a\ t =$ [] $\vee$ $a\ t =$ [*stop-vc*]
using *assms*
by (*metis stopType-nonempty*)

lemma *vcType-empty*:
  assumes *msg* (*Suc 0*) *a*
        and $a\ t \neq$ [*vc-com*]
  shows$a\ t =$ []
using *assms*
by (*metis* (*full-types*) *list-length-hint2 msg-nonempty2 vcType.exhaust*)

lemma *vcType-lemma*:
  assumes *msg* (*Suc 0*) *a*
  shows $a\ t =$ [] $\vee$ $a\ t =$ [*vc-com*]
using *assms*
by (*metis vcType-empty*)

15.2 Properties of the Delay component

lemma *Delay-L1*:
 assumes *h1*:$\forall$ *t1* < *t*. *i1 t1* = []
        and *h2*:*Delay y i1 d x i2*
        and *h3*:*t2* < *t* + *d*
   shows *i2 t2* = []
proof (*cases t2* < *d*)
  assume *a1*:*t2* < *d*
  from *h2* **have** *sg1*:*t2* < *d* $\longrightarrow$ *i2 t2* = []
    by (*simp add*: *Delay-def*)
  from *sg1* **and** *a1* **show** *?thesis* **by** *simp*
next

```
  assume a2:¬ t2 < d
  from h2 have sg2:d ≤ t2 ⟶ i2 t2 = i1 (t2 − d)
    by (simp add: Delay-def)
  from a2 and sg2 have i2 t2 = i1 (t2 − d) by simp
  from h1 and a2 and h3 and this show ?thesis by auto
qed

lemma Delay-L2:
 assumes ∀ t1 < t. i1 t1 = []
        and Delay y i1 d x i2
   shows ∀ t2 < t + d. i2 t2 = []
using assms by (clarify, rule Delay-L1, auto)

lemma Delay-L3:
 assumes h1:∀ t1 ≤ t. y t1 = []
        and h2:Delay y i1 d x i2
        and h3:t2 ≤ t + d
   shows x t2 = []
proof (cases t2 < d)
  assume a1:t2 < d
  from h2 have sg1:t2 < d ⟶ x t2 = []
    by (simp add: Delay-def)
  from sg1 and a1 show ?thesis by simp
next
  assume a2:¬ t2 < d
  from h2 have sg2:d ≤ t2 ⟶ x t2 = y (t2 − d)
    by (simp add: Delay-def)
  from a2 and sg2 have sg3:x t2 = y (t2 − d) by simp
  from h1 and a2 and h3 and sg3 show ?thesis by auto
qed

lemma Delay-L4:
 assumes ∀ t1 ≤ t. y t1 = []
     and Delay y i1 d x i2
   shows ∀ t2 ≤ t + d. x t2 = []
using assms by (clarify, rule Delay-L3, auto)

lemma Delay-lengthOut1:
  assumes h1:∀ t. length (x t) ≤ Suc 0
         and h2:Delay x i1 d y i2
  shows length (y t) ≤ Suc 0
proof (cases t < d)
  assume a1:t < d
  from h2 have sg1:t < d ⟶ y t = []
    by (simp add: Delay-def)
  from a1 and sg1 show ?thesis by auto
next
  assume a2:¬ t < d
  from h2 have sg2:t ≥ d ⟶ (y t = x (t−d))
```

```
   by (simp add: Delay-def)
  from a2 and sg2 and h1 show ?thesis by auto
qed

lemma Delay-msg1:
  assumes msg (Suc 0) x
          and Delay x i1 d y i2
  shows      msg (Suc 0) y
using assms
by (simp add: msg-def Delay-lengthOut1)
```

15.3 Properties of the Loss component

```
lemma Loss-L1:
 assumes ∀ t2<t. i2 t2 = []
         and Loss lose a i2 y i
         and t2 < t
         and ts lose
 shows i t2 = []
using assms
by (metis Loss-def)

lemma Loss-L2:
 assumes ∀ t2<t. i2 t2 = []
         and Loss lose a i2 y i
         and ts lose
 shows  ∀ t2<t. i t2 = []
using assms
by (metis Loss-def)

lemma Loss-L3:
 assumes ∀ t2<t. a t2 = []
         and Loss lose a i2 y i
         and t2 < t
         and ts lose
 shows y t2 = []
using assms
by (metis Loss-def)

lemma Loss-L4:
 assumes ∀ t2<t. a t2 = []
         and Loss lose a i2 y i
         and ts lose
 shows  ∀ t2<t. y t2 = []
using assms
by (metis Loss-def)

lemma Loss-L5:
 assumes ∀ t1 ≤ t. a t1 = []
```

```
        and Loss lose a i2 y i
        and t2 ≤ t
        and ts lose
 shows y t2 = []
using assms
by (metis Loss-def)

lemma Loss-L5Suc:
 assumes ∀ j ≤ d. a (t + Suc j) = []
        and Loss lose a i2 y i
        and Suc j ≤ d
        and tsLose:ts lose
 shows y (t + Suc j) = []
using assms
proof (cases lose (t + Suc j) = [False])
  assume lose (t + Suc j) = [False]
  from assms and this show ?thesis by (simp add: Loss-def)
next
  assume lose (t + Suc j) ≠ [False]
  from this and tsLose have lose (t + Suc j) = [True]
    by (simp add: ts-bool-True)
  from assms and this show ?thesis by (simp add: Loss-def)
qed

lemma Loss-L6:
 assumes ∀ t2 ≤ t. a t2 = []
        and Loss lose a i2 y i
        and ts lose
 shows  ∀ t2 ≤ t. y t2 = []
using assms
by (metis Loss-L5)

lemma Loss-lengthOut1:
  assumes h1:∀ t. length (a t) ≤ Suc 0
         and h2:Loss lose a i2 x i
  shows length (x t) ≤ Suc 0
proof (cases lose t =  [False])
  assume lose t =  [False]
  from this and h2 have sg1:x t = a t by (simp add: Loss-def)
  from h1 have sg2:length (a t) ≤ Suc 0 by auto
  from sg1 and sg2 show ?thesis by simp
next
  assume lose t ≠ [False]
  from this and h2 have x t = [] by (simp add: Loss-def)
  from this show ?thesis by simp
qed

lemma Loss-lengthOut2:
  assumes ∀ t. length (a t) ≤ Suc 0
```

```
       and Loss lose a i2 x i
  shows ∀ t. length (x t) ≤ Suc 0
using assms
by (simp add: Loss-lengthOut1)

lemma Loss-msg1:
  assumes msg (Suc 0) a
       and Loss lose a i2 x i
  shows      msg (Suc 0) x
using assms
by (simp add: msg-def Loss-def Loss-lengthOut1)
```

15.4 Properties of the composition of Delay and Loss components

```
lemma Loss-Delay-length-y:
  assumes ∀ t. length (a t) ≤ Suc 0
       and Delay x i1 d y i2
       and Loss lose a i2 x i
  shows length (y t) ≤ Suc 0
using assms
by (metis Delay-msg1 Loss-msg1 msg-def)

lemma Loss-Delay-msg-a:
  assumes msg (Suc 0) a
       and Delay x i1 d y i2
       and Loss lose a i2 x i
  shows  msg (Suc 0) y
using assms
by (simp add: msg-def Loss-Delay-length-y)
```

15.5 Auxiliary Lemmas

```
lemma inf-last-ti2:
  assumes inf-last-ti dt (Suc (Suc t)) ≠ []
  shows     inf-last-ti dt (Suc (Suc (t + k))) ≠ []
using assms
by (metis add-Suc inf-last-ti-nonempty-k)

lemma aux-ack-t2:
  assumes h1:∀ m≤k. ack (Suc (Suc (t + m))) = [connection-ok]
       and h2:Suc (Suc t) < t2
       and h3:t2 < t + 3 + k
  shows ack t2 = [connection-ok]
proof −
  from h3 have sg1:t2 − Suc (Suc t) ≤ k by arith
  from h1 and sg1
    obtain m where a1:m = t2 − Suc (Suc t)
              and a2:ack (Suc (Suc (t + m))) = [connection-ok]
    by auto
  from h2 have sg2:(Suc (Suc (t2 − 2))) =  t2 by arith
  from h2 have sg3:Suc (Suc (t + (t2 − Suc (Suc t)))) =  t2 by arith
  from sg1 and a1 and a2 and sg2 and sg3 show ?thesis by simp
qed

lemma aux-lemma-lose-1:
  assumes h1:∀ j≤((2::nat) ∗ d + ((4::nat) + k)). (lose (t + j) = x)
        and h2:ka≤Suc d
  shows lose (Suc (Suc (t + k + ka))) = x
proof −
  from h2 have sg1:k + (2::nat) + ka ≤ (2::nat) ∗ d + ((4::nat) + k) by auto
  from h2 and sg1 have sg2:Suc (Suc (k + ka)) ≤2 ∗ d + (4 + k) by auto
  from sg1 and sg2 and h1 and h2  obtain j where a1:j = k + (2::nat) + ka
                                  and a2:lose (t + j) = x
    by blast
  have sg3:Suc (Suc (t + (k + ka))) = Suc (Suc (t + k + ka)) by arith
  from a1 and a2 and sg3 show ?thesis  by simp
qed

lemma aux-lemma-lose-2:
  assumes ∀ j≤(2::nat) ∗ d + ((4::nat) + k). lose (t + j) = [False]
  shows    ∀ x≤d + (1::nat). lose (t + x) = [False]
using assms by auto

lemma aux-lemma-lose-3a:
  assumes h1:∀ j≤2 ∗ d + (4 + k). lose (t + j) = [False]
        and h2:ka ≤ Suc d
  shows lose (d + (t + (3 + k)) + ka) = [False]
proof −
  from h2 have sg1:(d + 3 + k + ka) ≤2 ∗ d + (4 + k)
    by arith
  from h1 and h2 and sg1  obtain j where a1:j = (d + 3 + k + ka) and
                                    a2:lose (t + j) = [False]
    by simp
  from h2 and sg1 have sg2:(t + (d + 3 + k + ka)) = (d + (t + (3 + k)) +
ka)
    by arith
  from h1 and h2 and a1 and a2 and sg2 show ?thesis
    by simp
qed

lemma aux-lemma-lose-3:
  assumes ∀ j≤2 ∗ d + (4 + k). lose (t + j) = [False]
  shows    ∀ ka≤Suc d. lose (d + (t + (3 + k)) + ka) = [False]
using assms
by (auto, simp add: aux-lemma-lose-3a)

lemma aux-arith1-Gateway7:
  assumes t2 − t ≤ (2::nat) ∗ d + (t + ((4::nat) + k))
      and t2 < t + (3::nat) + k + d
      and ¬ t2 − d < (0::nat)
  shows t2 − d < t + (3::nat) + k
using assms by arith

lemma ts-lose-ack-st1ts:
  assumes ts lose
      and lose t = [True]  ⟶  ack t = [x]  ∧ st-out t = x
      and lose t = [False] ⟶  ack t = [y]  ∧ st-out t = y
  shows ack t = [st-out t]
using assms
by (metis ts-bool-False)

lemma ts-lose-ack-st1:
  assumes h1:lose t = [True] ∨ lose t = [False]
  and h2:lose t = [True]  ⟶  ack t = [x]  ∧ st-out t = x
  and h3:lose t = [False] ⟶  ack t = [y]  ∧ st-out t = y
  shows ack t = [st-out t]
proof (cases lose t = [False])
  assume lose t = [False]
  from this and h3 show ?thesis by simp
next
  assume a2:lose t ≠ [False]
  from this and h1 have lose t = [True] by (simp add: ts-bool-True)
  from this and a2 and h2 show ?thesis by simp
qed

lemma ts-lose-ack-st2ts:
  assumes ts lose
      and lose t = [True] ⟶
          ack t = [x]  ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = x
     and lose t = [False] ⟶
         ack t = [y] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = y
  shows ack t = [st-out t]
using assms
by (metis ts-bool-True-False)

lemma ts-lose-ack-st2:
  assumes h1:lose t = [True] ∨ lose t = [False]
      and h2:lose t = [True] ⟶
          ack t = [x]  ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = x
      and h3:lose t = [False] ⟶
          ack t = [y] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = y
  shows ack t = [st-out t]
proof (cases lose t = [False])
  assume lose t = [False]
  from this and h3 show ?thesis by simp
next
```

```
  assume a2:lose t ≠ [False]
  from this and h1 have lose t = [True] by (simp add: ts-bool-True)
  from this and a2 and h2 show ?thesis by simp
qed

lemma ts-lose-ack-st2vc-com:
assumes h1:lose t = [True] ∨ lose t = [False]
       and h2:lose t = [True] ⟶
             ack t = [x]  ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = x
       and h3:lose t = [False] ⟶
             ack t = [y] ∧ i1 t = [] ∧ vc t = [vc-com] ∧ st-out t = y
shows ack t = [st-out t]
proof (cases lose t = [False])
  assume lose t = [False]
  from this and h3 show ?thesis by simp
next
  assume a2:lose t ≠ [False]
  from this and h1 have ag1:lose t = [True] by (simp add: ts-bool-True)
  from this and a2 and h2 show ?thesis by simp
qed

lemma ts-lose-ack-st2send:
  assumes h1:lose t = [True] ∨ lose t = [False]
  and h2:lose t = [True] ⟶
      ack t = [x]  ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = x
  and h3:lose t = [False] ⟶
      ack t = [y] ∧ i1 t = b t ∧ vc t = [] ∧ st-out t = y
  shows ack t = [st-out t]
proof (cases lose t = [False])
  assume lose t = [False]
  from this and h3 show ?thesis by simp
next
  assume a2:lose t ≠ [False]
  from this and h1 have lose t = [True] by (simp add: ts-bool-True)
  from this and a2 and h2 show ?thesis by simp
qed

lemma tiTable-ack-st-splitten:
  assumes h1:ts lose
      and h2:msg (Suc 0) a1
      and h3:msg (Suc 0) stop
      and h4:st-in t = init-state ∧ req t = [init] ⟶
          ack t = [call] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = call
      and h5:st-in t = init-state ∧ req t ≠ [init] ⟶
          ack t = [init-state] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = init-state
     and h6:(st-in t = call ∨ st-in t = connection-ok ∧ req t ≠ [send]) ∧ lose t =
[False] ⟶
          ack t = [connection-ok] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = connection-ok
      and h7:(st-in t = call ∨ st-in t = connection-ok ∨ st-in t = sending-data) ∧
```

```
lose t = [True] ⟶
        ack t = [init-state] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = init-state
    and h8:st-in t = connection-ok ∧ req t = [send] ∧ lose t = [False] ⟶
        ack t = [sending-data] ∧ i1 t = b t ∧ vc t = [] ∧ st-out t = sending-data
    and h9:st-in t = sending-data ∧ a1 t = [] ∧ lose t = [False] ⟶
        ack t = [sending-data] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = sending-data
    and h10:st-in t = sending-data ∧ a1 t = [sc-ack] ∧ lose t = [False] ⟶
        ack t = [voice-com] ∧ i1 t = [] ∧ vc t = [vc-com] ∧ st-out t = voice-com
    and h11:st-in t = voice-com ∧ stop t = [] ∧ lose t = [False] ⟶
        ack t = [voice-com] ∧ i1 t = [] ∧ vc t = [vc-com] ∧ st-out t = voice-com
    and h12:st-in t = voice-com ∧ stop t = [] ∧ lose t = [True] ⟶
        ack t = [voice-com] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = voice-com
    and h13:st-in t = voice-com ∧ stop t = [stop-vc] ⟶
        ack t = [init-state] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = init-state
  shows ack t = [st-out t]
proof −
  from h1 and h6 and h7 have sg1:lose t = [True] ∨ lose t = [False]
    by (simp add: ts-bool-True-False)
show ?thesis
proof (cases st-in t)
  assume a1:st-in t = init-state
  from a1 and h4 and h5 show ?thesis
  proof (cases req t = [init])
    assume a11:req t = [init]
    from a11 and a1 and h4 and h5 show ?thesis by simp
  next
    assume a12:req t ≠ [init]
    from a12 and a1 and h4 and h5 show ?thesis by simp
  qed
next
  assume a2:st-in t = call
  from a2 and sg1 and h6 and h7 show ?thesis
     apply simp
     by (rule ts-lose-ack-st2, assumption+)
next
  assume a3:st-in t = connection-ok
  from a3 and h6 and h7 and h8 show ?thesis apply simp
  proof (cases req t = [send])
    assume a31:req t = [send]
    from this and a3 and h6 and h7 and h8 and sg1 show ?thesis
      apply simp
      by (rule ts-lose-ack-st2send, assumption+)
  next
    assume a32:req t ≠ [send]
    from this and a3 and h6 and h7 and h8 and sg1 show ?thesis
      apply simp
      by (rule ts-lose-ack-st2, assumption+)
  qed
next
```

```
  assume a4:st-in t = sending-data
  from sg1 and a4 and h7 and h9 and h10  show ?thesis apply simp
  proof (cases a1 t = [])
    assume a41:a1 t = []
    from this and a4 and sg1 and h7 and h9 and h10 show ?thesis
      apply simp
      by (rule ts-lose-ack-st2, assumption+)
  next
    assume a42:a1 t ≠ []
    from this and h2  have a1 t = [sc-ack]  by (simp add: aType-nonempty)
    from this and a4 and a42 and sg1 and h7 and h9 and h10 show ?thesis
      apply simp
      by (rule ts-lose-ack-st2vc-com, assumption+)
  qed
next
  assume a5:st-in t = voice-com
  from a5 and h11 and h12 and h13 show ?thesis
  apply simp
  proof (cases stop t = [])
    assume a51:stop t = []
    from this and a5 and h11 and h12 and h13 and sg1 show ?thesis
      apply simp
      by (rule ts-lose-ack-st2vc-com, assumption+)
  next
    assume a52:stop t ≠ []
    from this and h3 have sg7:stop t = [stop-vc]
      by (simp add: stopType-nonempty)
    from this and a5 and a52 and h13 show ?thesis by simp
  qed
qed
qed

lemma tiTable-ack-st:
assumes tiTable-SampleT req a1 stop lose st-in b ack i1 vc st-out
       and tsLose:ts lose
       and a1Msg1:msg (Suc 0) a1
       and stopMsg1:msg (Suc 0) stop
  shows      ack t = [st-out t]
proof –
  from assms have sg1:
   st-in t = init-state ∧ req t = [init] ⟶
    ack t = [call] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = call
     by (simp add: tiTable-SampleT-def)
  from assms have sg2:
   st-in t = init-state ∧ req t ≠ [init] ⟶
    ack t = [init-state] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = init-state
     by (simp add: tiTable-SampleT-def)
  from assms have sg3:
   (st-in t = call ∨ st-in t = connection-ok ∧ req t ≠ [send]) ∧
```

lose t = [*False*] ⟶
ack t = [*connection-ok*] ∧ *i1 t* = [] ∧ *vc t* = [] ∧ *st-out t* = *connection-ok*
by (*simp add*: *tiTable-SampleT-def*)
from *assms* **have** *sg4*:
(*st-in t* = *call* ∨ *st-in t* = *connection-ok* ∨ *st-in t* = *sending-data*) ∧
lose t = [*True*] ⟶
ack t = [*init-state*] ∧ *i1 t* = [] ∧ *vc t* = [] ∧ *st-out t* = *init-state*
by (*simp add*: *tiTable-SampleT-def*)
from *assms* **have** *sg5*:
st-in t = *connection-ok* ∧ *req t* = [*send*] ∧ *lose t* = [*False*] ⟶
ack t = [*sending-data*] ∧ *i1 t* = *b t* ∧ *vc t* = [] ∧ *st-out t* = *sending-data*
by (*simp add*: *tiTable-SampleT-def*)
from *assms* **have** *sg6*:
st-in t = *sending-data* ∧ *a1 t* = [] ∧ *lose t* = [*False*] ⟶
ack t = [*sending-data*] ∧ *i1 t* = [] ∧ *vc t* = [] ∧ *st-out t* = *sending-data*
by (*simp add*: *tiTable-SampleT-def*)
from *assms* **have** *sg7*:
st-in t = *sending-data* ∧ *a1 t* = [*sc-ack*] ∧ *lose t* = [*False*] ⟶
ack t = [*voice-com*] ∧ *i1 t* = [] ∧ *vc t* = [*vc-com*] ∧ *st-out t* = *voice-com*
by (*simp add*: *tiTable-SampleT-def*)
from *assms* **have** *sg8*:
st-in t = *voice-com* ∧ *stop t* = [] ∧ *lose t* = [*False*] ⟶
ack t = [*voice-com*] ∧ *i1 t* = [] ∧ *vc t* = [*vc-com*] ∧ *st-out t* = *voice-com*
by (*simp add*: *tiTable-SampleT-def*)
from *assms* **have** *sg9*:
st-in t = *voice-com* ∧ *stop t* = [] ∧ *lose t* = [*True*] ⟶
ack t = [*voice-com*] ∧ *i1 t* = [] ∧ *vc t* = [] ∧ *st-out t* = *voice-com*
by (*simp add*: *tiTable-SampleT-def*)
from *assms* **have** *sg10*:
st-in t = *voice-com* ∧ *stop t* = [*stop-vc*] ⟶
ack t = [*init-state*] ∧ *i1 t* = [] ∧ *vc t* = [] ∧ *st-out t* = *init-state*
by (*simp add*: *tiTable-SampleT-def*)
from *tsLose* **and** *a1Msg1* **and** *stopMsg1* **and** *sg1* **and** *sg2* **and** *sg3* **and** *sg4*
and *sg5* **and**
sg6 **and** *sg7* **and** *sg8* **and** *sg9* **and** *sg10* **show** *?thesis*
by (*rule tiTable-ack-st-splitten*)
qed

lemma *tiTable-ack-st-hd*:
assumes *tiTable-SampleT req a1 stop lose st-in b ack i1 vc st-out*
and *ts lose*
and *msg* (*Suc 0*) *a1*
and *msg* (*Suc 0*) *stop*
shows *st-out t* = *hd* (*ack t*)
using *assms* **by** (*simp add*: *tiTable-ack-st*)

lemma *tiTable-ack-connection-ok*:
assumes *tbl*:*tiTable-SampleT req x stop lose st-in b ack i1 vc st-out*
and *ackCon*:*ack t* = [*connection-ok*]

```
      and xMsg1:msg (Suc 0) x
      and tsLose:ts lose
      and stopMsg1:msg (Suc 0) stop
  shows (st-in t = call ∨ st-in t = connection-ok ∧ req t ≠ [send]) ∧
        lose t = [False]
proof –
  from tbl and tsLose have sg1:lose t = [True] ∨ lose t = [False]
    by (simp add: ts-bool-True-False)
  from tbl and xMsg1 have sg2:x t = [] ∨ x t = [sc-ack]
    by (simp add: aType-lemma)
  from tbl and stopMsg1 have sg3:stop t = [] ∨ stop t = [stop-vc]
    by (simp add: stopType-lemma)
  show ?thesis
  proof (cases st-in t)
    assume a1:st-in t = init-state
    show ?thesis
    proof (cases req t = [init])
      assume a11:req t = [init]
    from tbl and a1 and a11 and ackCon show ?thesis by (simp add: tiTable-SampleT-def)
    next
      assume a12:req t ≠ [init]
    from tbl and a1 and a12 and ackCon show ?thesis by (simp add: tiTable-SampleT-def)
    qed
  next
    assume a2:st-in t = call
    show ?thesis
    proof (cases lose t = [True])
      assume a21:lose t = [True]
    from tbl and a2 and a21 and ackCon show ?thesis by (simp add: tiTable-SampleT-def)
    next
      assume a22:lose t ≠ [True]
     from this and tsLose have a22a:lose t = [False] by (simp add: ts-bool-False)
      from tbl have
       (st-in t = call ∨ st-in t = connection-ok ∧ req t ≠ [send]) ∧
         lose t = [False] ⟶
         ack t = [connection-ok] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = connection-ok
         by (simp add: tiTable-SampleT-def)
      from this and a2 and a22a and ackCon show ?thesis by simp
    qed
  next
    assume a3:st-in t = connection-ok
    show ?thesis
    proof (cases lose t = [True])
      assume a31:lose t = [True]
      from tbl have
       (st-in t = call ∨ st-in t = connection-ok ∨ st-in t = sending-data) ∧
         lose t = [True] ⟶
         ack t = [init-state] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = init-state
        by (simp add: tiTable-SampleT-def)
      from this and a3 and a31 and ackCon show ?thesis by simp
    next
      assume a32:lose t ≠ [True]
     from this and tsLose have a32a:lose t = [False] by (simp add: ts-bool-False)
      show ?thesis
      proof (cases req t = [send])
        assume a321:req t = [send]
        from tbl and a3 and a32a and a321 and ackCon show ?thesis
          by (simp add: tiTable-SampleT-def)
      next
        assume a322:req t ≠ [send]
        from tbl and a3 and a32a and a322 and ackCon show ?thesis
           by (simp add: tiTable-SampleT-def)
      qed
    qed
  next
    assume a4:st-in t = sending-data
    show ?thesis
    proof (cases lose t = [True])
      assume a41:lose t = [True]
      from tbl and a4 and a41 and ackCon show ?thesis
         by (simp add: tiTable-SampleT-def)
    next
      assume a42:lose t ≠ [True]
     from this and tsLose have a42a:lose t = [False] by (simp add: ts-bool-False)
      show ?thesis
      proof (cases x t = [sc-ack])
        assume a421:x t = [sc-ack]
        from tbl and a4 and a42a and a421 and ackCon show ?thesis
          by (simp add: tiTable-SampleT-def)
      next
        assume a422: x t ≠ [sc-ack]
        from this and xMsg1 have a422a:x t = [] by (simp add: aType-empty)
        from tbl and a4 and a42a and a422a and ackCon show ?thesis
          by (simp add: tiTable-SampleT-def)
      qed
    qed
  next
    assume a5:st-in t = voice-com
    show ?thesis
    proof (cases stop t = [stop-vc])
      assume a51:stop t = [stop-vc]
      from tbl and a5 and a51 and ackCon show ?thesis
         by (simp add: tiTable-SampleT-def)
    next
      assume a52:stop t ≠ [stop-vc]
    from this and stopMsg1 have a52a:stop t = [] by (simp add: stopType-empty)
      show ?thesis
      proof (cases lose t = [True])
```

```
        assume a521:lose t = [True]
        from tbl and a5 and a52a and a521 and ackCon show ?thesis
          by (simp add: tiTable-SampleT-def)
      next
        assume a522:lose t ≠ [True]
      from this and tsLose have a522a:lose t = [False] by (simp add: ts-bool-False)
        from tbl and a5 and a52a and a522a and ackCon show ?thesis
          by (simp add: tiTable-SampleT-def)
      qed
    qed
  qed
qed

lemma tiTable-i1-1:
assumes tbl:tiTable-SampleT req x stop lose st-in b ack i1 vc st-out
       and ts lose
       and msg (Suc 0) x
       and msg (Suc 0) stop
       and ack t = [connection-ok]
shows i1 t = []
proof –
  from assms have
   (st-in t = call ∨ st-in t = connection-ok ∧ req t ≠ [send]) ∧
    lose t = [False]
    by (simp add: tiTable-ack-connection-ok)
  from this and tbl show ?thesis by (simp add: tiTable-SampleT-def)
qed

lemma tiTable-ack-call:
assumes tbl:tiTable-SampleT req x stop lose st-in b ack i1 vc st-out
      and ackCall:ack t = [call]
      and xMsg1:msg (Suc 0) x
      and tsLose:ts lose
      and stopMsg1:msg (Suc 0) stop
shows st-in t = init-state ∧ req t = [init]
proof –
  from tbl and tsLose have sg1:lose t = [True] ∨ lose t = [False]
    by (simp add: ts-bool-True-False)
  from tbl and xMsg1 have sg2:x t = [] ∨ x t = [sc-ack]
    by (simp add: aType-lemma)
  from tbl and stopMsg1 have sg3:stop t = [] ∨ stop t = [stop-vc]
    by (simp add: stopType-lemma)
  show ?thesis
  proof (cases st-in t)
    assume a1:st-in t = init-state
    show ?thesis
    proof (cases req t = [init])
      assume a11:req t = [init]
      from tbl and a1 and a11 and ackCall show ?thesis
      by (simp add: tiTable-SampleT-def)
  next
    assume a12:req t ≠ [init]
    from tbl and a1 and a12 and ackCall show ?thesis
       by (simp add: tiTable-SampleT-def)
  qed
next
  assume a2:st-in t = call
  show ?thesis
  proof (cases lose t = [True])
    assume a21:lose t = [True]
    from tbl and a2 and a21 and ackCall show ?thesis
      by (simp add: tiTable-SampleT-def)
  next
    assume a22:lose t ≠ [True]
    from this and tsLose have a22a:lose t = [False]
       by (simp add: ts-bool-False)
    from tbl and a2 and a22a and ackCall show ?thesis
       by (simp add: tiTable-SampleT-def)
  qed
next
  assume a3:st-in t = connection-ok
  show ?thesis
  proof (cases lose t = [True])
    assume a31:lose t = [True]
    from tbl and a3 and a31 and ackCall show ?thesis
      by (simp add: tiTable-SampleT-def)
  next
    assume a32:lose t ≠ [True]
    from this and tsLose have a32a:lose t = [False]
      by (simp add: ts-bool-False)
    show ?thesis
    proof (cases req t = [send])
      assume a321:req t = [send]
      from tbl and a3 and a32a and a321 and ackCall show ?thesis
        by (simp add: tiTable-SampleT-def)
    next
      assume a322:req t ≠ [send]
      from tbl and a3 and a32a and a322 and ackCall show ?thesis
         by (simp add: tiTable-SampleT-def)
    qed
  qed
next
  assume a4:st-in t = sending-data
  show ?thesis
  proof (cases lose t = [True])
    assume a41:lose t = [True]
    from tbl and a4 and a41 and ackCall show ?thesis
      by (simp add: tiTable-SampleT-def)
    next
      assume a42:lose t ≠ [True]
      from this and tsLose have a42a:lose t = [False]
        by (simp add: ts-bool-False)
      show ?thesis
      proof (cases x t = [sc-ack])
        assume a421:x t = [sc-ack]
        from tbl and a4 and a42a and a421 and ackCall show ?thesis
          by (simp add: tiTable-SampleT-def)
      next
        assume a422: x t ≠ [sc-ack]
        from this and xMsg1  have a422a:x t = []
          by (simp add: aType-empty)
        from tbl and a4 and a42a and a422a and ackCall show ?thesis
          by (simp add: tiTable-SampleT-def)
      qed
    qed
  next
    assume a5:st-in t = voice-com
    show ?thesis
    proof (cases stop t = [stop-vc])
      assume a51:stop t = [stop-vc]
      from tbl and a5 and a51 and ackCall show ?thesis
        by (simp add: tiTable-SampleT-def)
    next
      assume a52:stop t ≠ [stop-vc]
     from this and stopMsg1 have a52a:stop t = [] by (simp add: stopType-empty)
      show ?thesis
      proof (cases lose t = [True])
        assume a521:lose t = [True]
        from tbl and a5 and a52a and a521 and ackCall show ?thesis
          by (simp add: tiTable-SampleT-def)
      next
        assume a522:lose t ≠ [True]
      from this and tsLose have a522a:lose t = [False] by (simp add: ts-bool-False)
        from tbl and a5 and a52a and a522a and ackCall show ?thesis
          by (simp add: tiTable-SampleT-def)
      qed
    qed
  qed
qed

lemma tiTable-i1-2:
assumes tbl:tiTable-SampleT req a1 stop lose st-in b ack i1 vc st-out
       and ts lose
       and msg (Suc 0) a1
       and msg (Suc 0) stop
       and ack t = [call]
shows i1 t = []
```

proof −
  from *assms* **have** *st-in t* = *init-state* ∧ *req t* = [*init*]
    by (*simp add*: *tiTable-ack-call*)
  from *this* **and** *tbl* **show** *?thesis*
    by (*simp add*: *tiTable-SampleT-def*)
qed

lemma *tiTable-ack-init0*:
assumes *tbl*:*tiTable-SampleT req a1 stop lose*
                  (*fin-inf-append* [*init-state*] *st*)
                   b ack i1 vc st
      and *req0*:*req 0* = []
shows *ack 0* = [*init-state*]
proof −
  have (*fin-inf-append* [*init-state*] *st*) (*0*::*nat*) = *init-state*
    by (*simp add*: *fin-inf-append-def*)
  from *tbl* **and** *this* **and** *req0* **show** *?thesis* **by** (*simp add*: *tiTable-SampleT-def*)
qed

lemma *tiTable-ack-init*:
assumes *tiTable-SampleT req a1 stop lose*
                  (*fin-inf-append* [*init-state*] *st*)
                   b ack i1 vc st
      and *ts lose*
      and *msg* (*Suc 0*) *a1*
      and *msg* (*Suc 0*) *stop*
      and ∀ *t1* ≤ *t*. *req t1* = []
shows *ack t* = [*init-state*]
using *assms*
proof (*induction t*)
  case *0*
  from *this* **show** *?case*
    by (*simp add*: *tiTable-ack-init0*)
next
  case (*Suc t*)
  from *Suc* **have** *sg1*: *st t* = *hd* (*ack t*)
    by (*simp add*: *tiTable-ack-st-hd*)
  from *Suc* **and** *sg1* **have** *sg2*:
   (*fin-inf-append* [*init-state*] *st*) (*Suc t*) = *init-state*
    by (*simp add*: *correct-fin-inf-append2*)
  from *Suc* **and** *sg1* **and** *sg2* **show** *?case*
    by (*simp add*: *tiTable-SampleT-def*)
qed

lemma *tiTable-i1-3*:
assumes *tbl*:*tiTable-SampleT req x stop lose*
                  (*fin-inf-append* [*init-state*] *st*) *b ack i1 vc st*
      and *tsLose*:*ts lose*
      and *xMsg1*:*msg* (*Suc 0*) *x*

```
    and stopMsg1:msg (Suc 0) stop
    and h5:∀ t1 ≤ t. req t1 = []
shows i1 t = []
proof –
  from assms have sg1:ack t = [init-state]
   by (simp add: tiTable-ack-init)
  from assms have sg2:st t = hd (ack t)
   by (simp add: tiTable-ack-st-hd)
  from sg1 and sg2 have sg3:
  (fin-inf-append [init-state] st) (Suc t) = init-state
   by (simp add: correct-fin-inf-append2)
  from tbl and tsLose have sg4:lose t = [True] ∨ lose t = [False]
   by (simp add: ts-bool-True-False)
  from tbl and xMsg1 have sg5:x t = [] ∨ x t = [sc-ack]
   by (simp add: aType-lemma)
  from tbl and stopMsg1 have sg6:stop t = [] ∨ stop t = [stop-vc]
   by (simp add: stopType-lemma)
  show ?thesis
  proof (cases fin-inf-append [init-state] st t)
   assume a1:fin-inf-append [init-state] st t = init-state
   from assms and sg1 and sg2 and sg3 and a1 show ?thesis
     by (simp add: tiTable-SampleT-def)
  next
   assume a2:fin-inf-append [init-state] st t = call
   show ?thesis
   proof (cases lose t = [True])
     assume a21:lose t = [True]
     from tbl and a2 and a21 show ?thesis
       by (simp add: tiTable-SampleT-def)
   next
     assume a22:lose t ≠ [True]
     from this and tsLose have a22a:lose t = [False]
      by (simp add: ts-bool-False)
     from tbl and a2 and a22a show ?thesis
      by (simp add: tiTable-SampleT-def)
   qed
  next
   assume a3:fin-inf-append [init-state] st t = connection-ok
   show ?thesis
   proof (cases lose t = [True])
     assume a31:lose t = [True]
     from tbl and a3 and a31 show ?thesis
       by (simp add: tiTable-SampleT-def)
   next
     assume a32:lose t ≠ [True]
     from this and tsLose have a32a:lose t = [False]
       by (simp add: ts-bool-False)
     from h5 have a322:req t ≠ [send] by auto
     from tbl and a3 and a32a and a322 show ?thesis
```

```
      by (simp add: tiTable-SampleT-def)
    qed
  next
    assume a4:fin-inf-append [init-state] st t = sending-data
    show ?thesis
    proof (cases lose t = [True])
      assume a41:lose t = [True]
     from tbl and a4 and a41 show ?thesis by (simp add: tiTable-SampleT-def)

    next
      assume a42:lose t ≠ [True]
     from this and tsLose have a42a:lose t = [False] by (simp add: ts-bool-False)
      show ?thesis
      proof (cases x t = [sc-ack])
        assume a421:x t = [sc-ack]
        from tbl and a4 and a42a and a421 and tsLose show ?thesis
          by (simp add: tiTable-SampleT-def)
      next
        assume a422: x t ≠ [sc-ack]
        from this and xMsg1 have a422a:x t = [] by (simp add: aType-empty)
        from tbl and a4 and a42a and a422a and tsLose show ?thesis
          by (simp add: tiTable-SampleT-def)
      qed
    qed
  next
    assume a5:fin-inf-append [init-state] st t = voice-com
    show ?thesis
    proof (cases stop t = [stop-vc])
      assume a51:stop t = [stop-vc]
      from tbl and a5 and a51 and tsLose show ?thesis
         by (simp add: tiTable-SampleT-def)
    next
      assume a52:stop t ≠ [stop-vc]
     from this and stopMsg1 have a52a:stop t = [] by (simp add: stopType-empty)
      show ?thesis
      proof (cases lose t = [True])
        assume a521:lose t = [True]
        from tbl and a5 and a52a and a521 and tsLose show ?thesis
          by (simp add: tiTable-SampleT-def)
      next
        assume a522:lose t ≠ [True]
      from this and tsLose have a522a:lose t = [False] by (simp add: ts-bool-False)
        from tbl and a5 and a52a and a522a and tsLose show ?thesis
          by (simp add: tiTable-SampleT-def)
      qed
    qed
  qed
qed
```

```
lemma tiTable-st-call-ok:
assumes tbl:tiTable-SampleT req x stop lose
                 (fin-inf-append [init-state] st)
                  b ack i1 vc st
      and tsLose:ts lose
      and h3:∀ m ≤ k. ack (Suc (Suc (t + m))) = [connection-ok]
      and h4:st (Suc t) = call
shows st (Suc (Suc t)) = connection-ok
proof −
    from h4 have sg1:
     (fin-inf-append [init-state] st) (Suc (Suc t)) = call
      by (simp add: correct-fin-inf-append2)
   from tbl and tsLose have sg2:lose (Suc (Suc t)) = [True] ∨ lose (Suc (Suc t))
= [False]
    by (simp add: ts-bool-True-False)
   show ?thesis
   proof (cases lose (Suc (Suc t)) = [False])
     assume a1:lose (Suc (Suc t)) = [False]
     from tbl and a1 and sg1 show ?thesis
       by (simp add: tiTable-SampleT-def)
   next
     assume a2:lose (Suc (Suc t)) ≠ [False]
     from h3 have sg3:ack (Suc (Suc t)) = [connection-ok] by auto
     from tbl and a2 and sg1 and sg2 and sg3 show ?thesis
       by (simp add: tiTable-SampleT-def)
   qed
qed

lemma tiTable-i1-4b:
assumes tiTable-SampleT req x stop lose
                 (fin-inf-append [init-state] st) b ack i1 vc st
      and ts lose
      and msg (Suc 0) x
      and msg (Suc 0) stop
      and ∀  t1 ≤ t. req t1 = []
      and req (Suc t) = [init]
      and ∀ m < k + 3. req (t + m) ≠ [send]
      and h7:∀ m ≤ k. ack (Suc (Suc (t + m))) = [connection-ok]
      and ∀ j ≤ k + 3. lose (t + j) = [False]
      and h9:t2 < (t + 3 + k)
shows i1 t2 = []
proof (cases t2 ≤ t)
  assume t2 ≤ t
  from assms and this show ?thesis by (simp add: tiTable-i1-3)
next
  assume a2:¬ t2 ≤ t
  from assms have sg1:ack t = [init-state] by (simp add: tiTable-ack-init)
  from assms have sg2:st t =  hd (ack t) by (simp add: tiTable-ack-st-hd)
  from sg1 and sg2 have sg3:
```

```
 (fin-inf-append [init-state] st) (Suc t) = init-state
   by (simp add: correct-fin-inf-append2)
  from assms and sg3 have sg4:st (Suc t) = call
   by (simp add: tiTable-SampleT-def)
  show ?thesis
  proof (cases t2 = Suc t)
   assume a3:t2 = Suc t
   from assms and sg3 and a3 show ?thesis
    by (simp add: tiTable-SampleT-def)
  next
   assume a4:t2 ≠ Suc t
  from assms and sg4 and a4 and a2 have sg7:st (Suc (Suc t)) = connection-ok
    by (simp add: tiTable-st-call-ok)
   from assms have sg8:ack (Suc (Suc t)) = [st (Suc (Suc t))]
    by (simp add: tiTable-ack-st)
   show ?thesis
   proof (cases t2 =  Suc (Suc t))
    assume a5:t2 =  Suc (Suc t)
    from h7 and h9 and a5 have sg9:ack t2 = [connection-ok] by auto
    from assms and sg9 show ?thesis by (simp add:  tiTable-i1-1)
   next
    assume a6:t2 ≠ Suc (Suc t)
    from a6 and a4 and a2 have sg10:Suc (Suc t) < t2 by arith
    from h7 and h9 and sg10 have sg11:ack t2 = [connection-ok]
     by (simp add: aux-ack-t2)
    from assms and a6 and sg7 and sg8 and sg11 show ?thesis
     by (simp add:  tiTable-i1-1)
   qed
  qed
qed

lemma tiTable-i1-4:
assumes tiTable-SampleT req a1 stop lose
               (fin-inf-append [init-state] st) b ack i1 vc st
     and ts lose
     and msg (Suc 0) a1
     and msg (Suc 0) stop
     and ∀ t1 ≤ t. req t1 = []
     and req (Suc t) = [init]
     and ∀ m < k + 3. req (t + m) ≠ [send]
     and ∀ m ≤ k. ack (Suc (Suc (t + m))) = [connection-ok]
     and ∀ j ≤ k + 3. lose (t + j) = [False]
shows ∀ t2 < (t + 3 + k). i1 t2 = []
using assms by (simp add: tiTable-i1-4b)

lemma tiTable-ack-ok:
  assumes h1:∀ j≤ d + 2. lose (t + j) = [False]
     and tsLose:ts lose
     and stopMsg1:msg (Suc 0) stop
```

```
      and a1Msg1:msg (Suc 0) a1
      and reqNsend:req (Suc t) ≠ [send]
      and ackCon:ack t = [connection-ok]
      and tbl:tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st) b ack
i1 vc st
  shows ack (Suc t) = [connection-ok]
proof −
  from tbl and tsLose and a1Msg1 and stopMsg1 have st t =  hd (ack t)
    by (simp add: tiTable-ack-st-hd)
  from this and ackCon have sg2:
   (fin-inf-append [init-state] st) (Suc t) =  connection-ok
    by (simp add: correct-fin-inf-append2)
  have sg3a:Suc 0 ≤ d + 2 by arith
  from h1 and sg3a have sg3:lose (t + Suc 0) = [False] by auto
  from sg2 and sg3 and reqNsend and tbl show ?thesis
    by (simp add: tiTable-SampleT-def)
qed

lemma Gateway-L7a:
  assumes gw:Gateway req dt a stop lose d ack i vc
      and aMsg1:msg (Suc 0) a
      and stopMsg1:msg (Suc 0) stop
      and reqMsg1:msg (Suc 0) req
      and tsLose:ts lose
      and loseFalse:∀ j≤ d + 2. lose (t + j) = [False]
      and nsend:req (Suc t) ≠ [send]
      and ackNCon:ack (t) = [connection-ok]
  shows ack (Suc t) = [connection-ok]
proof −
  from gw and stopMsg1 and reqMsg1 and nsend obtain i1 i2 a1 a2 where
    ah1:Sample req dt a1 stop lose ack i1 vc and
    ah2:Delay a2 i1 d a1 i2 and
    ah3:Loss lose a i2 a2 i
    by (simp add: Gateway-def, auto)
  from ah2 and ah3 and aMsg1 have sg1:msg (Suc 0) a1
    by (simp add: Loss-Delay-msg-a)
  from ah1 and sg1 and stopMsg1 and reqMsg1 obtain st buffer where
    ah4:Sample-L req dt a1 stop lose (fin-inf-append [init-state] st)
            (fin-inf-append [[]] buffer)
            ack i1 vc st buffer
    by (simp add: Sample-def, auto)
  from ah4 have sg2:
    tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st)
         (fin-inf-append [[]] buffer)
         ack i1 vc st
    by (simp add: Sample-L-def)
  from loseFalse and tsLose and stopMsg1 and sg1 and
          nsend and ackNCon and sg2 show ?thesis
    by (simp add: tiTable-ack-ok)
```

```
qed

lemma Sample-L-buffer:
  assumes
    Sample-L req dt a1 stop lose (fin-inf-append [init-state] st)
         (fin-inf-append [[]] buffer)
          ack i1 vc st buffer
  shows buffer t = inf-last-ti dt t
proof −
  from assms have
   ∀ t. buffer t =
    (if dt t = [] then fin-inf-append [[]] buffer t else dt t)
    by (simp add: Sample-L-def)
  from this show ?thesis
  proof (induct t)
    case 0
    from this show ?case
      by (simp add: fin-inf-append-def)
  next
    fix t
    case (Suc t)
    from this show ?case
    proof (cases dt t = [])
      assume dt t = []
      from this and Suc show ?thesis
        by (simp add: correct-fin-inf-append1)
    next
      assume dt t ≠ []
      from this and Suc show ?thesis
        by (simp add: correct-fin-inf-append1)
    qed
  qed
qed

lemma  tiTable-SampleT-i1-buffer:
 assumes ack t = [connection-ok]
        and reqSend:req (Suc t) = [send]
        and loseFalse:∀ k≤Suc d. lose (t + k) = [False]
        and buf: buffer t = inf-last-ti dt t
        and tbl:tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st)
      (fin-inf-append [[]] buffer) ack
      i1 vc st
     and conOk:fin-inf-append [init-state] st (Suc t) = connection-ok
shows i1 (Suc t) = inf-last-ti dt t
proof −
  have sg1:Suc 0 ≤Suc d by arith
  from loseFalse and sg1 have sg2:lose (Suc t) = [False] by auto
  from tbl have
   fin-inf-append [init-state] st (Suc t) = connection-ok ∧
```

```
    req (Suc t) = [send] ∧
    lose (Suc t) = [False] ⟶
    ack (Suc t) = [sending-data] ∧
    i1 (Suc t) = (fin-inf-append [[]] buffer) (Suc t) ∧
    vc (Suc t) = [] ∧ st (Suc t) = sending-data
    by (simp add: tiTable-SampleT-def)
  from this and conOk and reqSend and sg2 have
   i1 (Suc t) = (fin-inf-append [[]] buffer) (Suc t) by simp
  from this and buf show ?thesis by (simp add: correct-fin-inf-append1)
qed

lemma Sample-L-i1-buffer:
  assumes msg (Suc 0) req
      and msg (Suc 0) a
      and stopMsg1:msg (Suc 0) stop
      and a1Msg1:msg (Suc 0) a1
      and tsLose:ts lose
      and ackCon:ack t = [connection-ok]
      and reqSend:req (Suc t) = [send]
      and loseFalse:∀ k≤Suc d. lose (t + k) = [False]
      and smpl:Sample-L req dt a1 stop lose
               (fin-inf-append [init-state] st)
               (fin-inf-append [[]] buffer) ack i1 vc st buffer
  shows i1 (Suc t) =  buffer t
proof −
  from smpl have sg1:buffer t = inf-last-ti dt t
    by (simp add: Sample-L-buffer)
  from smpl have sg2:
    ∀ t. buffer t = (if dt t = [] then fin-inf-append [[]] buffer t else dt t)
    by (simp add: Sample-L-def)
  from smpl have sg3:
    tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st)
      (fin-inf-append [[]] buffer) ack
      i1 vc st
    by (simp add: Sample-L-def)
  from sg3 and tsLose and a1Msg1 and stopMsg1 have sg4:st t =  hd (ack t)
    by (simp add: tiTable-ack-st-hd)
  from ackCon and sg4 have sg5:
    (fin-inf-append [init-state] st) (Suc t) = connection-ok
    by (simp add: correct-fin-inf-append1)
  from ackCon and reqSend and loseFalse and sg1 and
         sg3 and sg4 and sg5 have sg6:
    i1 (Suc t) = inf-last-ti dt t
     by (simp add: tiTable-SampleT-i1-buffer)
  from this and sg1 show ?thesis by simp
qed

lemma tiTable-SampleT-sending-data:
  assumes tbl: tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st)
```

```
          (fin-inf-append [[]] buffer)
          ack i1 vc st
       and loseFalse:∀ j≤2 ∗ d. lose (t + j) = [False]
       and a1e:∀ t4≤t + d + d. a1 t4 = []
       and snd:fin-inf-append [init-state] st (Suc (t + x)) = sending-data
       and h6:Suc (t + x) ≤ 2 ∗ d + t
  shows ack (Suc (t + x)) = [sending-data]
proof −
  from h6 have Suc x ≤ 2 ∗ d by arith
  from this and loseFalse have sg1:lose (t + Suc x) = [False] by auto
  from h6 have Suc (t + x) ≤t + d + d by arith
  from this and a1e have sg2:a1 (Suc (t + x)) = [] by auto
  from tbl and sg1 and sg2 and snd show ?thesis
    by (simp add: tiTable-SampleT-def)
qed

lemma Sample-sending-data:
  assumes stopMsg1:msg (Suc 0) stop
      and tsLose:ts lose
      and reqMsg1:msg (Suc 0) req
      and a1Msg1:msg (Suc 0) a1
      and loseFalse:∀ j≤2 ∗ d. lose (t + j) = [False]
      and ackSnd:ack t = [sending-data]
      and smpl:Sample req dt a1 stop lose ack i1 vc
      and xdd:x ≤ d + d
      and h9:∀ t4 ≤ t + d + d. a1 t4 = []
 shows ack (t + x) = [sending-data]
using assms
proof −
  from stopMsg1 and reqMsg1 and a1Msg1 and smpl obtain st buffer where
a1:
   Sample-L req dt a1 stop lose (fin-inf-append [init-state] st)
            (fin-inf-append [[]] buffer) ack
            i1 vc st buffer
    by (simp add: Sample-def, auto)
  from a1 have sg1:
    tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st)
        (fin-inf-append [[]] buffer)
         ack i1 vc st
     by (simp add: Sample-L-def)
  from a1 have sg2:
    ∀ t. buffer t = (if dt t = [] then fin-inf-append [[]] buffer t else dt t)
     by (simp add: Sample-L-def)
 from stopMsg1 and tsLose and a1Msg1 and ackSnd and xdd and sg1 and sg2
show ?thesis
  proof (induct x)
    case 0
    from this show ?case by simp
  next
```

```
    fix x
    case (Suc x)
    from this have sg3:st (t + x) = hd (ack (t + x))
      by (simp add: tiTable-ack-st-hd)
    from Suc have sg4:x ≤ d + d by arith
    from Suc and sg3 and sg4 have sg5:
     (fin-inf-append [init-state] st) (Suc (t + x)) = sending-data
      by (simp add: fin-inf-append-def)
    from Suc have sg6:Suc (t + x) ≤ 2 ∗ d + t by simp
    from Suc have sg7:ack (t + x) = [sending-data] by simp
    from sg1 and loseFalse and h9 and sg7 and sg5 and sg6 have sg7:
     ack (Suc (t + x)) = [sending-data]
      by (simp add: tiTable-SampleT-sending-data)
    from this show ?case by simp
  qed
qed
```

15.6 Properties of the ServiceCenter component

```
lemma ServiceCenter-a-l:
  assumes ServiceCenter i a
  shows    length (a t) ≤ (Suc 0)
proof (cases t)
  case 0
  from this and assms show ?thesis by (simp add: ServiceCenter-def)
next
  fix m assume t = Suc m
  from this and assms show ?thesis by (simp add: ServiceCenter-def)
qed

lemma ServiceCenter-a-msg:
  assumes ServiceCenter i a
  shows    msg (Suc 0) a
using assms
by (simp add: msg-def ServiceCenter-a-l)

lemma ServiceCenter-L1:
assumes ∀ t2 < x. i t2 = []
       and ServiceCenter i a
       and t ≤ x
shows a t = []
using assms
proof (induct t)
   case 0
   from this show ?case by (simp add: ServiceCenter-def)
next
   case (Suc t)
   from this show ?case by (simp add: ServiceCenter-def)
qed
```

lemma *ServiceCenter-L2*:
assumes ∀ *t2* < *x*. *i t2* = []
    and *ServiceCenter i a*
shows ∀ *t3* ≤ *x*. *a t3* = []
using *assms*
by (*clarify*, *simp add*: *ServiceCenter-L1*)

15.7 General properties of stream values

lemma *streamValue1*:
assumes *h1*:∀ *j*≤ *D* + (*z*::*nat*). *str* (*t* + *j*) = *x*
    and *h2*: *j*≤ *D*
shows *str* (*t* + *j* + *z*) = *x*
proof −
  from *h2* **have** *sg1*: *j* + *z* ≤ *D* + *z* **by** *arith*
  have *sg2*:*t* + *j* + *z* = *t* + (*j* + *z*) **by** *arith*
  from *h1* **and** *sg1* **and** *sg2* **show** *?thesis* **by** (*simp* (*no-asm-simp*))
qed

lemma *streamValue2*:
 assumes ∀ *j*≤ *D* + (*z*::*nat*). *str* (*t* + *j*) = *x*
 shows ∀ *j*≤ *D*. *str* (*t* + *j* + *z*) = *x*
using *assms* **by** (*clarify*, *simp add*: *streamValue1*)

lemma *streamValue3*:
assumes ∀ *j*≤ *D*. *str* (*t* + *j* + (*Suc y*)) = *x*
    and *j* ≤ *D*
    and *h3*:*str* (*t* + *y*) = *x*
  shows *str* (*t* + *j* + *y*) = *x*
using *assms*
proof (*induct j*)
 case *0*
 from *h3* **show** *?case* **by** *simp*
next
 case (*Suc j*)
 from *this* **show** *?case* **by** *auto*
qed

lemma *streamValue4*:
assumes ∀ *j*≤ *D*. *str* (*t* + *j* + (*Suc y*)) = *x*
    and *str* (*t* + *y*) = *x*
shows ∀ *j*≤ *D*. *str* (*t* + *j* + *y*) = *x*
using *assms*
by (*clarify*, *hypsubst-thin*, *simp add*: *streamValue3*)

lemma *streamValue5*:
assumes ∀ *j*≤ *D*. *str* (*t* + *j* + ((*i*::*nat*) + *k*)) = *x*
    and *j*≤ *D*

```
shows      str (t + i + k + j) = x
using assms
by (metis add.commute add.left-commute)

lemma streamValue6:
  assumes ∀ j≤ D. str (t + j + ((i::nat) + k)) = x
  shows     ∀ j≤ D. str (t + (i::nat) + k + j) = x
using assms by (clarify, simp add: streamValue5)

lemma streamValue7:
  assumes h1:∀ j≤d. str (t + i + k + d + Suc j) = x
      and h2:str (t + i + k + d) = x
      and h3:j≤ Suc d
  shows      str (t + i + k + d + j) = x
proof −
  from h1 have sg1:str (t + i + k + d + Suc d) = x
     by (simp (no-asm-simp), simp)
  from assms show ?thesis
  proof (cases j = Suc d)
    assume a1:j = Suc d
    from a1 and sg1 show ?thesis by simp
  next
    assume a2:j ≠ Suc d
    from a2 and h3 have sg2:j≤d by auto
    from assms and sg2 show ?thesis
    proof (cases j > 0)
      assume a3:0 < j
      from a3 and h3 have sg3:j − (1::nat) ≤ d by simp
      from a3 have sg4:Suc (j − (1::nat)) = j  by arith
      from sg3 and h1 and sg4 have sg5:str (t + i + k + d + j) = x by auto
      from sg5 show ?thesis by simp
    next
      assume a4:¬ 0 < j
      from a4 have sg6:j = 0 by simp
      from h2 and sg6 show ?thesis by simp
    qed
  qed
qed

lemma streamValue8:
assumes ∀ j≤d. str (t + i + k + d + Suc j) = x
       and str (t + i + k + d) = x
shows ∀ j≤ Suc d. str (t + i + k + d + j) = x
using assms streamValue7
  by metis

lemma arith-streamValue9aux:
Suc (t + (j + d) + (i + k)) =  Suc (t + i + k + d + j)
by arith
```

```
lemma streamValue9:
assumes h1:∀ j≤2 ∗ d. str (t + j + Suc (i + k)) = x
      and h2:j≤d
shows      str (t + i + k + d + Suc j) = x
proof −
  from h2 have (j+d) ≤2 ∗ d by arith
  from h1 and this have str (t + (j + d) + Suc (i + k)) = x by auto
  from this show ?thesis  by (simp add: arith-streamValue9aux)
qed

lemma streamValue10:
  assumes ∀ j≤2 ∗ d. str (t + j + Suc (i + k)) = x
  shows    ∀ j≤d. str (t + i + k + d + Suc j) = x
using assms
  apply clarify
  by (rule streamValue9, auto)

lemma arith-sum1:(t::nat) + (i + k + d) =  t + i + k + d
by arith

lemma arith-sum2:Suc (Suc (t + k + j)) = Suc (Suc (t + (k + j)))
by arith

lemma arith-sum4:t + 3 + k + d = Suc (t + (2::nat) + k + d)
by arith

lemma streamValue11:
assumes h1:∀ j≤2 ∗ d + (4 + k). lose (t + j) = x
      and h2:j≤Suc d
shows      lose (t + 2 + k + j) = x
proof −
  from h2 have sg1:2 + k + j ≤2 ∗ d + (4 + k) by arith
  have sg2:Suc (Suc (t + k + j)) = Suc (Suc (t + (k + j))) by arith
  from sg1 and h1 have lose (t + (2 + k + j)) = x by blast
  from this and sg2 show ?thesis by (simp add: arith-sum2)
qed

lemma streamValue12:
 assumes  ∀ j≤2 ∗ d + (4 + k). lose (t + j) = x
 shows     ∀ j≤Suc d. lose (t + 2 + k + j) = x
using assms
  apply clarify
  by (rule streamValue11, auto)

lemma streamValue43:
  assumes ∀ j≤2 ∗ d + ((4::nat) + k). lose (t + j) = [False]
  shows    ∀ j≤2 ∗ d. lose ((t + (3::nat) + k) + j) = [False]
proof −
```

```
  from assms have sg1:∀ j≤2 ∗ d. lose (t + j + (4 + k)) = [False]
    by (simp add: streamValue2)
  have sg2:Suc (3 + k) = (4 + k) by arith
  from sg1 and sg2 have sg3:∀ j≤2 ∗ d. lose (t + j + Suc (3 + k)) = [False]
    by (simp (no-asm-simp))
  from assms have sg4:lose (t + (3 + k)) = [False] by auto
  from sg3 and sg4 have sg5:∀ j≤2 ∗ d. lose (t + j + (3 + k)) = [False]
    by (rule streamValue4)
  from sg5 show ?thesis by (rule streamValue6)
qed

end



theory Gateway-proof
imports Gateway-proof-aux
begin
```

15.8 Properties of the Gateway

```
lemma Gateway-L1:
  assumes h1:Gateway req dt a stop lose d ack i vc
      and h2:msg (Suc 0) req
      and h3:msg (Suc 0) a
      and h4:msg (Suc 0) stop
      and h5:ts lose
      and h6:ack t = [init-state]
      and h7:req (Suc t) = [init]
      and h8:lose (Suc t) = [False]
      and h9:lose (Suc (Suc t)) = [False]
  shows ack (Suc (Suc t)) = [connection-ok]
proof −
  from h1 obtain i1 i2 x y
    where a1:Sample req dt x stop lose ack i1 vc
      and a2:Delay y i1 d x i2
      and a3:Loss lose a i2 y i
    by (simp only: Gateway-def, auto)
  from a2 and a3 and h3 have sg1:msg (Suc 0) x
    by (simp add: Loss-Delay-msg-a)
  from a1 and h2 and h4 and sg1 obtain st buffer where a4:
    tiTable-SampleT req x stop lose
        (fin-inf-append [init-state] st) (fin-inf-append [[]] buffer) ack
         i1 vc st
     by (simp add: Sample-def Sample-L-def, auto)
  from a4 and h5 and sg1 and h4 have sg2:st t =  hd (ack t)
    by (simp add: tiTable-ack-st-hd)
  from h6 and sg1 and sg2 and h4 have sg3:
   (fin-inf-append [init-state] st) (Suc t) = init-state
```

```
    by (simp add: correct-fin-inf-append1)
  from a4 and h7 and sg3 have sg4:st (Suc t) = call
    by (simp add: tiTable-SampleT-def)
  from sg4 have sg5:(fin-inf-append [init-state] st) (Suc (Suc t)) = call
    by (simp add: correct-fin-inf-append1)
  from a4 and sg5 and assms show ?thesis
    by (simp add: tiTable-SampleT-def)
qed

lemma Gateway-L2:
  assumes h1:Gateway req dt a stop lose d ack i vc
      and h2:msg (Suc 0) req
      and h3:msg (Suc 0) a
      and h4:msg (Suc 0) stop
      and h5:ts lose
      and h6:ack t = [connection-ok]
      and h7:req (Suc t) = [send]
      and h8:∀ k≤Suc d. lose (t + k) = [False]
  shows i (Suc (t + d)) = inf-last-ti dt t
proof –
  from h1 obtain i1 i2 x y
    where a1:Sample req dt x stop lose ack i1 vc
      and a2:Delay y i1 d x i2
      and a3:Loss lose a i2 y i
    by (simp only: Gateway-def, auto)
  from a2 and a3 and h3 have sg1:msg (Suc 0) x
    by (simp add: Loss-Delay-msg-a)
  from a1 and h2 and h4 and sg1 obtain st buffer where a4:
    Sample-L req dt x stop lose (fin-inf-append [init-state] st)
        (fin-inf-append [[]] buffer) ack i1 vc st buffer
    by (simp add: Sample-def, auto)
  from a4 have sg2:buffer t = inf-last-ti dt t
    by (simp add: Sample-L-buffer)
  from assms and a1 and a4 and sg1 and sg2 have sg3:i1 (Suc t) = buffer t
    by (simp add: Sample-L-i1-buffer)
  from a2 and sg1 have sg4:i2 ((Suc t) + d) = i1 (Suc t)
    by (simp add: Delay-def)
  from a3 and h8 have sg5:i ((Suc t) + d) = i2 ((Suc t) + d)
    by (simp add: Loss-def, auto)
  from sg5 and sg4 and sg3 and sg2 show ?thesis by simp
qed

lemma Gateway-L3:
  assumes h1:Gateway req dt a stop lose d ack i vc
      and h2:msg (Suc 0) req
      and h3:msg (Suc 0) a
      and h4:msg (Suc 0) stop
      and h5:ts lose
      and h6:ack t = [connection-ok]
    and h7:req (Suc t) = [send]
    and h8:∀ k≤Suc d. lose (t + k) = [False]
  shows ack (Suc t) = [sending-data]
proof −
  from h1 obtain i1 i2 x y
    where a1:Sample req dt x stop lose ack i1 vc
      and a2:Delay y i1 d x i2
      and a3:Loss lose a i2 y i
    by (simp only: Gateway-def, auto)
  from a2 and a3 and h3 have sg1:msg (Suc 0) x
    by (simp add: Loss-Delay-msg-a)
  from a1 and h2 and h4 and sg1 obtain st buffer where a4:
    tiTable-SampleT req x stop lose
        (fin-inf-append [init-state] st) (fin-inf-append [[]] buffer) ack
         i1 vc st
     by (simp add: Sample-def Sample-L-def, auto)
  from a4 and h5 and sg1 and h4 have sg2:st t =  hd (ack t)
    by (simp add: tiTable-ack-st-hd)
 from sg2 and h6 have sg3:(fin-inf-append [init-state] st) (Suc t) = connection-ok
    by (simp add: correct-fin-inf-append1)
  from h8 have sg4:lose (Suc t) = [False] by auto
  from a4 and sg3 and sg4 and h7 have sg5:st (Suc t) = sending-data
    by (simp add: tiTable-SampleT-def)
  from a4 and h2 and sg1 and h4 and h5 have sg6:ack (Suc t) = [st (Suc t)]
    by (simp add: tiTable-ack-st)
  from sg5 and sg6 show ?thesis by simp
qed

lemma Gateway-L4:
  assumes h1:Gateway req dt a stop lose d ack i vc
    and h2:msg (Suc 0) req
    and h3:msg (Suc 0) a
    and h4:msg (Suc 0) stop
    and h5:ts lose
    and h6:ack (t + d) = [sending-data]
    and h7:a (Suc t) = [sc-ack]
    and h8:∀ k≤Suc d. lose (t + k) = [False]
  shows vc (Suc (t + d)) = [vc-com]
proof −
  from h1 obtain i1 i2 x y
    where a1:Sample req dt x stop lose ack i1 vc
       and a2:Delay y i1 d x i2
       and a3:Loss lose a i2 y i
    by (simp only: Gateway-def, auto)
  from a2 and a3 and h3 have sg1:msg (Suc 0) x
    by (simp add: Loss-Delay-msg-a)
  from a1 and h2 and h4 and sg1 obtain st buffer where a4:
    tiTable-SampleT req x stop lose
        (fin-inf-append [init-state] st) (fin-inf-append [[]] buffer) ack
         i1 vc st
     by (simp add: Sample-def Sample-L-def, auto)
  from a4 and h5 and sg1 and h4 have sg2:st (t+d) = hd (ack (t+d))
    by (simp add: tiTable-ack-st-hd)
  from sg2 and h6 have sg3:(fin-inf-append [init-state] st) (Suc (t+d)) = send-
  ing-data
    by (simp add: correct-fin-inf-append1)
  from a3 and h8 have sg4:y (Suc t) = a (Suc t)
    by (simp add: Loss-def, auto)
  from a2 and sg1 have sg5:x ((Suc t) + d) = y (Suc t)
    by (simp add: Delay-def)
  from sg5 and sg4 and h7 have sg6: x (Suc (t + d)) = [sc-ack] by simp
  from h8 have sg7:lose (Suc (t + d)) = [False] by auto
  from sg6 and a4 and h2 and sg1 and h4 and h5 and sg7 and sg3 show
  ?thesis
    by (simp add: tiTable-SampleT-def)
qed

lemma Gateway-L5:
  assumes h1:Gateway req dt a stop lose d ack i vc
         and h2:msg (Suc 0) req
         and h3:msg (Suc 0) a
         and h4:msg (Suc 0) stop
         and h5:ts lose
         and h6:ack (t + d) = [sending-data]
         and h7:∀ j ≤ Suc d. a (t+j) = []
         and h8:∀ k≤ (d + d). lose (t + k) = [False]
  shows j ≤ d ⟶ ack (t+d+j) = [sending-data]
proof −
  from h1 obtain i1 i2 x y
    where a1:Sample req dt x stop lose ack i1 vc
       and a2:Delay y i1 d x i2
       and a3:Loss lose a i2 y i
    by (simp only: Gateway-def, auto)
  from a2 and a3 and h3 have sg1:msg (Suc 0) x
    by (simp add: Loss-Delay-msg-a)
  from a1 and h2 and h4 and sg1 obtain st buffer where a4:
    tiTable-SampleT req x stop lose
        (fin-inf-append [init-state] st) (fin-inf-append [[]] buffer) ack
         i1 vc st
     by (simp add: Sample-def Sample-L-def, auto)
  from assms and a2 and a3 and sg1 and a4 show ?thesis
  proof (induct j)
    case 0 then show ?case by simp
  next
    case (Suc j)
    then show ?case
    proof (cases Suc j ≤ d)
        assume ¬ Suc j ≤ d then show ?thesis by simp
    next
        assume a0:Suc j ≤ d
        then have d + Suc j ≤ d + d   by arith
        then have sg3:Suc (d + j) ≤ d + d by arith
        from a4 and h2 and sg1 and h4 and h5 have sg4:
         st (t+d+j) =  hd (ack (t+d+j))
         by (simp add: tiTable-ack-st-hd)
        from Suc and a0 and sg4 have sg5:
         (fin-inf-append [init-state] st) (Suc (t+d+j)) = sending-data
         by (simp add: correct-fin-inf-append1)
        from h7 and a0  have sg6:∀ j≤ d. a (t + Suc j) = [] by auto
        from sg6 and a3 and a0 and h5 have sg7:y (t + (Suc j)) = []
         by (rule Loss-L5Suc)
        from sg7 and a2 have sg8a:x (t + d + (Suc j)) = []
         by (simp add: Delay-def)
        then  have sg8:x (Suc (t + d + j)) = [] by simp
        have sg9:Suc (t + d + j) = Suc (t + (d + j)) by arith
        from a4 have sg10:
         fin-inf-append [init-state] st (Suc (t + d + j)) = sending-data ∧
          x (Suc (t + d + j)) = [] ∧
          lose (Suc (t + d + j)) = [False] ⟶
          ack (Suc (t + d + j)) = [sending-data]
         by (simp add: tiTable-SampleT-def)
        from h8 and sg3 have sg11:lose (t + Suc (d + j)) = [False] by blast
        have Suc (t + d + j) = t + Suc (d + j)  by arith
        from this and sg11 have lose (Suc (t + d + j)) = [False]
         by  (simp (no-asm-simp), simp)
     from sg10 and sg5 and sg8a and this show ?thesis by simp
    qed
  qed
qed

lemma Gateway-L6-induction:
 assumes h1:msg (Suc 0) req
     and h2:msg (Suc 0) x
     and h3:msg (Suc 0) stop
     and h4:ts lose
     and h5:∀ j≤ k. lose (t + j) = [False]
     and h6:∀ m ≤ k. req (t + m) ≠ [send]
     and h7:ack t = [connection-ok]
     and h8:Sample req dt x1 stop lose ack i1 vc
     and h9:Delay x2 i1 d x1 i2
     and h10:Loss lose x i2 x2 i
     and h11:m ≤ k
 shows ack (t + m) = [connection-ok]
using assms
proof (induct m)
  case 0 then show ?case  by simp
next
```

```
  case (Suc m)
  then have sg1:msg (Suc 0) x1 by (simp add: Loss-Delay-msg-a)
  from Suc and this obtain st buffer where
    a1:tiTable-SampleT req x1 stop lose (fin-inf-append [init-state] st)
        (fin-inf-append [[]] buffer) ack i1 vc st  and
    a2:∀ t. buffer t = (if dt t = [] then fin-inf-append [[]] buffer t else dt t)
    by (simp add: Sample-def Sample-L-def, auto)
  from a1 and sg1 and h3 and h4 have sg2:st (t + m) =  hd (ack (t + m))
    by (simp add: tiTable-ack-st-hd)
  from Suc have sg3:ack (t + m) = [connection-ok] by simp
  from a1 and sg2 and sg3 have sg4:
  (fin-inf-append [init-state] st) (Suc (t + m)) = connection-ok
    by (simp add: fin-inf-append-def)
  from Suc have sg5:Suc m ≤ k by simp
  from sg5 and h5 have sg6:lose (Suc (t + m)) = [False] by auto
  from h6 and sg5 have sg7:req (Suc (t + m)) ≠ [send] by auto
  from a1 and sg3 and sg4 and sg5 and sg6 and sg7 show ?case
    by (simp add: tiTable-SampleT-def)
qed

lemma Gateway-L6:
 assumes Gateway req dt a stop lose d ack i vc
     and ∀ m≤k. req (t + m) ≠ [send]
     and ∀ j≤k. lose (t + j) = [False]
     and ack t = [connection-ok]
     and msg (Suc 0) req
     and msg (Suc 0) stop
     and msg (Suc 0) a
     and ts lose
 shows ∀ m≤k. ack (t + m) = [connection-ok]
using assms
by (simp add: Gateway-def, clarify, simp add: Gateway-L6-induction)

lemma Gateway-L6a:
 assumes Gateway req dt a stop lose d ack i vc
     and ∀ m≤k. req (t + 2 + m) ≠ [send]
     and ∀ j≤k. lose (t + 2 + j) = [False]
     and ack (t + 2) = [connection-ok]
     and msg (Suc 0) req
     and msg (Suc 0) stop
     and msg (Suc 0) a
     and ts lose
 shows ∀ m≤k. ack (t + 2 + m) = [connection-ok]
using assms by (rule Gateway-L6)

lemma aux-k3req:
assumes h1:∀ m<k + 3. req (t + m) ≠ [send]
      and h2:m ≤ k
shows req (Suc (Suc (t + m))) ≠ [send]
```

```
proof −
  from h2 have m + 2 < k + 3 by arith
  from h1 and this have req (t + (m + 2)) ≠ [send] by blast
  then show ?thesis by simp
qed

lemma aux3lose:
assumes h1:∀ j≤k + d + 3. lose (t + j) = [False]
       and h2:j ≤ k
shows lose (Suc (Suc (t + j))) = [False]
proof −
  from h2 have j + 2 ≤k + d + 3 by arith
  from h1 and this have lose (t + (j + 2)) = [False] by blast
  then show ?thesis by simp
qed


lemma Gateway-L7:
assumes h1:Gateway req dt a stop lose d ack i vc
     and h2:ts lose
     and h3:msg (Suc 0) a
     and h4:msg (Suc 0) stop
     and h5:msg (Suc 0) req
     and h6:req (Suc t) = [init]
     and h7:∀ m < (k + 3). req (t + m) ≠ [send]
     and h8:req (t + 3 + k) = [send]
     and h9:ack t = [init-state]
     and h10:∀ j≤k + d + 3. lose (t + j) = [False]
     and h11:∀  t1 ≤ t. req t1 = []
shows ∀  t2 < (t + 3 + k + d). i t2 = []
proof −
  have Suc 0 ≤ k + d + 3 by arith
  from h10 and this have lose (t + Suc 0) = [False] by blast
  then have sg1:lose (Suc t) = [False] by simp
  have Suc (Suc 0)≤ k + d + 3 by arith
  from h10 and this have lose (t + Suc (Suc 0)) = [False] by blast
  then have sg2:lose (Suc (Suc t)) = [False] by simp
  from h1 and h2 and h3 and h4 and h5 and h6 and h9 and sg1 and sg2
     have sg3:ack (t + 2) = [connection-ok]
    by (simp add: Gateway-L1)
  from h7 and this have sg4:∀ m≤ k. req ((t + 2) + m) ≠ [send]
    by (auto, simp add: aux-k3req)
  from h10 have sg5:∀ j≤ k. lose ((t + 2) + j) = [False]
    by (auto, simp add: aux3lose)
  from h1 and sg4 and sg5 and sg3 and h5 and h4 and h3 and h2 have sg6:
   ∀ m ≤  k. ack ((t + 2) + m) = [connection-ok]
    by (rule Gateway-L6a)
  from sg6 have sg7:ack (t + 2 + k) = [connection-ok]  by auto
  from h1 obtain i1 i2 x y where
```

```
    a1:Sample req dt x stop lose ack i1 vc and
    a2:Delay y i1 d x i2 and
    a3:Loss lose a i2 y i
    by (simp add: Gateway-def, auto)
  from h3 and a2 and a3  have sg8:msg (Suc 0) x
    by (simp add: Loss-Delay-msg-a)
  from a1 and sg8 and h4 and h5 obtain st buffer where
    a4:tiTable-SampleT req x stop lose (fin-inf-append [init-state] st)
         (fin-inf-append [[]] buffer) ack i1 vc st  and
    a5:∀ t. buffer t = (if dt t = [] then fin-inf-append [[]] buffer t else dt t)
    by (simp add: Sample-def Sample-L-def, auto)
  from a4 and h2 and sg8 and h4 and h11 and h6 and h7 and sg6 and h10
    have sg9:∀ t1 < (t + 3 + k). i1 t1 = []
    by (simp add: tiTable-i1-4)
  from sg9 and a2 have sg10:∀ t2 < (t + 3 + k + d). i2 t2 = []
    by (rule Delay-L2)
  from sg10 and a3 and h2 show ?thesis by (rule Loss-L2)
qed

lemma Gateway-L8a:
  assumes h1:Gateway req dt a stop lose d ack i vc
      and h2:msg (Suc 0) req
      and h3:msg (Suc 0) stop
      and h4:msg (Suc 0) a
      and h5:ts lose
      and h6:∀ j≤2 ∗ d. lose (t + j) = [False]
      and h7:ack t = [sending-data]
      and h8:∀ t3 ≤ t + d. a t3 = []
      and h9:x ≤ d + d
  shows ack (t + x) = [sending-data]
proof −
  from h1 obtain i1 i2 x y where
    a1:Sample req dt x stop lose ack i1 vc and
    a2:Delay y i1 d x i2 and
    a3:Loss lose a i2 y i
    by (simp add: Gateway-def, auto)
  from h8 and a3 and h5 have sg1:∀ t3 ≤ t + d. y t3 = [] by (rule Loss-L6)
  from sg1 and a2 have sg2:∀ t4 ≤ t + d + d. x t4 = []   by (rule Delay-L4)
 from h4 and a2 and a3 have sg3:msg (Suc 0) x by (simp add:  Loss-Delay-msg-a)
  from h3 and h5 and h2 and sg3 and h6 and h7 and a1 and h9 and sg2
show ?thesis
    by (simp add: Sample-sending-data)
qed


lemma Gateway-L8:
assumes Gateway req dt a stop lose d ack i vc
       and msg (Suc 0) req
       and msg (Suc 0) stop
       and msg (Suc 0) a
       and ts lose
       and ∀ j≤2 ∗ d. lose (t + j) = [False]
       and ack t = [sending-data]
       and ∀ t3 ≤ t + d. a t3 = []
shows ∀ x ≤ d + d. ack (t + x) = [sending-data]
using assms
by (simp add: Gateway-L8a)
```

15.9 Proof of the Refinement Relation for the Gateway Requirements

```
lemma Gateway-L0:
 assumes Gateway req dt a stop lose d ack i vc
 shows   GatewayReq req dt a stop lose d ack i vc
using assms
by (simp add: GatewayReq-def Gateway-L1 Gateway-L2 Gateway-L3 Gateway-L4)
```

15.10 Lemmas about Gateway Requirements

```
lemma GatewayReq-L1:
  assumes h1:msg (Suc 0) req
      and h2:msg (Suc 0) stop
      and h3:msg (Suc 0) a
      and h4:ts lose
      and h6:req (t + 3 + k) = [send]
      and h7:∀ j≤2 ∗ d + (4 + k). lose (t + j) = [False]
      and h9:∀ m≤ k. ack (t + 2 + m) = [connection-ok]
      and h10:GatewayReq req dt a stop lose d ack i vc
 shows ack (t + 3 + k) = [sending-data]
proof −
  from h9 have sg1:ack (Suc (Suc (t + k))) = [connection-ok] by auto
  from h7 have sg2:
   ∀ ka≤Suc d. lose (Suc (Suc (t + k + ka))) = [False]
    by (simp add: aux-lemma-lose-1)
  from h1 and h2 and h3 and h4 and h6 and h10 and sg1 and sg2 have sg3:
   ack (t + 2 + k) = [connection-ok] ∧
    req (Suc (t + 2 + k)) = [send] ∧ (∀ k≤Suc d. lose (t + k) = [False]) ⟶
    ack (Suc (t + 2 + k)) = [sending-data]
    by (simp add: GatewayReq-def)
  have t + 3 + k = Suc (Suc (Suc (t + k))) by arith
  from sg3 and sg1 and h6 and h7 and this show ?thesis
    by (simp add: eval-nat-numeral)
qed

lemma GatewayReq-L2:
 assumes  h1:msg (Suc 0) req
      and h2:msg (Suc 0) stop
      and h3:msg (Suc 0) a
```

  and *h4*:*ts lose*
  and *h5*:*GatewayReq req dt a stop lose d ack i vc*
  and *h6*:*req* $(t + 3 + k) = [send]$
  and *h7*:*inf-last-ti dt t* $\neq []$
  and *h8*:$\forall j \leq 2 * d + (4 + k).$ *lose* $(t + j) = [False]$
  and *h9*:$\forall m \leq k.$ *ack* $(t + 2 + m) = [connection\text{-}ok]$
shows *i* $(t + 3 + k + d) \neq []$
proof −
from *h8* **have** *sg1*:$(\forall\ (x::nat).\ x \leq (d+1) \longrightarrow lose\ (t+x) = [False])$
  by (*simp add*: *aux-lemma-lose-2*)
from *h8* **have** *sg2*:$\forall ka \leq Suc\ d.\ lose\ (Suc\ (Suc\ (t + k + ka))) = [False]$
  by (*simp add*: *aux-lemma-lose-1*)
from *h9* **have** *sg3*:*ack* $(t + 2 + k) = [connection\text{-}ok]$ **by** *simp*
from *h1* **and** *h2* **and** *h3* **and** *h4* **and** *h5* **and** *h6* **and** *sg2* **and** *sg3* **have** *sg4*:
 $ack\ (t + 2 + k) = [connection\text{-}ok] \wedge$
  $req\ (Suc\ (t + 2 + k)) = [send] \wedge (\forall k \leq Suc\ d.\ lose\ (t + k) = [False]) \longrightarrow$
  $i\ (Suc\ (t + 2 + k + d)) = inf\text{-}last\text{-}ti\ dt\ (t + 2 + k)$
  by (*simp add*: *GatewayReq-def*, *auto*)
from *h7* **have** *sg5*:*inf-last-ti dt* $(t + 2 + k) \neq []$
  by (*simp add*: *inf-last-ti-nonempty-k*)
have *sg6*:$t + 3 + k = Suc\ (Suc\ (Suc\ (t + k)))$ **by** *arith*
have $t + 2 + k = Suc\ (Suc\ (t + k))$ **by** *arith*
from *sg1* **and** *sg2* **and** *sg3* **and** *sg4* **and** *sg5* **and** *sg6* **and** *this* **and** *h6* **show** *?thesis*
  by (*simp add*: *eval-nat-numeral*)
qed

15.11 Properties of the Gateway System

lemma *GatewaySystem-L1aux*:
assumes *msg* (*Suc 0*) *req*
  and *msg* (*Suc 0*) *stop*
  and *msg* (*Suc 0*) *a*
  and *ts lose*
  and $msg\ (Suc\ 0)\ req \wedge msg\ (Suc\ 0)\ a \wedge msg\ (Suc\ 0)\ stop \wedge ts\ lose \longrightarrow$
  $(\forall t.\ (ack\ t = [init\text{-}state] \wedge$
   $req\ (Suc\ t) = [init] \wedge lose\ (Suc\ t) = [False] \wedge$
   $lose\ (Suc\ (Suc\ t)) = [False] \longrightarrow$
   $ack\ (Suc\ (Suc\ t)) = [connection\text{-}ok]) \wedge$
  $(ack\ t = [connection\text{-}ok] \wedge req\ (Suc\ t) = [send] \wedge$
  $(\forall k \leq Suc\ d.\ lose\ (t + k) = [False]) \longrightarrow$
   $i\ (Suc\ (t + d)) = inf\text{-}last\text{-}ti\ dt\ t \wedge ack\ (Suc\ t) = [sending\text{-}data]) \wedge$
  $(ack\ (t + d) = [sending\text{-}data] \wedge a\ (Suc\ t) = [sc\text{-}ack] \wedge$
   $(\forall k \leq Suc\ d.\ lose\ (t + k) = [False]) \longrightarrow$
   $vc\ (Suc\ (t + d)) = [vc\text{-}com]))$
shows $ack\ (t + 3 + k + d + d) = [sending\text{-}data] \wedge$
   $a\ (Suc\ (t + 3 + k + d)) = [sc\text{-}ack] \wedge$
  $(\forall ka \leq Suc\ d.\ lose\ (t + 3 + k + d + ka) = [False]) \longrightarrow$
  $vc\ (Suc\ (t + 3 + k + d + d)) = [vc\text{-}com]$

```
using assms by blast

lemma GatewaySystem-L3aux:
assumes msg (Suc 0) req
      and msg (Suc 0) stop
      and msg (Suc 0) a
      and ts lose
      and msg (Suc 0) req ∧ msg (Suc 0) a ∧ msg (Suc 0) stop ∧ ts lose ⟶
       (∀ t. (ack t = [init-state] ∧
         req (Suc t) = [init] ∧ lose (Suc t) = [False] ∧
         lose (Suc (Suc t)) = [False] ⟶
         ack (Suc (Suc t)) = [connection-ok]) ∧
        (ack t = [connection-ok] ∧ req (Suc t) = [send] ∧
        (∀ k≤Suc d. lose (t + k) = [False]) ⟶
         i (Suc (t + d)) = inf-last-ti dt t ∧ ack (Suc t) = [sending-data]) ∧
        (ack (t + d) = [sending-data] ∧ a (Suc t) = [sc-ack] ∧
         (∀ k≤Suc d. lose (t + k) = [False]) ⟶
         vc (Suc (t + d)) = [vc-com]))
shows ack (t + 2 + k) = [connection-ok] ∧
        req (Suc (t + 2 + k)) = [send] ∧
        (∀ j≤Suc d. lose (t + 2 + k + j) = [False]) ⟶
        i (Suc (t + 2 + k + d)) = inf-last-ti dt (t + 2 + k)
using assms by blast

lemma GatewaySystem-L1:
 assumes h2:ServiceCenter i a
     and h3:GatewayReq req dt a stop lose d ack i vc
     and h4:msg (Suc 0) req
     and h5:msg (Suc 0) stop
     and h6:msg (Suc 0) a
     and h7:ts lose
     and h9:∀ j≤2 ∗ d + (4 + k). lose (t + j) = [False]
     and h11:i (t + 3 + k + d) ≠ []
     and h14:∀ x ≤ d + d. ack (t + 3 + k + x) = [sending-data]
 shows vc (2 ∗ d + (t + (4 + k))) = [vc-com]
proof −
  from h2 have ∀ t. a (Suc t) = (if i t = [] then [] else [sc-ack])
    by (simp add:ServiceCenter-def)
  then have sg1:
    a (Suc (t + 3 + k + d)) = (if i (t + 3 + k + d) = [] then [] else [sc-ack])
     by blast
  from sg1 and h11 have sg2:a (Suc (t + 3 + k + d)) = [sc-ack] by auto
  from h14 have sg3:ack (t + 3 + k + 2∗d) = [sending-data] by simp
  from h4 and h5 and h6 and h7 and h3 have sg4:
     ack (t + 3 + k + d + d) = [sending-data] ∧ a (Suc (t + 3 + k + d)) =
[sc-ack] ∧
     (∀ ka≤Suc d. lose (t + 3 + k + d + ka) = [False]) ⟶
     vc (Suc (t + 3 + k + d + d)) = [vc-com]
    apply (simp only: GatewayReq-def)
    by (rule GatewaySystem-L1aux, auto)
  from h9 have sg5:∀ka≤Suc d. lose (d + (t + (3 + k)) + ka) = [False]
    by (simp add: aux-lemma-lose-3)
  have sg5a:d + (t + (3 + k)) = t + 3 + k + d by arith
  from sg5 and sg5a have sg5b:∀ka≤Suc d. lose (t + 3 + k + d + ka) = [False]

    by auto
  have sg6:(t + 3 + k + 2 ∗ d)  = (2 ∗ d + (t + (3 + k))) by arith
   have sg7:Suc (Suc (Suc (t + k + (d + d)))) = Suc (Suc (Suc (t + k + d +
d)))
    by arith
  have Suc (Suc (Suc (Suc (t + k + d + d)))) =
           Suc (Suc (Suc (Suc (d + d + (t + k))))) by arith
  from sg4 and sg3 and sg2 and sg5b and sg6 and sg7 and this show ?thesis
    by (simp add: eval-nat-numeral)
qed

lemma aux4lose1:
assumes h1:∀j≤2 ∗ d + (4 + k). lose (t + j) = [False]
       and h2:j ≤ k
shows lose (t + (2::nat) + j) = [False]
proof −
  from h2 have (2::nat) + j ≤ (2::nat) ∗ d + (4 + k) by arith
  from h1 and this have lose (t + (2 + j)) = [False] by blast
  then show ?thesis by simp
qed

lemma aux4lose2:
assumes ∀j≤2 ∗ d + (4 + k). lose (t + j) = [False]
       and 3 + k + d ≤ 2 ∗ d + (4 + k)
shows lose (t + (3::nat) + k + d) = [False]
proof −
  from assms have lose (t + ((3::nat) + k + d)) = [False] by blast
  then show ?thesis by (simp add: arith-sum1)
qed

lemma aux4req:
assumes h1:∀ (m::nat) ≤ k + 2. req (t + m) ≠ [send]
      and h2:m ≤ k
      and h3:req (t + 2 + m) = [send] shows False
proof −
  from h2 have (2::nat) + m ≤ k + (2::nat) by arith
  from h1 and this have req (t + (2 + m)) ≠ [send] by blast
  from this and h3 show ?thesis by simp
qed

lemma GatewaySystem-L2:
 assumes h1:Gateway req dt a stop lose d ack i vc
     and h2:ServiceCenter i a
     and h3:GatewayReq req dt a stop lose d ack i vc
     and h4:msg (Suc 0) req
     and h5:msg (Suc 0) stop
     and h6:msg (Suc 0) a
     and h7:ts lose
     and h8:ack t = [init-state]
     and h9:req (Suc t) = [init]
     and h10:∀ t1≤t. req t1 = []
     and h11:∀ m ≤ k + 2. req (t + m) ≠ [send]
     and h12:req (t + 3 + k) = [send]
     and h13:inf-last-ti dt t ≠ []
     and h14:∀ j≤2 ∗ d + (4 + k). lose (t + j) = [False]
 shows vc (2 ∗ d + (t + (4 + k))) = [vc-com]
proof −
  have Suc 0 ≤ 2 ∗ d + (4 + k) by arith
  from h14 and this have lose (t + Suc 0) = [False] by blast
  then have sg1:lose (Suc t) = [False] by simp
  have Suc (Suc 0) ≤ 2 ∗ d + (4 + k) by arith
  from h14 and this have lose (t + Suc (Suc 0)) = [False] by blast
  then have sg2:lose (Suc (Suc t)) = [False] by simp
   from h3 and h4 and h5 and h6 and h7 and h8 and h9 and sg1 and sg2
have sg3:
   ack (t + 2) = [connection-ok]
    by (simp add: GatewayReq-def)
  from h14 have sg4: ∀ j≤k. lose (t + 2 + j) = [False]
    by (clarify, rule aux4lose1, simp)
  from h11 have sg5:∀ m ≤ k. req (t + 2 + m) ≠ [send]
    by (clarify, rule aux4req, auto)

  from h1 and sg5 and sg4 and sg3 and h4 and h5 and h6 and h7 have sg6:
   ∀ m ≤ k. ack (t + 2 + m) = [connection-ok]
    by (rule Gateway-L6)

   from h3 and h4 and h5 and h6 and h7 and h12 and h14 and sg6 have
sg10:
   ack (t + 3 + k) = [sending-data]
    by (simp add: GatewayReq-L1)
  from h3 and h4 and h5 and h6 and h7 and h12 and h13 and h14 and sg6
have sg11:
   i (t + 3 + k + d) ≠ []
    by (simp add: GatewayReq-L2)

  from h11 have sg12:∀ m < k + 3. req (t + m) ≠ [send] by auto
  from h14 have sg13:∀ j≤k + d + 3. lose (t + j) = [False] by auto
  from h1 and h7 and h6 and h5 and h4 and h9 and sg12
        and h12 and h8 and sg13 and h10
    have sg14:∀ t2 < (t + 3 + k + d). i t2 = []
    by (simp add: Gateway-L7)
  from sg14 and h2 have sg15:∀ t3 ≤ (t + 3 + k + d). a t3 = []
```

```
    by (simp add: ServiceCenter-L2)
  from h14 have sg18:∀ j≤2 ∗ d. lose ((t + 3 + k) + j) = [False]
    by (simp add: streamValue43)
  from h14  have sg16a:∀ j≤2 ∗ d. lose (t + j + (4 + k)) = [False]
    by (simp add: streamValue2)
  have sg16b:Suc (3 + k) = (4 + k) by arith
   from sg16a and sg16b have sg16:∀ j≤2 ∗ d. lose (t + j + Suc (3 + k)) =
[False]
    by (simp (no-asm-simp))
  from h1 and h4 and h5 and h6 and h7 and sg18 and sg10 and sg15 have
sg19:
    ∀ x ≤ d + d. ack (t + 3 + k + x) = [sending-data]
     by (simp add: Gateway-L8)
  from sg19 have sg19a:ack (t + 3 + k + d + d) = [sending-data] by auto
  from sg16 have sg20a:∀ j≤ d. lose (t + 3 + k + d + (Suc j)) = [False]
    by (rule streamValue10)
  have sg20b:3 + k + d ≤ 2 ∗ d + (4 + k) by arith
  from h14 and sg20b have sg20c:lose (t + 3 + k + d) = [False]
    by (rule aux4lose2)
  from sg20a and sg20c have sg20:∀ j≤Suc d. lose (t + 3 + k + d + j) = [False]

    by (rule streamValue8)
  from h4 and h5 and h6 and h7 and h3 have sg21:
     ack (t + 3 + k + d + d) = [sending-data] ∧
      a (Suc (t + 3 + k + d)) = [sc-ack] ∧
      (∀ j≤Suc d. lose (t + 3 + k + d + j) = [False]) ⟶
      vc (Suc (t + 3 + k + d + d)) = [vc-com]
     apply (simp only: GatewayReq-def)
     by (rule GatewaySystem-L1aux, auto)
   from h2 and sg11 have sg22:a (Suc (t + 3 + k + d)) = [sc-ack]
    by (simp only: ServiceCenter-def, auto)
   from sg21 and sg19a and sg22 and sg20 have sg23:
     vc (Suc (t + 3 + k + d + d)) = [vc-com] by simp
   have 2 ∗ d + (t + (4 + k)) = (Suc (t + 3 + k + d + d)) by arith
   from sg23 and this show ?thesis
     by (simp (no-asm-simp), simp)
qed

lemma GatewaySystem-L3:
 assumes h1:Gateway req dt a stop lose d ack i vc
     and h2:ServiceCenter i a
     and h3:GatewayReq req dt a stop lose d ack i vc
     and h4:msg (Suc 0) req
     and h5:msg (Suc 0) stop
     and h6:msg (Suc 0) a
     and h7:ts lose
     and h8: dt (Suc t) ≠ [] ∨ dt (Suc (Suc t)) ≠ []
     and h9: ack t = [init-state]
     and h10:req (Suc t) = [init]
    and h11:∀ t1≤t. req t1 = []
    and h12:∀ m ≤ k + 2. req (t + m) ≠ [send]
    and h13:req (t + 3 + k) = [send]
    and h14:∀ j≤2 ∗ d + (4 + k). lose (t + j) = [False]
 shows vc (2 ∗ d + (t + (4 + k))) = [vc-com]
proof −
  have Suc 0 ≤ 2 ∗ d + (4 + k) by arith
  from h14 and this have lose (t + Suc 0) = [False] by blast
  then have sg1:lose (Suc t) = [False] by simp
  have Suc (Suc 0) ≤ 2 ∗ d + (4 + k) by arith
  from h14 and this have lose (t + Suc (Suc 0)) = [False] by blast
  then have sg2:lose (Suc (Suc t)) = [False] by simp
  from h3 and h4 and h5 and h6 and h7 and h10 and h9 and sg1 and sg2
have sg3:
   ack (t + 2) = [connection-ok]
    by (simp add: GatewayReq-def)
  from h14 have sg4: ∀ j≤k. lose (t + 2 + j) = [False]
    by (clarify, rule aux4lose1, simp)
  from h12 have sg5:∀ m ≤ k. req (t + 2 + m) ≠ [send]
    by (clarify, rule aux4req, auto)

  from h1 and sg5 and sg4 and sg3 and h4 and h5 and h6 and h7 have sg6:
   ∀ m ≤ k. ack (t + 2 + m) = [connection-ok]
    by (rule Gateway-L6)
  from sg6 have sg6a:ack (t + 2 + k) = [connection-ok] by simp

  from h3 and h4 and h5 and h6 and h7 and h13 and h14 and sg6 have
sg10:
   ack (t + 3 + k) = [sending-data]
    by (simp add: GatewayReq-L1)
  from h3 and h4 and h5 and h6 and h7 have sg11a:
   ack (t + 2 + k) = [connection-ok] ∧
    req (Suc (t + 2 + k)) = [send] ∧
    (∀ j≤Suc d. lose ((t + 2 + k) + j) = [False]) ⟶
    i (Suc (t + (2::nat) + k + d)) = inf-last-ti dt (t + 2 + k)
    apply (simp only: GatewayReq-def)
    by (rule GatewaySystem-L3aux, auto)
  have sg12:Suc (t + 2 + k) = t + 3 + k by arith
  from h13 and sg12 have sg12a:req (Suc (t + 2 + k)) = [send]
    by (simp add: eval-nat-numeral)
  from h14 have sg13:∀ j≤Suc d. lose ((t + 2 + k) + j) = [False]
    by (rule streamValue12)
  from sg11a and sg6a and h13 and sg12a and sg13 have sg14:
    i (Suc (t + (2::nat) + k + d)) = inf-last-ti dt (t + 2 + k) by simp
  from h8 have sg15:inf-last-ti dt (t + 2 + k) ≠ []
    by (rule inf-last-ti-Suc2)
  from sg14 and sg15 have sg16: i (t + 3 + k + d) ≠ []
    by (simp add: arith-sum4)
  from h14 have sg17:∀ j≤k + d + 3. lose (t + j) = [False] by auto
  from h12 have sg18:∀ m < (k + 3). req (t + m) ≠ [send]   by auto
  from h1 and h4 and h5 and h6 and h7 and h10 and sg18 and h13 and h9
and sg17 and h11
    have sg19:∀  t2 < (t + 3 + k + d). i t2 = []
    by (simp add: Gateway-L7)
  from h2 and sg19 have sg20:∀  t3 ≤ (t + 3 + k + d). a t3 = []
    by (simp add: ServiceCenter-L2)
  from h14 have sg21:∀ j≤2 ∗ d. lose (t + 3 + k + j) = [False]
    by (simp add: streamValue43)
  from h1 and h4 and h5 and h6 and h7 and sg21 and sg10 and sg20 have
    ∀ x ≤ d + d. ack (t + 3 + k + x) = [sending-data]
    by (simp add: Gateway-L8)
  from h2 and h3 and h4 and h5 and h6 and h7 and h14 and sg16 and this
show ?thesis
    by (simp add: GatewaySystem-L1)
qed
```

15.12 Proof of the Refinement for the Gateway System

```
lemma GatewaySystem-L0:
 assumes GatewaySystem req dt stop lose d ack vc
 shows    GatewaySystemReq req dt stop lose d ack vc
proof –
  from assms obtain x i where
    a1:Gateway req dt x stop lose d ack i vc and
    a2:ServiceCenter i x
    by (simp add: GatewaySystem-def, auto)
  from a1 have sg1:GatewayReq req dt x stop lose d ack i vc
    by (simp add: Gateway-L0)
  from a2 have sg2:msg (Suc 0) x
    by (simp add: ServiceCenter-a-msg)
  from assms and a1 and a2 and sg1 and sg2 show ?thesis
    apply (simp add: GatewaySystemReq-def, auto)
    apply (simp add: GatewaySystem-L3)
    apply (simp add: GatewaySystem-L3)
    apply (simp add: GatewaySystem-L3)
    by (simp add: GatewaySystem-L2)
qed

end
```

References

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

[2] FlexRay Consortium. http://www.flexray.com.

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

[4] C. Kühnel and M. Spichkova. Fault-Tolerant Communication for Distributed Embedded Systems. In *Software Engineering and Fault Tolerance*, Series on Software Engineering and Knowledge Engineering, 2007.

[5] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS. Springer, 2013.

[6] M. Spichkova. *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle*. PhD thesis, 2007.