

Stream processing components: Isabelle/HOL formalisation and case studies

Maria Spichkova

September 13, 2023

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	4
1.1	Stream processing components	4
1.2	Case Study 1: Steam Boiler System	5
1.3	Case Study 2: FlexRay Communication Protocol	7
1.4	Case Study 3: Automotive-Gateway	10
2	Theory ArithExtras.thy	15
3	Auxiliary Theory ListExtras.thy	15
4	Auxiliary arithmetic lemmas	18
5	FOCUS streams: operators and lemmas	20
5.1	Definition of the FOCUS stream types	20
5.2	Definitions of operators	20
5.3	Properties of operators	31
5.3.1	Lemmas for concatenation operator	32
5.3.2	Lemmas for operators <i>ts</i> and <i>msg</i>	33
5.3.3	Lemmas for <i>inf_truncate</i>	34

5.3.4	Lemmas for <i>fin_make_untimed</i>	34
5.3.5	Lemmas for <i>inf_disj</i> and <i>inf_disjS</i>	35
6	Properties of time-synchronous streams of types bool and bit	36
7	Changing time granularity of the streams	37
7.1	Join time units	37
7.2	Split time units	39
7.3	Duality of the split and the join operators	40
8	Steam Boiler System: Specification	40
9	Steam Boiler System: Verification	42
9.1	Properties of the Boiler Component	42
9.2	Properties of the Controller Component	43
9.3	Properties of the Converter Component	46
9.4	Properties of the System	46
9.5	Proof of the Refinement Relation	47
10	FlexRay: Types	47
11	FlexRay: Specification	48
11.1	Auxiliary predicates	48
11.2	Specifications of the FlexRay components	49
12	FlexRay: Verification	51
12.1	Properties of the function Send	51
12.2	Properties of the component Scheduler	51
12.3	Disjoint Frames	52
12.4	Properties of the sheaf of channels nSend	53
12.5	Properties of the sheaf of channels nGet	55
12.6	Properties of the sheaf of channels nStore	56
12.7	Refinement Properties	58
13	Gateway: Types	59
14	Gateway: Specification	60
15	Gateway: Verification	64
15.1	Properties of the defined data types	64
15.2	Properties of the Delay component	65
15.3	Properties of the Loss component	66
15.4	Properties of the composition of Delay and Loss components	67
15.5	Auxiliary Lemmas	68

15.6 Properties of the ServiceCenter component	74
15.7 General properties of stream values	75
15.8 Properties of the Gateway	77
15.9 Proof of the Refinement Relation for the Gateway Requirements	80
15.10 Lemmas about Gateway Requirements	80
15.11 Properties of the Gateway System	81
15.12 Proof of the Refinement for the Gateway System	83

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:

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

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:

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

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 $'a \text{ fstream}$, which is an abbreviation for the type $'a \text{ list list}$.
- Finite untimed streams of type $'a$ are represented by the list type: $'a \text{ list}$.
- Infinite timed streams of type $'a$ are represented by the type $'a \text{ istream}$, which represents the functional type $\text{nat} \Rightarrow 'a \text{ list}$.
- Infinite untimed streams of type $'a$ are represented by the functional type $\text{nat} \Rightarrow 'a$.

1.2 Case Study 1: Steam Boiler System

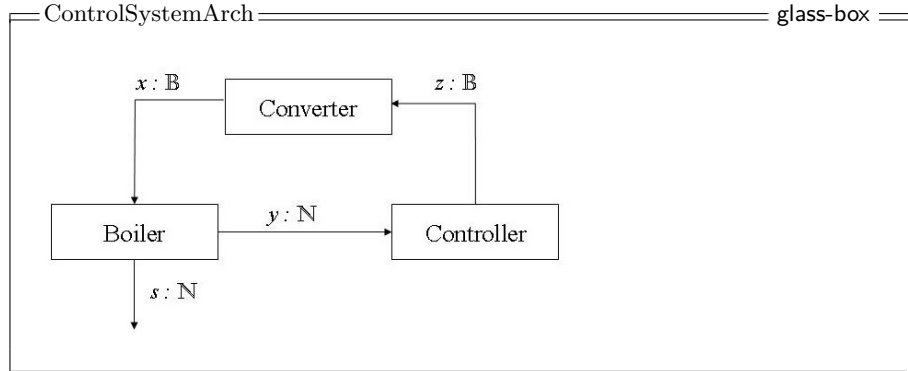
A steam boiler control system can be represent as a distributed system consisting of a number of communicating components and must fulfil real time requirements. This case study shows how we can deal with local variables (system's states) and in which way we can represent mutually recursive functions 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.



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

$$\llbracket ControlSystemArch \rrbracket \Rightarrow \llbracket ControlSystem \rrbracket$$

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

lemma *L0-ControlSystem*: $\llbracket ControlSystemArch \ s \rrbracket \Longrightarrow ControlSystem \ s$

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 *CNL_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).

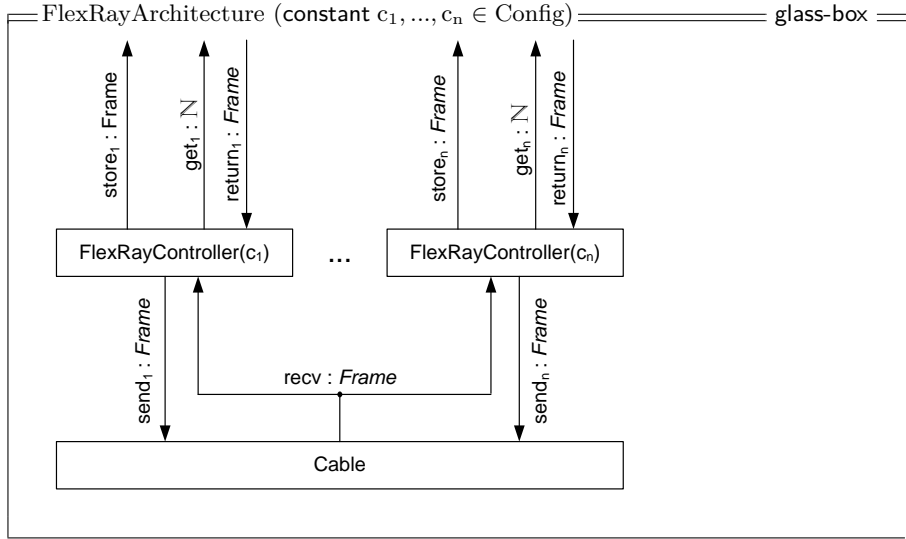
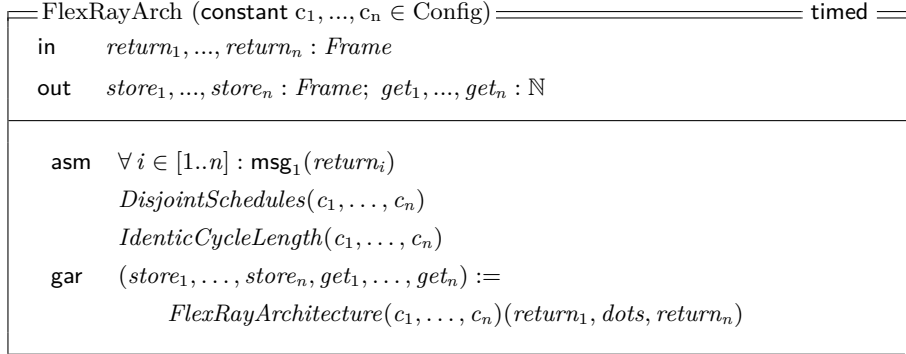
```

type Message = msg (message_id :  $\mathbb{N}$ , ftcdata : Data)
type Frame   = frm (slot :  $\mathbb{N}$ , data : Data)
type Config  = conf (schedule :  $\mathbb{N}^*$ , cycleLength :  $\mathbb{N}$ )

```

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 (*return₁, ..., return_n*), (*c₁, ..., c_n*), (*store₁, ..., store_n*), (*get₁, ..., get_n*), and (*send₁, ..., 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 at 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).



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 *FlexRayControllers* 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 t th 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

$$\llbracket FlexRayArch \rrbracket \Rightarrow \llbracket FlexRay \rrbracket$$

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*:

$$FlexRayArch \ n \ nReturn \ nC \ nStore \ nGet \implies FlexRay \ n \ nReturn \ nC \ nStore \ nGet$$

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 project¹. 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.

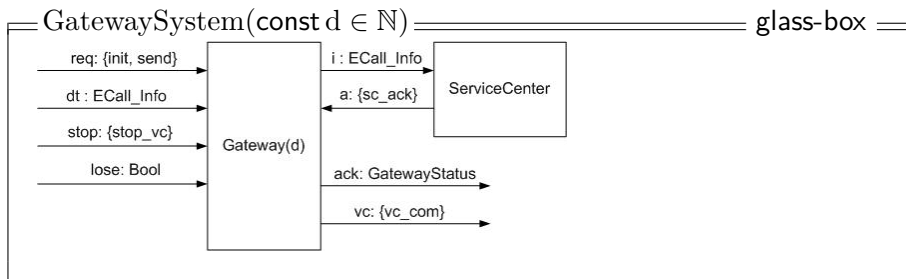
```

type Coordinates    =  N × N
type CollisionSpeed =  N
type ECall_Info     =  ecall(coord ∈ Coordinates, speed ∈ CollisionSpeed)
type GatewayStatus =  { init_state, call, connection_ok,
                       sending_data, voice_com }

```

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:



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.

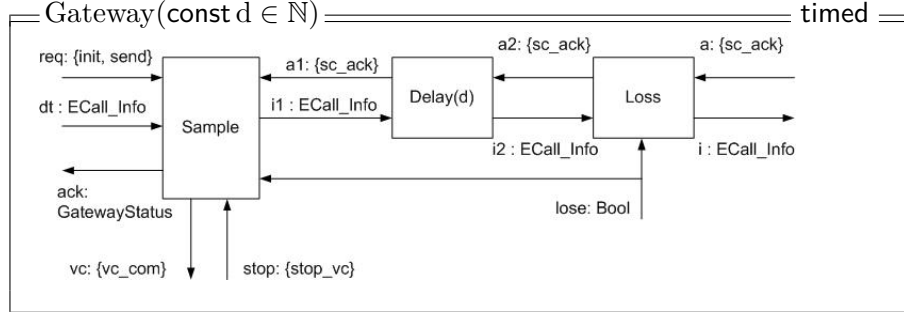
The component *ServiceCenter* represents the interface behaviour of the ESC (wrt. connection to the gateway): if at time t a message about a vehicle crash comes, it acknowledges this event by sending the at time $t + 1$ message *sc_ack* that represents the attempt to establish the voice communication 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*:



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

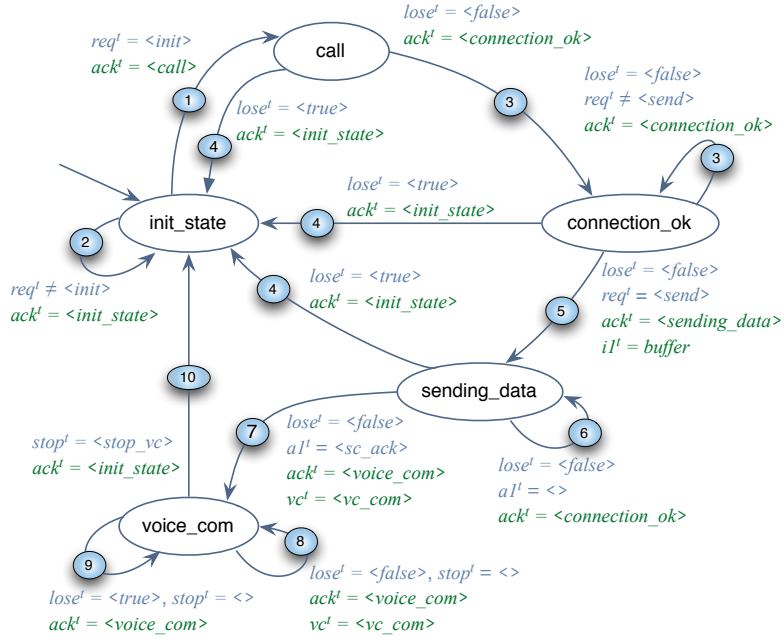


Figure 1: Timed state transition diagram for the component Sample

lemma:

lemma *Gateway-L0*:

$$\begin{aligned} & Gateway \ req \ dt \ a \ stop \ lose \ d \ ack \ i \ vc \\ \implies & GatewayReq \ req \ dt \ a \ stop \ lose \ d \ ack \ i \ vc \end{aligned}$$

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*:

$$\begin{aligned} & GatewaySystem \ req \ dt \ stop \ lose \ d \ ack \ vc \\ \implies & GatewaySystemReq \ req \ dt \ stop \ lose \ d \ ack \ vc \end{aligned}$$

2 Theory ArithExtras.thy

```
theory ArithExtras
imports Main
begin

datatype natInf = Fin nat
             | Infty                ( $\infty$ )
primrec
nat2inat :: nat list  $\Rightarrow$  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  $\Rightarrow$  'a list  $\Rightarrow$  bool
where
  disjoint x y  $\equiv$  (set x)  $\cap$  (set y) = {}

primrec
  mem :: 'a  $\Rightarrow$  'a list  $\Rightarrow$  bool (infixr mem 65)
where
  x mem [] = False |
  x mem (y # l) = ((x = y)  $\vee$  (x mem l))

definition
  memS :: 'a  $\Rightarrow$  'a list  $\Rightarrow$  bool
where
  memS x l  $\equiv$  x  $\in$  (set l)

lemma mem-memS-eq: x mem l  $\equiv$  memS x l
  <proof>

lemma mem-set-1:
assumes a mem l
shows a  $\in$  set l
  <proof>

lemma mem-set-2:
assumes a  $\in$  set l
shows a mem l
```

<proof>

lemma *set-inter-mem*:

assumes $x \text{ mem } l1$

and $x \text{ mem } l2$

shows $\text{set } l1 \cap \text{set } l2 \neq \{\}$

<proof>

lemma *mem-notdisjoint*:

assumes $x \text{ mem } l1$

and $x \text{ mem } l2$

shows $\neg \text{disjoint } l1 \ l2$

<proof>

lemma *mem-notdisjoint2*:

assumes $h1:\text{disjoint } (\text{schedule } A) (\text{schedule } B)$

and $h2:x \text{ mem } \text{schedule } A$

shows $\neg x \text{ mem } \text{schedule } B$

<proof>

lemma *Add-Less*:

assumes $0 < b$

shows $(\text{Suc } a - b < \text{Suc } a) = \text{True}$

<proof>

lemma *list-length-hint1*:

assumes $l \neq []$

shows $0 < \text{length } l$

<proof>

lemma *list-length-hint1a*:

assumes $l \neq []$

shows $0 < \text{length } l$

<proof>

lemma *list-length-hint2*:

assumes $\text{length } x = \text{Suc } 0$

shows $[\text{hd } x] = x$

<proof>

lemma *list-length-hint2a*:

assumes $\text{length } l = \text{Suc } 0$

shows $\text{tl } l = []$

<proof>

lemma *list-length-hint3*:

assumes $\text{length } l = \text{Suc } 0$

shows $l \neq []$

<proof>

lemma *list-length-hint4*:
assumes $\text{length } x \leq \text{Suc } 0$
and $x \neq []$
shows $\text{length } x = \text{Suc } 0$
 $\langle \text{proof} \rangle$

lemma *length-nonempty*:
assumes $x \neq []$
shows $\text{Suc } 0 \leq \text{length } x$
 $\langle \text{proof} \rangle$

lemma *last-nth-length*:
assumes $x \neq []$
shows $x ! ((\text{length } x) - \text{Suc } 0) = \text{last } x$
 $\langle \text{proof} \rangle$

lemma *list-nth-append0*:
assumes $i < \text{length } x$
shows $x ! i = (x \bullet z) ! i$
 $\langle \text{proof} \rangle$

lemma *list-nth-append1*:
assumes $i < \text{length } x$
shows $(b \# x) ! i = (b \# x \bullet y) ! i$
 $\langle \text{proof} \rangle$

lemma *list-nth-append2*:
assumes $i < \text{Suc } (\text{length } x)$
shows $(b \# x) ! i = (b \# x \bullet a \# y) ! i$
 $\langle \text{proof} \rangle$

lemma *list-nth-append3*:
assumes $h1: \neg i < \text{Suc } (\text{length } x)$
and $i - \text{Suc } (\text{length } x) < \text{Suc } (\text{length } y)$
shows $(a \# y) ! (i - \text{Suc } (\text{length } x)) = (b \# x \bullet a \# y) ! i$
 $\langle \text{proof} \rangle$

lemma *list-nth-append4*:
assumes $i < \text{Suc } (\text{length } x + \text{length } y)$
and $\neg i - \text{Suc } (\text{length } x) < \text{Suc } (\text{length } y)$
shows *False*
 $\langle \text{proof} \rangle$

lemma *list-nth-append5*:
assumes $i - \text{length } x < \text{Suc } (\text{length } y)$
and $\neg i - \text{Suc } (\text{length } x) < \text{Suc } (\text{length } y)$
shows $\neg i < \text{Suc } (\text{length } x + \text{length } y)$
 $\langle \text{proof} \rangle$

lemma *list-nth-append6*:
assumes $\neg i - \text{length } x < \text{Suc } (\text{length } y)$
and $\neg i - \text{Suc } (\text{length } x) < \text{Suc } (\text{length } y)$
shows $\neg i < \text{Suc } (\text{length } x + \text{length } y)$
 $\langle \text{proof} \rangle$

lemma *list-nth-append6a*:
assumes $i < \text{Suc } (\text{length } x + \text{length } y)$
and $\neg i - \text{length } x < \text{Suc } (\text{length } y)$
shows *False*
 $\langle \text{proof} \rangle$

lemma *list-nth-append7*:
assumes $i - \text{length } x < \text{Suc } (\text{length } y)$
and $i - \text{Suc } (\text{length } x) < \text{Suc } (\text{length } y)$
shows $i < \text{Suc } (\text{Suc } (\text{length } x + \text{length } y))$
 $\langle \text{proof} \rangle$

lemma *list-nth-append8*:
assumes $\neg i < \text{Suc } (\text{length } x + \text{length } y)$
and $i < \text{Suc } (\text{Suc } (\text{length } x + \text{length } y))$
shows $i = \text{Suc } (\text{length } x + \text{length } y)$
 $\langle \text{proof} \rangle$

lemma *list-nth-append9*:
assumes $i - \text{Suc } (\text{length } x) < \text{Suc } (\text{length } y)$
shows $i < \text{Suc } (\text{Suc } (\text{length } x + \text{length } y))$
 $\langle \text{proof} \rangle$

lemma *list-nth-append10*:
assumes $\neg i < \text{Suc } (\text{length } x)$
and $\neg i - \text{Suc } (\text{length } x) < \text{Suc } (\text{length } y)$
shows $\neg i < \text{Suc } (\text{Suc } (\text{length } x + \text{length } y))$
 $\langle \text{proof} \rangle$

end

4 Auxiliary arithmetic lemmas

theory *arith-hints*
imports *Main*
begin

lemma *arith-mod-neq*:
assumes $a \bmod n \neq b \bmod n$
shows $a \neq b$
 $\langle \text{proof} \rangle$

```

lemma arith-mod-nzero:
  fixes  $i :: nat$ 
  assumes  $i < n$  and  $0 < i$ 
  shows  $0 < (n * t + i) \bmod n$ 
   $\langle proof \rangle$ 

lemma arith-mult-neq-nzero1:
  fixes  $i :: nat$ 
  assumes  $i < n$ 
    and  $0 < i$ 
  shows  $i + n * t \neq n * q$ 
   $\langle proof \rangle$ 

lemma arith-mult-neq-nzero2:
  fixes  $i :: nat$ 
  assumes  $i < n$ 
    and  $0 < i$ 
  shows  $n * t + i \neq n * q$ 
   $\langle proof \rangle$ 

lemma arith-mult-neq-nzero3:
  fixes  $i :: nat$ 
  assumes  $i < n$ 
    and  $0 < i$ 
  shows  $n + n * t + i \neq n * q$ 
   $\langle proof \rangle$ 

lemma arith-modZero1:
   $(t + n * t) \bmod \text{Suc } n = 0$ 
   $\langle proof \rangle$ 

lemma arith-modZero2:
   $\text{Suc } (n + (t + n * t)) \bmod \text{Suc } n = 0$ 
   $\langle proof \rangle$ 

lemma arith1:
  assumes  $h1: \text{Suc } n * t = \text{Suc } n * q$ 
  shows  $t = q$ 
   $\langle proof \rangle$ 

lemma arith2:
  fixes  $t n q :: nat$ 
  assumes  $h1: t + n * t = q + n * q$ 
  shows  $t = q$ 
   $\langle proof \rangle$ 

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. ∃ i. x = (Fin i)} ∪ {∞}
```

— 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. ∃ i. x = (Fin i) ∧ 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

$finU\text{-range} :: 'a \text{ list} \Rightarrow 'a \text{ set}$

where

$finU\text{-range } x \equiv \text{set } x$

— range of an infinite timed stream

definition

$infT\text{-range} :: 'a \text{ istream} \Rightarrow 'a \text{ set}$

where

$infT\text{-range } s \equiv \{y. \exists i::nat. y \text{ mem } (s \ i)\}$

— range of a finite untimed stream

definition

$infU\text{-range} :: (nat \Rightarrow 'a) \Rightarrow 'a \text{ set}$

where

$infU\text{-range } s \equiv \{y. \exists i::nat. y = (s \ i)\}$

— range of a (general) stream

definition

$stream\text{-range} :: 'a \text{ stream} \Rightarrow 'a \text{ set}$

where

$stream\text{-range } s \equiv \text{case } s \text{ of}$
 $FinT \ x \Rightarrow finT\text{-range } x$
 $| FinU \ x \Rightarrow finU\text{-range } x$
 $| InfT \ x \Rightarrow infT\text{-range } x$
 $| InfU \ x \Rightarrow infU\text{-range } x$

— finite timed stream that consists of n empty time intervals

primrec

$nticks :: nat \Rightarrow 'a \text{ fstream}$

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

$inf\text{-tl} :: (nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a)$

where

$inf\text{-tl } s \equiv (\lambda \ i. \ s \ (Suc \ i))$

— 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

$inf\text{-drop} :: nat \Rightarrow (nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a)$

where

$inf-drop\ i\ s \equiv \lambda j. s\ (i+j)$

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

primrec

$fin-find1nonemp :: 'a\ fstream \Rightarrow 'a\ list$

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

$inf-find1nonemp :: 'a\ istream \Rightarrow 'a\ list$

where

$inf-find1nonemp\ s$
 \equiv
 (if $(\exists i. s\ i \neq [])$
 then $s\ (LEAST\ i. s\ i \neq [])$
 else $[]$)

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

primrec

$fin-find1nonemp-index :: 'a\ fstream \Rightarrow nat$

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

$inf-find1nonemp-index :: 'a\ istream \Rightarrow nat$

where

$inf-find1nonemp-index\ s$
 \equiv
 (if $(\exists i. s\ i \neq [])$
 then $(LEAST\ i. s\ i \neq [])$
 else 0)

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

primrec

$fin-length :: 'a\ fstream \Rightarrow nat$

where

$fin-length\ [] = 0$ |
 $fin-length\ (x\#\ xs) = (length\ x) + (fin-length\ xs)$

— length of a (general) stream

definition

$stream-length :: 'a stream \Rightarrow natInf$

where

$stream-length\ s \equiv$
 $case\ s\ of$
 $(FinT\ x) \Rightarrow Fin\ (fin-length\ x)$
 $| (FinU\ x) \Rightarrow Fin\ (length\ x)$
 $| (InfT\ x) \Rightarrow \infty$
 $| (InfU\ x) \Rightarrow \infty$

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

axiomatization

$fin-nth :: 'a\ fstream \Rightarrow nat \Rightarrow 'a$

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

$inf-nth :: 'a\ istream \Rightarrow nat \Rightarrow 'a$

where

$inf-nth\ s\ 0 = hd\ (s\ (LEAST\ i.(s\ i) \neq []))\ |$
 $inf-nth\ s\ (Suc\ k) =$
 $(\ if\ ((Suc\ k) < (length\ (s\ 0)))$
 $\ \ then\ (nth\ (s\ 0)\ (Suc\ k))$
 $\ \ else\ (\ if\ (s\ 0) = []$
 $\ \ \ \ then\ (inf-nth\ (inf-tl\ (inf-drop$
 $\ \ \ \ \ (LEAST\ i.(s\ i) \neq [])\ s))\ k$
 $\ \ \ \ else\ inf-nth\ (inf-tl\ s)\ k)$

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

definition

$stream-nth :: 'a\ stream \Rightarrow nat \Rightarrow 'a$

where

$stream-nth\ s\ k \equiv$
 $case\ s\ of\ (FinT\ x) \Rightarrow fin-nth\ x\ k$
 $| (FinU\ x) \Rightarrow nth\ x\ k$
 $| (InfT\ x) \Rightarrow inf-nth\ x\ k$
 $| (InfU\ x) \Rightarrow x\ k$

— prefix of an infinite stream

primrec

$inf-prefix :: 'a\ list \Rightarrow (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow bool$

where

$inf\text{-}prefix \ [] \ s \ k = True \ |$
 $inf\text{-}prefix \ (x\#\!xs) \ s \ k = ((x = (s \ k)) \wedge (inf\text{-}prefix \ xs \ s \ (Suc \ k)))$

— prefix of a finite stream

primrec

$fin\text{-}prefix :: 'a \ list \Rightarrow 'a \ list \Rightarrow bool$

where

$fin\text{-}prefix \ [] \ s = True \ |$
 $fin\text{-}prefix \ (x\#\!xs) \ s =$
 $(if \ (s = [])$
 $then \ False$
 $else \ (x = (hd \ s)) \wedge (fin\text{-}prefix \ xs \ s))$

— prefix of a (general) stream

definition

$stream\text{-}prefix :: 'a \ stream \Rightarrow 'a \ stream \Rightarrow bool$

where

$stream\text{-}prefix \ p \ s \equiv$
 $(case \ p \ of$
 $(FinT \ x) \Rightarrow$
 $(case \ s \ of \ (FinT \ y) \Rightarrow (fin\text{-}prefix \ x \ y)$
 $| \ (FinU \ y) \Rightarrow False$
 $| \ (InfT \ y) \Rightarrow inf\text{-}prefix \ x \ y \ 0$
 $| \ (InfU \ y) \Rightarrow False)$
 $| \ (FinU \ x) \Rightarrow$
 $(case \ s \ of \ (FinT \ y) \Rightarrow False$
 $| \ (FinU \ y) \Rightarrow (fin\text{-}prefix \ x \ y)$
 $| \ (InfT \ y) \Rightarrow False$
 $| \ (InfU \ y) \Rightarrow inf\text{-}prefix \ x \ y \ 0)$
 $| \ (InfT \ x) \Rightarrow$
 $(case \ s \ of \ (FinT \ y) \Rightarrow False$
 $| \ (FinU \ y) \Rightarrow False$
 $| \ (InfT \ y) \Rightarrow (\forall \ i. \ x \ i = y \ i)$
 $| \ (InfU \ y) \Rightarrow False)$
 $| \ (InfU \ x) \Rightarrow$
 $(case \ s \ of \ (FinT \ y) \Rightarrow False$
 $| \ (FinU \ y) \Rightarrow False$
 $| \ (InfT \ y) \Rightarrow False$
 $| \ (InfU \ y) \Rightarrow (\forall \ i. \ x \ i = y \ i)))$

— truncating a finite stream after the n-th element

primrec

$fin\text{-}truncate :: 'a \ list \Rightarrow nat \Rightarrow 'a \ list$

where

$fin\text{-}truncate \ [] \ n = [] \ |$
 $fin\text{-}truncate \ (x\#\!xs) \ i =$
 $(case \ i \ of \ 0 \Rightarrow []$
 $| \ (Suc \ n) \Rightarrow x \ \# \ (fin\text{-}truncate \ xs \ n))$

- 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 \Rightarrow natInf \Rightarrow 'a list

where

fin-truncate-plus s n

\equiv

case n of (Fin i) \Rightarrow *fin-truncate* s i
 | ∞ \Rightarrow s

- truncating an infinite stream after the n-th element

primrec

inf-truncate :: (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow '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 \Rightarrow natInf \Rightarrow 'a stream

where

inf-truncate-plus s n

\equiv

case n of (Fin i) \Rightarrow FinT (*inf-truncate* s i)
 | ∞ \Rightarrow InfT s

- concatenation of a finite and an infinite stream

definition

fin-inf-append ::

'a list \Rightarrow (nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a)

where

fin-inf-append us s \equiv

(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 \Rightarrow bool

where

ts s $\equiv \forall$ i. (length (s i) = 1)

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

definition

msg :: nat \Rightarrow 'a istream \Rightarrow bool

where

$msg\ n\ s \equiv \forall t. length\ (s\ t) \leq n$

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

primrec

$fin\text{-}msg :: nat \Rightarrow 'a\ list\ list \Rightarrow bool$

where

$fin\text{-}msg\ n\ [] = True$ |

$fin\text{-}msg\ n\ (x\#\!xs) = (((length\ x) \leq n) \wedge (fin\text{-}msg\ n\ xs))$

— making a finite timed stream to a finite untimed stream

definition

$fin\text{-}make\text{-}untimed :: 'a\ fstream \Rightarrow 'a\ list$

where

$fin\text{-}make\text{-}untimed\ x \equiv concat\ x$

— making an infinite timed stream to an infinite untimed stream

— (auxiliary function)

primrec

$inf\text{-}make\text{-}untimed1 :: 'a\ istream \Rightarrow nat \Rightarrow 'a$

where

$inf\text{-}make\text{-}untimed1\ 0:$

$inf\text{-}make\text{-}untimed1\ s\ 0 = hd\ (s\ (LEAST\ i. (s\ i) \neq []))$ |

$inf\text{-}make\text{-}untimed1\text{-}Suc:$

$inf\text{-}make\text{-}untimed1\ s\ (Suc\ k) =$

(if $((Suc\ k) < length\ (s\ 0))$

then $nth\ (s\ 0)\ (Suc\ k)$

else (if $(s\ 0) = []$

then $(inf\text{-}make\text{-}untimed1\ (inf\text{-}tl\ (inf\text{-}drop\ (LEAST\ i. \forall j. j < i \longrightarrow (s\ j) = [])\ s))\ k)$

else $inf\text{-}make\text{-}untimed1\ (inf\text{-}tl\ s)\ k$)

— making an infinite timed stream to an infinite untimed stream

— (main function)

definition

$inf\text{-}make\text{-}untimed :: 'a\ istream \Rightarrow (nat \Rightarrow 'a)$

where

$inf\text{-}make\text{-}untimed\ s$

\equiv

$\lambda i. inf\text{-}make\text{-}untimed1\ s\ i$

— making a (general) stream untimed

definition

$make\text{-}untimed :: 'a\ stream \Rightarrow 'a\ stream$

where

$make\text{-}untimed\ s \equiv$

case s of $(FinT\ x) \Rightarrow FinU\ (fin\text{-}make\text{-}untimed\ x)$

| $(FinU\ x) \Rightarrow FinU\ x$

$$\begin{aligned}
& | (\text{InfT } x) \Rightarrow \\
& \quad (\text{if } (\exists i. \forall j. i < j \longrightarrow (x j) = []) \\
& \quad \quad \text{then } \text{FinU } (\text{fin-make-untimed } (\text{inf-truncate } x \\
& \quad \quad \quad (\text{LEAST } i. \forall j. i < j \longrightarrow (x j) = []))) \\
& \quad \quad \text{else } \text{InfU } (\text{inf-make-untimed } x)) \\
& | (\text{InfU } x) \Rightarrow \text{InfU } x
\end{aligned}$$

— finding the index of the time interval that contains the k-th data element
— defined over a finite timed stream

primrec

$\text{fin-tm} :: 'a \text{ fstream} \Rightarrow \text{nat} \Rightarrow \text{nat}$

where

$$\begin{aligned}
& \text{fin-tm } [] \ k = k \ | \\
& \text{fin-tm } (x\#\text{xs}) \ k = \\
& \quad (\text{if } k = 0 \\
& \quad \quad \text{then } 0 \\
& \quad \quad \text{else } (\text{if } (k \leq \text{length } x) \\
& \quad \quad \quad \text{then } (\text{Suc } 0) \\
& \quad \quad \quad \text{else } \text{Suc}(\text{fin-tm } \text{xs } (k - \text{length } x))))
\end{aligned}$$

— auxiliary lemma for the definition of the truncate operator

lemma *inf-tm-hint1*:

$$\begin{aligned}
& \text{assumes } i2 = \text{Suc } i - \text{length } a \\
& \quad \text{and } \neg \text{Suc } i \leq \text{length } a \\
& \quad \text{and } a \neq [] \\
& \text{shows } i2 < \text{Suc } i
\end{aligned}$$

<proof>

definition

$\text{finT-filter} :: 'a \text{ set} \Rightarrow 'a \text{ fstream} \Rightarrow 'a \text{ fstream}$

where

$\text{finT-filter } m \ s \equiv \text{map } (\lambda s. \text{filter } (\lambda y. y \in m) \ s) \ s$

— filtering an infinite timed stream

definition

$\text{infT-filter} :: 'a \text{ set} \Rightarrow 'a \text{ istream} \Rightarrow 'a \text{ istream}$

where

$\text{infT-filter } m \ s \equiv (\lambda i. (\text{filter } (\lambda x. x \in m) \ (s \ i)))$

— removing duplications from a finite timed stream

definition

$\text{finT-remdups} :: 'a \text{ fstream} \Rightarrow 'a \text{ fstream}$

where

$\text{finT-remdups } s \equiv \text{map } (\lambda s. \text{remdups } \ s) \ s$

— removing duplications from an infinite timed stream

definition

$\text{infT-remdups} :: 'a \text{ istream} \Rightarrow 'a \text{ istream}$

where

$infT\text{-remdups } s \equiv (\lambda i.(\text{remdups } (s\ i)))$

— removing duplications from a time interval of a stream

primrec

$fst\text{-remdups} :: 'a\ list \Rightarrow 'a\ list$

where

$fst\text{-remdups } [] = [] \mid$
 $fst\text{-remdups } (x\#\!xs) =$
 $(if\ xs = []$
 $then [x]$
 $else (if\ x = (hd\ xs)$
 $then\ fst\text{-remdups } xs$
 $else\ (x\#\!xs)))$

— time interval operator

definition

$ti :: 'a\ fstream \Rightarrow nat \Rightarrow 'a\ list$

where

$ti\ s\ i \equiv$
 $(if\ s = []$
 $then []$
 $else\ (nth\ s\ i))$

— insuring that a sheaf of channels is correctly defined

definition

$CorrectSheaf :: nat \Rightarrow bool$

where

$CorrectSheaf\ n \equiv 0 < n$

— insuring that all channels in a sheaf are disjoint

— indices in the sheaf are represented using an extra specified set

definition

$inf\text{-disjS} :: 'b\ set \Rightarrow ('b \Rightarrow 'a\ istream) \Rightarrow bool$

where

$inf\text{-disjS } IdSet\ nS$
 \equiv
 $\forall\ (t::nat)\ i\ j. (i:IdSet) \wedge (j:IdSet) \wedge$
 $((nS\ i)\ t) \neq [] \longrightarrow ((nS\ j)\ t) = []$

— insuring that all channels in a sheaf are disjoint

— indices in the sheaf are represented using natural numbers

definition

$inf\text{-disj} :: nat \Rightarrow (nat \Rightarrow 'a\ istream) \Rightarrow bool$

where

$inf\text{-disj } n\ nS$
 \equiv
 $\forall\ (t::nat)\ (i::nat)\ (j::nat).$
 $i < n \wedge j < n \wedge i \neq j \wedge ((nS\ i)\ t) \neq [] \longrightarrow$
 $((nS\ j)\ t) = []$

— 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
⟨proof⟩

```

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)
⟨proof⟩

```

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 untyped 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 typed 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 untyped 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

$$\begin{aligned} \text{get-prefix } s \ k &\equiv \\ &\text{case } s \text{ of } (FinT \ x) \Rightarrow FinT \ (\text{fin-get-prefix-plus } x \ k) \\ &\quad | (FinU \ x) \Rightarrow FinU \ (\text{take-plus } k \ x) \\ &\quad | (InfT \ x) \Rightarrow \text{infT-get-prefix-plus } x \ k \\ &\quad | (InfU \ x) \Rightarrow \text{infU-get-prefix-plus } x \ k \end{aligned}$$

— merging time intervals of two finite timed streams

primrec

$$\text{fin-merge-ti} :: 'a \ \text{fstream} \Rightarrow 'a \ \text{fstream} \Rightarrow 'a \ \text{fstream}$$

where

$$\begin{aligned} \text{fin-merge-ti} \ [] \ y &= y \ | \\ \text{fin-merge-ti} \ (x\#\!xs) \ y &= \\ &(\text{case } y \ \text{of} \ [] \Rightarrow (x\#\!xs) \\ &\quad | (z\#\!zs) \Rightarrow (x\bullet z) \ \# \ (\text{fin-merge-ti } xs \ zs)) \end{aligned}$$

— merging time intervals of two infinite timed streams

definition

$$\text{inf-merge-ti} :: 'a \ \text{istream} \Rightarrow 'a \ \text{istream} \Rightarrow 'a \ \text{istream}$$

where

$$\begin{aligned} \text{inf-merge-ti } x \ y & \\ &\equiv \\ &\lambda \ i. \ (x \ i) \bullet (y \ i) \end{aligned}$$

— the last time interval of a finite timed stream

primrec

$$\text{fin-last-ti} :: ('a \ \text{list}) \ \text{list} \Rightarrow \text{nat} \Rightarrow 'a \ \text{list}$$

where

$$\begin{aligned} \text{fin-last-ti } s \ 0 &= \text{hd } s \ | \\ \text{fin-last-ti } s \ (\text{Suc } i) &= \\ &(\text{if } s!(\text{Suc } i) \neq [] \\ &\quad \text{then } s!(\text{Suc } i) \\ &\quad \text{else } \text{fin-last-ti } s \ i) \end{aligned}$$

— 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

$$\text{inf-last-ti} :: 'a \ \text{istream} \Rightarrow \text{nat} \Rightarrow 'a \ \text{list}$$

where

$$\begin{aligned} \text{inf-last-ti } s \ 0 &= s \ 0 \ | \\ \text{inf-last-ti } s \ (\text{Suc } i) &= \\ &(\text{if } s \ (\text{Suc } i) \neq [] \\ &\quad \text{then } s \ (\text{Suc } i) \\ &\quad \text{else } \text{inf-last-ti } s \ i) \end{aligned}$$

5.3 Properties of operators

lemma *inf-last-ti-nonempty-k*:

assumes $\text{inf-last-ti dt } t \neq []$
shows $\text{inf-last-ti dt } (t + k) \neq []$
 $\langle \text{proof} \rangle$

lemma $\text{inf-last-ti-nonempty}$:
assumes $s \neq []$
shows $\text{inf-last-ti } s (t + k) \neq []$
 $\langle \text{proof} \rangle$

lemma arith-sum-t2k :
 $t + 2 + k = (\text{Suc } t) + (\text{Suc } k)$
 $\langle \text{proof} \rangle$

lemma inf-last-ti-Suc2 :
assumes $\text{dt } (\text{Suc } t) \neq [] \vee \text{dt } (\text{Suc } (\text{Suc } t)) \neq []$
shows $\text{inf-last-ti dt } (t + 2 + k) \neq []$
 $\langle \text{proof} \rangle$

5.3.1 Lemmas for concatenation operator

lemma fin-length-append :
 $\text{fin-length } (x \bullet y) = (\text{fin-length } x) + (\text{fin-length } y)$
 $\langle \text{proof} \rangle$

lemma fin-append-Nil : $\text{fin-inf-append } [] \ z = z$
 $\langle \text{proof} \rangle$

lemma $\text{correct-fin-inf-append1}$:
assumes $s1 = \text{fin-inf-append } [x] \ s$
shows $s1 (\text{Suc } i) = s \ i$
 $\langle \text{proof} \rangle$

lemma $\text{correct-fin-inf-append2}$:
 $\text{fin-inf-append } [x] \ s (\text{Suc } i) = s \ i$
 $\langle \text{proof} \rangle$

lemma $\text{fin-append-com-Nil1}$:
 $\text{fin-inf-append } [] \ (\text{fin-inf-append } y \ z)$
 $= \text{fin-inf-append } ([] \bullet y) \ z$
 $\langle \text{proof} \rangle$

lemma $\text{fin-append-com-Nil2}$:
 $\text{fin-inf-append } x \ (\text{fin-inf-append } [] \ z)$
 $= \text{fin-inf-append } (x \bullet []) \ z$
 $\langle \text{proof} \rangle$

lemma fin-append-com-i :
 $\text{fin-inf-append } x \ (\text{fin-inf-append } y \ z) \ i = \text{fin-inf-append } (x \bullet y) \ z \ i$
 $\langle \text{proof} \rangle$

5.3.2 Lemmas for operators ts and msg

lemma $ts\text{-}msg1$:

assumes $ts\ p$

shows $msg\ 1\ p$

$\langle proof \rangle$

lemma $ts\text{-}inf\text{-}tl$:

assumes $ts\ x$

shows $ts\ (inf\text{-}tl\ x)$

$\langle proof \rangle$

lemma $ts\text{-}length\text{-}hint1$:

assumes $ts\ x$

shows $x\ i \neq []$

$\langle proof \rangle$

lemma $ts\text{-}length\text{-}hint2$:

assumes $ts\ x$

shows $length\ (x\ i) = Suc\ (0::nat)$

$\langle proof \rangle$

lemma $ts\text{-}Least\text{-}0$:

assumes $ts\ x$

shows $(LEAST\ i.\ (x\ i) \neq []) = (0::nat)$

$\langle proof \rangle$

lemma $inf\text{-}tl\text{-}Suc$: $inf\text{-}tl\ x\ i = x\ (Suc\ i)$

$\langle proof \rangle$

lemma $ts\text{-}Least\text{-}Suc0$:

assumes $ts\ x$

shows $(LEAST\ i.\ x\ (Suc\ i) \neq []) = 0$

$\langle proof \rangle$

lemma $ts\text{-}inf\text{-}make\text{-}untimed\text{-}inf\text{-}tl$:

assumes $ts\ x$

shows $inf\text{-}make\text{-}untimed\ (inf\text{-}tl\ x)\ i = inf\text{-}make\text{-}untimed\ x\ (Suc\ i)$

$\langle proof \rangle$

lemma $ts\text{-}inf\text{-}make\text{-}untimed1\text{-}inf\text{-}tl$:

assumes $ts\ x$

shows $inf\text{-}make\text{-}untimed1\ (inf\text{-}tl\ x)\ i = inf\text{-}make\text{-}untimed1\ x\ (Suc\ i)$

$\langle proof \rangle$

lemma $msg\text{-}nonempty1$:

assumes $h1:msg\ (Suc\ 0)\ a$

and $h2:a\ t = aa\ \# \ l$

shows $l = []$

$\langle proof \rangle$

lemma *msg-nonempty2*:
assumes $h1:msg (Suc\ 0)\ a$
and $h2:a\ t \neq []$
shows $length\ (a\ t) = (Suc\ 0)$
 $\langle proof \rangle$

5.3.3 Lemmas for *inf_truncate*

lemma *inf-truncate-nonempty*:
assumes $z\ i \neq []$
shows $inf-truncate\ z\ i \neq []$
 $\langle proof \rangle$

lemma *concat-inf-truncate-nonempty*:
assumes $z\ i \neq []$
shows $concat\ (inf-truncate\ z\ i) \neq []$
 $\langle proof \rangle$

lemma *concat-inf-truncate-nonempty-a*:
assumes $z\ i = [a]$
shows $concat\ (inf-truncate\ z\ i) \neq []$
 $\langle proof \rangle$

lemma *concat-inf-truncate-nonempty-el*:
assumes $z\ i \neq []$
shows $concat\ (inf-truncate\ z\ i) \neq []$
 $\langle proof \rangle$

lemma *inf-truncate-append*:
 $(inf-truncate\ z\ i \bullet [z\ (Suc\ i)]) = inf-truncate\ z\ (Suc\ i)$
 $\langle proof \rangle$

5.3.4 Lemmas for *fin_make_untimed*

lemma *fin-make-untimed-append*:
assumes $fin-make-untimed\ x \neq []$
shows $fin-make-untimed\ (x \bullet y) \neq []$
 $\langle proof \rangle$

lemma *fin-make-untimed-inf-truncate-Nonempty*:
assumes $z\ k \neq []$
and $k \leq i$
shows $fin-make-untimed\ (inf-truncate\ z\ i) \neq []$
 $\langle proof \rangle$

lemma *last-fin-make-untimed-append*:
 $last\ (fin-make-untimed\ (z \bullet [[a]])) = a$

$\langle proof \rangle$

lemma *last-fin-make-untimed-inf-truncate*:

assumes $z\ i = [a]$

shows $last\ (fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i)) = a$

$\langle proof \rangle$

lemma *fin-make-untimed-append-empty*:

$fin\text{-}make\text{-}untimed\ (z \bullet []) = fin\text{-}make\text{-}untimed\ z$

$\langle proof \rangle$

lemma *fin-make-untimed-inf-truncate-append-a*:

$fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i \bullet [[a]]) !$

$(length\ (fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i \bullet [[a]])) - Suc\ 0) = a$

$\langle proof \rangle$

lemma *fin-make-untimed-inf-truncate-Nonempty-all*:

assumes $z\ k \neq []$

shows $\forall i. k \leq i \longrightarrow fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i) \neq []$

$\langle proof \rangle$

lemma *fin-make-untimed-inf-truncate-Nonempty-all0*:

assumes $z\ 0 \neq []$

shows $\forall i. fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i) \neq []$

$\langle proof \rangle$

lemma *fin-make-untimed-inf-truncate-Nonempty-all0a*:

assumes $z\ 0 = [a]$

shows $\forall i. fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i) \neq []$

$\langle proof \rangle$

lemma *fin-make-untimed-inf-truncate-Nonempty-all-app*:

assumes $z\ 0 = [a]$

shows $\forall i. fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i \bullet [z\ (Suc\ i)]) \neq []$

$\langle proof \rangle$

lemma *fin-make-untimed-nth-length*:

assumes $z\ i = [a]$

shows

$fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i) !$

$(length\ (fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i)) - Suc\ 0)$

$= a$

$\langle proof \rangle$

5.3.5 Lemmas for *inf_disj* and *inf_disjS*

lemma *inf-disj-index*:

assumes $h1 : inf_disj\ n\ nS$

and $nS\ k\ t \neq []$

and $k < n$
shows $(\text{SOME } i. i < n \wedge nS\ i\ t \neq []) = k$
 $\langle \text{proof} \rangle$

lemma *inf-disjS-index*:
assumes $h1: \text{inf-disjS } IdSet\ nS$
and $k: IdSet$
and $nS\ k\ t \neq []$
shows $(\text{SOME } i. (i: IdSet) \wedge nSend\ i\ t \neq []) = k$
 $\langle \text{proof} \rangle$

end

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

theory *BitBoolTS*
imports *Main stream*
begin

datatype *bit* = *Zero* | *One*

primrec
 $\text{negation} :: \text{bit} \Rightarrow \text{bit}$
where
 $\text{negation } Zero = One$ |
 $\text{negation } One = Zero$

lemma *ts-bit-stream-One*:
assumes $h1: ts\ x$
and $h2: x\ i \neq [Zero]$
shows $x\ i = [One]$
 $\langle \text{proof} \rangle$

lemma *ts-bit-stream-Zero*:
assumes $h1: ts\ x$
and $h2: x\ i \neq [One]$
shows $x\ i = [Zero]$
 $\langle \text{proof} \rangle$

lemma *ts-bool-True*:
assumes $h1: ts\ x$
and $h2: x\ i \neq [False]$
shows $x\ i = [True]$
 $\langle \text{proof} \rangle$

lemma *ts-bool-False*:

```

assumes  $h1:ts\ x$ 
and  $h2:x\ i \neq [True]$ 
shows  $x\ i = [False]$ 
<proof>

```

```

lemma ts-bool-True-False:
fixes  $x::bool\ istream$ 
assumes  $ts\ x$ 
shows  $x\ i = [True] \vee x\ i = [False]$ 
<proof>

```

end

7 Changing time granularity of the streams

```

theory JoinSplitTime
imports stream arith-hints
begin

```

7.1 Join time units

```

primrec
 $join-ti :: 'a\ istream \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list$ 
where
 $join-ti-0:$ 
 $join-ti\ s\ x\ 0 = s\ x\ |$ 
 $join-ti-Suc:$ 
 $join-ti\ s\ x\ (Suc\ i) = (join-ti\ s\ x\ i) \bullet (s\ (x + (Suc\ i)))$ 

```

```

primrec
 $fin-join-ti :: 'a\ fstream \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list$ 
where
 $fin-join-ti-0:$ 
 $fin-join-ti\ s\ x\ 0 = nth\ s\ x\ |$ 
 $fin-join-ti-Suc:$ 
 $fin-join-ti\ s\ x\ (Suc\ i) = (fin-join-ti\ s\ x\ i) \bullet (nth\ s\ (x + (Suc\ i)))$ 

```

```

definition
 $join-time :: 'a\ istream \Rightarrow nat \Rightarrow 'a\ istream$ 
where
 $join-time\ s\ n\ t \equiv$ 
 $(case\ n\ of$ 
 $0 \Rightarrow []$ 
 $| (Suc\ i) \Rightarrow join-ti\ s\ (n*t)\ i)$ 

```

```

lemma join-ti-hint1:
assumes  $join-ti\ s\ x\ (Suc\ i) = []$ 
shows  $join-ti\ s\ x\ i = []$ 
<proof>

```

lemma *join-ti-hint2*:

assumes *join-ti s x (Suc i) = []*

shows *s (x + (Suc i)) = []*

<proof>

lemma *join-ti-hint3*:

assumes *join-ti s x (Suc i) = []*

shows *s (x + i) = []*

<proof>

lemma *join-ti-empty-join*:

assumes *i ≤ n*

and *join-ti s x n = []*

shows *s (x+i) = []*

<proof>

lemma *join-ti-empty-ti*:

assumes $\forall i \leq n. s (x+i) = []$

shows *join-ti s x n = []*

<proof>

lemma *join-ti-1nempty*:

assumes $\forall i. 0 < i \wedge i < \text{Suc } n \longrightarrow s (x+i) = []$

shows *join-ti s x n = s x*

<proof>

lemma *join-time1t*: $\forall t. \text{join-time } s (1::\text{nat}) t = s t$

<proof>

lemma *join-time1*: *join-time s 1 = s*

<proof>

lemma *join-time-empty1*:

assumes *h1:i < n*

and *h2:join-time s n t = []*

shows *s (n*t + i) = []*

<proof>

lemma *fin-join-ti-hint1*:

assumes *fin-join-ti s x (Suc i) = []*

shows *fin-join-ti s x i = []*

<proof>

lemma *fin-join-ti-hint2*:

assumes *fin-join-ti s x (Suc i) = []*

shows *nth s (x + (Suc i)) = []*

<proof>

lemma *fin-join-ti-hint3*:
assumes *fin-join-ti s x (Suc i) = []*
shows *nth s (x + i) = []*
<proof>

lemma *fin-join-ti-empty-join*:
assumes $i \leq n$
and *fin-join-ti s x n = []*
shows *nth s (x+i) = []*
<proof>

lemma *fin-join-ti-empty-ti*:
assumes $\forall i \leq n. \text{nth } s \text{ (x+i) = []}$
shows *fin-join-ti s x n = []*
<proof>

lemma *fin-join-ti-1nempty*:
assumes $\forall i. 0 < i \wedge i < \text{Suc } n \longrightarrow \text{nth } s \text{ (x+i) = []}$
shows *fin-join-ti s x n = nth s x*
<proof>

7.2 Split time units

definition

split-time :: 'a istream \Rightarrow nat \Rightarrow 'a istream

where

split-time s n t \equiv
 (if (t mod n = 0)
 then s (t div n)
 else [])

lemma *split-time1t*: $\forall t. \text{split-time } s \ 1 \ t = s \ t$
<proof>

lemma *split-time1*: *split-time s 1 = s*
<proof>

lemma *split-time-mod*:
assumes $t \bmod n \neq 0$
shows *split-time s n t = []*
<proof>

lemma *split-time-nempty*:
assumes $0 < n$
shows *split-time s n (n * t) = s t*
<proof>

lemma *split-time-nempty-Suc*:

assumes $0 < n$
shows $\text{split-time } s \text{ (Suc } n \text{) ((Suc } n \text{) * } t \text{) = split-time } s \text{ } n \text{ (} n * t \text{)}$
 <proof>

lemma *split-time-empty*:
assumes $i < n$ **and** $h2:0 < i$
shows $\text{split-time } s \text{ } n \text{ (} n * t + i \text{) = []}$
 <proof>

lemma *split-time-empty-Suc*:
assumes $h1:i < n$
and $h2:0 < i$
shows $\text{split-time } s \text{ (Suc } n \text{) ((Suc } n \text{) * } t + i \text{) = split-time } s \text{ } n \text{ (} n * t + i \text{)}$
 <proof>

lemma *split-time-hint1*:
assumes $n = \text{Suc } m$
shows $\text{split-time } s \text{ (Suc } n \text{) (} i + n * i + n \text{) = []}$
 <proof>

7.3 Duality of the split and the join operators

lemma *join-split-i*:
assumes $0 < n$
shows $\text{join-time (split-time } s \text{ } n \text{) } n \text{ } i = s \text{ } i$
 <proof>

lemma *join-split*:
assumes $0 < n$
shows $\text{join-time (split-time } s \text{ } n \text{) } n = s$
 <proof>

end

8 Steam Boiler System: Specification

theory *SteamBoiler*
imports *stream BitBoolTS*
begin

definition
 $\text{ControlSystem} :: \text{nat } \text{istream} \Rightarrow \text{bool}$
where
 $\text{ControlSystem } s \equiv$
 $(\text{ts } s) \wedge$
 $(\forall (j::\text{nat}). (200::\text{nat}) \leq \text{hd } (s \text{ } j) \wedge \text{hd } (s \text{ } j) \leq (800::\text{nat}))$

definition
 $\text{SteamBoiler} :: \text{bit } \text{istream} \Rightarrow \text{nat } \text{istream} \Rightarrow \text{nat } \text{istream} \Rightarrow \text{bool}$

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

$Controller :: nat\ istream \Rightarrow bit\ istream \Rightarrow bool$
where
 $Controller\ y\ z$
 \equiv
 $(ts\ y)$
 \longrightarrow
 $(\exists\ l.\ Controller-L\ y\ (fin-inf-append\ [Zero]\ l)\ l\ z)$

definition

$ControlSystemArch :: nat\ istream \Rightarrow bool$
where
 $ControlSystemArch\ s$
 \equiv
 $\exists\ x\ z :: bit\ istream.\ \exists\ y :: nat\ istream.$
 $(SteamBoiler\ x\ s\ y \wedge Controller\ y\ z \wedge Converter\ z\ x)$

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 $ts\ s$
 $\langle proof \rangle$

lemma *L2-Boiler*:
assumes $SteamBoiler\ x\ s\ y$
and $ts\ x$
shows $ts\ y$
 $\langle proof \rangle$

lemma *L3-Boiler*:
assumes $SteamBoiler\ x\ s\ y$
and $ts\ x$
shows $200 \leq hd\ (s\ 0)$
 $\langle proof \rangle$

lemma *L4-Boiler*:
assumes $SteamBoiler\ x\ s\ y$
and $ts\ x$
shows $hd\ (s\ 0) \leq 800$
 $\langle proof \rangle$

lemma L5-Boiler:
assumes $h1:SteamBoiler\ x\ s\ y$
and $h2:ts\ x$
and $h3:hd\ (x\ j) = Zero$
shows $(hd\ (s\ j)) \leq hd\ (s\ (Suc\ j)) + (10::nat)$
 $\langle proof \rangle$

lemma L6-Boiler:
assumes $h1:SteamBoiler\ x\ s\ y$
and $h2:ts\ x$
and $h3:hd\ (x\ j) = Zero$
shows $(hd\ (s\ j)) - (10::nat) \leq hd\ (s\ (Suc\ j))$
 $\langle proof \rangle$

lemma L7-Boiler:
assumes $h1:SteamBoiler\ x\ s\ y$
and $h2:ts\ x$
and $h3:hd\ (x\ j) \neq Zero$
shows $(hd\ (s\ j)) \geq hd\ (s\ (Suc\ j)) - (10::nat)$
 $\langle proof \rangle$

lemma L8-Boiler:
assumes $h1:SteamBoiler\ x\ s\ y$
and $h2:ts\ x$
and $h3:hd\ (x\ j) \neq Zero$
shows $(hd\ (s\ j)) + (10::nat) \geq hd\ (s\ (Suc\ j))$
 $\langle proof \rangle$

9.2 Properties of the Controller Component

lemma L1-Controller:
assumes $Controller-L\ s\ (fin-inf-append\ [Zero]\ l)\ l\ z$
shows $fin-make-untimed\ (inf-truncate\ z\ i) \neq []$
 $\langle proof \rangle$

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 $z\ (Suc\ t) = []$
 $\langle proof \rangle$

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 $z\ (Suc\ t) = []$
 $\langle proof \rangle$

lemma *L3-Controller-Zero*:

assumes *Controller-L* y (*fin-inf-append* [Zero] l) l z
and $l\ t = \text{Zero}$
and $\neg\ 300 < \text{hd}\ (y\ (\text{Suc}\ t))$
shows $z\ (\text{Suc}\ t) = [\text{One}]$
<proof>

lemma *L3-Controller-One*:

assumes *Controller-L* y (*fin-inf-append* [Zero] l) l z
and $l\ t = \text{One}$
and $\neg\ \text{hd}\ (y\ (\text{Suc}\ t)) < 700$
shows $z\ (\text{Suc}\ t) = [\text{Zero}]$
<proof>

lemma *L4-Controller-Zero*:

assumes $h1:\text{Controller-L}$ y (*fin-inf-append* [Zero] l) l z
and $h2:l\ (\text{Suc}\ t) = \text{Zero}$
shows $(z\ (\text{Suc}\ t) = [] \wedge l\ t = \text{Zero}) \vee (z\ (\text{Suc}\ t) = [\text{Zero}] \wedge l\ t = \text{One})$
<proof>

lemma *L4-Controller-One*:

assumes $h1:\text{Controller-L}$ y (*fin-inf-append* [Zero] l) l z
and $h2:l\ (\text{Suc}\ t) = \text{One}$
shows $(z\ (\text{Suc}\ t) = [] \wedge l\ t = \text{One}) \vee (z\ (\text{Suc}\ t) = [\text{One}] \wedge l\ t = \text{Zero})$
<proof>

lemma *L5-Controller-Zero*:

assumes $h1:\text{Controller-L}$ y lIn $lOut$ z
and $h2:lOut\ t = \text{Zero}$
and $h3:z\ t = []$
shows $lIn\ t = \text{Zero}$
<proof>

lemma *L5-Controller-One*:

assumes $h1:\text{Controller-L}$ y lIn $lOut$ z
and $h2:lOut\ t = \text{One}$
and $h3:z\ t = []$
shows $lIn\ t = \text{One}$
<proof>

lemma *L5-Controller*:

assumes *Controller-L* y lIn $lOut$ z
and $lOut\ t = a$
and $z\ t = []$
shows $lIn\ t = a$
<proof>

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 $l\ t = Zero$
 $\langle proof \rangle$

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 $l\ t = One$
 $\langle proof \rangle$

lemma *L6-Controller*:
assumes *Controller-L y (fin-inf-append [Zero] l) l z*
and $l (Suc\ t) = a$
and $z (Suc\ t) = []$
shows $l\ t = a$
 $\langle proof \rangle$

lemma *L7-Controller-Zero*:
assumes $h1: Controller-L\ y\ (fin-inf-append\ [Zero]\ l)\ l\ z$
and $h2: l\ t = Zero$
shows $last\ (fin-make-untimed\ (inf-truncate\ z\ t)) = Zero$
 $\langle proof \rangle$

lemma *L7-Controller-One-l0*:
assumes *Controller-L y (fin-inf-append [Zero] l) l z*
and $y\ 0 = [500::nat]$
shows $l\ 0 = Zero$
 $\langle proof \rangle$

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 $last\ (fin-make-untimed\ (inf-truncate\ z\ t)) = One$
 $\langle proof \rangle$

lemma *L7-Controller*:
assumes *Controller-L y (fin-inf-append [Zero] l) l z*
and $y\ 0 = [500::nat]$
shows $last\ (fin-make-untimed\ (inf-truncate\ z\ t)) = l\ t$
 $\langle proof \rangle$

lemma *L8-Controller*:
assumes *Controller-L y (fin-inf-append [Zero] l) l z*
shows $z\ t = [] \vee z\ t = [Zero] \vee z\ t = [One]$

<proof>

lemma *L9-Controller:*

assumes *h1:Controller-L s (fin-inf-append [Zero] l) l z*
and *h2:fin-make-untimed (inf-truncate z i) !*
 $(\text{length } (\text{fin-make-untimed } (\text{inf-truncate } z \ i)) - \text{Suc } 0) = \text{Zero}$
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 $200 \leq \text{hd } (s \ (\text{Suc } i))$

<proof>

lemma *L10-Controller:*

assumes *h1:Controller-L s (fin-inf-append [Zero] l) l z*
and *h2:fin-make-untimed (inf-truncate z i) !*
 $(\text{length } (\text{fin-make-untimed } (\text{inf-truncate } z \ i)) - \text{Suc } 0) \neq \text{Zero}$
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 $\text{hd } (s \ (\text{Suc } i)) \leq 800$

<proof>

9.3 Properties of the Converter Component

lemma *L1-Converter:*

assumes *Converter z x*
and *fin-make-untimed (inf-truncate z t) ≠ []*
shows $\text{hd } (x \ t) = (\text{fin-make-untimed } (\text{inf-truncate } z \ t)) !$
 $((\text{length } (\text{fin-make-untimed } (\text{inf-truncate } z \ t))) - (1::\text{nat}))$

<proof>

lemma *L1a-Converter:*

assumes *Converter z x*
and *fin-make-untimed (inf-truncate z t) ≠ []*
and *hd (x t) = Zero*
shows $(\text{fin-make-untimed } (\text{inf-truncate } z \ t)) !$
 $((\text{length } (\text{fin-make-untimed } (\text{inf-truncate } z \ t))) - (1::\text{nat}))$
 $= \text{Zero}$

<proof>

9.4 Properties of the System

lemma *L1-ControlSystem:*

assumes *ControlSystemArch s*
shows *ts s*

<proof>

lemma *L2-ControlSystem:*

```

assumes ControlSystemArch s
shows  $(200::nat) \leq hd (s i)$ 
<proof>

```

```

lemma L3-ControlSystem:
assumes ControlSystemArch s
shows  $hd (s i) \leq (800:: nat)$ 
<proof>

```

9.5 Proof of the Refinement Relation

```

lemma L0-ControlSystem:
assumes  $h1:ControlSystemArch s$ 
shows ControlSystem s
<proof>

```

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  $\Rightarrow$  ('a Frame) istream

```

```

type-synonym nNat = nat  $\Rightarrow$  nat istream

```

```

type-synonym nConfig = nat  $\Rightarrow$  Config

```

```

consts sN :: nat

```

```

definition
  sheafNumbers :: nat list
where
  sheafNumbers  $\equiv$  [sN]

```

end

11 FlexRay: Specification

```
theory FR
imports FR-types
begin
```

11.1 Auxiliary predicates

definition

DisjointSchedules :: $nat \Rightarrow nConfig \Rightarrow bool$

where

DisjointSchedules n nC

\equiv

$\forall i j. i < n \wedge j < n \wedge i \neq j \longrightarrow$

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 :: $nat \Rightarrow nConfig \Rightarrow bool$

where

IdenticCycleLength n nC

\equiv

$\forall i j. i < n \wedge j < n \longrightarrow$

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 ::

$nat \Rightarrow 'a nFrame \Rightarrow 'a nFrame \Rightarrow nNat \Rightarrow nConfig \Rightarrow bool$

where

FrameTransmission n $nStore$ $nReturn$ $nGet$ nC

\equiv

$\forall (t::nat) (k::nat). k < n \longrightarrow$

(*let* $s = t \bmod (\text{cycleLength } (nC) k)$

in

(*s mem* (*schedule* (nC k))

\longrightarrow

(*nGet* k t) = [s] \wedge

($\forall j. j < n \wedge j \neq k \longrightarrow$

((*nStore* j) t) = ((*nReturn* k) t))))

— The predicate *Broadcast* describes properties of FlexRay broadcast.

definition

Broadcast ::
 $\text{nat} \Rightarrow 'a \text{ nFrame} \Rightarrow 'a \text{ Frame istream} \Rightarrow \text{bool}$

where

Broadcast $n \text{ nSend} \text{ recv}$
 \equiv
 $\forall (t::\text{nat}).$
 (*if* $\exists k. k < n \wedge ((\text{nSend } k) t) \neq []$
 then $(\text{recv } t) = ((\text{nSend } (\text{SOME } k. k < n \wedge ((\text{nSend } k) t) \neq [])) t)$
 else $(\text{recv } t) = []$)

— The predicate Receive defines the relations on the streams to represent
 — data receive by FlexRay controller.

definition

Receive ::
 $'a \text{ Frame istream} \Rightarrow 'a \text{ Frame istream} \Rightarrow \text{nat istream} \Rightarrow \text{bool}$

where

Receive $\text{recv} \text{ store} \text{ activation}$
 \equiv
 $\forall (t::\text{nat}).$
 (*if* $(\text{activation } t) = []$
 then $(\text{store } t) = (\text{recv } t)$
 else $(\text{store } t) = []$)

— The predicate Send defines the relations on the streams to represent
 — sending data by FlexRay controller.

definition

Send ::
 $'a \text{ Frame istream} \Rightarrow 'a \text{ Frame istream} \Rightarrow \text{nat istream} \Rightarrow \text{nat istream} \Rightarrow \text{bool}$

where

Send $\text{return} \text{ send} \text{ get} \text{ activation}$
 \equiv
 $\forall (t::\text{nat}).$
 (*if* $(\text{activation } t) = []$
 then $(\text{get } t) = [] \wedge (\text{send } t) = []$
 else $(\text{get } t) = (\text{activation } t) \wedge (\text{send } t) = (\text{return } t)$)

11.2 Specifications of the FlexRay components

definition

FlexRay ::
 $\text{nat} \Rightarrow 'a \text{ nFrame} \Rightarrow \text{nConfig} \Rightarrow 'a \text{ nFrame} \Rightarrow \text{nNat} \Rightarrow \text{bool}$

where

FlexRay $n \text{ nReturn} \text{ nC} \text{ nStore} \text{ nGet}$
 \equiv
 $(\text{CorrectSheaf } n) \wedge$
 $(\forall (i::\text{nat}). i < n \longrightarrow (\text{msg } 1 \text{ (nReturn } i))) \wedge$
 $(\text{DisjointSchedules } n \text{ nC}) \wedge (\text{IdenticalCycleLength } n \text{ nC})$
 \longrightarrow

$$(FrameTransmission\ n\ nStore\ nReturn\ nGet\ nC) \wedge$$

$$(\forall (i::nat). i < n \longrightarrow (msg\ 1\ (nGet\ i)) \wedge (msg\ 1\ (nStore\ i))))$$

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) ∧ (IdenticalCycleLength 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) ≠ []
  ⟨proof⟩

```

```

lemma Send-L2:
assumes Send return send get activation
  and (activation t) ≠ []
  and return t ≠ []
shows (send t) ≠ []
  ⟨proof⟩

```

12.2 Properties of the component Scheduler

```

lemma Scheduler-L1:

```

assumes $h1$:Scheduler C activation
and $h2$:(activation t) $\neq \square$
shows $(t \bmod (\text{cycleLength } C)) \text{ mem } (\text{schedule } C)$
 $\langle \text{proof} \rangle$

lemma Scheduler-L2:
assumes Scheduler C activation
and $\neg (t \bmod \text{cycleLength } C) \text{ mem } (\text{schedule } C)$
shows activation $t = \square$
 $\langle \text{proof} \rangle$

lemma Scheduler-L3:
assumes Scheduler C activation
and $(t \bmod \text{cycleLength } C) \text{ mem } (\text{schedule } C)$
shows activation $t \neq \square$
 $\langle \text{proof} \rangle$

lemma Scheduler-L4:
assumes Scheduler C activation
and $(t \bmod \text{cycleLength } C) \text{ mem } (\text{schedule } C)$
shows activation $t = [t \bmod \text{cycleLength } C]$
 $\langle \text{proof} \rangle$

lemma correct-DisjointSchedules1:
assumes $h1$:DisjointSchedules n nC
and $h2$:IdenticCycleLength n nC
and $h3$:($t \bmod \text{cycleLength } (nC \ i)$) mem schedule ($nC \ i$)
and $h4$: $i < n$
and $h5$: $j < n$
and $h6$: $i \neq j$
shows $\neg (t \bmod \text{cycleLength } (nC \ j)) \text{ mem } \text{schedule } (nC \ j)$
 $\langle \text{proof} \rangle$

12.3 Disjoint Frames

lemma disjointFrame-L1:
assumes $h1$:DisjointSchedules n nC
and $h2$:IdenticCycleLength n nC
and $h3$: $\forall i < n. \text{FlexRayController } (n\text{Return } i) \text{ rcv}$
 $(nC \ i) (n\text{Store } i) (n\text{Send } i) (n\text{Get } i)$
and $h4$: $n\text{Send } i \ t \neq \square$
and $h5$: $i < n$
and $h6$: $j < n$
and $h7$: $i \neq j$
shows $n\text{Send } j \ t = \square$
 $\langle \text{proof} \rangle$

lemma disjointFrame-L2:
assumes DisjointSchedules n nC

and *IdenticCycleLength* n nC
and $\forall i < n$. *FlexRayController* $(nReturn\ i)$ rcv
 $(nC\ i)$ $(nStore\ i)$ $(nSend\ i)$ $(nGet\ i)$
shows *inf-disj* n $nSend$
 $\langle proof \rangle$

lemma *disjointFrame-L3*:
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 $h4:t\ mod\ cycleLength\ (nC\ i)\ mem\ schedule\ (nC\ i)$
and $h5:i < n$
and $h6:j < n$
and $h7:i \neq j$
shows $nSend\ j\ t = \square$
 $\langle proof \rangle$

12.4 Properties of the sheaf of channels $nSend$

lemma *fr-Send1*:
assumes $frc:FlexRayController\ (nReturn\ i)\ rcv\ (nC\ i)\ (nStore\ i)\ (nSend\ i)\ (nGet\ i)$
and $h1:\neg (t\ mod\ cycleLength\ (nC\ i)\ mem\ schedule\ (nC\ i))$
shows $(nSend\ i)\ t = \square$
 $\langle proof \rangle$

lemma *fr-Send2*:
assumes $h1:\forall i < n$. *FlexRayController* $(nReturn\ i)$ $rcv\ (nC\ i)\ (nStore\ i)\ (nSend\ i)\ (nGet\ i)$
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 $nSend\ k\ t = nReturn\ k\ t$
 $\langle proof \rangle$

lemma *fr-Send3*:
assumes $\forall i < n$. *FlexRayController* $(nReturn\ i)$ $rcv\ (nC\ i)\ (nStore\ i)\ (nSend\ i)\ (nGet\ i)$
and *DisjointSchedules* $n\ nC$
and *IdenticCycleLength* $n\ nC$
and $t\ mod\ cycleLength\ (nC\ k)\ mem\ schedule\ (nC\ k)$
and $k < n$
and $nReturn\ k\ t \neq \square$
shows $nSend\ k\ t \neq \square$
 $\langle proof \rangle$

lemma *fr-Send4*:

assumes $\forall i < n. \text{FlexRayController } (n\text{Return } i) \text{ recv } (nC \ i) \ (n\text{Store } i) \ (n\text{Send } i) \ (n\text{Get } i)$
and *DisjointSchedules* $n \ nC$
and *IdenticCycleLength* $n \ nC$
and $t \bmod \text{cycleLength } (nC \ k) \text{ mem schedule } (nC \ k)$
and $k < n$
and $n\text{Return } k \ t \neq []$
shows $\exists k. k < n \longrightarrow n\text{Send } k \ t \neq []$
<proof>

lemma *fr-Send5*:

assumes $h1: \forall i < n. \text{FlexRayController } (n\text{Return } i) \text{ recv } (nC \ i) \ (n\text{Store } i) \ (n\text{Send } i) \ (n\text{Get } i)$
and $h2: \text{DisjointSchedules } n \ nC$
and $h3: \text{IdenticCycleLength } n \ nC$
and $h4: t \bmod \text{cycleLength } (nC \ k) \text{ mem schedule } (nC \ k)$
and $h5: k < n$
and $h6: n\text{Return } k \ t \neq []$
and $h7: \forall k < n. n\text{Send } k \ t = []$
shows *False*
<proof>

lemma *fr-Send6*:

assumes $\forall i < n. \text{FlexRayController } (n\text{Return } i) \text{ recv } (nC \ i) \ (n\text{Store } i) \ (n\text{Send } i) \ (n\text{Get } i)$
and *DisjointSchedules* $n \ nC$
and *IdenticCycleLength* $n \ nC$
and $t \bmod \text{cycleLength } (nC \ k) \text{ mem schedule } (nC \ k)$
and $k < n$
and $n\text{Return } k \ t \neq []$
shows $\exists k < n. n\text{Send } k \ t \neq []$
<proof>

lemma *fr-Send7*:

assumes $\forall i < n. \text{FlexRayController } (n\text{Return } i) \text{ recv } (nC \ i) \ (n\text{Store } i) \ (n\text{Send } i) \ (n\text{Get } i)$
and *DisjointSchedules* $n \ nC$
and *IdenticCycleLength* $n \ nC$
and $t \bmod \text{cycleLength } (nC \ k) \text{ mem schedule } (nC \ k)$
and $k < n$
and $j < n$
and $n\text{Return } k \ t = []$
shows $n\text{Send } j \ t = []$
<proof>

lemma *fr-Send8*:

assumes $\forall i < n. \text{FlexRayController } (n\text{Return } i) \text{ recv } (nC \ i) \ (n\text{Store } i) \ (n\text{Send } i) \ (n\text{Get } i)$

and *DisjointSchedules* n nC
and *IdenticCycleLength* n nC
and $t \bmod \text{cycleLength } (nC\ k) \text{ mem schedule } (nC\ k)$
and $k < n$
and $n\text{Return } k\ t = []$
shows $\neg (\exists k < n. n\text{Send } k\ t \neq [])$
 $\langle \text{proof} \rangle$

lemma *fr-nC-Send*:

assumes $\forall i < n. \text{FlexRayController } (n\text{Return } i) \text{ recv } (nC\ i) (n\text{Store } i) (n\text{Send } i) (n\text{Get } i)$
and $k < n$
and *DisjointSchedules* n nC
and *IdenticCycleLength* n nC
and $t \bmod \text{cycleLength } (nC\ k) \text{ mem schedule } (nC\ k)$
shows $\forall j. j < n \wedge j \neq k \longrightarrow (n\text{Send } j) t = []$
 $\langle \text{proof} \rangle$

lemma *length-nSend*:

assumes $h1: \text{BusInterface activation } (n\text{Return } i) \text{ recv } (n\text{Store } i) (n\text{Send } i) (n\text{Get } i)$
and $h2: \forall t. \text{length } (n\text{Return } i\ t) \leq \text{Suc } 0$
shows $\text{length } (n\text{Send } i\ t) \leq \text{Suc } 0$
 $\langle \text{proof} \rangle$

lemma *msg-nSend*:

assumes *BusInterface activation* $(n\text{Return } i) \text{ recv } (n\text{Store } i) (n\text{Send } i) (n\text{Get } i)$
and $\text{msg } (\text{Suc } 0) (n\text{Return } i)$
shows $\text{msg } (\text{Suc } 0) (n\text{Send } i)$
 $\langle \text{proof} \rangle$

lemma *Broadcast-nSend-empty1*:

assumes $h1: \text{Broadcast } n\ n\text{Send } \text{recv}$
and $h2: \forall k < n. n\text{Send } k\ t = []$
shows $\text{recv } t = []$
 $\langle \text{proof} \rangle$

12.5 Properties of the sheaf of channels nGet

lemma *fr-nGet1a*:

assumes $h1: \text{FlexRayController } (n\text{Return } k) \text{ recv } (nC\ k) (n\text{Store } k) (n\text{Send } k) (n\text{Get } k)$
and $h2: t \bmod \text{cycleLength } (nC\ k) \text{ mem schedule } (nC\ k)$
shows $n\text{Get } k\ t = [t \bmod \text{cycleLength } (nC\ k)]$
 $\langle \text{proof} \rangle$

lemma *fr-nGet1*:

assumes $\forall i < n. \text{FlexRayController } (n\text{Return } i) \text{ recv } (nC\ i) (n\text{Store } i) (n\text{Send } i) (n\text{Get } i)$

and $t \bmod \text{cycleLength } (nC \ k) \text{ mem schedule } (nC \ k)$
and $k < n$
shows $n\text{Get } k \ t = [t \bmod \text{cycleLength } (nC \ k)]$
 $\langle \text{proof} \rangle$

lemma *fr-nGet2a*:

assumes $h1: \text{FlexRayController } (n\text{Return } k) \text{ recv } (nC \ k) (n\text{Store } k) (n\text{Send } k)$
 $(n\text{Get } k)$
and $h2: \neg (t \bmod \text{cycleLength } (nC \ k) \text{ mem schedule } (nC \ k))$
shows $n\text{Get } k \ t = []$
 $\langle \text{proof} \rangle$

lemma *fr-nGet2*:

assumes $h1: \forall i < n. \text{FlexRayController } (n\text{Return } i) \text{ recv } (nC \ i) (n\text{Store } i) (n\text{Send } i)$
 $(n\text{Get } i)$
and $h2: \neg (t \bmod \text{cycleLength } (nC \ k) \text{ mem schedule } (nC \ k))$
and $h3: k < n$
shows $n\text{Get } k \ t = []$
 $\langle \text{proof} \rangle$

lemma *length-nGet1*:

assumes $\text{FlexRayController } (n\text{Return } k) \text{ recv } (nC \ k) (n\text{Store } k) (n\text{Send } k) (n\text{Get } k)$
shows $\text{length } (n\text{Get } k \ t) \leq \text{Suc } 0$
 $\langle \text{proof} \rangle$

lemma *msg-nGet1*:

assumes $\text{FlexRayController } (n\text{Return } k) \text{ recv } (nC \ k) (n\text{Store } k) (n\text{Send } k) (n\text{Get } k)$
shows $\text{msg } (\text{Suc } 0) (n\text{Get } k)$
 $\langle \text{proof} \rangle$

lemma *msg-nGet2*:

assumes $\forall i < n. \text{FlexRayController } (n\text{Return } i) \text{ recv } (nC \ i) (n\text{Store } i) (n\text{Send } i)$
 $(n\text{Get } i)$
and $k < n$
shows $\text{msg } (\text{Suc } 0) (n\text{Get } k)$
 $\langle \text{proof} \rangle$

12.6 Properties of the sheaf of channels nStore

lemma *fr-nStore-nReturn1*:

assumes $h0: \text{Broadcast } n \ n\text{Send } \text{recv}$
and $h1: \text{inf-disj } n \ n\text{Send}$
and $h2: \forall i < n. \text{FlexRayController } (n\text{Return } i) \text{ recv } (nC \ i) (n\text{Store } i) (n\text{Send } i)$
 $(n\text{Get } i)$
and $h3: \text{DisjointSchedules } n \ nC$
and $h4: \text{IdenticalCycleLength } n \ nC$
and $h5: t \bmod \text{cycleLength } (nC \ k) \text{ mem schedule } (nC \ k)$

and $h6:k < n$
and $h7:j < n$
and $h8:j \neq k$
shows $nStore\ j\ t = nReturn\ k\ t$
 $\langle proof \rangle$

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 $nStore\ j\ t = nReturn\ k\ t$
 $\langle proof \rangle$

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 $nStore\ k\ t = []$
 $\langle proof \rangle$

lemma *fr-nStore-nReturn3:*

assumes $Cable\ n\ nSend\ recv$
and $\forall i < n. FlexRayController\ (nReturn\ i)\ recv\ (nC\ i)\ (nStore\ i)\ (nSend\ i)\ (nGet\ i)$
and $DisjointSchedules\ n\ nC$
and $IdenticCycleLength\ n\ nC$
and $t\ mod\ cycleLength\ (nC\ k)\ mem\ schedule\ (nC\ k)$
and $k < n$
shows $\forall j. j < n \wedge j \neq k \longrightarrow nStore\ j\ t = nReturn\ k\ t$
 $\langle proof \rangle$

lemma *length-nStore:*

assumes $h1:\forall i < n. FlexRayController\ (nReturn\ i)\ recv\ (nC\ i)\ (nStore\ i)\ (nSend\ i)\ (nGet\ i)$
and $h2:DisjointSchedules\ n\ nC$
and $h3:IdenticCycleLength\ n\ nC$
and $h4:inf-disj\ n\ nSend$
and $h5:i < n$

and $h6:\forall i < n. \text{msg} (\text{Suc } 0) (n\text{Return } i)$
and $h7:\text{Broadcast } n \text{ nSend } \text{recv}$
shows $\text{length} (n\text{Store } i \ t) \leq \text{Suc } 0$
 $\langle \text{proof} \rangle$

lemma *msg-nStore:*

assumes $\forall i < n. \text{FlexRayController} (n\text{Return } i) \text{recv} (nC \ i) (n\text{Store } i) (n\text{Send } i)$
 $(n\text{Get } i)$
and $\text{DisjointSchedules } n \ nC$
and $\text{IdenticCycleLength } n \ nC$
and $\text{inf-disj } n \ n\text{Send}$
and $i < n$
and $\forall i < n. \text{msg} (\text{Suc } 0) (n\text{Return } i)$
and $\text{Cable } n \ n\text{Send } \text{recv}$
shows $\text{msg} (\text{Suc } 0) (n\text{Store } i)$
 $\langle \text{proof} \rangle$

12.7 Refinement Properties

lemma *fr-refinement-FrameTransmission:*

assumes $\text{Cable } n \ n\text{Send } \text{recv}$
and $\forall i < n. \text{FlexRayController} (n\text{Return } i) \text{recv} (nC \ i) (n\text{Store } i) (n\text{Send } i)$
 $(n\text{Get } i)$
and $\text{DisjointSchedules } n \ nC$
and $\text{IdenticCycleLength } n \ nC$
shows $\text{FrameTransmission } n \ n\text{Store } n\text{Return } n\text{Get } nC$
 $\langle \text{proof} \rangle$

lemma *FlexRayArch-CorrectSheaf:*

assumes $\text{FlexRayArch } n \ n\text{Return } nC \ n\text{Store } n\text{Get}$
shows $\text{CorrectSheaf } n$
 $\langle \text{proof} \rangle$

lemma *FlexRayArch-FrameTransmission:*

assumes $h1:\text{FlexRayArch } n \ n\text{Return } nC \ n\text{Store } n\text{Get}$
and $h2:\forall i < n. \text{msg} (\text{Suc } 0) (n\text{Return } i)$
and $h3:\text{DisjointSchedules } n \ nC$
and $h4:\text{IdenticCycleLength } n \ nC$
shows $\text{FrameTransmission } n \ n\text{Store } n\text{Return } n\text{Get } nC$
 $\langle \text{proof} \rangle$

lemma *FlexRayArch-nGet:*

assumes $h1:\text{FlexRayArch } n \ n\text{Return } nC \ n\text{Store } n\text{Get}$
and $h2:\forall i < n. \text{msg} (\text{Suc } 0) (n\text{Return } i)$
and $h3:\text{DisjointSchedules } n \ nC$
and $h4:\text{IdenticCycleLength } n \ nC$
and $h5:i < n$
shows $\text{msg} (\text{Suc } 0) (n\text{Get } i)$
 $\langle \text{proof} \rangle$

```

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:IdenticalCycleLength n nC
    and h5:i < n
  shows      msg (Suc 0) (nStore i)
  <proof>

```

```

theorem main-fr-refinement:
  assumes FlexRayArch n nReturn nC nStore nGet
  shows   FlexRay n nReturn nC nStore nGet
  <proof>

```

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 = [] ∧ a (Suc t) = (if (i t) = [] then [] else [sc-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
  ≡

```

$$\begin{aligned}
& \forall (t::\text{nat}) \\
& \quad (r::\text{reqType list}) (x::\text{aType list}) \\
& \quad (y::\text{stopType list}) (z::\text{bool list}). \\
& \text{--- 1:} \\
& \quad (st\text{-in } t = \text{init-state} \wedge \text{req } t = [\text{init}] \\
& \quad \quad \rightarrow \text{ack } t = [\text{call}] \wedge i1 \ t = [] \wedge \text{vc } t = [] \\
& \quad \quad \wedge \text{st-out } t = \text{call}) \\
& \wedge \\
& \text{--- 2:} \\
& \quad (st\text{-in } t = \text{init-state} \wedge \text{req } t \neq [\text{init}] \\
& \quad \quad \rightarrow \text{ack } t = [\text{init-state}] \wedge i1 \ t = [] \wedge \text{vc } t = [] \\
& \quad \quad \wedge \text{st-out } t = \text{init-state}) \\
& \wedge \\
& \text{--- 3:} \\
& \quad ((st\text{-in } t = \text{call} \vee (st\text{-in } t = \text{connection-ok} \wedge r \neq [\text{send}])) \wedge \\
& \quad \quad \text{req } t = r \wedge \text{lose } t = [\text{False}] \\
& \quad \quad \rightarrow \text{ack } t = [\text{connection-ok}] \wedge i1 \ t = [] \wedge \text{vc } t = [] \\
& \quad \quad \wedge \text{st-out } t = \text{connection-ok}) \\
& \wedge \\
& \text{--- 4:} \\
& \quad ((st\text{-in } t = \text{call} \vee st\text{-in } t = \text{connection-ok} \vee st\text{-in } t = \text{sending-data}) \\
& \quad \quad \wedge \text{lose } t = [\text{True}] \\
& \quad \quad \rightarrow \text{ack } t = [\text{init-state}] \wedge i1 \ t = [] \wedge \text{vc } t = [] \\
& \quad \quad \wedge \text{st-out } t = \text{init-state}) \\
& \wedge \\
& \text{--- 5:} \\
& \quad (st\text{-in } t = \text{connection-ok} \wedge \text{req } t = [\text{send}] \wedge \text{lose } t = [\text{False}] \\
& \quad \quad \rightarrow \text{ack } t = [\text{sending-data}] \wedge i1 \ t = \text{buffer-in } t \wedge \text{vc } t = [] \\
& \quad \quad \wedge \text{st-out } t = \text{sending-data}) \\
& \wedge \\
& \text{--- 6:} \\
& \quad (st\text{-in } t = \text{sending-data} \wedge a1 \ t = [] \wedge \text{lose } t = [\text{False}] \\
& \quad \quad \rightarrow \text{ack } t = [\text{sending-data}] \wedge i1 \ t = [] \wedge \text{vc } t = [] \\
& \quad \quad \wedge \text{st-out } t = \text{sending-data}) \\
& \wedge \\
& \text{--- 7:} \\
& \quad (st\text{-in } t = \text{sending-data} \wedge a1 \ t = [\text{sc-ack}] \wedge \text{lose } t = [\text{False}] \\
& \quad \quad \rightarrow \text{ack } t = [\text{voice-com}] \wedge i1 \ t = [] \wedge \text{vc } t = [\text{vc-com}] \\
& \quad \quad \wedge \text{st-out } t = \text{voice-com}) \\
& \wedge \\
& \text{--- 8:} \\
& \quad (st\text{-in } t = \text{voice-com} \wedge \text{stop } t = [] \wedge \text{lose } t = [\text{False}] \\
& \quad \quad \rightarrow \text{ack } t = [\text{voice-com}] \wedge i1 \ t = [] \wedge \text{vc } t = [\text{vc-com}] \\
& \quad \quad \wedge \text{st-out } t = \text{voice-com}) \\
& \wedge \\
& \text{--- 9:} \\
& \quad (st\text{-in } t = \text{voice-com} \wedge \text{stop } t = [] \wedge \text{lose } t = [\text{True}] \\
& \quad \quad \rightarrow \text{ack } t = [\text{voice-com}] \wedge i1 \ t = [] \wedge \text{vc } t = [] \\
& \quad \quad \wedge \text{st-out } t = \text{voice-com})
\end{aligned}$$

$$\wedge$$

— 10:

$$(st\text{-in } t = \text{voice-com} \wedge stop\ t = [stop\text{-vc}]$$

$$\longrightarrow ack\ t = [init\text{-state}] \wedge i1\ t = [] \wedge vc\ t = []$$

$$\wedge st\text{-out } t = init\text{-state})$$

definition

Sample-L ::

$$reqType\ istream \Rightarrow ECall\text{-Info}\ istream \Rightarrow aType\ istream \Rightarrow$$

$$stopType\ istream \Rightarrow bool\ istream \Rightarrow$$

$$(nat \Rightarrow GatewayStatus) \Rightarrow (nat \Rightarrow ECall\text{-Info}\ list) \Rightarrow$$

$$GatewayStatus\ istream \Rightarrow ECall\text{-Info}\ istream \Rightarrow vcType\ istream$$

$$\Rightarrow (nat \Rightarrow GatewayStatus) \Rightarrow (nat \Rightarrow ECall\text{-Info}\ list)$$

$$\Rightarrow bool$$

where

Sample-L req dt a1 stop lose st-in buffer-in
ack i1 vc st-out buffer-out

$$\equiv$$

$$(\forall (t::nat).$$

$$buffer\text{-out } t =$$

$$(if\ dt\ t = []\ then\ buffer\text{-in } t\ else\ dt\ t))$$

$$\wedge$$

$$(tiTable\text{-SampleT}\ req\ a1\ stop\ lose\ st\text{-in}\ buffer\text{-in}$$

$$ack\ i1\ vc\ st\text{-out})$$

definition

Sample ::

$$reqType\ istream \Rightarrow ECall\text{-Info}\ istream \Rightarrow aType\ istream \Rightarrow$$

$$stopType\ istream \Rightarrow bool\ istream \Rightarrow$$

$$GatewayStatus\ istream \Rightarrow ECall\text{-Info}\ istream \Rightarrow vcType\ istream$$

$$\Rightarrow bool$$

where

Sample req dt a1 stop lose ack i1 vc

$$\equiv$$

$$((msg\ (1::nat)\ req) \wedge$$

$$(msg\ (1::nat)\ a1) \wedge$$

$$(msg\ (1::nat)\ stop))$$

$$\longrightarrow$$

$$(\exists\ st\ buffer.$$

$$(Sample\text{-L}\ req\ dt\ a1\ stop\ lose$$

$$(fin\text{-inf}\text{-append } [init\text{-state}]\ st)$$

$$(fin\text{-inf}\text{-append } []\ buffer)$$

$$ack\ i1\ vc\ st\ buffer))$$

definition

Gateway ::

$$reqType\ istream \Rightarrow ECall\text{-Info}\ istream \Rightarrow aType\ istream \Rightarrow$$

$$stopType\ istream \Rightarrow bool\ istream \Rightarrow nat \Rightarrow$$

$$GatewayStatus\ istream \Rightarrow ECall\text{-Info}\ istream \Rightarrow vcType\ istream$$

$\Rightarrow \text{bool}$

where

$\text{Gateway req dt a stop lose d ack i vc}$
 $\equiv \exists i1 i2 x y.$
 $(\text{Sample req dt x stop lose ack i1 vc}) \wedge$
 $(\text{Delay y i1 d x i2}) \wedge$
 $(\text{Loss lose a i2 y i})$

definition

$\text{GatewaySystem} ::$
 $\text{reqType istream} \Rightarrow \text{ECall-Info istream} \Rightarrow$
 $\text{stopType istream} \Rightarrow \text{bool istream} \Rightarrow \text{nat} \Rightarrow$
 $\text{GatewayStatus istream} \Rightarrow \text{vcType istream}$
 $\Rightarrow \text{bool}$

where

$\text{GatewaySystem req dt stop lose d ack vc}$
 \equiv
 $\exists a i.$
 $(\text{Gateway req dt a stop lose d ack i vc}) \wedge$
 $(\text{ServiceCenter i a})$

definition

$\text{GatewayReq} ::$
 $\text{reqType istream} \Rightarrow \text{ECall-Info istream} \Rightarrow \text{aType istream} \Rightarrow$
 $\text{stopType istream} \Rightarrow \text{bool istream} \Rightarrow \text{nat} \Rightarrow$
 $\text{GatewayStatus istream} \Rightarrow \text{ECall-Info istream} \Rightarrow \text{vcType istream}$
 $\Rightarrow \text{bool}$

where

$\text{GatewayReq req dt a stop lose d ack i vc}$
 \equiv
 $(\text{msg } (1::\text{nat}) \text{ req}) \wedge (\text{msg } (1::\text{nat}) \text{ a}) \wedge$
 $(\text{msg } (1::\text{nat}) \text{ stop}) \wedge (\text{ts lose})$
 \longrightarrow
 $(\forall (t::\text{nat}).$
 $(\text{ack } t = [\text{init-state}] \wedge \text{req } (\text{Suc } t) = [\text{init}] \wedge$
 $\text{lose } (t+1) = [\text{False}] \wedge \text{lose } (t+2) = [\text{False}]$
 $\longrightarrow \text{ack } (t+2) = [\text{connection-ok}])$
 \wedge
 $(\text{ack } t = [\text{connection-ok}] \wedge \text{req } (\text{Suc } t) = [\text{send}] \wedge$
 $(\forall (k::\text{nat}). k \leq (d+1) \longrightarrow \text{lose } (t+k) = [\text{False}])$
 $\longrightarrow i ((\text{Suc } t) + d) = \text{inf-last-ti dt } t$
 $\wedge \text{ack } (\text{Suc } t) = [\text{sending-data}])$
 \wedge
 $(\text{ack } (t+d) = [\text{sending-data}] \wedge \text{a } (\text{Suc } t) = [\text{sc-ack}] \wedge$
 $(\forall (k::\text{nat}). k \leq (d+1) \longrightarrow \text{lose } (t+k) = [\text{False}])$
 $\longrightarrow \text{vc } ((\text{Suc } t) + d) = [\text{vc-com}])$)

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⟩

lemma aType-nonempty:
  assumes h1:msg (Suc 0) a
  and h2: a t ≠ []
  shows a t = [sc-ack]
  ⟨proof⟩

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

lemma stopType-empty:
  assumes msg (Suc 0) a

```


and $a\ t \neq [stop-vc]$
shows $a\ t = []$
 $\langle proof \rangle$

lemma *stopType-nonempty*:
assumes $msg\ (Suc\ 0)\ a$
and $a\ t \neq []$
shows $a\ t = [stop-vc]$
 $\langle proof \rangle$

lemma *stopType-lemma*:
assumes $msg\ (Suc\ 0)\ a$
shows $a\ t = [] \vee a\ t = [stop-vc]$
 $\langle proof \rangle$

lemma *vcType-empty*:
assumes $msg\ (Suc\ 0)\ a$
and $a\ t \neq [vc-com]$
shows $a\ t = []$
 $\langle proof \rangle$

lemma *vcType-lemma*:
assumes $msg\ (Suc\ 0)\ a$
shows $a\ t = [] \vee a\ t = [vc-com]$
 $\langle proof \rangle$

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 $i2\ t2 = []$
 $\langle proof \rangle$

lemma *Delay-L2*:
assumes $\forall t1 < t. i1\ t1 = []$
and $Delay\ y\ i1\ d\ x\ i2$
shows $\forall t2 < t + d. i2\ t2 = []$
 $\langle proof \rangle$

lemma *Delay-L3*:
assumes $h1:\forall t1 \leq t. y\ t1 = []$
and $h2:Delay\ y\ i1\ d\ x\ i2$
and $h3:t2 \leq t + d$
shows $x\ t2 = []$
 $\langle proof \rangle$

lemma *Delay-L4*:

assumes $\forall t1 \leq t. y\ t1 = []$
and $Delay\ y\ i1\ d\ x\ i2$
shows $\forall t2 \leq t + d. x\ t2 = []$
<proof>

lemma *Delay-lengthOut1*:
assumes $h1:\forall t. length\ (x\ t) \leq Suc\ 0$
and $h2:Delay\ x\ i1\ d\ y\ i2$
shows $length\ (y\ t) \leq Suc\ 0$
<proof>

lemma *Delay-msg1*:
assumes $msg\ (Suc\ 0)\ x$
and $Delay\ x\ i1\ d\ y\ i2$
shows $msg\ (Suc\ 0)\ y$
<proof>

15.3 Properties of the Loss component

lemma *Loss-L1*:
assumes $\forall t2 < t. i2\ t2 = []$
and $Loss\ lose\ a\ i2\ y\ i$
and $t2 < t$
and $ts\ lose$
shows $i\ t2 = []$
<proof>

lemma *Loss-L2*:
assumes $\forall t2 < t. i2\ t2 = []$
and $Loss\ lose\ a\ i2\ y\ i$
and $ts\ lose$
shows $\forall t2 < t. i\ t2 = []$
<proof>

lemma *Loss-L3*:
assumes $\forall t2 < t. a\ t2 = []$
and $Loss\ lose\ a\ i2\ y\ i$
and $t2 < t$
and $ts\ lose$
shows $y\ t2 = []$
<proof>

lemma *Loss-L4*:
assumes $\forall t2 < t. a\ t2 = []$
and $Loss\ lose\ a\ i2\ y\ i$
and $ts\ lose$
shows $\forall t2 < t. y\ t2 = []$
<proof>

lemma *Loss-L5*:
assumes $\forall t1 \leq t. a\ t1 = []$
and *Loss lose a i2 y i*
and $t2 \leq t$
and *ts lose*
shows $y\ t2 = []$
 $\langle proof \rangle$

lemma *Loss-L5Suc*:
assumes $\forall j \leq d. a\ (t + Suc\ j) = []$
and *Loss lose a i2 y i*
and $Suc\ j \leq d$
and *tsLose:ts lose*
shows $y\ (t + Suc\ j) = []$
 $\langle proof \rangle$

lemma *Loss-L6*:
assumes $\forall t2 \leq t. a\ t2 = []$
and *Loss lose a i2 y i*
and *ts lose*
shows $\forall t2 \leq t. y\ t2 = []$
 $\langle proof \rangle$

lemma *Loss-lengthOut1*:
assumes $h1:\forall t. length\ (a\ t) \leq Suc\ 0$
and $h2:Loss\ lose\ a\ i2\ x\ i$
shows $length\ (x\ t) \leq Suc\ 0$
 $\langle proof \rangle$

lemma *Loss-lengthOut2*:
assumes $\forall t. length\ (a\ t) \leq Suc\ 0$
and *Loss lose a i2 x i*
shows $\forall t. length\ (x\ t) \leq Suc\ 0$
 $\langle proof \rangle$

lemma *Loss-msg1*:
assumes $msg\ (Suc\ 0)\ a$
and *Loss lose a i2 x i*
shows $msg\ (Suc\ 0)\ x$
 $\langle proof \rangle$

15.4 Properties of the composition of Delay and Loss components

lemma *Loss-Delay-length-y*:
assumes $\forall t. length\ (a\ t) \leq Suc\ 0$
and *Delay x i1 d y i2*
and *Loss lose a i2 x i*
shows $length\ (y\ t) \leq Suc\ 0$

<proof>

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*
<proof>

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))) ≠ []*
<proof>

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>

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>

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]*
<proof>

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>

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]*
<proof>

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 $t2 - d < t + (3::nat) + k$
 $\langle proof \rangle$

lemma *ts-lose-ack-st1ts*:

assumes *ts lose*
and $lose\ t = [True] \longrightarrow ack\ t = [x] \wedge st-out\ t = x$
and $lose\ t = [False] \longrightarrow ack\ t = [y] \wedge st-out\ t = y$
shows $ack\ t = [st-out\ t]$
 $\langle proof \rangle$

lemma *ts-lose-ack-st1*:

assumes $h1:lose\ t = [True] \vee lose\ t = [False]$
and $h2:lose\ t = [True] \longrightarrow ack\ t = [x] \wedge st-out\ t = x$
and $h3:lose\ t = [False] \longrightarrow ack\ t = [y] \wedge st-out\ t = y$
shows $ack\ t = [st-out\ t]$
 $\langle proof \rangle$

lemma *ts-lose-ack-st2ts*:

assumes *ts lose*
and $lose\ t = [True] \longrightarrow$
 $ack\ t = [x] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st-out\ t = x$
and $lose\ t = [False] \longrightarrow$
 $ack\ t = [y] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st-out\ t = y$
shows $ack\ t = [st-out\ t]$
 $\langle proof \rangle$

lemma *ts-lose-ack-st2*:

assumes $h1:lose\ t = [True] \vee lose\ t = [False]$
and $h2:lose\ t = [True] \longrightarrow$
 $ack\ t = [x] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st-out\ t = x$
and $h3:lose\ t = [False] \longrightarrow$
 $ack\ t = [y] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st-out\ t = y$
shows $ack\ t = [st-out\ t]$
 $\langle proof \rangle$

lemma *ts-lose-ack-st2vc-com*:

assumes $h1:lose\ t = [True] \vee lose\ t = [False]$
and $h2:lose\ t = [True] \longrightarrow$
 $ack\ t = [x] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st-out\ t = x$
and $h3:lose\ t = [False] \longrightarrow$
 $ack\ t = [y] \wedge i1\ t = [] \wedge vc\ t = [vc-com] \wedge st-out\ t = y$
shows $ack\ t = [st-out\ t]$
 $\langle proof \rangle$

lemma *ts-lose-ack-st2send*:

assumes $h1:lose\ t = [True] \vee lose\ t = [False]$
and $h2:lose\ t = [True] \longrightarrow$
 $ack\ t = [x] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st-out\ t = x$
and $h3:lose\ t = [False] \longrightarrow$

$ack\ t = [y] \wedge i1\ t = b\ t \wedge vc\ t = [] \wedge st-out\ t = y$
shows $ack\ t = [st-out\ t]$
 <proof>

lemma *tiTable-ack-st-splitten*:

assumes $h1:ts\ lose$
and $h2:msg\ (Suc\ 0)\ a1$
and $h3:msg\ (Suc\ 0)\ stop$
and $h4:st-in\ t = init-state \wedge req\ t = [init] \longrightarrow$
 $ack\ t = [call] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st-out\ t = call$
and $h5:st-in\ t = init-state \wedge req\ t \neq [init] \longrightarrow$
 $ack\ t = [init-state] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st-out\ t = init-state$
and $h6:(st-in\ t = call \vee st-in\ t = connection-ok \wedge req\ t \neq [send]) \wedge lose\ t =$
 $[False] \longrightarrow$
 $ack\ t = [connection-ok] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st-out\ t = connection-ok$
and $h7:(st-in\ t = call \vee st-in\ t = connection-ok \vee st-in\ t = sending-data) \wedge$
 $lose\ t = [True] \longrightarrow$
 $ack\ t = [init-state] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st-out\ t = init-state$
and $h8:st-in\ t = connection-ok \wedge req\ t = [send] \wedge lose\ t = [False] \longrightarrow$
 $ack\ t = [sending-data] \wedge i1\ t = b\ t \wedge vc\ t = [] \wedge st-out\ t = sending-data$
and $h9:st-in\ t = sending-data \wedge a1\ t = [] \wedge lose\ t = [False] \longrightarrow$
 $ack\ t = [sending-data] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st-out\ t = sending-data$
and $h10:st-in\ t = sending-data \wedge a1\ t = [sc-ack] \wedge lose\ t = [False] \longrightarrow$
 $ack\ t = [voice-com] \wedge i1\ t = [] \wedge vc\ t = [vc-com] \wedge st-out\ t = voice-com$
and $h11:st-in\ t = voice-com \wedge stop\ t = [] \wedge lose\ t = [False] \longrightarrow$
 $ack\ t = [voice-com] \wedge i1\ t = [] \wedge vc\ t = [vc-com] \wedge st-out\ t = voice-com$
and $h12:st-in\ t = voice-com \wedge stop\ t = [] \wedge lose\ t = [True] \longrightarrow$
 $ack\ t = [voice-com] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st-out\ t = voice-com$
and $h13:st-in\ t = voice-com \wedge stop\ t = [stop-vc] \longrightarrow$
 $ack\ t = [init-state] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st-out\ t = init-state$
shows $ack\ t = [st-out\ t]$
 <proof>

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>

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 $st-out\ t = hd\ (ack\ t)$
 <proof>

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 $(st-in\ t = call \vee st-in\ t = connection-ok \wedge req\ t \neq [send]) \wedge$
 $lose\ t = [False]$
 $\langle proof \rangle$

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 = []*
 $\langle proof \rangle$

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 $st-in\ t = init-state \wedge req\ t = [init]$
 $\langle proof \rangle$

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 = []*
 $\langle proof \rangle$

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]*
 $\langle proof \rangle$

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 $\forall t1 \leq t. req\ t1 = []$
shows $ack\ t = [init-state]$
 $\langle proof \rangle$

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 $h5:\forall t1 \leq t. req\ t1 = []$
shows $i1\ t = []$
 $\langle proof \rangle$

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 $h3:\forall m \leq k. ack$ (Suc (Suc ($t + m$))) = $[connection-ok]$
and $h4:st$ ($Suc\ t$) = $call$
shows st (Suc ($Suc\ t$)) = $connection-ok$
 $\langle proof \rangle$

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 $\forall t1 \leq t. req\ t1 = []$
and req ($Suc\ t$) = $[init]$
and $\forall m < k + 3. req$ ($t + m$) $\neq [send]$
and $h7:\forall m \leq k. ack$ (Suc (Suc ($t + m$))) = $[connection-ok]$
and $\forall j \leq k + 3. lose$ ($t + j$) = $[False]$
and $h9:t2 < (t + 3 + k)$
shows $i1\ t2 = []$
 $\langle proof \rangle$

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 $\forall t1 \leq t. req\ t1 = []$

and $\text{req } (\text{Suc } t) = [\text{init}]$
and $\forall m < k + 3. \text{req } (t + m) \neq [\text{send}]$
and $\forall m \leq k. \text{ack } (\text{Suc } (\text{Suc } (t + m))) = [\text{connection-ok}]$
and $\forall j \leq k + 3. \text{lose } (t + j) = [\text{False}]$
shows $\forall t2 < (t + 3 + k). i1 \ t2 = []$
 $\langle \text{proof} \rangle$

lemma *tiTable-ack-ok*:

assumes $h1: \forall j \leq d + 2. \text{lose } (t + j) = [\text{False}]$
and $ts\text{Lose}: ts \ \text{lose}$
and $stop\text{Msg1}: \text{msg } (\text{Suc } 0) \ \text{stop}$
and $a1\text{Msg1}: \text{msg } (\text{Suc } 0) \ a1$
and $\text{reqNsend}: \text{req } (\text{Suc } t) \neq [\text{send}]$
and $\text{ackCon}: \text{ack } t = [\text{connection-ok}]$
and $tbl: \text{tiTable-SampleT } \text{req } a1 \ \text{stop } \text{lose } (\text{fin-inf-append } [\text{init-state}] \ st) \ b \ \text{ack}$
 $i1 \ vc \ st$
shows $\text{ack } (\text{Suc } t) = [\text{connection-ok}]$
 $\langle \text{proof} \rangle$

lemma *Gateway-L7a*:

assumes $gw: \text{Gateway } \text{req } dt \ a \ \text{stop } \text{lose } d \ \text{ack } i \ vc$
and $a\text{Msg1}: \text{msg } (\text{Suc } 0) \ a$
and $stop\text{Msg1}: \text{msg } (\text{Suc } 0) \ \text{stop}$
and $\text{reqMsg1}: \text{msg } (\text{Suc } 0) \ \text{req}$
and $ts\text{Lose}: ts \ \text{lose}$
and $\text{loseFalse}: \forall j \leq d + 2. \text{lose } (t + j) = [\text{False}]$
and $\text{nsend}: \text{req } (\text{Suc } t) \neq [\text{send}]$
and $\text{ackNCon}: \text{ack } (t) = [\text{connection-ok}]$
shows $\text{ack } (\text{Suc } t) = [\text{connection-ok}]$
 $\langle \text{proof} \rangle$

lemma *Sample-L-buffer*:

assumes
 $\text{Sample-L } \text{req } dt \ a1 \ \text{stop } \text{lose } (\text{fin-inf-append } [\text{init-state}] \ st)$
 $(\text{fin-inf-append } [[]] \ \text{buffer})$
 $\text{ack } i1 \ vc \ st \ \text{buffer}$
shows $\text{buffer } t = \text{inf-last-ti } dt \ t$
 $\langle \text{proof} \rangle$

lemma *tiTable-SampleT-i1-buffer*:

assumes $\text{ack } t = [\text{connection-ok}]$
and $\text{reqSend}: \text{req } (\text{Suc } t) = [\text{send}]$
and $\text{loseFalse}: \forall k \leq \text{Suc } d. \text{lose } (t + k) = [\text{False}]$
and $\text{buf}: \text{buffer } t = \text{inf-last-ti } dt \ t$
and $tbl: \text{tiTable-SampleT } \text{req } a1 \ \text{stop } \text{lose } (\text{fin-inf-append } [\text{init-state}] \ st)$
 $(\text{fin-inf-append } [[]] \ \text{buffer}) \ \text{ack}$
 $i1 \ vc \ st$
and $\text{conOk}: \text{fin-inf-append } [\text{init-state}] \ st \ (\text{Suc } t) = \text{connection-ok}$
shows $i1 \ (\text{Suc } t) = \text{inf-last-ti } dt \ t$

<proof>

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: $\forall k \leq \text{Suc } d. \text{lose } (t + k) = [\text{False}]$*
and *smp1: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>

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: $\forall j \leq 2 * d. \text{lose } (t + j) = [\text{False}]$*

and *a1e: $\forall t_4 \leq t + d + d. a1 t_4 = []$*

and *snd:fin-inf-append [init-state] st (Suc (t + x)) = sending-data*

and *h6:Suc (t + x) $\leq 2 * d + t$*

shows *ack (Suc (t + x)) = [sending-data]*

<proof>

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: $\forall j \leq 2 * d. \text{lose } (t + j) = [\text{False}]$*

and *ackSnd:ack t = [sending-data]*

and *smp1:Sample req dt a1 stop lose ack i1 vc*

and *xdd:x $\leq d + d$*

and *h9: $\forall t_4 \leq t + d + d. a1 t_4 = []$*

shows *ack (t + x) = [sending-data]*

<proof>

15.6 Properties of the ServiceCenter component

lemma *ServiceCenter-a-l:*

assumes *ServiceCenter i a*

shows *length (a t) $\leq (\text{Suc } 0)$*

<proof>

lemma *ServiceCenter-a-msg:*

assumes *ServiceCenter i a*
shows $\text{msg } (\text{Suc } 0) a$
 $\langle \text{proof} \rangle$

lemma *ServiceCenter-L1*:
assumes $\forall t2 < x. i t2 = []$
and *ServiceCenter i a*
and $t \leq x$
shows $a t = []$
 $\langle \text{proof} \rangle$

lemma *ServiceCenter-L2*:
assumes $\forall t2 < x. i t2 = []$
and *ServiceCenter i a*
shows $\forall t3 \leq x. a t3 = []$
 $\langle \text{proof} \rangle$

15.7 General properties of stream values

lemma *streamValue1*:
assumes $h1: \forall j \leq D + (z::\text{nat}). \text{str } (t + j) = x$
and $h2: j \leq D$
shows $\text{str } (t + j + z) = x$
 $\langle \text{proof} \rangle$

lemma *streamValue2*:
assumes $\forall j \leq D + (z::\text{nat}). \text{str } (t + j) = x$
shows $\forall j \leq D. \text{str } (t + j + z) = x$
 $\langle \text{proof} \rangle$

lemma *streamValue3*:
assumes $\forall j \leq D. \text{str } (t + j + (\text{Suc } y)) = x$
and $j \leq D$
and $h3: \text{str } (t + y) = x$
shows $\text{str } (t + j + y) = x$
 $\langle \text{proof} \rangle$

lemma *streamValue4*:
assumes $\forall j \leq D. \text{str } (t + j + (\text{Suc } y)) = x$
and $\text{str } (t + y) = x$
shows $\forall j \leq D. \text{str } (t + j + y) = x$
 $\langle \text{proof} \rangle$

lemma *streamValue5*:
assumes $\forall j \leq D. \text{str } (t + j + ((i::\text{nat}) + k)) = x$
and $j \leq D$
shows $\text{str } (t + i + k + j) = x$
 $\langle \text{proof} \rangle$

lemma *streamValue6*:

assumes $\forall j \leq D. \text{str } (t + j + ((i::\text{nat}) + k)) = x$
shows $\forall j \leq D. \text{str } (t + (i::\text{nat}) + k + j) = x$
<proof>

lemma *streamValue7*:

assumes $h1: \forall j \leq d. \text{str } (t + i + k + d + \text{Suc } j) = x$
and $h2: \text{str } (t + i + k + d) = x$
and $h3: j \leq \text{Suc } d$
shows $\text{str } (t + i + k + d + j) = x$
<proof>

lemma *streamValue8*:

assumes $\forall j \leq d. \text{str } (t + i + k + d + \text{Suc } j) = x$
and $\text{str } (t + i + k + d) = x$
shows $\forall j \leq \text{Suc } d. \text{str } (t + i + k + d + j) = x$
<proof>

lemma *arith-streamValue9aux*:

$\text{Suc } (t + (j + d) + (i + k)) = \text{Suc } (t + i + k + d + j)$
<proof>

lemma *streamValue9*:

assumes $h1: \forall j \leq 2 * d. \text{str } (t + j + \text{Suc } (i + k)) = x$
and $h2: j \leq d$
shows $\text{str } (t + i + k + d + \text{Suc } j) = x$
<proof>

lemma *streamValue10*:

assumes $\forall j \leq 2 * d. \text{str } (t + j + \text{Suc } (i + k)) = x$
shows $\forall j \leq d. \text{str } (t + i + k + d + \text{Suc } j) = x$
<proof>

lemma *arith-sum1*: $(t::\text{nat}) + (i + k + d) = t + i + k + d$

<proof>

lemma *arith-sum2*: $\text{Suc } (\text{Suc } (t + k + j)) = \text{Suc } (\text{Suc } (t + (k + j)))$

<proof>

lemma *arith-sum4*: $t + 3 + k + d = \text{Suc } (t + (2::\text{nat}) + k + d)$

<proof>

lemma *streamValue11*:

assumes $h1: \forall j \leq 2 * d + (4 + k). \text{lose } (t + j) = x$
and $h2: j \leq \text{Suc } d$
shows $\text{lose } (t + 2 + k + j) = x$
<proof>

lemma *streamValue12*:

assumes $\forall j \leq 2 * d + (4 + k). \text{lose } (t + j) = x$
shows $\forall j \leq \text{Suc } d. \text{lose } (t + 2 + k + j) = x$
 <proof>

lemma *streamValue43*:
assumes $\forall j \leq 2 * d + ((4::\text{nat}) + k). \text{lose } (t + j) = [\text{False}]$
shows $\forall j \leq 2 * d. \text{lose } ((t + (3::\text{nat}) + k) + j) = [\text{False}]$
 <proof>

end

theory *Gateway-proof*
imports *Gateway-proof-aux*
begin

15.8 Properties of the Gateway

lemma *Gateway-L1*:
assumes $h1:\text{Gateway req dt a stop lose d ack i vc}$
and $h2:\text{msg } (\text{Suc } 0) \text{ req}$
and $h3:\text{msg } (\text{Suc } 0) \text{ a}$
and $h4:\text{msg } (\text{Suc } 0) \text{ stop}$
and $h5:\text{ts lose}$
and $h6:\text{ack } t = [\text{init-state}]$
and $h7:\text{req } (\text{Suc } t) = [\text{init}]$
and $h8:\text{lose } (\text{Suc } t) = [\text{False}]$
and $h9:\text{lose } (\text{Suc } (\text{Suc } t)) = [\text{False}]$
shows $\text{ack } (\text{Suc } (\text{Suc } t)) = [\text{connection-ok}]$
 <proof>

lemma *Gateway-L2*:
assumes $h1:\text{Gateway req dt a stop lose d ack i vc}$
and $h2:\text{msg } (\text{Suc } 0) \text{ req}$
and $h3:\text{msg } (\text{Suc } 0) \text{ a}$
and $h4:\text{msg } (\text{Suc } 0) \text{ stop}$
and $h5:\text{ts lose}$
and $h6:\text{ack } t = [\text{connection-ok}]$
and $h7:\text{req } (\text{Suc } t) = [\text{send}]$
and $h8:\forall k \leq \text{Suc } d. \text{lose } (t + k) = [\text{False}]$
shows $i (\text{Suc } (t + d)) = \text{inf-last-ti dt t}$
 <proof>

lemma *Gateway-L3*:
assumes $h1:\text{Gateway req dt a stop lose d ack i vc}$
and $h2:\text{msg } (\text{Suc } 0) \text{ req}$
and $h3:\text{msg } (\text{Suc } 0) \text{ a}$
and $h4:\text{msg } (\text{Suc } 0) \text{ stop}$

and $h5:ts\ lose$
and $h6:ack\ t = [connection-ok]$
and $h7:req\ (Suc\ t) = [send]$
and $h8:\forall k \leq Suc\ d. lose\ (t + k) = [False]$
shows $ack\ (Suc\ t) = [sending-data]$
 $\langle proof \rangle$

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 $h8:\forall k \leq Suc\ d. lose\ (t + k) = [False]$
shows $vc\ (Suc\ (t + d)) = [vc-com]$
 $\langle proof \rangle$

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 $h7:\forall j \leq Suc\ d. a\ (t+j) = []$
and $h8:\forall k \leq (d + d). lose\ (t + k) = [False]$
shows $j \leq d \longrightarrow ack\ (t+d+j) = [sending-data]$
 $\langle proof \rangle$

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 $h5:\forall j \leq k. lose\ (t + j) = [False]$
and $h6:\forall m \leq k. req\ (t + m) \neq [send]$
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 $h11:m \leq k$
shows $ack\ (t + m) = [connection-ok]$
 $\langle proof \rangle$

lemma *Gateway-L6:*

assumes $Gateway\ req\ dt\ a\ stop\ lose\ d\ ack\ i\ vc$
and $\forall m \leq k. req\ (t + m) \neq [send]$

and $\forall j \leq k. \text{lose}(t + j) = [\text{False}]$
and $\text{ack } t = [\text{connection-ok}]$
and $\text{msg}(\text{Suc } 0) \text{ req}$
and $\text{msg}(\text{Suc } 0) \text{ stop}$
and $\text{msg}(\text{Suc } 0) a$
and $ts \text{ lose}$
shows $\forall m \leq k. \text{ack}(t + m) = [\text{connection-ok}]$
 $\langle \text{proof} \rangle$

lemma *Gateway-L6a:*

assumes *Gateway req dt a stop lose d ack i vc*
and $\forall m \leq k. \text{req}(t + 2 + m) \neq [\text{send}]$
and $\forall j \leq k. \text{lose}(t + 2 + j) = [\text{False}]$
and $\text{ack}(t + 2) = [\text{connection-ok}]$
and $\text{msg}(\text{Suc } 0) \text{ req}$
and $\text{msg}(\text{Suc } 0) \text{ stop}$
and $\text{msg}(\text{Suc } 0) a$
and $ts \text{ lose}$
shows $\forall m \leq k. \text{ack}(t + 2 + m) = [\text{connection-ok}]$
 $\langle \text{proof} \rangle$

lemma *aux-k3req:*

assumes $h1: \forall m < k + 3. \text{req}(t + m) \neq [\text{send}]$
and $h2: m \leq k$
shows $\text{req}(\text{Suc}(\text{Suc}(t + m))) \neq [\text{send}]$
 $\langle \text{proof} \rangle$

lemma *aux3lose:*

assumes $h1: \forall j \leq k + d + 3. \text{lose}(t + j) = [\text{False}]$
and $h2: j \leq k$
shows $\text{lose}(\text{Suc}(\text{Suc}(t + j))) = [\text{False}]$
 $\langle \text{proof} \rangle$

lemma *Gateway-L7:*

assumes $h1: \text{Gateway req dt a stop lose d ack i vc}$
and $h2: ts \text{ lose}$
and $h3: \text{msg}(\text{Suc } 0) a$
and $h4: \text{msg}(\text{Suc } 0) \text{ stop}$
and $h5: \text{msg}(\text{Suc } 0) \text{ req}$
and $h6: \text{req}(\text{Suc } t) = [\text{init}]$
and $h7: \forall m < (k + 3). \text{req}(t + m) \neq [\text{send}]$
and $h8: \text{req}(t + 3 + k) = [\text{send}]$
and $h9: \text{ack } t = [\text{init-state}]$
and $h10: \forall j \leq k + d + 3. \text{lose}(t + j) = [\text{False}]$
and $h11: \forall t1 \leq t. \text{req } t1 = []$
shows $\forall t2 < (t + 3 + k + d). i \text{ } t2 = []$
 $\langle \text{proof} \rangle$

lemma *Gateway-L8a*:
assumes $h1: \text{Gateway req dt a stop lose d ack i vc}$
and $h2: \text{msg (Suc 0) req}$
and $h3: \text{msg (Suc 0) stop}$
and $h4: \text{msg (Suc 0) a}$
and $h5: \text{ts lose}$
and $h6: \forall j \leq 2 * d. \text{lose (t + j) = [False]}$
and $h7: \text{ack t = [sending-data]}$
and $h8: \forall t3 \leq t + d. a t3 = []$
and $h9: x \leq d + d$
shows $\text{ack (t + x) = [sending-data]}$
 $\langle \text{proof} \rangle$

lemma *Gateway-L8*:
assumes $\text{Gateway req dt a stop lose d ack i vc}$
and msg (Suc 0) req
and msg (Suc 0) stop
and msg (Suc 0) a
and ts lose
and $\forall j \leq 2 * d. \text{lose (t + j) = [False]}$
and $\text{ack t = [sending-data]}$
and $\forall t3 \leq t + d. a t3 = []$
shows $\forall x \leq d + d. \text{ack (t + x) = [sending-data]}$
 $\langle \text{proof} \rangle$

15.9 Proof of the Refinement Relation for the Gateway Requirements

lemma *Gateway-L0*:
assumes $\text{Gateway req dt a stop lose d ack i vc}$
shows $\text{GatewayReq req dt a stop lose d ack i vc}$
 $\langle \text{proof} \rangle$

15.10 Lemmas about Gateway Requirements

lemma *GatewayReq-L1*:
assumes $h1: \text{msg (Suc 0) req}$
and $h2: \text{msg (Suc 0) stop}$
and $h3: \text{msg (Suc 0) a}$
and $h4: \text{ts lose}$
and $h6: \text{req (t + 3 + k) = [send]}$
and $h7: \forall j \leq 2 * d + (4 + k). \text{lose (t + j) = [False]}$
and $h9: \forall m \leq k. \text{ack (t + 2 + m) = [connection-ok]}$
and $h10: \text{GatewayReq req dt a stop lose d ack i vc}$
shows $\text{ack (t + 3 + k) = [sending-data]}$
 $\langle \text{proof} \rangle$

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 $h9:\forall m \leq k. ack\ (t + 2 + m) = [connection-ok]$
shows $i\ (t + 3 + k + d) \neq []$
 $\langle proof \rangle$

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 $msg (Suc\ 0)\ req \wedge msg (Suc\ 0)\ a \wedge msg (Suc\ 0)\ stop \wedge ts\ lose \longrightarrow$
 $(\forall t. (ack\ t = [init-state] \wedge$
 $req\ (Suc\ t) = [init] \wedge lose\ (Suc\ t) = [False] \wedge$
 $lose\ (Suc\ (Suc\ t)) = [False] \longrightarrow$
 $ack\ (Suc\ (Suc\ t)) = [connection-ok]) \wedge$
 $(ack\ t = [connection-ok] \wedge req\ (Suc\ t) = [send] \wedge$
 $(\forall k \leq Suc\ d. lose\ (t + k) = [False]) \longrightarrow$
 $i\ (Suc\ (t + d)) = inf-last-ti\ dt\ t \wedge ack\ (Suc\ t) = [sending-data]) \wedge$
 $(ack\ (t + d) = [sending-data] \wedge a\ (Suc\ t) = [sc-ack] \wedge$
 $(\forall k \leq Suc\ d. lose\ (t + k) = [False]) \longrightarrow$
 $vc\ (Suc\ (t + d)) = [vc-com]))$
shows $ack\ (t + 3 + k + d + d) = [sending-data] \wedge$
 $a\ (Suc\ (t + 3 + k + d)) = [sc-ack] \wedge$
 $(\forall ka \leq Suc\ d. lose\ (t + 3 + k + d + ka) = [False]) \longrightarrow$
 $vc\ (Suc\ (t + 3 + k + d + d)) = [vc-com]$
 $\langle proof \rangle$

lemma *GatewaySystem-L3aux:*

assumes $msg (Suc\ 0)\ req$
and $msg (Suc\ 0)\ stop$
and $msg (Suc\ 0)\ a$
and $ts\ lose$
and $msg (Suc\ 0)\ req \wedge msg (Suc\ 0)\ a \wedge msg (Suc\ 0)\ stop \wedge ts\ lose \longrightarrow$
 $(\forall t. (ack\ t = [init-state] \wedge$
 $req\ (Suc\ t) = [init] \wedge lose\ (Suc\ t) = [False] \wedge$
 $lose\ (Suc\ (Suc\ t)) = [False] \longrightarrow$
 $ack\ (Suc\ (Suc\ t)) = [connection-ok]) \wedge$
 $(ack\ t = [connection-ok] \wedge req\ (Suc\ t) = [send] \wedge$
 $(\forall k \leq Suc\ d. lose\ (t + k) = [False]) \longrightarrow$
 $i\ (Suc\ (t + d)) = inf-last-ti\ dt\ t \wedge ack\ (Suc\ t) = [sending-data]) \wedge$

$(ack (t + d) = [sending-data] \wedge a (Suc t) = [sc-ack] \wedge$
 $(\forall k \leq Suc d. lose (t + k) = [False]) \longrightarrow$
 $vc (Suc (t + d)) = [vc-com])$)
shows $ack (t + 2 + k) = [connection-ok] \wedge$
 $req (Suc (t + 2 + k)) = [send] \wedge$
 $(\forall j \leq Suc d. lose (t + 2 + k + j) = [False]) \longrightarrow$
 $i (Suc (t + 2 + k + d)) = inf-last-ti dt (t + 2 + k)$
 $\langle proof \rangle$

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 $h9:\forall j \leq 2 * d + (4 + k). lose (t + j) = [False]$
and $h11:i (t + 3 + k + d) \neq []$
and $h14:\forall x \leq d + d. ack (t + 3 + k + x) = [sending-data]$
shows $vc (2 * d + (t + (4 + k))) = [vc-com]$
 $\langle proof \rangle$

lemma aux4lose1:
assumes $h1:\forall j \leq 2 * d + (4 + k). lose (t + j) = [False]$
and $h2:j \leq k$
shows $lose (t + (2::nat) + j) = [False]$
 $\langle proof \rangle$

lemma aux4lose2:
assumes $\forall j \leq 2 * d + (4 + k). lose (t + j) = [False]$
and $3 + k + d \leq 2 * d + (4 + k)$
shows $lose (t + (3::nat) + k + d) = [False]$
 $\langle proof \rangle$

lemma aux4req:
assumes $h1:\forall (m::nat) \leq k + 2. req (t + m) \neq [send]$
and $h2:m \leq k$
and $h3:req (t + 2 + m) = [send]$ **shows** $False$
 $\langle proof \rangle$

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 $h10: \forall t1 \leq t. req\ t1 = []$
and $h11: \forall m \leq k + 2. req\ (t + m) \neq [send]$
and $h12: req\ (t + 3 + k) = [send]$
and $h13: inf-last-ti\ dt\ t \neq []$
and $h14: \forall j \leq 2 * d + (4 + k). lose\ (t + j) = [False]$
shows $vc\ (2 * d + (t + (4 + k))) = [vc-com]$
 $\langle proof \rangle$

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 $h8: dt\ (Suc\ t) \neq [] \vee dt\ (Suc\ (Suc\ t)) \neq []$
and $h9: ack\ t = [init-state]$
and $h10: req\ (Suc\ t) = [init]$
and $h11: \forall t1 \leq t. req\ t1 = []$
and $h12: \forall m \leq k + 2. req\ (t + m) \neq [send]$
and $h13: req\ (t + 3 + k) = [send]$
and $h14: \forall j \leq 2 * d + (4 + k). lose\ (t + j) = [False]$
shows $vc\ (2 * d + (t + (4 + k))) = [vc-com]$
 $\langle proof \rangle$

15.12 Proof of the Refinement for the Gateway System

lemma *GatewaySystem-L0*:
assumes $GatewaySystem\ req\ dt\ stop\ lose\ d\ ack\ vc$
shows $GatewaySystemReq\ req\ dt\ stop\ lose\ d\ ack\ vc$
 $\langle proof \rangle$

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.