Stream processing components: Isabelle/HOL formalisation and case studies

Maria Spichkova

March 17, 2025

Abstract

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

Contents

1 Introduction

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

1.1 Stream processing components

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

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

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

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

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 $(I_S \rhd O_S)$.

To specify the behaviour of a real-time system we use *infinite timed streams* to represent the input and the output streams. The type of *finite timed streams* will be used only if some argumentation about a timed stream that was truncated at some point of time is needed. The type of *finite*

untimed streams will be used to argue about a sequence of messages that are transmitted during a time interval. The type of *infinite untimed streams* will be used in the case of timed specifications only to represent local variables of FOCUS specification. Our definition in Isabelle/HOL of corresponding types is given below:

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

1.2 Case Study 1: Steam Boiler System

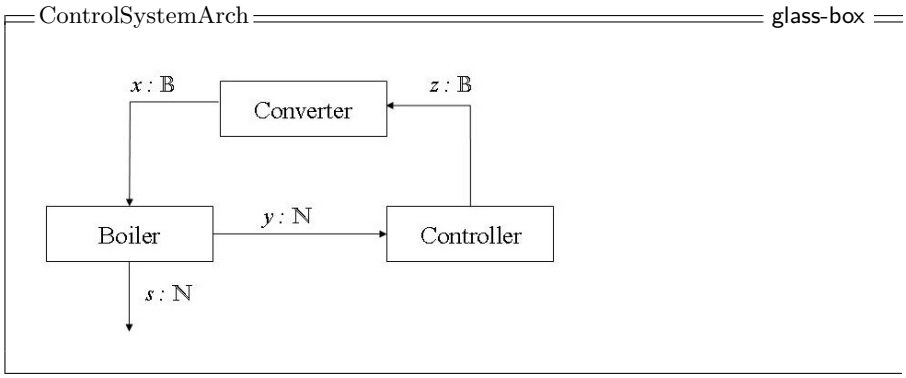
A steam boiler control system can be represent as a distributed system consisting of a number of communicating components and must fulfil real time requirements. This case study shows how we can deal with local variables (system's states) and in which way we can represent mutually recursive functions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

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

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

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

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



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

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

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

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

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

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

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

1.3 Case Study 2: FlexRay Communication Protocol

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

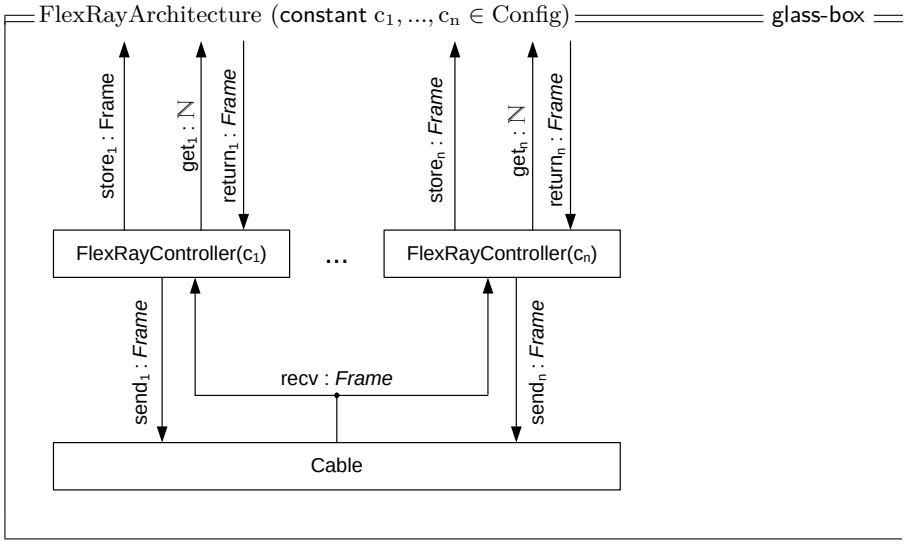
asm $\forall\, i \in [1..n] : \mathsf{msg}_1(return_i)$

$DisjointSchedules(c_1, \ldots, c_n)$

$IdenticCycleLength(c_1, \ldots, c_n)$

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

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



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

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

communication cycle to some FlexRay frame identifier (that corresponds to the number of the slot in the communication round) from the scheduler table, this frame identifier.

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

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

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

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

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

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

1.4 Case Study 3: Automotive-Gateway

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

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

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

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

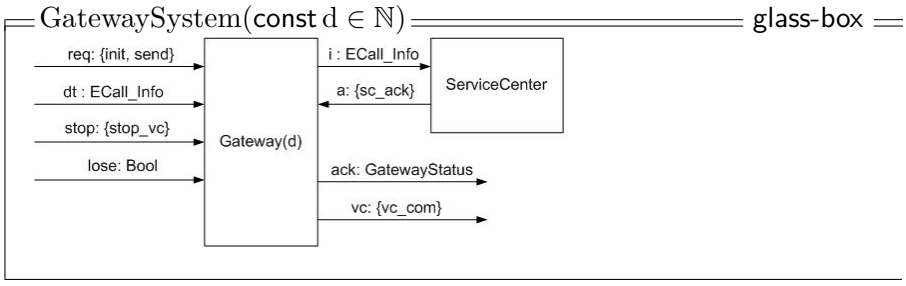
- *Gateway_types.thy* – datatype definitions,
- *Gateway.thy* – specifications of the system components,
- *Gateway_proof* – proofs of refinement relations between the requirements and the architecture specifications (for the components *Gateway* and *GatewaySystem*).

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

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

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

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



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.

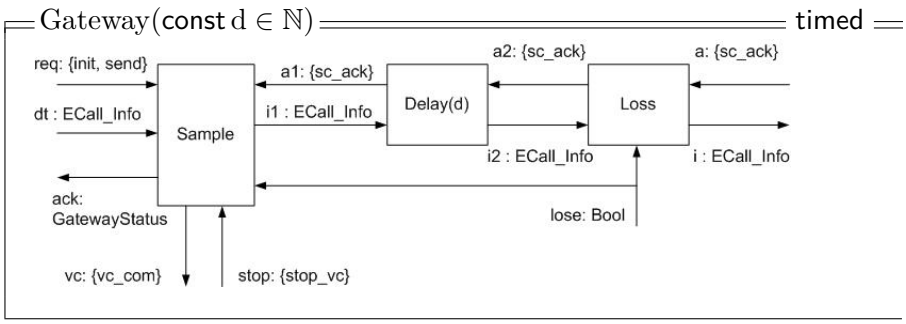
We specify the gateway requirements (*GatewayReq*) as follows:

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 lemma:

lemma *Gateway-L0*:

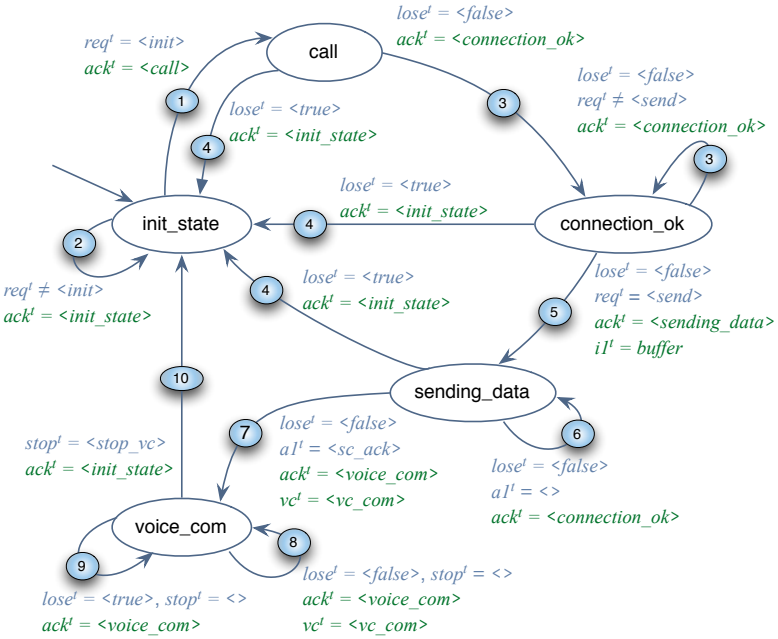


Figure 1: Timed state transition diagram for the component Sample

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

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

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

2 Theory ArithExtras.thy

theory *ArithExtras*
imports *Main*
begin

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

end

3 Auxiliary Theory ListExtras.thy

theory *ListExtras*
imports *Main*
begin

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

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

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

lemma *mem-memS-eq*: *x mem l* ≡ *memS x l*
⟨*proof*⟩

lemma *mem-set-1*:
assumes *a mem l*
shows *a* ∈ *set l*
⟨*proof*⟩

lemma *mem-set-2*:
assumes *a* ∈ *set l*
shows *a mem l*

⟨*proof*⟩

lemma *set-inter-mem*:
assumes *x mem l1*
and *x mem l2*
shows *set l1 ∩ set l2 ≠ {}*
⟨*proof*⟩

lemma *mem-notdisjoint*:
assumes *x mem l1*
and *x mem l2*
shows ¬ *disjoint l1 l2*
⟨*proof*⟩

lemma *mem-notdisjoint2*:
assumes *h1*:*disjoint* (*schedule A*) (*schedule B*)
and *h2*:*x mem schedule A*
shows ¬ *x mem schedule B*
⟨*proof*⟩

lemma *Add-Less*:
assumes *0 < b*
shows (*Suc a − b < Suc a*) = *True*
⟨*proof*⟩

lemma *list-length-hint1*:
assumes *l ≠ []*
shows *0 < length l*
⟨*proof*⟩

lemma *list-length-hint1a*:
assumes *l ≠ []*
shows *0 < length l*
⟨*proof*⟩

lemma *list-length-hint2*:
assumes *length x = Suc 0*
shows [*hd x*] = *x*
⟨*proof*⟩

lemma *list-length-hint2a*:
assumes *length l = Suc 0*
shows *tl l = []*
⟨*proof*⟩

lemma *list-length-hint3*:
assumes *length l = Suc 0*
shows *l ≠ []*
⟨*proof*⟩

lemma *list-length-hint4*:
assumes *length x* ≤ *Suc 0*
and *x* ≠ []
shows *length x* = *Suc 0*
⟨*proof*⟩

lemma *length-nonempty*:
assumes *x* ≠ []
shows *Suc 0* ≤ *length x*
⟨*proof*⟩

lemma *last-nth-length*:
assumes *x* ≠ []
shows *x* ! ((*length x*) − *Suc 0*) = *last x*
⟨*proof*⟩

lemma *list-nth-append0*:
assumes *i* < *length x*
shows *x* ! *i* = (*x* • *z*) ! *i*
⟨*proof*⟩

lemma *list-nth-append1*:
assumes *i* < *length x*
shows (*b* # *x*) ! *i* = (*b* # *x* • *y*) ! *i*
⟨*proof*⟩

lemma *list-nth-append2*:
assumes *i* < *Suc* (*length x*)
shows (*b* # *x*) ! *i* = (*b* # *x* • *a* # *y*) ! *i*
⟨*proof*⟩

lemma *list-nth-append3*:
assumes *h1*:¬ *i* < *Suc* (*length x*)
and *i* − *Suc* (*length x*) < *Suc* (*length y*)
shows (*a* # *y*) ! (*i* − *Suc* (*length x*)) = (*b* # *x* • *a* # *y*) ! *i*
⟨*proof*⟩

lemma *list-nth-append4*:
assumes *i* < *Suc* (*length x* + *length y*)
and ¬ *i* − *Suc* (*length x*) < *Suc* (*length y*)
shows *False*
⟨*proof*⟩

lemma *list-nth-append5*:
assumes *i* − *length x* < *Suc* (*length y*)
and ¬ *i* − *Suc* (*length x*) < *Suc* (*length y*)
shows ¬ *i* < *Suc* (*length x* + *length y*)
⟨*proof*⟩

lemma *list-nth-append6*:
assumes $\neg\ i - length\ x < Suc\ (length\ y)$
and $\neg\ i - Suc\ (length\ x) < Suc\ (length\ y)$
shows $\neg\ i < Suc\ (length\ x + length\ y)$
⟨*proof*⟩

lemma *list-nth-append6a*:
assumes $i < Suc\ (length\ x + length\ y)$
and $\neg\ i - length\ x < Suc\ (length\ y)$
shows *False*
⟨*proof*⟩

lemma *list-nth-append7*:
assumes $i - length\ x < Suc\ (length\ y)$
and $i - Suc\ (length\ x) < Suc\ (length\ y)$
shows $i < Suc\ (Suc\ (length\ x + length\ y))$
⟨*proof*⟩

lemma *list-nth-append8*:
assumes $\neg\ i < Suc\ (length\ x + length\ y)$
and $i < Suc\ (Suc\ (length\ x + length\ y))$
shows $i = Suc\ (length\ x + length\ y)$
⟨*proof*⟩

lemma *list-nth-append9*:
assumes $i - Suc\ (length\ x) < Suc\ (length\ y)$
shows $i < Suc\ (Suc\ (length\ x + length\ y))$
⟨*proof*⟩

lemma *list-nth-append10*:
assumes $\neg\ i < Suc\ (length\ x)$
and $\neg\ i - Suc\ (length\ x) < Suc\ (length\ y)$
shows $\neg\ i < Suc\ (Suc\ (length\ x + length\ y))$
⟨*proof*⟩

end

4 Auxiliary arithmetic lemmas

theory *arith-hints*
imports *Main*
begin

lemma *arith-mod-neq*:
assumes $a\ mod\ n \neq b\ mod\ n$
shows $a \neq b$
⟨*proof*⟩

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

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

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

lemma *arith-mult-neq-nzero3*:
fixes i::nat
assumes $i < n$
and $0 < i$
shows $n + n * t + i \neq n * qc$
⟨*proof*⟩

lemma *arith-modZero1*:
$(t + n * t)\ mod\ Suc\ n = 0$
⟨*proof*⟩

lemma *arith-modZero2*:
$Suc\ (n + (t + n * t))\ mod\ Suc\ n = 0$
⟨*proof*⟩

lemma *arith1*:
assumes *h1*:$Suc\ n * t = Suc\ n * q$
shows $t = q$
⟨*proof*⟩

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

end

5 FOCUS streams: operators and lemmas

theory *stream*
imports *ListExtras ArithExtras*
begin

5.1 Definition of the FOCUS stream types

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

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

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

— FOCUS stream (general)
datatype *'a stream =*
FinT 'a fstream — finite timed streams
| *FinU 'a list* — finite untimed streams
| *InfT 'a istream* — infinite timed streams
| *InfU 'a iustream* — infinite untimed streams

5.2 Definitions of operators

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

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

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

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

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

— range of an infinite timed stream
definition
infT-range :: *'a istream* ⇒ *'a set*
where
infT-range s ≡ {*y*. ∃ *i*::*nat*. *y mem* (*s i*)}

— range of a finite untimed stream
definition
infU-range :: (*nat* ⇒ *'a*) ⇒ *'a set*
where
infU-range s ≡ { *y*. ∃ *i*::*nat*. *y* = (*s i*) }

— range of a (general) stream
definition
stream-range :: *'a stream* ⇒ *'a set*
where
stream-range s ≡ *case s of*
FinT x ⇒ *finT-range x*
| *FinU x* ⇒ *finU-range x*
| *InfT x* ⇒ *infT-range x*
| *InfU x* ⇒ *infU-range x*

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

— removing the first element from an infinite stream
— in the case of an untimed stream: removing the first data element
— in the case of a timed stream: removing the first time interval
definition
inf-tl :: (*nat* ⇒ *'a*) ⇒ (*nat* ⇒ *'a*)
where
inf-tl s ≡ (λ *i*. *s* (*Suc i*))

— removing i first elements from an infinite stream s
— in the case of an untimed stream: removing i first data elements
— in the case of a timed stream: removing i first time intervals
definition
inf-drop :: *nat* ⇒ (*nat* ⇒ *'a*) ⇒ (*nat* ⇒ *'a*)
where

```
  inf-drop i s ≡  λ j. s (i+j)

— finding the first nonempty time interval in a finite timed stream
primrec
 fin-find1nonemp :: 'a fstream ⇒ 'a list
where
 fin-find1nonemp [] = [] |
 fin-find1nonemp (x#xs) =
    ( if x = []
      then fin-find1nonemp xs
      else x )

— finding the first nonempty time interval in an infinite timed stream
definition
  inf-find1nonemp :: 'a istream ⇒ 'a list
where
 inf-find1nonemp s
  ≡
  ( if (∃ i. s i ≠ [])
    then s (LEAST i. s i ≠ [])
    else [] )

— finding the index of the first nonempty time interval in a finite timed stream
primrec
 fin-find1nonemp-index :: 'a fstream ⇒ nat
where
 fin-find1nonemp-index [] = 0 |
 fin-find1nonemp-index (x#xs) =
    ( if x = []
      then Suc (fin-find1nonemp-index xs)
      else 0 )

— finding the index of the first nonempty time interval in an infinite timed stream
definition
  inf-find1nonemp-index :: 'a istream ⇒ nat
where
 inf-find1nonemp-index s
  ≡
  ( if (∃ i. s i ≠ [])
    then (LEAST i. s i ≠ [])
    else 0 )

— length of a finite timed stream: number of data elements in this stream
primrec
  fin-length :: 'a fstream ⇒ nat
where
  fin-length [] = 0 |
  fin-length (x#xs) = (length x) + (fin-length xs)
```

— length of a (general) stream
definition
stream-length :: *'a stream* ⇒ *natInf*
where
stream-length s ≡
case s of
(*FinT x*) ⇒ *Fin* (*fin-length x*)
| (*FinU x*) ⇒ *Fin* (*length x*)
| (*InfT x*) ⇒ ∞
| (*InfU x*) ⇒ ∞

— removing the first k elements from a finite (nonempty) timed stream
axiomatization
fin-nth :: *'a fstream* ⇒ *nat* ⇒ *'a*
where
fin-nth-Cons:
fin-nth (*hds* # *tls*) *k* =
(*if hds* = []
then fin-nth tls k
else (*if* (*k* < (*length hds*))
then nth hds k
else fin-nth tls (*k* − *length hds*)))

— removing i first data elements from an infinite timed stream s
primrec
inf-nth :: *'a istream* ⇒ *nat* ⇒ *'a*
where
inf-nth s 0 = *hd* (*s* (*LEAST i.*(*s i*) ≠ [])) |
inf-nth s (*Suc k*) =
(*if* ((*Suc k*) < (*length* (*s 0*)))
then (*nth* (*s 0*) (*Suc k*))
else (*if* (*s 0*) = []
then (*inf-nth* (*inf-tl* (*inf-drop*
(*LEAST i.* (*s i*) ≠ []) *s*)) *k*)
else inf-nth (*inf-tl s*) *k*))

— removing the first k data elements from a (general) stream
definition
stream-nth :: *'a stream* ⇒ *nat* ⇒ *'a*
where
stream-nth s k ≡
case s of (*FinT x*) ⇒ *fin-nth x k*
| (*FinU x*) ⇒ *nth x k*
| (*InfT x*) ⇒ *inf-nth x k*
| (*InfU x*) ⇒ *x k*

— prefix of an infinite stream
primrec
inf-prefix :: *'a list* ⇒ (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *bool*

```
where
  inf-prefix [] s k = True |
  inf-prefix (x#xs) s k = ( (x = (s k)) ∧ (inf-prefix xs s (Suc k)) )

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

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

— truncating a finite stream after the n-th element
primrec
fin-truncate :: 'a list ⇒ nat ⇒ 'a list
where
  fin-truncate [] n = [] |
  fin-truncate (x#xs) i =
      (case i of 0 ⇒ []
        | (Suc n) ⇒ x # (fin-truncate xs n))
```

— truncating a finite stream after the n-th element
— n is of type of natural numbers enriched by Infinity
definition
fin-truncate-plus :: *'a list* ⇒ *natInf* ⇒ *'a list*
where
fin-truncate-plus s n
≡
case n of (*Fin i*) ⇒ *fin-truncate s i*
| ∞ ⇒ *s*

— truncating an infinite stream after the n-th element
primrec
inf-truncate :: (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *'a list*
where
inf-truncate s 0 = [*s 0*] |
inf-truncate s (*Suc k*) = (*inf-truncate s k*) • [*s* (*Suc k*)]

— truncating an infinite stream after the n-th element
— n is of type of natural numbers enriched by Infinity
definition
inf-truncate-plus :: *'a istream* ⇒ *natInf* ⇒ *'a stream*
where
inf-truncate-plus s n
≡
case n of (*Fin i*) ⇒ *FinT* (*inf-truncate s i*)
| ∞ ⇒ *InfT s*

— concatanation of a finite and an infinite stream
definition
fin-inf-append ::
'a list ⇒ (*nat* ⇒ *'a*) ⇒ (*nat* ⇒ *'a*)
where
fin-inf-append us s ≡
(λ *i*. (*if* (*i* < (*length us*))
then (*nth us i*)
else s (*i* − (*length us*))))

— insuring that the infinite timed stream is time-synchronous
definition
ts :: *'a istream* ⇒ *bool*
where
ts s ≡ ∀ *i*. (*length* (*s i*) = *1*)

— insuring that each time interval of an infinite timed stream contains at most n data elements
definition
msg :: *nat* ⇒ *'a istream* ⇒ *bool*
where

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

— 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-make-untimed1-0:
$inf\text{-}make\text{-}untimed1\ s\ 0 =\ hd\ (s\ (LEAST\ i.(s\ i) \neq []))\ \ |$
inf-make-untimed1-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
$make\text{-}untimed\ s \equiv$
$case\ s\ of\ (FinT\ x) \Rightarrow FinU\ (fin\text{-}make\text{-}untimed\ x)$
$|\ (FinU\ x) \Rightarrow FinU\ x$

```
| (InfT x) ⇒
  (if (∃ i.∀ j. i < j ⟶ (x j) = [])
   then FinU (fin-make-untimed (inf-truncate x
                (LEAST i.∀ j. i < j ⟶ (x j) = [])))
   else InfU (inf-make-untimed x))
| (InfU x) ⇒ InfU x
```

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

```
primrec
  fin-tm :: 'a fstream ⇒ nat ⇒ nat
where
  fin-tm [] k = k |
  fin-tm (x#xs) k =
    (if k = 0
     then 0
     else (if (k ≤ length x)
           then (Suc 0)
           else Suc(fin-tm xs (k − length x))))
```

— auxiliary lemma for the definition of the truncate operator

```
lemma inf-tm-hint1:
  assumes i2 = Suc i − length a
      and ¬ Suc i ≤ length a
      and a ≠ []
  shows i2 < Suc i
⟨proof⟩
definition
   finT-filter :: 'a set => 'a fstream => 'a fstream
where
  finT-filter m s ≡  map (λ s. filter (λ y. y ∈ m) s) s
```

— filtering an infinite timed stream

```
definition
   infT-filter :: 'a set => 'a istream => 'a istream
where
  infT-filter m s ≡  (λi.( filter (λ x. x ∈ m) (s i)))
```

— removing duplications from a finite timed stream

```
definition
   finT-remdups :: 'a fstream => 'a fstream
where
  finT-remdups s ≡  map (λ s. remdups s) s
```

— removing duplications from an infinite timed stream

```
definition
   infT-remdups :: 'a istream => 'a istream
where
```

```
  infT-remdups s ≡  (λi.( remdups (s i)))

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

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

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

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

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

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

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

— auxiliary lemmas
lemma length-inf-drop-hint1:
  assumes s k ≠ []
  shows    length (inf-drop k s 0) ≠ 0
⟨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 untimed stream
— (defined over natural numbers)
primrec
  infU-get-prefix :: (nat ⇒ 'a) ⇒ nat ⇒ 'a list
where
  infU-get-prefix s 0 = [] |
  infU-get-prefix s (Suc i)
        = (infU-get-prefix s i) • [s i]

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

— taking the prefix of n data elements from an infinite untimed stream
— (defined over natural numbers enriched by Infinity)
definition
  infU-get-prefix-plus :: (nat ⇒ 'a) ⇒ natInf ⇒ 'a stream
where
 infU-get-prefix-plus s n
  ≡
  case n of (Fin i) ⇒ FinU (infU-get-prefix s i)
          | ∞       ⇒ InfU s

— taking the prefix of n data elements from an infinite stream
— (defined over natural numbers enriched by Infinity)
definition
  take-plus :: natInf ⇒ 'a list ⇒ 'a list
where
 take-plus n s
  ≡
  case n of (Fin i) ⇒ (take i s)
          | ∞       ⇒ s

— taking the prefix of n data elements from a (general) stream
— (defined over natural numbers enriched by Infinity)
definition
   get-prefix :: 'a stream ⇒ natInf ⇒ 'a stream
```

where
get-prefix s k ≡
case s of (*FinT x*) ⇒ *FinT* (*fin-get-prefix-plus x k*)
| (*FinU x*) ⇒ *FinU* (*take-plus k x*)
| (*InfT x*) ⇒ *infT-get-prefix-plus x k*
| (*InfU x*) ⇒ *infU-get-prefix-plus x k*

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

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

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

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

5.3 Properties of operators

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

assumes *inf-last-ti dt t* $\neq$ []
shows *inf-last-ti dt* (*t* + *k*) $\neq$ []
⟨*proof*⟩

lemma *inf-last-ti-nonempty*:
assumes *s t* $\neq$ []
shows *inf-last-ti s* (*t* + *k*) $\neq$ []
⟨*proof*⟩

lemma *arith-sum-t2k*:
t + *2* + *k* = (*Suc t*) + (*Suc k*)
⟨*proof*⟩

lemma *inf-last-ti-Suc2*:
assumes *dt* (*Suc t*) $\neq$ [] $\vee$ *dt* (*Suc* (*Suc t*)) $\neq$ []
shows *inf-last-ti dt* (*t* + *2* + *k*) $\neq$ []
⟨*proof*⟩

5.3.1 Lemmas for concatenation operator

lemma *fin-length-append*:
fin-length (*x*•*y*) = (*fin-length x*) + (*fin-length y*)
⟨*proof*⟩

lemma *fin-append-Nil*: *fin-inf-append* [] *z* = *z*
⟨*proof*⟩

lemma *correct-fin-inf-append1*:
assumes *s1* = *fin-inf-append* [*x*] *s*
shows *s1* (*Suc i*) = *s i*
⟨*proof*⟩

lemma *correct-fin-inf-append2*:
fin-inf-append [*x*] *s* (*Suc i*) = *s i*
⟨*proof*⟩

lemma *fin-append-com-Nil1*:
fin-inf-append [] (*fin-inf-append y z*)
= *fin-inf-append* ([] • *y*) *z*
⟨*proof*⟩

lemma *fin-append-com-Nil2*:
fin-inf-append x (*fin-inf-append* [] *z*)
= *fin-inf-append* (*x* • []) *z*
⟨*proof*⟩

lemma *fin-append-com-i*:
fin-inf-append x (*fin-inf-append y z*) *i* = *fin-inf-append* (*x* • *y*) *z i*
⟨*proof*⟩

5.3.2 Lemmas for operators *ts* and *msg*

lemma *ts-msg1*:
assumes *ts p*
shows *msg 1 p*
⟨*proof*⟩

lemma *ts-inf-tl*:
assumes *ts x*
shows *ts* (*inf-tl x*)
⟨*proof*⟩

lemma *ts-length-hint1*:
assumes *ts x*
shows $x\ i \neq []$
⟨*proof*⟩

lemma *ts-length-hint2*:
assumes *ts x*
shows *length* (*x i*) = *Suc* (*0*::*nat*)
⟨*proof*⟩

lemma *ts-Least-0*:
assumes *ts x*
shows (*LEAST i*. (*x i*) ≠ []) = (*0*::*nat*)
⟨*proof*⟩

lemma *inf-tl-Suc*: *inf-tl x i* = *x* (*Suc i*)
⟨*proof*⟩

lemma *ts-Least-Suc0*:
assumes *ts x*
shows (*LEAST i*. *x* (*Suc i*) ≠ []) = *0*
⟨*proof*⟩

lemma *ts-inf-make-untimed-inf-tl*:
assumes *ts x*
shows *inf-make-untimed* (*inf-tl x*) *i* = *inf-make-untimed x* (*Suc i*)
⟨*proof*⟩

lemma *ts-inf-make-untimed1-inf-tl*:
assumes *ts x*
shows *inf-make-untimed1* (*inf-tl x*) *i* = *inf-make-untimed1 x* (*Suc i*)
⟨*proof*⟩

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

lemma *msg-nonempty2*:
assumes *h1*:*msg* (*Suc 0*) *a*
and *h2*:*a t* ≠ []
shows *length* (*a t*) = (*Suc 0*)
⟨*proof*⟩

5.3.3 Lemmas for *inf_truncate*

lemma *inf-truncate-nonempty*:
assumes *z i* ≠ []
shows *inf-truncate z i* ≠ []
⟨*proof*⟩

lemma *concat-inf-truncate-nonempty*:
assumes *z i* ≠ []
shows *concat* (*inf-truncate z i*) ≠ []
⟨*proof*⟩

lemma *concat-inf-truncate-nonempty-a*:
assumes *z i* = [*a*]
shows *concat* (*inf-truncate z i*) ≠ []
⟨*proof*⟩

lemma *concat-inf-truncate-nonempty-el*:
assumes *z i* ≠ []
shows *concat* (*inf-truncate z i*) ≠ []
⟨*proof*⟩

lemma *inf-truncate-append*:
(*inf-truncate z i* • [*z* (*Suc i*)]) = *inf-truncate z* (*Suc i*)
⟨*proof*⟩

5.3.4 Lemmas for *fin_make_untimed*

lemma *fin-make-untimed-append*:
assumes *fin-make-untimed x* ≠ []
shows *fin-make-untimed* (*x* • *y*) ≠ []
⟨*proof*⟩

lemma *fin-make-untimed-inf-truncate-Nonempty*:
assumes *z k* ≠ []
and *k* ≤ *i*
shows *fin-make-untimed* (*inf-truncate z i*) ≠ []
⟨*proof*⟩

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

⟨*proof*⟩

lemma *last-fin-make-untimed-inf-truncate*:
assumes $z\ i = [a]$
shows $last\ (fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i)) = a$
⟨*proof*⟩

lemma *fin-make-untimed-append-empty*:
$fin\text{-}make\text{-}untimed\ (z \bullet [[]]) = fin\text{-}make\text{-}untimed\ z$
⟨*proof*⟩

lemma *fin-make-untimed-inf-truncate-append-a*:
$fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i \bullet [[a]])\ !$
$(length\ (fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i \bullet [[a]])) - Suc\ 0) = a$
⟨*proof*⟩

lemma *fin-make-untimed-inf-truncate-Nonempty-all*:
assumes $z\ k \neq []$
shows $\forall\ i.\ k \leq i \longrightarrow fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i) \neq []$
⟨*proof*⟩

lemma *fin-make-untimed-inf-truncate-Nonempty-all0*:
assumes $z\ 0 \neq []$
shows $\forall\ i.\ fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i) \neq []$
⟨*proof*⟩

lemma *fin-make-untimed-inf-truncate-Nonempty-all0a*:
assumes $z\ 0 = [a]$
shows $\forall\ i.\ fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i) \neq []$
⟨*proof*⟩

lemma *fin-make-untimed-inf-truncate-Nonempty-all-app*:
assumes $z\ 0 = [a]$
shows $\forall\ i.\ fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i \bullet [z\ (Suc\ i)]) \neq []$
⟨*proof*⟩

lemma *fin-make-untimed-nth-length*:
assumes $z\ i = [a]$
shows
$fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i)\ !$
$(length\ (fin\text{-}make\text{-}untimed\ (inf\text{-}truncate\ z\ i)) - Suc\ 0)$
$= a$
⟨*proof*⟩

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 $(SOME\ i.\ i < n \wedge\ \ nS\ i\ t \neq []) = k$
⟨*proof*⟩

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

end

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

theory *BitBoolTS*
imports *Main stream*
begin

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

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

lemma *ts-bit-stream-One*:
assumes *h1*:*ts x*
and *h2*:$x\ i \neq [Zero]$
shows $x\ i = [One]$
⟨*proof*⟩

lemma *ts-bit-stream-Zero*:
assumes *h1*:*ts x*
and *h2*:$x\ i \neq [One]$
shows $x\ i = [Zero]$
⟨*proof*⟩

lemma *ts-bool-True*:
assumes *h1*:*ts x*
and *h2*:$x\ i \neq [False]$
shows $x\ i = [True]$
⟨*proof*⟩

lemma *ts-bool-False*:

assumes *h1*:*ts x*
and *h2*:*x i* ≠ [*True*]
shows *x i* = [*False*]
⟨*proof*⟩

lemma *ts-bool-True-False*:
fixes *x*::*bool istream*
assumes *ts x*
shows *x i* = [*True*] ∨ *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* ⇒ *nat* ⇒ *nat* ⇒ '*a list*
where
join-ti-0:
join-ti s x 0 = *s x* |
join-ti-Suc:
join-ti s x (*Suc i*) = (*join-ti s x i*) • (*s* (*x* + (*Suc i*)))

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

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

lemma *join-ti-hint1*:
assumes *join-ti s x* (*Suc i*) = []
shows *join-ti s x i* = []
⟨*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 ∀ *i* ≤ *n*. *s* (*x*+*i*) = []
shows *join-ti s x n* = []
⟨*proof*⟩

lemma *join-ti-1nempty*:
assumes ∀ *i*. *0* < *i* ∧ *i* < *Suc n* ⟶ *s* (*x*+*i*) = []
shows *join-ti s x n* = *s x*
⟨*proof*⟩

lemma *join-time1t*: ∀ *t*. *join-time s* (*1*::*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 $\forall\ i \leq n.$ *nth s (x+i)* = []
shows *fin-join-ti s x n* = []
⟨*proof*⟩

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

7.2 Split time units

definition
split-time :: *'a istream* ⇒ *nat* ⇒ *'a istream*
where
split-time s n t ≡
(*if* (*t mod n* = *0*)
then s (*t div n*)
else [])

lemma *split-time1t*: $\forall\ t.$ *split-time s 1 t* = *s t*
⟨*proof*⟩

lemma *split-time1*: *split-time s 1* = *s*
⟨*proof*⟩

lemma *split-time-mod*:
assumes *t mod n* ≠ *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 *split-time s* (*Suc n*) ((*Suc n*) ∗ *t*) = *split-time s n* (*n* ∗ *t*)
⟨*proof*⟩

lemma *split-time-empty*:
assumes $i < n$ **and** *h2*:$0 < i$
shows *split-time s n* (*n* ∗ *t* + *i*) = []
⟨*proof*⟩

lemma *split-time-empty-Suc*:
assumes *h1*:$i < n$
and *h2*:$0 < i$
shows *split-time s* (*Suc n*) ((*Suc n*)∗ *t* + *i*) = *split-time s n* (*n* ∗ *t* + *i*)
⟨*proof*⟩

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

7.3 Duality of the split and the join operators

lemma *join-split-i*:
assumes $0 < n$
shows *join-time* (*split-time s n*) *n i* = *s i*
⟨*proof*⟩

lemma *join-split*:
assumes $0 < n$
shows *join-time* (*split-time s n*) *n* = *s*
⟨*proof*⟩

end

8 Steam Boiler System: Specification

theory *SteamBoiler*
imports *stream BitBoolTS*
begin

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

definition
SteamBoiler :: *bit istream* ⇒ *nat istream* ⇒ *nat istream* ⇒ *bool*

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

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

definition
Controller-L ::
nat istream ⇒ *bit iustream* ⇒ *bit iustream* ⇒ *bit istream* ⇒ *bool*
where
Controller-L y lIn lOut z
≡
(*z 0* = [*Zero*])
∧
(∀ (*t*::*nat*).
(*if* (*lIn t*) = *Zero*
then (*if* *300* < *hd* (*y t*)
then (*z t*) = [] ∧ (*lOut t*) = *Zero*
else (*z t*) = [*One*] ∧ (*lOut t*) = *One*
)
else (*if* *hd* (*y t*) < *700*
then (*z t*) = [] ∧ (*lOut t*) = *One*
else (*z t*) = [*Zero*] ∧ (*lOut t*) = *Zero*)))

definition

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

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

end

9 Steam Boiler System: Verification

theory *SteamBoiler-proof*
imports *SteamBoiler*
begin

9.1 Properties of the Boiler Component

lemma *L1-Boiler*:
assumes *SteamBoiler x s y*
and *ts x*
shows *ts s*
⟨*proof*⟩

lemma *L2-Boiler*:
assumes *SteamBoiler x s y*
and *ts x*
shows *ts y*
⟨*proof*⟩

lemma *L3-Boiler*:
assumes *SteamBoiler x s y*
and *ts x*
shows *200* ≤ *hd* (*s 0*)
⟨*proof*⟩

lemma *L4-Boiler*:
assumes *SteamBoiler x s y*
and *ts x*
shows *hd* (*s 0*) ≤ *800*
⟨*proof*⟩

lemma *L5-Boiler*:
assumes *h1*:*SteamBoiler x s y*
and *h2*:*ts x*
and *h3*:*hd* (*x j*) = *Zero*
shows (*hd* (*s j*)) ≤ *hd* (*s* (*Suc j*)) + (*10*::*nat*)
⟨*proof*⟩

lemma *L6-Boiler*:
assumes *h1*:*SteamBoiler x s y*
and *h2*:*ts x*
and *h3*:*hd* (*x j*) = *Zero*
shows (*hd* (*s j*)) − (*10*::*nat*) ≤ *hd* (*s* (*Suc j*))
⟨*proof*⟩

lemma *L7-Boiler*:
assumes *h1*:*SteamBoiler x s y*
and *h2*:*ts x*
and *h3*:*hd* (*x j*) ≠ *Zero*
shows (*hd* (*s j*)) ≥ *hd* (*s* (*Suc j*)) − (*10*::*nat*)
⟨*proof*⟩

lemma *L8-Boiler*:
assumes *h1*:*SteamBoiler x s y*
and *h2*:*ts x*
and *h3*:*hd* (*x j*) ≠ *Zero*
shows (*hd* (*s j*)) + (*10*::*nat*) ≥ *hd* (*s* (*Suc j*))
⟨*proof*⟩

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*) ≠ []
⟨*proof*⟩

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*) = []
⟨*proof*⟩

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*) = []
⟨*proof*⟩

lemma *L3-Controller-Zero*:
assumes *Controller-L y* (*fin-inf-append* [*Zero*] *l*) *l z*
and *l t* = *Zero*
and ¬ *300* < *hd* (*y* (*Suc t*))
shows *z* (*Suc t*) = [*One*]
⟨*proof*⟩

lemma *L3-Controller-One*:
assumes *Controller-L y* (*fin-inf-append* [*Zero*] *l*) *l z*
and *l t* = *One*
and ¬ *hd* (*y* (*Suc t*)) < *700*
shows *z* (*Suc t*) = [*Zero*]
⟨*proof*⟩

lemma *L4-Controller-Zero*:
assumes *h1*:*Controller-L y* (*fin-inf-append* [*Zero*] *l*) *l z*
and *h2*:*l* (*Suc t*) = *Zero*
shows (*z* (*Suc t*) = [] ∧ *l t* = *Zero*) ∨ (*z* (*Suc t*) = [*Zero*] ∧ *l t* = *One*)
⟨*proof*⟩

lemma *L4-Controller-One*:
assumes *h1*:*Controller-L y* (*fin-inf-append* [*Zero*] *l*) *l z*
and *h2*:*l* (*Suc t*) = *One*
shows (*z* (*Suc t*) = [] ∧ *l t* = *One*) ∨ (*z* (*Suc t*) = [*One*] ∧ *l t* = *Zero*)
⟨*proof*⟩

lemma *L5-Controller-Zero*:
assumes *h1*:*Controller-L y lIn lOut z*
and *h2*:*lOut t* = *Zero*
and *h3*:*z t* = []
shows *lIn t* = *Zero*
⟨*proof*⟩

lemma *L5-Controller-One*:
assumes *h1*:*Controller-L y lIn lOut z*
and *h2*:*lOut t* = *One*
and *h3*:*z t* = []
shows *lIn t* = *One*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *L8-Controller*:
assumes *Controller-L y* (*fin-inf-append* [*Zero*] *l*) *l z*
shows *z t* = [] ∨ *z t* = [*Zero*] ∨ *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*) !
(*length* (*fin-make-untimed* (*inf-truncate z i*)) − *Suc 0*) = *Zero*
and *h3*:*last* (*fin-make-untimed* (*inf-truncate z i*)) = *l i*
and *h5*:*hd* (*s* (*Suc i*)) = *hd* (*s i*) − *r*
and *h6*:*fin-make-untimed* (*inf-truncate z i*) ≠ []
and *h8*:*r* ≤ *10*
shows *200* ≤ *hd* (*s* (*Suc i*))
⟨*proof*⟩

lemma *L10-Controller*:
assumes *h1*:*Controller-L s* (*fin-inf-append* [*Zero*] *l*) *l z*
and *h2*:*fin-make-untimed* (*inf-truncate z i*) !
(*length* (*fin-make-untimed* (*inf-truncate z i*)) − *Suc 0*) ≠ *Zero*
and *h3*:*last* (*fin-make-untimed* (*inf-truncate z i*)) = *l i*
and *h5*:*hd* (*s* (*Suc i*)) = *hd* (*s i*) + *r*
and *h6*:*fin-make-untimed* (*inf-truncate z i*) ≠ []
and *h8*:*r* ≤ *10*
shows *hd* (*s* (*Suc i*)) ≤ *800*
⟨*proof*⟩

9.3 Properties of the Converter Component

lemma *L1-Converter*:
assumes *Converter z x*
and *fin-make-untimed* (*inf-truncate z t*) ≠ []
shows *hd* (*x t*) = (*fin-make-untimed* (*inf-truncate z t*)) !
((*length* (*fin-make-untimed* (*inf-truncate z t*))) − (*1*::*nat*))
⟨*proof*⟩

lemma *L1a-Converter*:
assumes *Converter z x*
and *fin-make-untimed* (*inf-truncate z t*) ≠ []
and *hd* (*x t*) = *Zero*
shows (*fin-make-untimed* (*inf-truncate z t*)) !
((*length* (*fin-make-untimed* (*inf-truncate z t*))) − (*1*::*nat*))
= *Zero*
⟨*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*) ≤ *hd* (*s i*)
⟨*proof*⟩

lemma *L3-ControlSystem*:
assumes *ControlSystemArch s*
shows *hd* (*s i*) ≤ (*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* ⇒ (*'a Frame*) *istream*

type-synonym *nNat* = *nat* ⇒ *nat istream*

type-synonym *nConfig* = *nat* ⇒ *Config*

consts *sN* :: *nat*

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

end

11 FlexRay: Specification

theory *FR*
imports *FR-types*
begin

11.1 Auxiliary predicates

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

— The predicate IdenticCycleLength is true for sheaf of channels of type Config,
— if all bus configurations have the equal length of the communication round.
definition
IdenticCycleLength :: *nat* ⇒ *nConfig* ⇒ *bool*
where
IdenticCycleLength n nC
≡
∀ *i j*. *i* < *n* ∧ *j* < *n* ⟶
cycleLength (*nC i*) = *cycleLength* (*nC j*)

— The predicate FrameTransmission defines the correct message transmission:
— if the time t is equal modulo the length of the cycle (Flexray communication round)
— to the element of the scheduler table of the node k, then this and only this node
— can send a data atn the *t*th time interval.
definition
FrameTransmission ::
nat ⇒ *'a nFrame* ⇒ *'a nFrame* ⇒ *nNat* ⇒ *nConfig* ⇒ *bool*
where
FrameTransmission n nStore nReturn nGet nC
≡
∀ (*t*::*nat*) (*k*::*nat*). *k* < *n* ⟶
(*let s* = *t mod* (*cycleLength* (*nC k*))
in
(*s mem* (*schedule* (*nC k*))
⟶
(*nGet k t*) = [*s*] ∧
(∀ *j*. *j* < *n* ∧ *j* ≠ *k* ⟶
((*nStore j*) *t*) = ((*nReturn k*) *t*))))

— The predicate Broadcast describes properties of FlexRay broadcast.

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

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

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

11.2 Specifications of the FlexRay components

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

$(FrameTransmission\ n\ nStore\ nReturn\ nGet\ nC) \wedge$
$(\forall\ (i{::}nat).\ i < n \longrightarrow (msg\ 1\ (nGet\ i)) \wedge (msg\ 1\ (nStore\ i)))\ )$

definition
$Cable :: nat \Rightarrow 'a\ nFrame \Rightarrow 'a\ Frame\ istream \Rightarrow bool$
where
$Cable\ n\ nSend\ recv$
$\equiv$
$(CorrectSheaf\ n)$
$\wedge$
$((inf\text{-}disj\ n\ nSend) \longrightarrow (Broadcast\ n\ nSend\ recv))$

definition
$Scheduler :: Config \Rightarrow nat\ istream \Rightarrow bool$
where
$Scheduler\ c\ activation$
$\equiv$
$\forall\ (t{::}nat).$
$(\ let\ s = (t\ mod\ (cycleLength\ c))$
in
$(\ if\ \ (s\ mem\ (schedule\ c))$
$then\ (activation\ t) = [s]$
$else\ (activation\ t) = [])\ )$

definition
$BusInterface ::$
$nat\ istream \Rightarrow 'a\ Frame\ istream \Rightarrow 'a\ Frame\ istream \Rightarrow$
$'a\ Frame\ istream \Rightarrow 'a\ Frame\ istream \Rightarrow nat\ istream \Rightarrow bool$
where
$BusInterface\ activation\ return\ recv\ \ store\ send\ get$
$\equiv$
$(Receive\ recv\ store\ activation) \wedge$
$(Send\ return\ send\ get\ activation)$

definition
$FlexRayController ::$
$'a\ Frame\ istream \Rightarrow 'a\ Frame\ istream \Rightarrow\ \ Config \Rightarrow$
$'a\ Frame\ istream \Rightarrow 'a\ Frame\ istream \Rightarrow nat\ istream \Rightarrow bool$
where
$FlexRayController\ return\ recv\ c\ store\ send\ get$
$\equiv$
$(\exists\ activation.$
$(Scheduler\ c\ activation) \wedge$
$(BusInterface\ activation\ return\ recv\ store\ send\ get))$

definition
$FlexRayArchitecture ::$

$nat \Rightarrow 'a\ nFrame \Rightarrow nConfig \Rightarrow 'a\ nFrame \Rightarrow nNat \Rightarrow bool$
where
$FlexRayArchitecture\ n\ nReturn\ nC\ nStore\ nGet$
$\equiv$
$(CorrectSheaf\ n) \wedge$
$(\exists\ nSend\ recv.$
$(Cable\ n\ nSend\ recv) \wedge$
$(\forall\ (i{::}nat).\ i < n \longrightarrow$
$FlexRayController\ (nReturn\ i)\ recv\ (nC\ i)$
$(nStore\ i)\ (nSend\ i)\ (nGet\ i)))$

definition
$FlexRayArch ::$
$nat \Rightarrow 'a\ nFrame \Rightarrow nConfig \Rightarrow 'a\ nFrame \Rightarrow nNat \Rightarrow bool$
where
$FlexRayArch\ n\ nReturn\ nC\ nStore\ nGet$
$\equiv$
$(CorrectSheaf\ n)\ \wedge$
$((\forall\ (i{::}nat).\ i < n \longrightarrow msg\ 1\ (nReturn\ i)) \wedge$
$(DisjointSchedules\ n\ nC) \wedge (IdenticCycleLength\ n\ nC)$
$\longrightarrow$
$(FlexRayArchitecture\ n\ nReturn\ nC\ nStore\ nGet))$

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 $send\ t \neq []$
shows $(activation\ t) \neq []$
⟨*proof*⟩

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

12.2 Properties of the component Scheduler

lemma *Scheduler-L1*:

assumes *h1*:*Scheduler C activation*
and *h2*:(*activation t*) ≠ []
shows (*t mod* (*cycleLength C*)) *mem* (*schedule C*)
⟨*proof*⟩

lemma *Scheduler-L2*:
assumes *Scheduler C activation*
and ¬ (*t mod cycleLength C*) *mem* (*schedule C*)
shows *activation t* = []
⟨*proof*⟩

lemma *Scheduler-L3*:
assumes *Scheduler C activation*
and (*t mod cycleLength C*) *mem* (*schedule C*)
shows *activation t* ≠ []
⟨*proof*⟩

lemma *Scheduler-L4*:
assumes *Scheduler C activation*
and (*t mod cycleLength C*) *mem* (*schedule C*)
shows *activation t* = [*t mod cycleLength C*]
⟨*proof*⟩

lemma *correct-DisjointSchedules1*:
assumes *h1*:*DisjointSchedules n nC*
and *h2*:*IdenticCycleLength n nC*
and *h3*:(*t mod cycleLength* (*nC i*)) *mem schedule* (*nC i*)
and *h4*:*i* < *n*
and *h5*:*j* < *n*
and *h6*:*i* ≠ *j*
shows ¬ (*t mod cycleLength* (*nC j*) *mem schedule* (*nC j*))
⟨*proof*⟩

12.3 Disjoint Frames

lemma *disjointFrame-L1*:
assumes *h1*:*DisjointSchedules n nC*
and *h2*:*IdenticCycleLength n nC*
and *h3*:∀ *i* < *n*. *FlexRayController* (*nReturn i*) *rcv*
(*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
and *h4*:*nSend i t* ≠ []
and *h5*:*i* < *n*
and *h6*:*j* < *n*
and *h7*:*i* ≠ *j*
shows *nSend j t* = []
⟨*proof*⟩

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*
⟨*proof*⟩

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* = []
⟨*proof*⟩

12.4 Properties of the sheaf of channels nSend

lemma *fr-Send1*:
assumes *frc*:*FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
and *h1*:¬ (*t mod cycleLength* (*nC i*) *mem schedule* (*nC i*))
shows (*nSend i*) *t* = []
⟨*proof*⟩

lemma *fr-Send2*:
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*:*t mod cycleLength* (*nC k*) *mem schedule* (*nC k*)
and *h5*:$k < n$
shows *nSend k t* = *nReturn k t*
⟨*proof*⟩

lemma *fr-Send3*:
assumes $\forall\, i<n$. *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 $k < n$
and *nReturn k t* ≠ []
shows *nSend k t* ≠ []
⟨*proof*⟩

lemma *fr-Send4*:
assumes $\forall\, i{<}n.$ *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 $k < n$
and $nReturn\ k\ t \neq []$
shows $\exists\, k.\ k < n \longrightarrow nSend\ k\ t \neq []$
⟨*proof*⟩

lemma *fr-Send5*:
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*:*t mod cycleLength* (*nC k*) *mem schedule* (*nC k*)
and *h5*:$k < n$
and *h6*:$nReturn\ k\ t \neq []$
and *h7*:$\forall\, k{<}n.\ nSend\ k\ t = []$
shows *False*
⟨*proof*⟩

lemma *fr-Send6*:
assumes $\forall\, i{<}n.$ *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 $k < n$
and $nReturn\ k\ t \neq []$
shows $\exists\, k{<}n.\ nSend\ k\ t \neq []$
⟨*proof*⟩

lemma *fr-Send7*:
assumes $\forall\, i{<}n.$ *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 $k < n$
and $j < n$
and $nReturn\ k\ t = []$
shows $nSend\ j\ t = []$
⟨*proof*⟩

lemma *fr-Send8*:
assumes $\forall\, i{<}n.$ *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 $k < n$
and *nReturn k t* = []
shows ¬ (∃ *k*<*n*. *nSend k t* ≠ [])
⟨*proof*⟩

lemma *fr-nC-Send*:
assumes ∀ *i*<*n*. *FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)*
and $k < n$
and *DisjointSchedules n nC*
and *IdenticCycleLength n nC*
and *t mod cycleLength (nC k) mem schedule (nC k)*
shows ∀ *j*. $j < n \wedge j \neq k \longrightarrow$ *(nSend j) t* = []
⟨*proof*⟩

lemma *length-nSend*:
assumes *h1*:*BusInterface activation (nReturn i) recv (nStore i) (nSend i) (nGet i)*
and *h2*:∀ *t*. *length (nReturn i t)* ≤ *Suc 0*
shows *length (nSend i t)* ≤ *Suc 0*
⟨*proof*⟩

lemma *msg-nSend*:
assumes *BusInterface activation (nReturn i) recv (nStore i) (nSend i) (nGet i)*
and *msg (Suc 0) (nReturn i)*
shows *msg (Suc 0) (nSend i)*
⟨*proof*⟩

lemma *Broadcast-nSend-empty1*:
assumes *h1*:*Broadcast n nSend recv*
and *h2*:∀ *k*<*n*. *nSend k t* = []
shows *recv t* = []
⟨*proof*⟩

12.5 Properties of the sheaf of channels nGet

lemma *fr-nGet1a*:
assumes *h1*:*FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)*
and *h2*:*t mod cycleLength (nC k) mem schedule (nC k)*
shows *nGet k t* = [*t mod cycleLength (nC k)*]
⟨*proof*⟩

lemma *fr-nGet1*:
assumes ∀ *i*<*n*. *FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)*

and t *mod cycleLength* (nC k) *mem schedule* (nC k)
and $k < n$
shows *nGet* k t = [t *mod cycleLength* (nC k)]
⟨*proof*⟩

lemma *fr-nGet2a*:
assumes *h1*:*FlexRayController* (*nReturn* k) *recv* (nC k) (*nStore* k) (*nSend* k) (*nGet* k)
and *h2*:¬ (t *mod cycleLength* (nC k) *mem schedule* (nC k))
shows *nGet* k t = []
⟨*proof*⟩

lemma *fr-nGet2*:
assumes *h1*:∀ i<n. *FlexRayController* (*nReturn* i) *recv* (nC i) (*nStore* i) (*nSend* i) (*nGet* i)
and *h2*:¬ (t *mod cycleLength* (nC k) *mem schedule* (nC k))
and *h3*:$k < n$
shows *nGet* k t = []
⟨*proof*⟩

lemma *length-nGet1*:
assumes *FlexRayController* (*nReturn* k) *recv* (nC k) (*nStore* k) (*nSend* k) (*nGet* k)
shows *length* (*nGet* k t) ≤ *Suc 0*
⟨*proof*⟩

lemma *msg-nGet1*:
assumes *FlexRayController* (*nReturn* k) *recv* (nC k) (*nStore* k) (*nSend* k) (*nGet* k)
shows *msg* (*Suc 0*) (*nGet* k)
⟨*proof*⟩

lemma *msg-nGet2*:
assumes ∀ i<n. *FlexRayController* (*nReturn* i) *recv* (nC i) (*nStore* i) (*nSend* i) (*nGet* i)
and $k < n$
shows *msg* (*Suc 0*) (*nGet* k)
⟨*proof*⟩

12.6 Properties of the sheaf of channels nStore

lemma *fr-nStore-nReturn1*:
assumes *h0*:*Broadcast* n *nSend recv*
and *h1*:*inf-disj* n *nSend*
and *h2*:∀ i<n. *FlexRayController* (*nReturn* i) *recv* (nC i) (*nStore* i) (*nSend* i) (*nGet* i)
and *h3*:*DisjointSchedules* n nC
and *h4*:*IdenticCycleLength* n nC
and *h5*:t *mod cycleLength* (nC k) *mem schedule* (nC k)

and *h6*:$k < n$
and *h7*:$j < n$
and *h8*:$j \neq k$
shows *nStore j t* = *nReturn k t*
⟨*proof*⟩

lemma *fr-nStore-nReturn2*:
assumes *h1*:*Cable n nSend recv*
and *h2*:∀ *i*<*n*. *FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
and *h3*:*DisjointSchedules n nC*
and *h4*:*IdenticCycleLength n nC*
and *h5*:*t mod cycleLength* (*nC k*) *mem schedule* (*nC k*)
and *h6*:$k < n$
and *h7*:$j < n$
and *h8*:$j \neq k$
shows *nStore j t* = *nReturn k t*
⟨*proof*⟩

lemma *fr-nStore-empty1*:
assumes *h1*:*Cable n nSend recv*
and *h2*:∀ *i*<*n*. *FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
and *h3*:*DisjointSchedules n nC*
and *h4*:*IdenticCycleLength n nC*
and *h5*:(*t mod cycleLength* (*nC k*) *mem schedule* (*nC k*))
and *h6*:$k < n$
shows *nStore k t* = []
⟨*proof*⟩

lemma *fr-nStore-nReturn3*:
assumes *Cable n nSend recv*
and ∀ *i*<*n*. *FlexRayController* (*nReturn i*) *recv* (*nC i*) (*nStore i*) (*nSend i*) (*nGet i*)
and *DisjointSchedules n nC*
and *IdenticCycleLength n nC*
and *t mod cycleLength* (*nC k*) *mem schedule* (*nC k*)
and $k < n$
shows ∀ *j*. $j < n \wedge j \neq k \longrightarrow$ *nStore j t* = *nReturn k t*
⟨*proof*⟩

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

and *h6*:$\forall\ i{<}n.\ msg\ (Suc\ 0)\ (nReturn\ i)$
and *h7*:*Broadcast n nSend recv*
shows $length\ (nStore\ i\ t) \leq Suc\ 0$
⟨*proof*⟩

lemma *msg-nStore*:
assumes $\forall\, i{<}n.\ FlexRayController\ (nReturn\ i)\ recv\ (nC\ i)\ (nStore\ i)\ (nSend\ i)\ (nGet\ i)$
and *DisjointSchedules n nC*
and *IdenticCycleLength n nC*
and *inf-disj n nSend*
and $i < n$
and $\forall\ i{<}n.\ msg\ (Suc\ 0)\ (nReturn\ i)$
and *Cable n nSend recv*
shows $msg\ (Suc\ 0)\ (nStore\ i)$
⟨*proof*⟩

12.7 Refinement Properties

lemma *fr-refinement-FrameTransmission*:
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*
shows *FrameTransmission n nStore nReturn nGet nC*
⟨*proof*⟩

lemma *FlexRayArch-CorrectSheaf*:
assumes *FlexRayArch n nReturn nC nStore nGet*
shows *CorrectSheaf n*
⟨*proof*⟩

lemma *FlexRayArch-FrameTransmission*:
assumes *h1*:*FlexRayArch n nReturn nC nStore nGet*
and *h2*:$\forall\, i{<}n.\ msg\ (Suc\ 0)\ (nReturn\ i)$
and *h3*:*DisjointSchedules n nC*
and *h4*:*IdenticCycleLength n nC*
shows *FrameTransmission n nStore nReturn nGet nC*
⟨*proof*⟩

lemma *FlexRayArch-nGet*:
assumes *h1*:*FlexRayArch n nReturn nC nStore nGet*
and *h2*:$\forall\, i{<}n.\ msg\ (Suc\ 0)\ (nReturn\ i)$
and *h3*:*DisjointSchedules n nC*
and *h4*:*IdenticCycleLength n nC*
and *h5*:$i < n$
shows $msg\ (Suc\ 0)\ (nGet\ i)$
⟨*proof*⟩

lemma *FlexRayArch-nStore*:
assumes *h1*:*FlexRayArch n nReturn nC nStore nGet*
and *h2*:$\forall$ *i*<*n*. *msg* (*Suc 0*) (*nReturn i*)
and *h3*:*DisjointSchedules n nC*
and *h4*:*IdenticCycleLength n nC*
and *h5*:*i* < *n*
shows *msg* (*Suc 0*) (*nStore i*)
⟨*proof*⟩

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
≡

$\forall$ (t::*nat*)
(r::*reqType list*) (x::*aType list*)
(y::*stopType list*) (z::*bool list*).
— 1:
(*st-in* t = *init-state* $\wedge$ *req* t = [*init*]
$\longrightarrow$ *ack* t = [*call*] $\wedge$ *i1* t = [] $\wedge$ *vc* t = []
$\wedge$ *st-out* t = *call*)
$\wedge$
— 2:
(*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*)
$\wedge$
— 3:
((*st-in* t = *call* $\vee$ (*st-in* t = *connection-ok* $\wedge$ $r \neq$ [*send*])) $\wedge$
req t = r $\wedge$ *lose* t = [*False*]
$\longrightarrow$ *ack* t = [*connection-ok*] $\wedge$ *i1* t = [] $\wedge$ *vc* t = []
$\wedge$ *st-out* t = *connection-ok*)
$\wedge$
— 4:
((*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*)
$\wedge$
— 5:
(*st-in* t = *connection-ok* $\wedge$ *req* t = [*send*] $\wedge$ *lose* t = [*False*]
$\longrightarrow$ *ack* t = [*sending-data*] $\wedge$ *i1* t = *buffer-in* t $\wedge$ *vc* t = []
$\wedge$ *st-out* t = *sending-data*)
$\wedge$
— 6:
(*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*)
$\wedge$
— 7:
(*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*)
$\wedge$
— 8:
(*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*)
$\wedge$
— 9:
(*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*)

$\wedge$
— 10:
(*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*)

definition
Sample-L ::
reqType istream $\Rightarrow$ *ECall-Info istream* $\Rightarrow$ *aType istream* $\Rightarrow$
stopType istream $\Rightarrow$ *bool istream* $\Rightarrow$
(*nat* $\Rightarrow$ *GatewayStatus*) $\Rightarrow$ (*nat* $\Rightarrow$ *ECall-Info list*) $\Rightarrow$
GatewayStatus istream $\Rightarrow$ *ECall-Info istream* $\Rightarrow$ *vcType istream*
$\Rightarrow$ (*nat* $\Rightarrow$ *GatewayStatus*) $\Rightarrow$ (*nat* $\Rightarrow$ *ECall-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-out t =
(*if dt t* = [] *then buffer-in t else dt t*))
$\wedge$
(*tiTable-SampleT req a1 stop lose st-in buffer-in*
ack i1 vc st-out)

definition
Sample ::
reqType istream $\Rightarrow$ *ECall-Info istream* $\Rightarrow$ *aType istream* $\Rightarrow$
stopType istream $\Rightarrow$ *bool istream* $\Rightarrow$
GatewayStatus istream $\Rightarrow$ *ECall-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-L req dt a1 stop lose*
(*fin-inf-append* [*init-state*] *st*)
(*fin-inf-append* [[]] *buffer*)
ack i1 vc st buffer))

definition
Gateway ::
reqType istream $\Rightarrow$ *ECall-Info istream* $\Rightarrow$ *aType istream* $\Rightarrow$
stopType istream $\Rightarrow$ *bool istream* $\Rightarrow$ *nat* $\Rightarrow$
GatewayStatus istream $\Rightarrow$ *ECall-Info istream* $\Rightarrow$ *vcType istream*

$\Rightarrow$ *bool*

where

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

definition

GatewaySystem ::
reqType istream $\Rightarrow$ *ECall-Info istream* $\Rightarrow$
stopType istream $\Rightarrow$ *bool istream* $\Rightarrow$ *nat* $\Rightarrow$
GatewayStatus istream $\Rightarrow$ *vcType istream*
$\Rightarrow$ *bool*

where

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

definition

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

where

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

definition

$GatewaySystemReq$ ::
$reqType\ istream \Rightarrow ECall\text{-}Info\ istream \Rightarrow$
$stopType\ istream \Rightarrow bool\ istream \Rightarrow nat \Rightarrow$
$GatewayStatus\ istream \Rightarrow vcType\ istream$
$\Rightarrow bool$
where
$GatewaySystemReq\ req\ dt\ \ stop\ lose\ d\ ack\ vc$
$\equiv$
$((msg\ (1{::}nat)\ req) \wedge (msg\ (1{::}nat)\ stop) \wedge (ts\ lose))$
$\longrightarrow$
$(\forall\ (t{::}nat)\ (k{::}nat).$
$(\ ack\ t = [init\text{-}state] \wedge req\ (Suc\ t) = [init]$
$\wedge\ (\forall\ t1.\ t1 \leq t \longrightarrow req\ t1 = [])$
$\wedge\ req\ (t{+}2) = []$
$\wedge\ (\forall\ m.\ m < k + 3 \longrightarrow req\ (t + m) \neq [send])$
$\wedge\ \ req\ (t{+}3{+}k)\ = [send] \wedge\ \ inf\text{-}last\text{-}ti\ dt\ (t{+}2) \neq []$
$\wedge\ (\forall\ (j{::}nat).$
$j \leq (4 + k + d + d) \longrightarrow lose\ (t{+}j) = [False])$
$\longrightarrow vc\ (t + 4 + k + d + d) = [vc\text{-}com])\ )$

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*: $a\ t \neq [sc\text{-}ack]$
shows $a\ t = []$
⟨*proof*⟩

lemma *aType-nonempty*:
assumes *h1*:$msg\ (Suc\ 0)\ a$
and *h2*: $a\ t \neq []$
shows $a\ t = [sc\text{-}ack]$
⟨*proof*⟩

lemma *aType-lemma*:
assumes $msg\ (Suc\ 0)\ a$
shows $a\ t = [] \vee a\ t = [sc\text{-}ack]$
⟨*proof*⟩

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

and $a\ t \neq [stop\text{-}vc]$
shows $a\ t = []$
⟨*proof*⟩

lemma *stopType-nonempty*:
assumes $msg\ (Suc\ 0)\ a$
and $a\ t \neq []$
shows $a\ t = [stop\text{-}vc]$
⟨*proof*⟩

lemma *stopType-lemma*:
assumes $msg\ (Suc\ 0)\ a$
shows $a\ t = [] \vee a\ t = [stop\text{-}vc]$
⟨*proof*⟩

lemma *vcType-empty*:
assumes $msg\ (Suc\ 0)\ a$
and $a\ t \neq [vc\text{-}com]$
shows$a\ t = []$
⟨*proof*⟩

lemma *vcType-lemma*:
assumes $msg\ (Suc\ 0)\ a$
shows $a\ t = [] \vee a\ t = [vc\text{-}com]$
⟨*proof*⟩

15.2 Properties of the Delay component

lemma *Delay-L1*:
assumes *h1*:$\forall\, t1 < t.\ i1\ t1 = []$
and *h2*:$Delay\ y\ i1\ d\ x\ i2$
and *h3*:$t2 < t + d$
shows $i2\ t2 = []$
⟨*proof*⟩

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

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 $x\ t2 = []$
⟨*proof*⟩

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 $y\ t2 = []$
⟨*proof*⟩

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 $y\ (t + Suc\ j) = []$
⟨*proof*⟩

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

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

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

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

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 $t2 - d < t + (3::nat) + k$
⟨*proof*⟩

lemma *ts-lose-ack-st1ts*:
assumes *ts lose*
and $lose\ t = [True] \longrightarrow ack\ t = [x] \wedge st\text{-}out\ t = x$
and $lose\ t = [False] \longrightarrow ack\ t = [y] \wedge st\text{-}out\ t = y$
shows $ack\ t = [st\text{-}out\ t]$
⟨*proof*⟩

lemma *ts-lose-ack-st1*:
assumes $h1{:}lose\ t = [True] \vee lose\ t = [False]$
and $h2{:}lose\ t = [True] \longrightarrow ack\ t = [x] \wedge st\text{-}out\ t = x$
and $h3{:}lose\ t = [False] \longrightarrow ack\ t = [y] \wedge st\text{-}out\ t = y$
shows $ack\ t = [st\text{-}out\ t]$
⟨*proof*⟩

lemma *ts-lose-ack-st2ts*:
assumes *ts lose*
and $lose\ t = [True] \longrightarrow$
$ack\ t = [x] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st\text{-}out\ t = x$
and $lose\ t = [False] \longrightarrow$
$ack\ t = [y] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st\text{-}out\ t = y$
shows $ack\ t = [st\text{-}out\ t]$
⟨*proof*⟩

lemma *ts-lose-ack-st2*:
assumes $h1{:}lose\ t = [True] \vee lose\ t = [False]$
and $h2{:}lose\ t = [True] \longrightarrow$
$ack\ t = [x] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st\text{-}out\ t = x$
and $h3{:}lose\ t = [False] \longrightarrow$
$ack\ t = [y] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st\text{-}out\ t = y$
shows $ack\ t = [st\text{-}out\ t]$
⟨*proof*⟩

lemma *ts-lose-ack-st2vc-com*:
assumes $h1{:}lose\ t = [True] \vee lose\ t = [False]$
and $h2{:}lose\ t = [True] \longrightarrow$
$ack\ t = [x] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st\text{-}out\ t = x$
and $h3{:}lose\ t = [False] \longrightarrow$
$ack\ t = [y] \wedge i1\ t = [] \wedge vc\ t = [vc\text{-}com] \wedge st\text{-}out\ t = y$
shows $ack\ t = [st\text{-}out\ t]$
⟨*proof*⟩

lemma *ts-lose-ack-st2send*:
assumes $h1{:}lose\ t = [True] \vee lose\ t = [False]$
and $h2{:}lose\ t = [True] \longrightarrow$
$ack\ t = [x] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st\text{-}out\ t = x$
and $h3{:}lose\ t = [False] \longrightarrow$

$ack\ t = [y] \wedge i1\ t = b\ t \wedge vc\ t = [] \wedge st\text{-}out\ t = y$
shows $ack\ t = [st\text{-}out\ t]$
⟨*proof*⟩

lemma *tiTable-ack-st-splitten*:
assumes *h1*:*ts lose*
and *h2*:*msg* (*Suc 0*) *a1*
and *h3*:*msg* (*Suc 0*) *stop*
and *h4*:$st\text{-}in\ t = init\text{-}state \wedge req\ t = [init] \longrightarrow$
$ack\ t = [call] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st\text{-}out\ t = call$
and *h5*:$st\text{-}in\ t = init\text{-}state \wedge req\ t \neq [init] \longrightarrow$
$ack\ t = [init\text{-}state] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st\text{-}out\ t = init\text{-}state$
and *h6*:$(st\text{-}in\ t = call \vee st\text{-}in\ t = connection\text{-}ok \wedge req\ t \neq [send]) \wedge lose\ t = [False] \longrightarrow$
$ack\ t = [connection\text{-}ok] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st\text{-}out\ t = connection\text{-}ok$
and *h7*:$(st\text{-}in\ t = call \vee st\text{-}in\ t = connection\text{-}ok \vee st\text{-}in\ t = sending\text{-}data) \wedge lose\ t = [True] \longrightarrow$
$ack\ t = [init\text{-}state] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st\text{-}out\ t = init\text{-}state$
and *h8*:$st\text{-}in\ t = connection\text{-}ok \wedge req\ t = [send] \wedge lose\ t = [False] \longrightarrow$
$ack\ t = [sending\text{-}data] \wedge i1\ t = b\ t \wedge vc\ t = [] \wedge st\text{-}out\ t = sending\text{-}data$
and *h9*:$st\text{-}in\ t = sending\text{-}data \wedge a1\ t = [] \wedge lose\ t = [False] \longrightarrow$
$ack\ t = [sending\text{-}data] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st\text{-}out\ t = sending\text{-}data$
and *h10*:$st\text{-}in\ t = sending\text{-}data \wedge a1\ t = [sc\text{-}ack] \wedge lose\ t = [False] \longrightarrow$
$ack\ t = [voice\text{-}com] \wedge i1\ t = [] \wedge vc\ t = [vc\text{-}com] \wedge st\text{-}out\ t = voice\text{-}com$
and *h11*:$st\text{-}in\ t = voice\text{-}com \wedge stop\ t = [] \wedge lose\ t = [False] \longrightarrow$
$ack\ t = [voice\text{-}com] \wedge i1\ t = [] \wedge vc\ t = [vc\text{-}com] \wedge st\text{-}out\ t = voice\text{-}com$
and *h12*:$st\text{-}in\ t = voice\text{-}com \wedge stop\ t = [] \wedge lose\ t = [True] \longrightarrow$
$ack\ t = [voice\text{-}com] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st\text{-}out\ t = voice\text{-}com$
and *h13*:$st\text{-}in\ t = voice\text{-}com \wedge stop\ t = [stop\text{-}vc] \longrightarrow$
$ack\ t = [init\text{-}state] \wedge i1\ t = [] \wedge vc\ t = [] \wedge st\text{-}out\ t = init\text{-}state$
shows $ack\ t = [st\text{-}out\ t]$
⟨*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 $ack\ t = [st\text{-}out\ t]$
⟨*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 $st\text{-}out\ t = hd\ (ack\ t)$
⟨*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* ∨ *st-in t* = *connection-ok* ∧ *req t* ≠ [*send*]) ∧
        lose t = [*False*]
⟨*proof*⟩

lemma *tiTable-i1-1*:
assumes *tbl*:*tiTable-SampleT req x stop lose st-in b ack i1 vc st-out*
      and *ts lose*
      and *msg* (*Suc 0*) *x*
      and *msg* (*Suc 0*) *stop*
      and *ack t* = [*connection-ok*]
shows *i1 t* = []
⟨*proof*⟩

lemma *tiTable-ack-call*:
assumes *tbl*:*tiTable-SampleT req x stop lose st-in b ack i1 vc st-out*
      and *ackCall*:*ack t* = [*call*]
      and *xMsg1*:*msg* (*Suc 0*) *x*
      and *tsLose*:*ts lose*
      and *stopMsg1*:*msg* (*Suc 0*) *stop*
shows *st-in t* = *init-state* ∧ *req t* = [*init*]
⟨*proof*⟩

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

lemma *tiTable-ack-init0*:
assumes *tbl*:*tiTable-SampleT req a1 stop lose*
                (*fin-inf-append* [*init-state*] *st*)
                 b ack i1 vc st
      and *req0*:*req 0* = []
shows *ack 0* = [*init-state*]
⟨*proof*⟩

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*]
⟨*proof*⟩

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* = []
⟨*proof*⟩

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*
⟨*proof*⟩

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* = []
⟨*proof*⟩

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 *req* (*Suc t*) = [*init*]
and ∀ *m* < *k* + *3*. *req* (*t* + *m*) ≠ [*send*]
and ∀ *m* ≤ *k*. *ack* (*Suc* (*Suc* (*t* + *m*))) = [*connection-ok*]
and ∀ *j* ≤ *k* + *3*. *lose* (*t* + *j*) = [*False*]
shows ∀ *t2* < (*t* + *3* + *k*). *i1 t2* = []
⟨*proof*⟩

lemma *tiTable-ack-ok*:
assumes *h1*:∀ *j*≤ *d* + *2*. *lose* (*t* + *j*) = [*False*]
and *tsLose*:*ts lose*
and *stopMsg1*:*msg* (*Suc 0*) *stop*
and *a1Msg1*:*msg* (*Suc 0*) *a1*
and *reqNsend*:*req* (*Suc t*) ≠ [*send*]
and *ackCon*:*ack t* = [*connection-ok*]
and *tbl*:*tiTable-SampleT req a1 stop lose* (*fin-inf-append* [*init-state*] *st*) *b ack i1 vc st*
shows *ack* (*Suc t*) = [*connection-ok*]
⟨*proof*⟩

lemma *Gateway-L7a*:
assumes *gw*:*Gateway req dt a stop lose d ack i vc*
and *aMsg1*:*msg* (*Suc 0*) *a*
and *stopMsg1*:*msg* (*Suc 0*) *stop*
and *reqMsg1*:*msg* (*Suc 0*) *req*
and *tsLose*:*ts lose*
and *loseFalse*:∀ *j*≤ *d* + *2*. *lose* (*t* + *j*) = [*False*]
and *nsend*:*req* (*Suc t*) ≠ [*send*]
and *ackNCon*:*ack* (*t*) = [*connection-ok*]
shows *ack* (*Suc t*) = [*connection-ok*]
⟨*proof*⟩

lemma *Sample-L-buffer*:
assumes
Sample-L req dt a1 stop lose (*fin-inf-append* [*init-state*] *st*)
(*fin-inf-append* [[]] *buffer*)
ack i1 vc st buffer
shows *buffer t* = *inf-last-ti dt t*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *Sample-L-i1-buffer*:
assumes *msg* (*Suc 0*) *req*
and *msg* (*Suc 0*) *a*
and *stopMsg1*:*msg* (*Suc 0*) *stop*
and *a1Msg1*:*msg* (*Suc 0*) *a1*
and *tsLose*:*ts lose*
and *ackCon*:*ack t* = [*connection-ok*]
and *reqSend*:*req* (*Suc t*) = [*send*]
and *loseFalse*:∀ *k*≤*Suc d*. *lose* (*t* + *k*) = [*False*]
and *smpl*:*Sample-L req dt a1 stop lose*
(*fin-inf-append* [*init-state*] *st*)
(*fin-inf-append* [[]] *buffer*) *ack i1 vc st buffer*
shows *i1* (*Suc t*) = *buffer t*
⟨*proof*⟩

lemma *tiTable-SampleT-sending-data*:
assumes *tbl*: *tiTable-SampleT req a1 stop lose* (*fin-inf-append* [*init-state*] *st*)
(*fin-inf-append* [[]] *buffer*)
ack i1 vc st
and *loseFalse*:∀ *j*≤*2* ∗ *d*. *lose* (*t* + *j*) = [*False*]
and *a1e*:∀ *t4*≤*t* + *d* + *d*. *a1 t4* = []
and *snd*:*fin-inf-append* [*init-state*] *st* (*Suc* (*t* + *x*)) = *sending-data*
and *h6*:*Suc* (*t* + *x*) ≤ *2* ∗ *d* + *t*
shows *ack* (*Suc* (*t* + *x*)) = [*sending-data*]
⟨*proof*⟩

lemma *Sample-sending-data*:
assumes *stopMsg1*:*msg* (*Suc 0*) *stop*
and *tsLose*:*ts lose*
and *reqMsg1*:*msg* (*Suc 0*) *req*
and *a1Msg1*:*msg* (*Suc 0*) *a1*
and *loseFalse*:∀ *j*≤*2* ∗ *d*. *lose* (*t* + *j*) = [*False*]
and *ackSnd*:*ack t* = [*sending-data*]
and *smpl*:*Sample req dt a1 stop lose ack i1 vc*
and *xdd*:*x* ≤ *d* + *d*
and *h9*:∀ *t4* ≤ *t* + *d* + *d*. *a1 t4* = []
shows *ack* (*t* + *x*) = [*sending-data*]
⟨*proof*⟩

15.6 Properties of the ServiceCenter component

lemma *ServiceCenter-a-l*:
assumes *ServiceCenter i a*
shows *length* (*a t*) ≤ (*Suc 0*)
⟨*proof*⟩

lemma *ServiceCenter-a-msg*:

assumes *ServiceCenter i a*
shows *msg (Suc 0) a*
⟨*proof*⟩

lemma *ServiceCenter-L1*:
assumes ∀ *t2* < *x*. *i t2* = []
and *ServiceCenter i a*
and *t* ≤ *x*
shows *a t* = []
⟨*proof*⟩

lemma *ServiceCenter-L2*:
assumes ∀ *t2* < *x*. *i t2* = []
and *ServiceCenter i a*
shows ∀ *t3* ≤ *x*. *a t3* = []
⟨*proof*⟩

15.7 General properties of stream values

lemma *streamValue1*:
assumes *h1*:∀*j*≤ *D* + (*z*::*nat*). *str* (*t* + *j*) = *x*
and *h2*: *j*≤ *D*
shows *str* (*t* + *j* + *z*) = *x*
⟨*proof*⟩

lemma *streamValue2*:
assumes ∀*j*≤ *D* + (*z*::*nat*). *str* (*t* + *j*) = *x*
shows ∀*j*≤ *D*. *str* (*t* + *j* + *z*) = *x*
⟨*proof*⟩

lemma *streamValue3*:
assumes ∀*j*≤ *D*. *str* (*t* + *j* + (*Suc y*)) = *x*
and *j* ≤ *D*
and *h3*:*str* (*t* + *y*) = *x*
shows *str* (*t* + *j* + *y*) = *x*
⟨*proof*⟩

lemma *streamValue4*:
assumes ∀*j*≤ *D*. *str* (*t* + *j* + (*Suc y*)) = *x*
and *str* (*t* + *y*) = *x*
shows ∀*j*≤ *D*. *str* (*t* + *j* + *y*) = *x*
⟨*proof*⟩

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

lemma *streamValue6*:
  assumes $\forall j{\leq}\ D.\ str\ (t + j + ((i::nat) + k)) = x$
  shows $\forall j{\leq}\ D.\ str\ (t + (i::nat) + k + j) = x$
⟨*proof*⟩

lemma *streamValue7*:
  assumes *h1*:$\forall j{\leq}d.\ str\ (t + i + k + d + Suc\ j) = x$
    and *h2*:$str\ (t + i + k + d) = x$
    and *h3*:$j{\leq}\ Suc\ d$
  shows $str\ (t + i + k + d + j) = x$
⟨*proof*⟩

lemma *streamValue8*:
assumes $\forall j{\leq}d.\ str\ (t + i + k + d + Suc\ j) = x$
    and $str\ (t + i + k + d) = x$
shows $\forall\ j{\leq}\ Suc\ d.\ str\ (t + i + k + d + j) = x$
⟨*proof*⟩

lemma *arith-streamValue9aux*:
$Suc\ (t + (j + d) + (i + k)) =\ Suc\ (t + i + k + d + j)$
⟨*proof*⟩

lemma *streamValue9*:
assumes *h1*:$\forall j{\leq}2 * d.\ str\ (t + j + Suc\ (i + k)) = x$
    and *h2*:$j{\leq}d$
shows $str\ (t + i + k + d + Suc\ j) = x$
⟨*proof*⟩

lemma *streamValue10*:
  assumes $\forall j{\leq}2 * d.\ str\ (t + j + Suc\ (i + k)) = x$
  shows $\forall j{\leq}d.\ str\ (t + i + k + d + Suc\ j) = x$
⟨*proof*⟩

lemma *arith-sum1*:$(t::nat) + (i + k + d) =\ t + i + k + d$
⟨*proof*⟩

lemma *arith-sum2*:$Suc\ (Suc\ (t + k + j)) = Suc\ (Suc\ (t + (k + j)))$
⟨*proof*⟩

lemma *arith-sum4*:$t + 3 + k + d = Suc\ (t + (2::nat) + k + d)$
⟨*proof*⟩

lemma *streamValue11*:
assumes *h1*:$\forall j{\leq}2 * d + (4 + k).\ lose\ (t + j) = x$
    and *h2*:$j{\leq}Suc\ d$
shows $lose\ (t + 2 + k + j) = x$
⟨*proof*⟩

lemma *streamValue12*:

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

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

end

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

15.8 Properties of the Gateway

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

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

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

and *h5*:*ts lose*
and *h6*:*ack t* = [*connection-ok*]
and *h7*:*req* (*Suc t*) = [*send*]
and *h8*:∀ *k*≤*Suc d*. *lose* (*t* + *k*) = [*False*]
shows *ack* (*Suc t*) = [*sending-data*]
⟨*proof*⟩

lemma *Gateway-L4*:
assumes *h1*:*Gateway req dt a stop lose d ack i vc*
and *h2*:*msg* (*Suc 0*) *req*
and *h3*:*msg* (*Suc 0*) *a*
and *h4*:*msg* (*Suc 0*) *stop*
and *h5*:*ts lose*
and *h6*:*ack* (*t* + *d*) = [*sending-data*]
and *h7*:*a* (*Suc t*) = [*sc-ack*]
and *h8*:∀ *k*≤*Suc d*. *lose* (*t* + *k*) = [*False*]
shows *vc* (*Suc* (*t* + *d*)) = [*vc-com*]
⟨*proof*⟩

lemma *Gateway-L5*:
assumes *h1*:*Gateway req dt a stop lose d ack i vc*
and *h2*:*msg* (*Suc 0*) *req*
and *h3*:*msg* (*Suc 0*) *a*
and *h4*:*msg* (*Suc 0*) *stop*
and *h5*:*ts lose*
and *h6*:*ack* (*t* + *d*) = [*sending-data*]
and *h7*:∀ *j* ≤ *Suc d*. *a* (*t*+*j*) = []
and *h8*:∀ *k*≤ (*d* + *d*). *lose* (*t* + *k*) = [*False*]
shows *j* ≤ *d* ⟶ *ack* (*t*+*d*+*j*) = [*sending-data*]
⟨*proof*⟩

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

lemma *Gateway-L6*:
assumes *Gateway req dt a stop lose d ack i vc*
and ∀ *m*≤*k*. *req* (*t* + *m*) ≠ [*send*]

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

lemma *Gateway-L6a*:
assumes $Gateway\ req\ dt\ a\ stop\ lose\ d\ ack\ i\ vc$
and $\forall m{\leq}k.\ req\ (t + 2 + m) \neq [send]$
and $\forall j{\leq}k.\ lose\ (t + 2 + j) = [False]$
and $ack\ (t + 2) = [connection\text{-}ok]$
and $msg\ (Suc\ 0)\ req$
and $msg\ (Suc\ 0)\ stop$
and $msg\ (Suc\ 0)\ a$
and $ts\ lose$
shows $\forall m{\leq}k.\ ack\ (t + 2 + m) = [connection\text{-}ok]$
⟨*proof*⟩

lemma *aux-k3req*:
assumes $h1{:}\forall m{<}k + 3.\ req\ (t + m) \neq [send]$
and $h2{:}m \leq k$
shows $req\ (Suc\ (Suc\ (t + m))) \neq [send]$
⟨*proof*⟩

lemma *aux3lose*:
assumes $h1{:}\forall j{\leq}k + d + 3.\ lose\ (t + j) = [False]$
and $h2{:}j \leq k$
shows $lose\ (Suc\ (Suc\ (t + j))) = [False]$
⟨*proof*⟩

lemma *Gateway-L7*:
assumes $h1{:}Gateway\ req\ dt\ a\ stop\ lose\ d\ ack\ i\ vc$
and $h2{:}ts\ lose$
and $h3{:}msg\ (Suc\ 0)\ a$
and $h4{:}msg\ (Suc\ 0)\ stop$
and $h5{:}msg\ (Suc\ 0)\ req$
and $h6{:}req\ (Suc\ t) = [init]$
and $h7{:}\forall m < (k + 3).\ req\ (t + m) \neq [send]$
and $h8{:}req\ (t + 3 + k) = [send]$
and $h9{:}ack\ t = [init\text{-}state]$
and $h10{:}\forall j{\leq}k + d + 3.\ lose\ (t + j) = [False]$
and $h11{:}\forall\ t1 \leq t.\ req\ t1 = []$
shows $\forall\ t2 < (t + 3 + k + d).\ i\ t2 = []$
⟨*proof*⟩

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

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

15.9 Proof of the Refinement Relation for the Gateway Requirements

lemma *Gateway-L0*:
assumes *Gateway req dt a stop lose d ack i vc*
shows *GatewayReq req dt a stop lose d ack i vc*
⟨*proof*⟩

15.10 Lemmas about Gateway Requirements

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

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* ≠ []
and *h8*:∀ *j*≤*2* ∗ *d* + (*4* + *k*). *lose* (*t* + *j*) = [*False*]
and *h9*:∀ *m*≤*k*. *ack* (*t* + *2* + *m*) = [*connection-ok*]
shows *i* (*t* + *3* + *k* + *d*) ≠ []
⟨*proof*⟩

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* ∧ *msg* (*Suc 0*) *a* ∧ *msg* (*Suc 0*) *stop* ∧ *ts lose* ⟶
(∀ *t*. (*ack t* = [*init-state*] ∧
req (*Suc t*) = [*init*] ∧ *lose* (*Suc t*) = [*False*] ∧
lose (*Suc* (*Suc t*)) = [*False*] ⟶
ack (*Suc* (*Suc t*)) = [*connection-ok*]) ∧
(*ack t* = [*connection-ok*] ∧ *req* (*Suc t*) = [*send*] ∧
(∀ *k*≤*Suc d*. *lose* (*t* + *k*) = [*False*]) ⟶
i (*Suc* (*t* + *d*)) = *inf-last-ti dt t* ∧ *ack* (*Suc t*) = [*sending-data*]) ∧
(*ack* (*t* + *d*) = [*sending-data*] ∧ *a* (*Suc t*) = [*sc-ack*] ∧
(∀ *k*≤*Suc d*. *lose* (*t* + *k*) = [*False*]) ⟶
vc (*Suc* (*t* + *d*)) = [*vc-com*]))
shows *ack* (*t* + *3* + *k* + *d* + *d*) = [*sending-data*] ∧
a (*Suc* (*t* + *3* + *k* + *d*)) = [*sc-ack*] ∧
(∀ *ka*≤*Suc d*. *lose* (*t* + *3* + *k* + *d* + *ka*) = [*False*]) ⟶
vc (*Suc* (*t* + *3* + *k* + *d* + *d*)) = [*vc-com*]
⟨*proof*⟩

lemma *GatewaySystem-L3aux*:
assumes *msg* (*Suc 0*) *req*
and *msg* (*Suc 0*) *stop*
and *msg* (*Suc 0*) *a*
and *ts lose*
and *msg* (*Suc 0*) *req* ∧ *msg* (*Suc 0*) *a* ∧ *msg* (*Suc 0*) *stop* ∧ *ts lose* ⟶
(∀ *t*. (*ack t* = [*init-state*] ∧
req (*Suc t*) = [*init*] ∧ *lose* (*Suc t*) = [*False*] ∧
lose (*Suc* (*Suc t*)) = [*False*] ⟶
ack (*Suc* (*Suc t*)) = [*connection-ok*]) ∧
(*ack t* = [*connection-ok*] ∧ *req* (*Suc t*) = [*send*] ∧
(∀ *k*≤*Suc d*. *lose* (*t* + *k*) = [*False*]) ⟶
i (*Suc* (*t* + *d*)) = *inf-last-ti dt t* ∧ *ack* (*Suc t*) = [*sending-data*]) ∧

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

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*:$\forall x \leq d + d.\ ack\ (t + 3 + k + x) = [sending\text{-}data]$
shows $vc\ (2 * d + (t + (4 + k))) = [vc\text{-}com]$
⟨*proof*⟩

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

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

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*
⟨*proof*⟩

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*:$ack\ t = [init\text{-}state]$

and *h9*:*req* (*Suc t*) = [*init*]
and *h10*:∀ *t1*≤*t*. *req t1* = []
and *h11*:∀ *m* ≤ *k* + *2*. *req* (*t* + *m*) ≠ [*send*]
and *h12*:*req* (*t* + *3* + *k*) = [*send*]
and *h13*:*inf-last-ti dt t* ≠ []
and *h14*:∀ *j*≤*2* ∗ *d* + (*4* + *k*). *lose* (*t* + *j*) = [*False*]
shows *vc* (*2* ∗ *d* + (*t* + (*4* + *k*))) = [*vc-com*]
⟨*proof*⟩

lemma *GatewaySystem-L3*:
assumes *h1*:*Gateway req dt a stop lose d ack i vc*
and *h2*:*ServiceCenter i a*
and *h3*:*GatewayReq req dt a stop lose d ack i vc*
and *h4*:*msg* (*Suc 0*) *req*
and *h5*:*msg* (*Suc 0*) *stop*
and *h6*:*msg* (*Suc 0*) *a*
and *h7*:*ts lose*
and *h8*: *dt* (*Suc t*) ≠ [] ∨ *dt* (*Suc* (*Suc t*)) ≠ []
and *h9*: *ack t* = [*init-state*]
and *h10*:*req* (*Suc t*) = [*init*]
and *h11*:∀ *t1*≤*t*. *req t1* = []
and *h12*:∀ *m* ≤ *k* + *2*. *req* (*t* + *m*) ≠ [*send*]
and *h13*:*req* (*t* + *3* + *k*) = [*send*]
and *h14*:∀ *j*≤*2* ∗ *d* + (*4* + *k*). *lose* (*t* + *j*) = [*False*]
shows *vc* (*2* ∗ *d* + (*t* + (*4* + *k*))) = [*vc-com*]
⟨*proof*⟩

15.12 Proof of the Refinement for the Gateway System

lemma *GatewaySystem-L0*:
assumes *GatewaySystem req dt stop lose d ack vc*
shows *GatewaySystemReq req dt stop lose d ack vc*
⟨*proof*⟩

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.