Stream processing components: 
Isabelle/HOL formalisation and case studies

Maria Spichkova
August 16, 2018

Abstract

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

Contents

1 Introduction 4
   1.1 Stream processing components . . . . . . . . . . . . . . 4
   1.2 Case Study 1: Steam Boiler System . . . . . . . . . . . . 5
   1.3 Case Study 2: FlexRay Communication Protocol . . . . . 7
   1.4 Case Study 3: Automotive-Gateway . . . . . . . . . . . . 10

2 Theory ArithExtras.thy 15

3 Auxiliary Theory ListExtras.thy 15

4 Auxiliary arithmetic lemmas 18

5 FOCUS streams: operators and lemmas 20
   5.1 Definition of the FOCUS stream types . . . . . . . . . . 20
   5.2 Definitions of operators . . . . . . . . . . . . . . . . . . 20
   5.3 Properties of operators . . . . . . . . . . . . . . . . . . 32
      5.3.1 Lemmas for concatenation operator . . . . . . . . . 32
      5.3.2 Lemmas for operators ts and msg . . . . . . . . . 33
      5.3.3 Lemmas for inf_truncate . . . . . . . . . . . . . . 34
15.6 Properties of the ServiceCenter component . . . . . . . . . . 75
15.7 General properties of stream values . . . . . . . . . . . . . . 75
15.8 Properties of the Gateway . . . . . . . . . . . . . . . . . . 77
15.9 Proof of the Refinement Relation for the Gateway Requirements 80
15.10 Lemmas about Gateway Requirements . . . . . . . . . . . . 80
15.11 Properties of the Gateway System . . . . . . . . . . . . . . 81
15.12 Proof of the Refinement for the Gateway System . . . . . . . 83
1 Introduction

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

1.1 Stream processing components

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

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

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

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

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

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

- Finite timed streams of type ‘a are represented by the type ‘a 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 fulfill 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 the 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 \(\text{ControlSystemArch}\) is a refinement of the specification \(\text{ControlSystem}\). It follows from the definition of behavioral refinement that in order to verify that \(\text{ControlSystem} \sim \text{ControlSystemArch}\) it is enough to prove that

\[
\llbracket \text{ControlSystemArch} \rrbracket \Rightarrow \llbracket \text{ControlSystem} \rrbracket
\]

Therefore, we have to prove a lemma that says the specification \(\text{ControlSystemArch}\) is a refinement of the specification \(\text{ControlSystem}\):}

**Lemma** \(L_0\)-ControlSystem: \(\llbracket \text{ControlSystemArch} \rrbracket \Rightarrow \llbracket \text{ControlSystem} \rrbracket \Rightarrow \text{ControlSystem} s \)
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 the formalisation.
5. The system contains only one FlexRay channel. Adding a second channel would mean simply doubling the FlexRay component with a different configuration and adding extra channels for the access to the CNL_Buffer component.

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

- FlexRay (general requirements specification),
- FlexRayArch (system architecture),
- FlexRayArchitecture (guarantee part of the system architecture),
Cable, Controller, Scheduler, and BusInterface.

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

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

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

\[
\text{type } \text{Message} = \text{msg} (\text{message}_\text{id} : \mathbb{N}, \text{ftcdata} : \text{Data})
\]

\[
\text{type } \text{Frame} = \text{frm} (\text{slot} : \mathbb{N}, \text{data} : \text{Data})
\]

\[
\text{type } \text{Config} = \text{conf} (\text{schedule} : \mathbb{N}^*, \text{cycleLength} : \mathbb{N})
\]

We do not specify the type Data here to have a polymorphic specification of FlexRay (this type can be underspecified later to any datatype), therefore, in Isabelle/HOL it will be also defined as a polymorphic type $\forall a$. The types $\forall a$ nFrame, nNat and nConfig are used to represent sheaves of channels of types Frame, N and Config respectively. In the specification group will be used channels recv and activations, as well as sheaves of channels $(\text{return}_1, \ldots, \text{return}_n), (c_1, \ldots, c_n), (\text{store}_1, \ldots, \text{store}_n), (\text{get}_1, \ldots, \text{get}_n)$, and $(\text{send}_1, \ldots, \text{send}_n)$. We also need to declare some constant, $sN$, for the number of specification replication and the corresponding number of channels in sheaves, as well as to define the list of sheaf upper bounds, sheafNumbers.

The architecture of the FlexRay communication protocol is specified as the Focus specification FlexRayArch. Its assumption-part consists of three constraints: (i) all bus configurations have disjoint scheduling tables, (ii) all bus configurations have the equal length of the communication round, (iii) each FlexRay controller can receive at most one data frame each time interval from the environment of the FlexRay system. The guarantee-part of FlexRayArch is represented by the specification FlexRayArchitecture (see below).
The component *Cable* simulate the broadcast properties of the physical network cable – every received FlexRay frame is resent to all connected nodes. Thus, if one *FlexRayController* send some frame, this frame will be resent to all nodes (to all *FlexRayControllers* of the system). The assumption is that all input streams of the component *Cable* are disjoint – this holds by the properties of the *FlexRayController* components and the overall system assumption that the scheduling tables of all nodes are disjoint. The guarantee is specified by the predicate *Broadcast*.

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

The specification FlexRay represents requirements on the protocol: If the scheduling tables are correct in terms of the predicates DisjointSchedules (all bus configurations have disjoint scheduling tables) and IdenticalCycleLength (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 ≤ i ≤ 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 at 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 ≤ i ≤ 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 ～ FlexRayArch it is enough to prove that

\[ \text{[FlexRayArch]} \Rightarrow \text{[FlexRay]} \]

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 ⇒ 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.

\[
\begin{align*}
\text{type } \text{Coordinates} &= \mathbb{N} \times \mathbb{N} \\
\text{type } \text{CollisionSpeed} &= \mathbb{N} \\
\text{type } \text{ECall_Info} &= \text{ecall}(\text{coord} \in \text{Coordinates}, \text{speed} \in \text{CollisionSpeed}) \\
\text{type } \text{GatewayStatus} &= \{ \text{init\_state}, \text{call}, \text{connection\_ok}, \\
&\quad \text{sending\_data}, \text{voice\_com} \}
\end{align*}
\]

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

The specification \textit{GatewaySystemReq} specifies the requirements for the component \textit{GatewaySystem}: Assuming that the input streams \textit{req} and \textit{stop} can contain at every time interval at most one message, and assuming that the stream \textit{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 \textit{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 \textit{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 (\textit{GatewayReq}) as follows:

1. If at time \( t \) the gateway is in the initial state \textit{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, \textit{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 $i$ 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 (TSTD) and an expression which says how the local variable buffer is computed, or using the corresponding table representation, which is semantically equivalent to the TSTD.

To show that the specified gateway architecture fulfils the requirements we need to show that the specification Gateway is a refinement of the specification GatewayReq. Therefore, we need to define and to prove the following
Figure 1: Timed state transition diagram for the component Sample

lemma:

**lemma** Gateway-L0:

\[ \text{Gateway req dt a stop lose d ack i vc} \quad \Rightarrow \quad \text{GatewayReq req dt a stop lose d ack i vc} \]

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:

\[ \text{GatewaySystem req dt stop lose d ack vc} \quad \Rightarrow \quad \text{GatewaySystemReq req dt stop lose d ack vc} \]
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
lemma set-inter-mem:
assumes x mem l1
and x mem l2
shows set l1 ∩ set l2 ≠ {}
(\textit{proof})

lemma mem-notdisjoint:
assumes x mem l1
and x mem l2
shows ¬ disjoint l1 l2
(\textit{proof})

lemma mem-notdisjoint2:
assumes h1: disjoint (schedule A) (schedule B)
and h2: x mem schedule A
shows ¬ x mem schedule B
(\textit{proof})

lemma Add-Less:
assumes 0 < b
shows (Suc a - b < Suc a) = True
(\textit{proof})

lemma list-length-hint1:
assumes l ≠ []
shows 0 < length l
(\textit{proof})

lemma list-length-hint1a:
assumes l ≠ []
shows 0 < length l
(\textit{proof})

lemma list-length-hint2:
assumes length x = Suc 0
shows [hd x] = x
(\textit{proof})

lemma list-length-hint2a:
assumes length l = Suc 0
shows tl l = []
(\textit{proof})

lemma list-length-hint3:
assumes length l = Suc 0
shows l ≠ []
(\textit{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 \(\neg i - \text{length} \; x < \text{Suc} \; \text{(length} \; y)\)
  \(\text{and} \; \neg i - \text{Suc} \; \text{(length} \; x) < \text{Suc} \; \text{(length} \; y)\)
  shows \(\neg i < \text{Suc} \; \text{(length} \; x + \text{length} \; y)\)
  ⟨proof⟩

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

lemma list-nth-append7:
  assumes \(i - \text{length} \; x < \text{Suc} \; \text{(length} \; y)\)
  \(\text{and} \; i - \text{Suc} \; \text{(length} \; x) < \text{Suc} \; \text{(length} \; y)\)
  shows \(i < \text{Suc} \; \text{(Suc} \; \text{(length} \; x + \text{length} \; y))\)
  ⟨proof⟩

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

lemma list-nth-append9:
  assumes \(i - \text{Suc} \; \text{(length} \; x) < \text{Suc} \; \text{(length} \; y)\)
  shows \(i < \text{Suc} \; \text{(Suc} \; \text{(length} \; x + \text{length} \; y))\)
  ⟨proof⟩

lemma list-nth-append10:
  assumes \(\neg i < \text{Suc} \; \text{(length} \; x)\)
  \(\text{and} \; \neg i - \text{Suc} \; \text{(length} \; x) < \text{Suc} \; \text{(length} \; y)\)
  shows \(\neg i < \text{Suc} \; \text{(Suc} \; \text{(length} \; x + \text{length} \; y))\)
  ⟨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-mull-neq-nzero1:
  fixes i :: nat
  assumes i < n
      and 0 < i
  shows i + n * t ≠ n * q
⟨proof⟩

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

lemma arith-mull-neq-nzero3:
  fixes i :: nat
  assumes i < n
      and 0 < i
  shows n + n * t + i ≠ n * q
⟨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

— Finite timed FOCUS stream
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

— domain of an infinite untimed stream
definition
  infU-dom :: natInf set
where
  infU-dom ≡ {x. ∃ i. x = (Fin i)} ∪ {∞}

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

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

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

— range of a finite untimed stream

definition
\[ \text{finU-range :: } 'a \text{ list } \Rightarrow 'a \text{ set} \]
where
\[ \text{finU-range } x \equiv \text{set } x \]

— range of an infinite untimed stream

definition
\[ \text{infT-range :: } 'a \text{ istream } \Rightarrow 'a \text{ set} \]
where
\[ \text{infT-range } s \equiv \{ y. \exists \, i :: \text{nat}. \ y \ \text{mem} \ (s \ i) \} \]

— range of a finite untimed stream

definition
\[ \text{infU-range :: } (\text{nat } \Rightarrow 'a) \Rightarrow 'a \text{ set} \]
where
\[ \text{infU-range } s \equiv \{ \ y. \exists \, i :: \text{nat}. \ y = (s \ i) \ \} \]

— range of a (general) stream

definition
\[ \text{stream-range :: } 'a \text{ stream } \Rightarrow 'a \text{ set} \]
where
\[ \text{stream-range } s \equiv \text{case } s \text{ of} \]
\[ \quad \text{FinT } x \Rightarrow \text{finT-range } x \]
\[ \quad | \text{FinU } x \Rightarrow \text{finU-range } x \]
\[ \quad | \text{InfT } x \Rightarrow \text{infT-range } x \]
\[ \quad | \text{InfU } x \Rightarrow \text{infU-range } x \]

— finite timed stream that consists of n empty time intervals

primrec
\[ \text{nticks :: } \text{nat } \Rightarrow 'a \text{ fstream} \]
where
\[ \text{nticks } 0 = [] \ | \]
\[ \text{nticks } (\text{Suc } i) = [] \ # \ (\text{nticks } i) \]

— 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
\[ \text{inf-tl :: } (\text{nat } \Rightarrow 'a) \Rightarrow (\text{nat } \Rightarrow 'a) \]
where
\[ \text{inf-tl } s \equiv (\lambda \ i. \ s \ (\text{Suc } i)) \]

— 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**

```haskell
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
— n is of type of natural numbers enriched by Infinity

**primrec**

```haskell
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

**definition**

```haskell
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
```

— concatenation of a finite and an infinite stream

**definition**

```haskell
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**

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

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

**definition**
msg :: nat ⇒ 'a istream ⇒ bool
where
msg n s ≡ ∀ t. length (s t) ≤ n
— insuring that each time interval of a finite timed stream contains at most n data elements

primrec
fin-msg :: nat ⇒ 'a list list ⇒ bool
where
fin-msg n [] = True |
fin-msg n (x#xs) = (((length x) ≤ n) ∧ (fin-msg n xs))
— making a finite timed stream to a finite untimed stream

definition
fin-make-untimed :: 'a fstream ⇒ 'a list
where
fin-make-untimed x ≡ concat x
— making an infinite timed stream to an infinite untimed stream
— (auxiliary function)

primrec
inf-make-untimed1 :: 'a istream ⇒ nat ⇒ 'a
where
inf-make-untimed1-0:
inf-make-untimed1 s 0 =  hd (LEAST i.(s i) ≠ []) |
inf-make-untimed1-Suc:
inf-make-untimed1 s (Suc k) =
( if ((Suc k) < length (s 0))
then nth (s 0) (Suc k)
else ( if (s 0) = []
then (inf-make-untimed1 (inf-tl (inf-drop (LEAST i. ∀ j. j < i → (s j) = [])) s)) k )
else inf-make-untimed1 (inf-tl s) k ))
— making an infinite timed stream to an infinite untimed stream
— (main function)
definition
inf-make-untimed :: 'a istream ⇒ (nat ⇒ 'a)
where
inf-make-untimed s ≡
λ i. inf-make-untimed1 s i
— making a (general) stream untimed
definition
make-untimed :: 'a stream ⇒ 'a stream
where
make-untimed s ≡
\[
\begin{align*}
&\text{case } s \text{ of } (\text{FinT } x) \Rightarrow \text{FinU } (\text{fin-make-untimed } x) \\
&\quad | \ (\text{FinU } x) \Rightarrow \text{FinU } x \\
&\quad | \ (\text{InfT } x) \Rightarrow \\
&\quad \quad \text{(if } (\exists \ i. \forall \ j. \ i < j \rightarrow (x j) = []) \text{ then } \text{FinU } (\text{fin-make-untimed } (\text{inf-truncate } x) \\
&\quad \quad \quad (\text{LEAST } i. \forall \ j. \ i < j \rightarrow (x j) = []))) \text{ else } \text{InfU } (\text{inf-make-untimed } x)) \\
&\quad | \ (\text{InfU } x) \Rightarrow \text{InfU } x
\end{align*}
\]

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

\textbf{primrec} \ 
\begin{align*}
\textit{fin-tm} :: \ 'a \ fstream & \Rightarrow \ \textit{nat} \Rightarrow \textit{nat} \\
\textbf{where} \\
\textit{fin-tm} \ [] \ k = k & | \\
\textit{fin-tm} \ (x \# xs) \ k = \ \\
\quad (\text{if } k = 0 & \text{ then } 0 \\
\quad & \text{ else } (\text{if } (k \leq \text{length } x) \text{ then } (\text{Suc } 0) \\
\quad & \text{ else } \text{Suc}(\text{fin-tm } xs \ (k - \text{length } x))))
\end{align*}

— auxiliary lemma for the definition of the truncate operator

\textbf{lemma} \ \textit{inf-tm-hint1}: \ 
\begin{align*}
\textbf{assumes} \ i2 = \text{Suc } i - \text{length } a & \quad \text{and } \neg \text{Suc } i \leq \text{length } a \\
\text{and } a \neq [] & \\
\textbf{shows} \ i2 < \text{Suc } i
\end{align*}

\textbf{definition} \ \textit{finT-filter} :: \ 'a \ set => \ 'a \ fstream => \ 'a \ fstream \\
\textbf{where} \\
\textit{finT-filter } m \ s \equiv \ \text{map } (\lambda s. \ \text{filter } (\lambda y. \ y \in m) \ s) \ s

— filtering an infinite timed stream

\textbf{definition} \ \textit{infT-filter} :: \ 'a \ set => \ 'a \ istream => \ 'a \ istream \\
\textbf{where} \\
\textit{infT-filter } m \ s \equiv (\lambda i. \ (\text{filter } (\lambda x. \ x \in m) \ (s \ i)))

— removing duplications from a finite timed stream

\textbf{definition} \ \textit{finT-remdups} :: \ 'a \ fstream => \ 'a \ fstream \\
\textbf{where} \\
\textit{finT-remdups } s \equiv \ \text{map } (\lambda s. \ \text{remdups } s) \ s

— removing duplications from an infinite timed stream

\textbf{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).

28
\[ i < n \land j < n \land i \neq j \land ((nS i) t) \neq \[] \rightarrow
\]
\[ ((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 \times nat) \Rightarrow 'a fstream**

**where**

\[
\begin{align*}
\text{fin-get-prefix}([], n) &= [] | \\
\text{fin-get-prefix}(x#xs, i) &= \\
&\quad (\text{if} \ (\text{length} \ x) < i \\
&\quad \text{then} \ x \# \text{fin-get-prefix}(xs, (i - (\text{length} \ x))) \\
&\quad \text{else} \ \text{[take} \ i \ x] ) \\
\end{align*}
\]

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

**definition**

**fun fin-get-prefix-plus :: 'a fstream \Rightarrow natInf \Rightarrow 'a fstream**

**where**

\[
\text{fin-get-prefix-plus} \ s n \equiv \\
\begin{cases} \\
\text{case} \ n \text{ of} \ (\text{Fin} \ i) \Rightarrow \text{fin-get-prefix}(s, i) \\
\infty \Rightarrow s \\
\end{cases}
\]

— auxiliary lemmas

**lemma length-inf-drop-hint1:**

**assumes** \( s k \neq \[] \)

**shows** \( \text{length} \ (\text{inf-drop} \ k \ s \ 0) \neq 0 \)

(proof)

**lemma length-inf-drop-hint2:**

\[
\begin{align*}
(s 0 \neq \[] & \rightarrow \text{length} \ (\text{inf-drop} \ 0 \ s \ 0) < \text{Suc} \ i \\
& \rightarrow \text{Suc} \ i - \text{length} \ (\text{inf-drop} \ 0 \ s \ 0) < \text{Suc} \ i) \\
\end{align*}
\]

(proof)

**fun infT-get-prefix :: ('a istream \times nat) \Rightarrow 'a fstream**

**where**

\[
\begin{align*}
\text{infT-get-prefix}(s, 0) &= [] | \\
\text{infT-get-prefix}(s, \text{Suc} \ i) &= \\
&\quad (\text{if} \ (s 0) = [] \\
&\quad \quad \text{then} \ (\text{if} \ (\forall \ i. \ s i = [])) \\
&\quad \quad \text{then} \ [] \\
&\quad \quad \text{else} \ (\text{let} \\
&\quad \quad \quad k = (\text{LEAST} \ k. \ s k \neq [] \land (\forall \ i. \ i < k \rightarrow s i = [])); \\
&\quad \quad \quad s2 = \text{inf-drop} \ (k+1) \ s \\
&\quad \quad \quad \text{in} \ (\text{if} \ (\text{length} \ (s k)=0) \\
&\quad \quad \quad \quad \text{then} \ [] \\
&\quad \quad \quad \quad \text{else} \ (\text{if} \ (\text{length} \ (s k) < (\text{Suc} \ i)) \\
&\quad \quad \quad \quad \quad \text{then} \ s k \neq \text{infT-get-prefix} \ (s2, \text{Suc} \ i - \text{length} \ (s k)) \\
&\quad \quad \quad \quad \quad \text{else} \ [\text{take} \ (\text{Suc} \ i) \ (s k)]) ) \\
\end{align*}
\]

29
```haskell
else
  (if (Suc i) < (length s)
      then (s) ≠ infT-get-prefix (inf-drop 1 s, (Suc i) - (length s))
      else [take (Suc i) (s)]
  )
)

— 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 |
infinf-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 ≠ []
sows inf-last-ti dt (t + k) ≠ []
⟨proof⟩

lemma inf-last-ti-nonempty:
assumes s t ≠ []
sows inf-last-ti s (t + k) ≠ []
⟨proof⟩

lemma arith-sum-t2k:
\[ t + 2 + k = (\text{Suc } t) + (\text{Suc } k) \]
⟨proof⟩

lemma inf-last-ti-Suc2:
assumes dt (Suc t) ≠ [] ∨ dt (Suc (Suc t)) ≠ []
sows inf-last-ti dt (t + 2 + k) ≠ []
⟨proof⟩

5.3.1 Lemmas for concatenation operator

lemma fin-length-append:
\[ \text{fin-length} (x \bullet y) = (\text{fin-length } x) + (\text{fin-length } y) \]
⟨proof⟩

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

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

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

lemma fin-append-com-nil1:
\[ \text{fin-inf-append } [] (\text{fin-inf-append } y z) = \text{fin-inf-append } ([] \bullet y) z \]
⟨proof⟩

lemma fin-append-com-nil2:
\[ \text{fin-inf-append } x (\text{fin-inf-append } [] z) = \text{fin-inf-append } (x \bullet []) z \]
⟨proof⟩

lemma fin-append-com-i:
\( \text{fin-inf-append } x (\text{fin-inf-append } y z) i = \text{fin-inf-append } (x \cdot y) z i \)  
(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 \( \text{length } (x i) = \text{Suc } (0::\text{nat}) \)  
(proof)

**lemma** ts-Least-0:  
assumes ts x  
shows \( (\text{LEAST } i. \ (x i) \neq []) = (0::\text{nat}) \)  
(proof)

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

**lemma** ts-Least-Suc0:  
assumes ts x  
shows \( (\text{LEAST } i. \ (\text{Suc } i) \neq []) = 0 \)  
(proof)

**lemma** ts-inf-make-untimed-inf-tl:  
assumes ts x  
shows \( \text{inf-make-untimed } (\text{inf-tl } x) i = \text{inf-make-untimed } x (\text{Suc } i) \)  
(proof)

**lemma** ts-inf-make-untimed1-inf-tl:  
assumes ts x  
shows \( \text{inf-make-untimed1 } (\text{inf-tl } x) i = \text{inf-make-untimed1 } x (\text{Suc } i) \)  
(proof)

**lemma** msg-nonempty1:
assumes \(h1: \text{msg} \ (\text{Suc} \ 0) \ a\)
and \(h2: a \ t = aa \neq l\)
shows \(l = []\)
(proof)

lemma \(\text{msg-nonempty2}:\)
assumes \(h1: \text{msg} \ (\text{Suc} \ 0) \ a\)
and \(h2: a \ t \neq []\)
shows \(\text{length} \ (a \ t) = (\text{Suc} \ 0)\)
(proof)

5.3.3 Lemmas for \(\text{inf_truncate}\)

lemma \(\text{inf-truncate-nonempty}:\)
assumes \(z \ i \neq []\)
shows \(\text{inf-truncate} \ z \ i \neq []\)
(proof)

lemma \(\text{concat-inf-truncate-nonempty}:\)
assumes \(z \ i \neq []\)
shows \(\text{concat} \ (\text{inf-truncate} \ z \ i) \neq []\)
(proof)

lemma \(\text{concat-inf-truncate-nonempty-a}:\)
assumes \(z \ i = [a]\)
shows \(\text{concat} \ (\text{inf-truncate} \ z \ i) \neq []\)
(proof)

lemma \(\text{concat-inf-truncate-nonempty-el}:\)
assumes \(z \ i \neq []\)
shows \(\text{concat} \ (\text{inf-truncate} \ z \ i) \neq []\)
(proof)

lemma \(\text{inf-truncate-append}:\)
\((\text{inf-truncate} \ z \ i \bullet [z \ (\text{Suc} \ i)]) = \text{inf-truncate} \ z \ (\text{Suc} \ i)\)
(proof)

5.3.4 Lemmas for \(\text{fin_make_untimed}\)

lemma \(\text{fin-make-untimed-append}:\)
assumes \(\text{fin-make-untimed} \ x \neq []\)
shows \(\text{fin-make-untimed} \ (x \bullet y) \neq []\)
(proof)

lemma \(\text{fin-make-untimed-inf-truncate-Nonempty}:\)
assumes \(z \ k \neq []\)
and \(k \leq i\)
shows \(\text{fin-make-untimed} \ (\text{inf-truncate} \ z \ i) \neq []\)

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-make-untimed (inf-truncate z i)) = a
(proof)

lemma fin-make-untimed-append-empty:
  fin-make-untimed (z • [[]]) = fin-make-untimed z
(proof)

lemma fin-make-untimed-inf-truncate-append-a:
  fin-make-untimed (inf-truncate z i • [[a]]) !
  (length (fin-make-untimed (inf-truncate z i • [[a]])) − Suc 0) = a
(proof)

lemma fin-make-untimed-inf-truncate-Nonempty-all:
  assumes z k = [[]]
  shows ∀ i. k ≤ i → fin-make-untimed (inf-truncate z i) ≠ []
(proof)

lemma fin-make-untimed-inf-truncate-Nonempty-all0:
  assumes z 0 = []
  shows ∀ i. fin-make-untimed (inf-truncate z i) ≠ []
(proof)

lemma fin-make-untimed-inf-truncate-Nonempty-all0a:
  assumes z 0 = [a]
  shows ∀ i. fin-make-untimed (inf-truncate z i) ≠ []
(proof)

lemma fin-make-untimed-inf-truncate-Nonempty-all-app:
  assumes z 0 = [a]
  shows ∀ i. fin-make-untimed (inf-truncate z i • [z (Suc i)]) ≠ []
(proof)

lemma fin-make-untimed-nth-length:
  assumes z i = [a]
  shows fin-make-untimed (inf-truncate z i) !
  (length (fin-make-untimed (inf-truncate z i)) − Suc 0)
  = a
(proof)
5.3.5 Lemmas for inf_disj and inf_disjS

lemma inf-disj-index:
  assumes h1:inf-disj n nS
  and nS k t ≠ []
  and k < n
  shows (SOME i. i < n ∧ nS i t ≠ []) = k
 ⟨proof⟩

lemma inf-disjS-index:
  assumes h1:inf-disjS IdSet nS
  and k:IdSet
  and nS k t ≠ []
  shows (SOME i. (i:IdSet) ∧ nSend i t ≠ []) = k
 ⟨proof⟩

end

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

theory BitBoolTS
imports Main stream
begin

datatype bit = Zero | One

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

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

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

lemma ts-bool-True:
  assumes h1:ts x
  and h2:x i ≠ [False]

36
shows $x \cdot i = [True]$
(proof)

lemma ts-bool-False:
assumes h1:ts $x$
and h2:$x \cdot i \neq [True]$
shows $x \cdot i = [False]$
(proof)

lemma ts-bool-True-False:
fixes $x :: bool$ istream
assumes ts $x$
shows $x \cdot i = [True] \lor x \cdot i = [False]$
(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 $s\cdot x\cdot 0 = s\cdot x$
join-ti-Suc:
join-ti $s\cdot x\cdot (\operatorname{Suc}\cdot i) = (\operatorname{join-ti}\cdot s\cdot x\cdot i) \cdot (s\cdot (x + (\operatorname{Suc}\cdot i)))$

primrec fin-join-ti :: 'a fstream ⇒ nat ⇒ nat ⇒ 'a list
where
fin-join-ti-0:
fin-join-ti $s\cdot x\cdot 0 = \operatorname{nth}\cdot s\cdot x$
fin-join-ti-Suc:
fin-join-ti $s\cdot x\cdot (\operatorname{Suc}\cdot i) = (\operatorname{fin-join-ti}\cdot s\cdot x\cdot i) \cdot (\operatorname{nth}\cdot s\cdot (x + (\operatorname{Suc}\cdot i)))$

definition join-time :: 'a istream ⇒ nat ⇒ 'a istream
where
join-time $s\cdot n\cdot t\equiv$
(case $n$ of
\(0\ ⇒ []\)
|(Suc $i$) ⇒ join-ti $s\cdot (n\cdot t)\cdot i$)
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 fin-join-ti-hint1:
  assumes fin-join-ti s x (Suc i) = []
  shows  fin-join-ti s x i = []
⟨proof⟩

lemma join-time1: ∀ 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⟩

38
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 ≤ n
      and fin-join-ti s x n = []
  shows  nth s (x+i) = []
⟨proof⟩

lemma fin-join-ti-empty-ti:
  assumes ∀ i ≤ n. nth s (x+i) = []
  shows  fin-join-ti s x n = []
⟨proof⟩

lemma fin-join-ti-1nempty:
  assumes ∀ i. 0 < i ∧ i < Suc n → nth s (x+i) = []
  shows  fin-join-ti s x n = nth s x
⟨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: ∀ t. split-time s 1 t = s t
⟨proof⟩

lemma split-time1: split-time s 1 t = s t
⟨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 \( \text{split-time } s \ n \ (n \ast t) = s \ t \)

\(\langle \text{proof} \rangle\)

lemma \(\text{split-time-nempty-Suc}\):
assumes \(\emptyset < n\)
shows \(\text{split-time } s \ (\text{Suc } n) \ ((\text{Suc } n) \ast t) = \text{split-time } s \ n \ (n \ast t)\)

\(\langle \text{proof} \rangle\)

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

\(\langle \text{proof} \rangle\)

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

\(\langle \text{proof} \rangle\)

lemma \(\text{split-time-hint1}\):
assumes \(n = \text{Suc } m\)
shows \(\text{split-time } s \ (\text{Suc } n) \ (i + n \ast i + n) = []\)

\(\langle \text{proof} \rangle\)

\(\text{7.3 Duality of the split and the join operators}\)

lemma \(\text{join-split-i}\):
assumes \(\emptyset < n\)
shows \(\text{join-time } (\text{split-time } s \ n) \ n \ i = s \ i\)

\(\langle \text{proof} \rangle\)

lemma \(\text{join-split}\):
assumes \(\emptyset < n\)
shows \(\text{join-time } (\text{split-time } s \ n) \ n = s\)

\(\langle \text{proof} \rangle\)

end

\(\text{8 Steam Boiler System: Specification}\)

theory \(\text{SteamBoiler}\)
imports \(\text{stream BitBoolTS}\)
begin

definition \(\text{stream BitBoolTS}\)
where
\(\text{ControlSystem } :: \text{nat istream } \Rightarrow \text{bool}\)

\(\text{ControlSystem } s \equiv\)
\((\text{ts } s) \ \land\)
(\forall (j::\text{n}at). (200::\text{n}at) \leq \text{hd} (s \ j) \wedge \text{hd} (s \ j) \leq (800::\text{n}at))

definition
SteamBoiler :: bit istream \Rightarrow \text{n}at istream \Rightarrow \text{n}at istream \Rightarrow \text{bool}
where
SteamBoiler x s y \equiv
ts x
\longrightarrow
((ts \ y) \wedge (ts \ s) \wedge (y = s) \wedge
(s \ 0) = [500::\text{n}at]) \wedge
(\forall (j::\text{n}at). (\exists (r::\text{n}at).
(0::\text{n}at) < r \wedge r \leq (10::\text{n}at) \wedge
\text{hd} (s \ (\text{Suc} \ j)) =
(if \ \text{hd} (x \ j) = \text{Zero}
then (\text{hd} (s \ j)) - r
else (\text{hd} (s \ j)) + r)) ))

definition
Converter :: bit istream \Rightarrow bit istream \Rightarrow \text{bool}
where
Converter z x \equiv
(t x)
\wedge
(\forall (t::\text{n}at).
\text{hd} \ (x \ t) =
(if \ (\text{fin-make-untimed} \ (\text{inf-truncate} \ z \ t) = [\ ])
then 0
else (\text{fin-make-untimed} \ (\text{inf-truncate} \ z \ t)) !
((\text{length} \ (\text{fin-make-untimed} \ (\text{inf-truncate} \ z \ t))) - (1::\text{n}at))
))

definition
Controller-L ::
\text{n}at istream \Rightarrow \text{bit i}ustream \Rightarrow \text{bit i}ustream \Rightarrow \text{bit istream} \Rightarrow \text{bool}
where
Controller-L y lIn lOut z \equiv
(z \ 0 = [\text{Zero}])
\wedge
(\forall (t::\text{n}at).
(if \ (lIn \ t) = \text{Zero}
then (if \ 300 < \text{hd} (y \ t)
then (z \ t) = [\ ] \wedge (lOut \ t) = \text{Zero}
else (z \ t) = [\text{One}] \wedge (lOut \ t) = \text{One}
)
else (if \ \text{hd} (y \ t) < 700
else (if \ (lOut \ t) = \text{One}
then (z \ t) = [\ ] \wedge (lOut \ t) = \text{Zero}
else (z \ t) = [\text{One}] \wedge (lOut \ t) = \text{One}
)\)
then \( (z \ t) = [] \land (t\text{Out} \ t) = \text{One} \)
else \( (z \ t) = [\text{Zero}] \land (t\text{Out} \ t) = \text{Zero} \))}

**definition**

**Controller ::** nat istream ⇒ bit istream ⇒ bool

**where**

**Controller y z**

\[ \equiv \]

\((ts \ y)\rightarrow (\exists \ l. \text{Controller-L y} \ (\text{fin-inf-append} \ [\text{Zero}] \ l) \ l \ z)\)

**definition**

**ControlSystemArch ::** nat istream ⇒ bool

**where**

**ControlSystemArch s**

\[ \equiv \]

\(\exists \ x z :: \text{bit istream}, \exists \ y :: \text{nat istream}.
\)

( SteamBoiler \ x \ s \ y \ \land \text{Controller y z} \ \land \text{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**

\(\text{SteamBoiler \ x \ s \ y} \ \land \ ts \ x\)

**shows**

\(ts \ s\)

(proof)

**lemma L2-Boiler:**

**assumes**

\(\text{SteamBoiler \ x \ s \ y} \ \land \ ts \ x\)

**shows**

\(ts \ y\)

(proof)

**lemma L3-Boiler:**

**assumes**

\(\text{SteamBoiler \ x \ s \ y} \ \land \ ts \ x\)

**shows**

\(200 \leq \text{hd}(s \ 0)\)

(proof)

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

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

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

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

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

\section{9.2 Properties of the Controller Component}

\textbf{lemma L1-Controller:}
assumes \(\text{Controller-L} \ s \ (\text{fin-inf-append} \ [\text{Zero}] \ l) \ l \ z\)
shows \(\text{fin-make-antimed} \ (\text{inf-truncate} \ z \ i) \neq []\)
\(\langle \text{proof} \rangle\)

\textbf{lemma L2-Controller-Zero:}
assumes \(\text{Controller-L} \ y \ (\text{fin-inf-append} \ [\text{Zero}] \ l) \ l \ z\)
and \(l \ t = \text{Zero}\)
and \(300 < \text{hd} (y \ (\text{Suc} \ t))\)
shows \(z \ (\text{Suc} \ t) = []\)
\(\langle \text{proof} \rangle\)

\textbf{lemma L2-Controller-One:}
assumes \(\text{Controller-L} \ y \ (\text{fin-inf-append} \ [\text{Zero}] \ l) \ l \ z\)
and \( t = One \)
and \( \text{hd} \ (y \ (\text{Suc} \ t)) < 700 \)
shows \( z \ (\text{Suc} \ t) = [] \)

(\text{proof})

**Lemma L3-Controller-Zero:**
assumes \( \text{Controller-L} \ y \ (\text{fin-inf-append} \ [\text{Zero}] \ l) \ l z \)
and \( t = \text{Zero} \)
and \( \neg 300 < \text{hd} \ (y \ (\text{Suc} \ t)) \)
shows \( z \ (\text{Suc} \ t) = [\text{One}] \)
(\text{proof})

**Lemma L3-Controller-One:**
assumes \( \text{Controller-L} \ y \ (\text{fin-inf-append} \ [\text{Zero}] \ l) \ l z \)
and \( t = \text{One} \)
and \( \neg \text{hd} \ (y \ (\text{Suc} \ t)) < 700 \)
shows \( z \ (\text{Suc} \ t) = [\text{Zero}] \)
(\text{proof})

**Lemma L4-Controller-Zero:**
assumes \( h1: \text{Controller-L} \ y \ (\text{fin-inf-append} \ [\text{Zero}] \ l) \ l z \)
and \( h2:l \ (\text{Suc} \ t) = \text{Zero} \)
shows \( (z \ (\text{Suc} \ t) = [] \land l t = \text{Zero}) \lor (z \ (\text{Suc} \ t) = [\text{Zero}] \land l t = \text{One}) \)
(\text{proof})

**Lemma L4-Controller-One:**
assumes \( h1: \text{Controller-L} \ y \ (\text{fin-inf-append} \ [\text{Zero}] \ l) \ l z \)
and \( h2:l \ (\text{Suc} \ t) = \text{One} \)
shows \( (z \ (\text{Suc} \ t) = [] \land l t = \text{One}) \lor (z \ (\text{Suc} \ t) = [\text{One}] \land l t = \text{Zero}) \)
(\text{proof})

**Lemma L5-Controller-Zero:**
assumes \( h1: \text{Controller-L} \ y \ \text{lIn} \ \text{lOut} \ z \)
and \( h2:lOut \ t = \text{Zero} \)
and \( h3:z \ t = [] \)
shows \( \text{lIn} \ t = \text{Zero} \)
(\text{proof})

**Lemma L5-Controller-One:**
assumes \( h1: \text{Controller-L} \ y \ \text{lIn} \ \text{lOut} \ z \)
and \( h2:lOut \ t = \text{One} \)
and \( h3:z \ t = [] \)
shows \( \text{lIn} \ t = \text{One} \)
(\text{proof})

**Lemma L5-Controller:**
assumes \( \text{Controller-L} \ y \ \text{lIn} \ \text{lOut} \ z \)
and \( \text{lOut} \ t = a \)
and $z \cdot t = []$
shows $l \cdot n \cdot t = a$
(proof)

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

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

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

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

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

lemma L7-Controller-One:
assumes $h1$ : Controller-L y (fin-inf-append [Zero] l) l z
and $h2$ : $l \cdot t = One$
and $h3$ : $y \cdot 0 = [500::nat]$
shows $last \cdot (fin-make-untimed \cdot (inf-truncate \cdot z \cdot t)) = One$
(proof)

lemma L7-Controller:
assumes Controller-L y (fin-inf-append [Zero] l) l z
and $y \cdot 0 = [500::nat]$
shows $last \cdot (fin-make-untimed \cdot (inf-truncate \cdot z \cdot t)) = l \cdot t$
(proof)
lemma L8-Controller:
\[\text{assumes } \text{Controller-L} y (\text{fin-inf-append} \ [\text{Zero}] \ l) \ l \ z\]
\[\text{shows } z \ t = \emptyset \lor z \ t = [\text{Zero}] \lor z \ t = [\text{One}]\]
\langle proof \rangle

lemma L9-Controller:
\[\text{assumes } h1: \text{Controller-L} s (\text{fin-inf-append} \ [\text{Zero}] \ l) \ l \ z\]
\[\text{and } h2: \text{fin-make-untimed} (\text{inf-truncate} z \ i) \neq \emptyset\]
\[\text{and } \text{length} (\text{fin-make-untimed} (\text{inf-truncate} z \ i)) - \text{Suc} 0 = \text{Zero}\]
\[\text{and } h3: \text{last} (\text{fin-make-untimed} (\text{inf-truncate} z \ i)) = l \ i\]
\[\text{and } h5: \text{hd} (s \ (\text{Suc} \ i)) = \text{hd} (s \ i) - r\]
\[\text{and } h6: \text{fin-make-untimed} (\text{inf-truncate} z \ i) \neq \emptyset\]
\[\text{and } h8: r \leq 10\]
\[\text{shows } 200 \leq \text{hd} (s \ (\text{Suc} \ i))\]
\langle proof \rangle

lemma L10-Controller:
\[\text{assumes } h1: \text{Controller-L} s (\text{fin-inf-append} \ [\text{Zero}] \ l) \ l \ z\]
\[\text{and } h2: \text{fin-make-untimed} (\text{inf-truncate} z \ i) \neq \emptyset\]
\[\text{and } \text{length} (\text{fin-make-untimed} (\text{inf-truncate} z \ i)) - \text{Suc} 0 \neq \text{Zero}\]
\[\text{and } h3: \text{last} (\text{fin-make-untimed} (\text{inf-truncate} z \ i)) = l \ i\]
\[\text{and } h5: \text{hd} (s \ (\text{Suc} \ i)) = \text{hd} (s \ i) + r\]
\[\text{and } h6: \text{fin-make-untimed} (\text{inf-truncate} z \ i) \neq \emptyset\]
\[\text{and } h8: r \leq 10\]
\[\text{shows } \text{hd} (s \ (\text{Suc} \ i)) \leq 800\]
\langle proof \rangle

9.3 Properties of the Converter Component

lemma L1-Converter:
\[\text{assumes } \text{Converter} z \ x\]
\[\text{and } \text{fin-make-untimed} (\text{inf-truncate} z \ t) \neq \emptyset\]
\[\text{shows } \text{hd} (x \ t) = (\text{fin-make-untimed} (\text{inf-truncate} z \ t)) !\]
\[((\text{length} (\text{fin-make-untimed} (\text{inf-truncate} z \ t))) - (1 :: \text{nat}))\]
\langle proof \rangle

lemma L1a-Converter:
\[\text{assumes } \text{Converter} z \ x\]
\[\text{and } \text{fin-make-untimed} (\text{inf-truncate} z \ t) \neq \emptyset\]
\[\text{and } \text{hd} (x \ t) = \text{Zero}\]
\[\text{shows } (\text{fin-make-untimed} (\text{inf-truncate} z \ t)) !\]
\[((\text{length} (\text{fin-make-untimed} (\text{inf-truncate} z \ t))) - (1 :: \text{nat}))
\]
\[= \text{Zero}\]
\langle proof \rangle

9.4 Properties of the System

lemma L1-ControlSystem:
\[\text{assumes } \text{ControlSystemArch} s\]
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
  ftdata :: ′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
— The predicate DisjointSchedules is true for sheaf of channels of type Config,
— if all bus configurations have disjoint scheduling tables.
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 at the tth 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)))
\[ (nGet \ k \ t) = \{s\} \land \\
(\forall \ j. \ j < n \land j \neq k \implies \\
((nStore \ j) \ t) = ((nReturn \ k) \ t)) \]

— The predicate \texttt{Broadcast} describes properties of FlexRay broadcast.

\textbf{definition}
\begin{align*}
\texttt{Broadcast} :: \\
nat \Rightarrow \text{'a nFrame} \Rightarrow \text{'a Frame istream} \Rightarrow \text{bool}
\end{align*}
where
\begin{align*}
\texttt{Broadcast} \ n \ nSend \ recv \\
\equiv \\
\forall \ (t::nat).
\begin{cases}
(\text{if } \exists \ k. \ k < n \land ((nSend \ k) \ t) \neq \tag{\{\}}) \\
\text{then } (recv \ t) = ((nSend \ \text{SOME} \ k. \ k < n \land ((nSend \ k) \ t) \neq \tag{\{\}}) \ t) \\
\text{else } (recv \ t) = \tag{\{\} })
\end{cases}
\end{align*}

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

\textbf{definition}
\begin{align*}
\texttt{Receive} :: \\
\text{'a Frame istream} \Rightarrow \text{'a Frame istream} \Rightarrow \text{nat istream} \Rightarrow \text{bool}
\end{align*}
where
\begin{align*}
\texttt{Receive} \ recv \ store \ activation \\
\equiv \\
\forall \ (t::nat).
\begin{cases}
(\text{if } (activation \ t) = \tag{\{\} }) \\
\text{then } (store \ t) = (recv \ t) \\
\text{else } (store \ t) = \tag{\{\} }
\end{cases}
\end{align*}

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

\textbf{definition}
\begin{align*}
\texttt{Send} :: \\
\text{'a Frame istream} \Rightarrow \text{'a Frame istream} \Rightarrow \text{nat istream} \Rightarrow \text{nat istream} \Rightarrow \text{bool}
\end{align*}
where
\begin{align*}
\texttt{Send} \ return \ send \ get \ activation \\
\equiv \\
\forall \ (t::nat).
\begin{cases}
(\text{if } (activation \ t) = \tag{\{\} }) \\
\text{then } (get \ t) = \tag{\{\} } \land (send \ t) = \tag{\{\} } \\
\text{else } (get \ t) = (activation \ t) \land (send \ t) = (return \ t) \}
\end{cases}
\end{align*}

\subsection{Specifications of the FlexRay components}

\textbf{definition}
\begin{align*}
\texttt{FlexRay} :: \\
nat \Rightarrow \text{'a nFrame} \Rightarrow \text{nConfig} \Rightarrow \text{'a nFrame} \Rightarrow \text{nNat} \Rightarrow \text{bool}
\end{align*}
where
\textit{FlexRay n nReturn nC nStore nGet} \\
\equiv \\
(\text{CorrectSheaf n}) \land \\
((\forall (i:\text{nat}). \ i < n \rightarrow (msg 1 (nReturn i)))) \land \\
(\text{DisjointSchedules n nC}) \land (\text{IdenticCycleLength n nC}) \\
\rightarrow \\
(\text{FrameTransmission n nStore nReturn nGet nC}) \land \\
(\forall (i:\text{nat}). \ i < n \rightarrow (msg 1 (nGet i)) \land (msg 1 (nStore i)))) \\
\}

definition
\textit{Cable} :: \text{nat} \Rightarrow \text{'a nFrame} \Rightarrow \text{'a Frame istream} \Rightarrow \text{bool} \\
where
\text{Cable} n \text{nSend recv} \\
\equiv \\
(\text{CorrectSheaf n}) \\
\land \\
((\text{inf-disj n nSend}) \rightarrow (\text{Broadcast n nSend recv})) \\
definition
\textit{Scheduler} :: \text{Config} \Rightarrow \text{nat istream} \Rightarrow \text{bool} \\
where
\text{Scheduler c activation} \\
\equiv \\
\forall (t:\text{nat}). \ \\
(\text{let s = (t mod (cycleLength c))} \\
in \\
(\text{if} (s \text{ mem (schedule c)}) \\
\text{then} \ (\text{activation t}) = [s] \\
\text{else} \ (\text{activation t}) = [] )) \\
definition
\textit{BusInterface} :: \\
\text{nat istream} \Rightarrow \text{'a Frame istream} \Rightarrow \text{'a Frame istream} \Rightarrow \\
\text{'a Frame istream} \Rightarrow \text{'a Frame istream} \Rightarrow \text{nat istream} \Rightarrow \text{bool} \\
where
\text{BusInterface activation return recv store send get} \\
\equiv \\
(\text{Receive recv store activation}) \land \\
(\text{Send return send get activation}) \\
definition
\textit{FlexRayController} :: \\
\text{'a Frame istream} \Rightarrow \text{'a Frame istream} \Rightarrow \text{Config} \Rightarrow \\
\text{'a Frame istream} \Rightarrow \text{'a Frame istream} \Rightarrow \text{nat istream} \Rightarrow \text{bool} \\
where
\text{FlexRayController return recv c store send get} \\
\equiv \\
(\exists \text{ activation}. \\
\text{'exists activation}
(Scheduler c activation) ∧
(BusInterface activation return recv store send get))

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

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

12 FlexRay: Verification

theory FR-proof
imports FR
begin

12.1 Properties of the function Send

lemma Send-L1:
assumes Send return send get activation
and send t ≠ []
shows (activation t) ≠ []
(proof)

lemma Send-L2:
assumes Send return send get activation
and (activation t) ≠ []
and return t ≠ []
shows \((send t) \neq []\)

\begin{proof}
\end{proof}

12.2 Properties of the component Scheduler

**Lemma Scheduler-L1:**
**Assumptions**
- \(h1: \text{Scheduler } C \text{ activation}\)
- \(h2: (activation t) \neq []\)

**Shows**
- \((t \mod (\text{cycleLength } C)) \text{ mem (schedule } C)\)

\begin{proof}
\end{proof}

**Lemma Scheduler-L2:**
**Assumptions**
- \(\text{Scheduler } C \text{ activation}\)
- \(\neg (t \mod \text{cycleLength } C) \text{ mem (schedule } C)\)

**Shows**
- \(activation t = []\)

\begin{proof}
\end{proof}

**Lemma Scheduler-L3:**
**Assumptions**
- \(\text{Scheduler } C \text{ activation}\)
- \((t \mod \text{cycleLength } C) \text{ mem (schedule } C)\)

**Shows**
- \(activation t \neq []\)

\begin{proof}
\end{proof}

**Lemma Scheduler-L4:**
**Assumptions**
- \(\text{Scheduler } C \text{ activation}\)
- \((t \mod \text{cycleLength } C) \text{ mem (schedule } C)\)

**Shows**
- \(activation t = [t \mod \text{cycleLength } C]\)

\begin{proof}
\end{proof}

**Lemma correct-DisjointSchedules1:**
**Assumptions**
- \(h1: \text{DisjointSchedules } n \ nC\)
- \(h2: \text{IdentcCycleLength } n \ nC\)
- \(h3: (t \mod \text{cycleLength } (nC i)) \text{ mem schedule } (nC i)\)
- \(h4: i < n\)
- \(h5: j < n\)
- \(h6: i \neq j\)

**Shows**
- \(\neg (t \mod \text{cycleLength } (nC j) \text{ mem schedule } (nC j))\)

\begin{proof}
\end{proof}

12.3 Disjoint Frames

**Lemma disjointFrame-L1:**
**Assumptions**
- \(h1: \text{DisjointSchedules } n \ nC\)
- \(h2: \text{IdentcCycleLength } n \ nC\)
- \(h3: \forall \ i < n. \text{FlexRayController } (n\text{Return } i) \text{ rcv } (nC i) (n\text{Store } i) (n\text{Send } i) (n\text{Get } i)\)
- \(h4: n\text{Send } i \ t \neq []\)
- \(h5: i < n\)
- \(h6: j < n\)
- \(h7: i \neq j\)
shows \( nSend j t = [] \)

(proof)

lemma disjointFrame-L2:
assumes DisjointSchedules \( n \) \( nC \)
and IdenticalCycleLength \( n \) \( nC \)
and \( \forall i < n. \text{FlexRayController} \ (nReturn i) \text{rcv} \ (nC i) \ (nStore i) \ (nSend i) \ (nGet i) \)
shows inf-disj \( n \) \( nSend \)
(proof)

lemma disjointFrame-L3:
assumes \( h1: \text{DisjointSchedules} \ (n \) \( nC \) \)
and \( h2: \text{IdenticalCycleLength} \ (n \) \( nC \) \)
and \( h3 : \forall i < n. \text{FlexRayController} \ (nReturn i) \text{rcv} \ (nC i) \ (nStore i) \ (nSend i) \ (nGet i) \)
and \( h4 : t \ mod \text{cycleLength} \ (nC i) \ \text{mem schedule} \ (nC i) \)
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 \( h1: \text{FlexRayController} \ (nReturn i) \text{rcv} \ (nC i) \ (nStore i) \ (nSend i) \ (nGet i) \)
and \( h1 : \neg (t \ mod \text{cycleLength} \ (nC i) \ \text{mem schedule} \ (nC i)) \)
shows \( (nSend i) \ t = [] \)
(proof)

lemma fr-Send2:
assumes \( h1 : \forall i < n. \text{FlexRayController} \ (nReturn i) \text{rcv} \ (nC i) \ (nStore i) \ (nSend i) \ (nGet i) \)
and \( h2 : \text{DisjointSchedules} \ n \ nC \)
and \( h3 : \text{IdenticalCycleLength} \ n \ nC \)
and \( h4 : t \ mod \text{cycleLength} \ (nC k) \ \text{mem schedule} \ (nC k) \)
and \( h5 : k < n \)
shows \( nSend k t = nReturn k t \)
(proof)

lemma fr-Send3:
assumes \( \forall i < n. \text{FlexRayController} \ (nReturn i) \text{rcv} \ (nC i) \ (nStore i) \ (nSend i) \ (nGet i) \)
and \( \text{DisjointSchedules} \ n \ nC \)
and \( \text{IdenticalCycleLength} \ n \ nC \)
and \( t \ mod \text{cycleLength} \ (nC k) \ \text{mem schedule} \ (nC k) \)

53
and \( k < n \)
and \( \text{nReturn} \ k \ t \neq \[] \)
shows \( \text{nSend} \ k \ t \neq \[] \)

(\text{proof})

**lemma** fr\text{-}Send4:
\begin{align*}
\text{assumes } & \forall \ i < n \ . \ \ FlexRayController \ (\text{nReturn} \ i) \ \text{recv} \ (\text{nC} \ i) \ (\text{nStore} \ i) \ (\text{nSend} \ i) \\
& (\text{nGet} \ i) \\
& \text{and} \ \ DisjointSchedules \ n \ \text{nC} \\
& \text{and} \ \ IdenticCycleLength \ n \ \text{nC} \\
& \text{and} \ \ t \ \text{mod} \ \text{cycleLength} \ (\text{nC} \ k) \ \text{mem} \ \text{schedule} \ (\text{nC} \ k) \\
& \text{and} \ \ k < n \\
& \text{and} \ \text{nReturn} \ k \ t \neq \[] \\
\text{shows } & \exists \ k \ . \ k < n \implies \text{nSend} \ k \ t \neq \[]
\end{align*}

(\text{proof})

**lemma** fr\text{-}Send5:
\begin{align*}
\text{assumes } & h1: \forall \ i < n \ . \ \ FlexRayController \ (\text{nReturn} \ i) \ \text{recv} \ (\text{nC} \ i) \ (\text{nStore} \ i) \ (\text{nSend} \ i) \\
& (\text{nGet} \ i) \\
& \text{and} \ h2: \ \text{DisjointSchedules} \ n \ \text{nC} \\
& \text{and} \ h3: \ \text{IdenticCycleLength} \ n \ \text{nC} \\
& \text{and} \ h4: \ t \ \text{mod} \ \text{cycleLength} \ (\text{nC} \ k) \ \text{mem} \ \text{schedule} \ (\text{nC} \ k) \\
& \text{and} \ h5: k < n \\
& \text{and} \ h6: \ \text{nReturn} \ k \ t \neq \[] \\
& \text{and} \ h7: \forall \ k < n \ . \ \text{nSend} \ k \ t = \[] \\
\text{shows } & \text{False}
\end{align*}

(\text{proof})

**lemma** fr\text{-}Send6:
\begin{align*}
\text{assumes } & \forall \ i < n \ . \ \ FlexRayController \ (\text{nReturn} \ i) \ \text{recv} \ (\text{nC} \ i) \ (\text{nStore} \ i) \ (\text{nSend} \ i) \\
& (\text{nGet} \ i) \\
& \text{and} \ DisjointSchedules \ n \ \text{nC} \\
& \text{and} \ IdenticCycleLength \ n \ \text{nC} \\
& \text{and} \ t \ \text{mod} \ \text{cycleLength} \ (\text{nC} \ k) \ \text{mem} \ \text{schedule} \ (\text{nC} \ k) \\
& \text{and} \ k < n \\
& \text{and} \ \text{nReturn} \ k \ t \neq \[] \\
\text{shows } & \exists \ k < n \ . \ \text{nSend} \ k \ t \neq \[]
\end{align*}

(\text{proof})

**lemma** fr\text{-}Send7:
\begin{align*}
\text{assumes } & \forall \ i < n \ . \ \ FlexRayController \ (\text{nReturn} \ i) \ \text{recv} \ (\text{nC} \ i) \ (\text{nStore} \ i) \ (\text{nSend} \ i) \\
& (\text{nGet} \ i) \\
& \text{and} \ DisjointSchedules \ n \ \text{nC} \\
& \text{and} \ IdenticCycleLength \ n \ \text{nC} \\
& \text{and} \ t \ \text{mod} \ \text{cycleLength} \ (\text{nC} \ k) \ \text{mem} \ \text{schedule} \ (\text{nC} \ k) \\
& \text{and} \ k < n \\
& \text{and} \ j < n \\
& \text{and} \ \text{nReturn} \ k \ t = \[] \\
\text{shows } & \text{nSend} \ j \ t = \[]
\end{align*}
lemma fr-Send8:
assumes \( \forall i < n. \) FlexRayController \((n\text{Return } i) \) recv \((nC \ i) \) (nStore \( i \)) (nSend \( i \))
\((n\text{Get } i) \)
and DisjointSchedules \( n \) nC
and IdenticCycleLength \( n \) nC
and \( t \mod \text{cycleLength} \ (nC \ k) \) mem schedule \((nC \ k) \)
and \( k < n \)
and \( \text{nReturn } k \ t = [] \)
shows \( \neg (\exists k < n. \text{nSend } k \ t \neq []) \)
(proof)

lemma fr-nC-Send:
assumes \( \forall i < n. \) FlexRayController \((n\text{Return } i) \) recv \((nC \ i) \) (nStore \( i \)) (nSend \( i \))
\((n\text{Get } i) \)
and \( k < n \)
and DisjointSchedules \( n \) nC
and IdenticCycleLength \( n \) nC
and \( t \mod \text{cycleLength} \ (nC \ k) \) mem schedule \((nC \ k) \)
shows \( \forall j. \ j < n \land j \neq k \rightarrow (\text{nSend } j \ t = []) \)
(proof)

lemma length-nSend:
assumes \( h1: \) BusInterface activation \((n\text{Return } i) \) recv \((n\text{Store } i) \) (nSend \( i \)) (nGet \( i) \)
and \( h2: \forall t. \) length \((n\text{Return } i \ t) \leq \text{Suc } 0 \)
shows \( \text{length } (n\text{Send } i \ t) \leq \text{Suc } 0 \)
(proof)

lemma msg-nSend:
assumes BusInterface activation \((n\text{Return } i) \) recv \((n\text{Store } i) \) (nSend \( i \)) (nGet \( i) \)
and msg \((\text{Suc } 0) \) \((n\text{Return } i) \)
shows msg \((\text{Suc } 0) \) \((n\text{Send } i) \)
(proof)

lemma Broadcast-nSend-empty1:
assumes \( h1: \) Broadcast \( n \) nSend recv
and \( h2: \forall k < n. \) nSend \( k \ t = [] \)
shows \( \text{recv } t = [] \)
(proof)

12.5 Properties of the sheaf of channels nGet

lemma fr-nGet1a:
assumes \( h1: \) FlexRayController \((n\text{Return } k) \) recv \((nC \ k) \) (nStore \( k \)) (nSend \( k \))
\((n\text{Get } k) \)
and \( h2: t \mod \text{cycleLength} \ (nC \ k) \) mem schedule \((nC \ k) \)
shows \( n\text{Get } k \ t = [t \mod \text{cycleLength} \ (nC \ k)] \)
lemma fr-nGet1:
assumes \( \forall i < n. \) 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 \( nGet k t = [t \ mod \ cycleLength \ (nC k)] \)

(\textit{proof})

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

(\textit{proof})

lemma fr-nGet2:
assumes \( h1: \forall i < n. \) FlexRayController \((nReturn i) \ recv (nC i) \ (nStore i) \ (nSend i) \ (nGet i) \)
and \( h2: \neg (t \ mod \ cycleLength \ (nC k) \ mem \ schedule \ (nC k)) \)
and \( h3: k < n \)
shows \( nGet k t = [] \)

(\textit{proof})

lemma length-nGet1:
assumes FlexRayController \((nReturn k) \ recv (nC k) \ (nStore k) \ (nSend k) \ (nGet k) \)
shows \( \text{length} \ (nGet k t) \leq \text{Suc} \ 0 \)

(\textit{proof})

lemma msg-nGet1:
assumes FlexRayController \((nReturn k) \ recv (nC k) \ (nStore k) \ (nSend k) \ (nGet k) \)
shows \( \text{msg} \ (\text{Suc} \ 0) \ (nGet k) \)

(\textit{proof})

lemma msg-nGet2:
assumes \( \forall i < n. \) FlexRayController \((nReturn i) \ recv (nC i) \ (nStore i) \ (nSend i) \ (nGet i) \)
and \( k < n \)
shows \( \text{msg} \ (\text{Suc} \ 0) \ (nGet k) \)

(\textit{proof})

12.6 Properties of the sheaf of channels \( nStore \)

lemma fr-nStore-nReturn1:
assumes \( h0: \) Broadcast \( n \) \( nSend \ recv \)
and \( h1: \text{inf-disj} \ n \) \( nSend \)

56
and $h_2 : \forall i < n. \text{FlexRayController} (\text{nReturn } i) \text{ recv } (\text{nC } i) (\text{nStore } i) (\text{nSend } i) (\text{nGet } i)$

and $h_3 : \text{DisjointSchedules } n \text{nC}$

and $h_4 : \text{IdenticalCycleLength } n \text{nC}$

and $h_5 : t \mod \text{cycleLength } (\text{nC } k) \text{ mem schedule } (\text{nC } k)$

and $h_6 : k < n$

and $h_7 : j < n$

and $h_8 : j \neq k$

shows $\text{nStore } j \ t = \text{nReturn } k \ t$

(\text{proof})

\text{lemma fr-nStore-nReturn2:}
\text{assumes } h_1 : \text{Cable } n \text{nSend recv}

and $h_2 : \forall i < n. \text{FlexRayController} (\text{nReturn } i) \text{ recv } (\text{nC } i) (\text{nStore } i) (\text{nSend } i) (\text{nGet } i)$

and $h_3 : \text{DisjointSchedules } n \text{nC}$

and $h_4 : \text{IdenticalCycleLength } n \text{nC}$

and $h_5 : (t \mod \text{cycleLength } (\text{nC } k) \text{ mem schedule } (\text{nC } k))$

and $h_6 : k < n$

and $h_7 : j < n$

and $h_8 : j \neq k$

shows $\text{nStore } j \ t = \text{nReturn } k \ t$

(\text{proof})

\text{lemma fr-nStore-empty1:}
\text{assumes } h_1 : \text{Cable } n \text{nSend recv}

and $h_2 : \forall i < n. \text{FlexRayController} (\text{nReturn } i) \text{ recv } (\text{nC } i) (\text{nStore } i) (\text{nSend } i) (\text{nGet } i)$

and $h_3 : \text{DisjointSchedules } n \text{nC}$

and $h_4 : \text{IdenticalCycleLength } n \text{nC}$

and $h_5 : (t \mod \text{cycleLength } (\text{nC } k) \text{ mem schedule } (\text{nC } k))$

and $h_6 : k < n$

shows $\text{nStore } k \ t = []$

(\text{proof})

\text{lemma fr-nStore-nReturn3:}
\text{assumes } \text{Cable } n \text{nSend recv}

and $\forall i < n. \text{FlexRayController} (\text{nReturn } i) \text{ recv } (\text{nC } i) (\text{nStore } i) (\text{nSend } i) (\text{nGet } i)$

and $\text{DisjointSchedules } n \text{nC}$

and $\text{IdenticalCycleLength } n \text{nC}$

and $t \mod \text{cycleLength } (\text{nC } k) \text{ mem schedule } (\text{nC } k)$

and $k < n$

shows $\forall j, j < n \land j \neq k \Rightarrow \text{nStore } j \ t = \text{nReturn } k \ t$

(\text{proof})

\text{lemma length-nStore:}
\text{assumes } h_1 : \forall i < n. \text{FlexRayController} (\text{nReturn } i) \text{ recv } (\text{nC } i) (\text{nStore } i) (\text{nSend } i) (\text{nGet } i)
lemma \textit{msg-nStore}:
\begin{itemize}
\item \textbf{assumes} \forall i < n. \textit{FlexRayController} (\textit{nReturn} i) \textit{recv} (\textit{nC} i) (\textit{nStore} i) (\textit{nSend} i) (\textit{nGet} i)
\item \textbf{and} \textit{DisjointSchedules} \textit{n} \textit{nC}
\item \textbf{and} \textit{IdenticCycleLength} \textit{n} \textit{nC}
\item \textbf{and} \textit{inf-disj} \textit{n} \textit{nSend}
\item \textbf{and} \textit{i} < \textit{n}
\item \textbf{and} \textbf{\forall} \textit{i} < \textit{n}. \textit{msg} (\textit{Suc} 0) (\textit{nReturn} i)
\item \textbf{and} \textit{Cable} \textit{n} \textit{nSend} \textit{recv}
\end{itemize}
\textbf{shows} \textit{msg} (\textit{Suc} 0) (\textit{nStore} i)
\hfill (proof)

12.7 Refinement Properties

lemma \textit{fr-refinement-FrameTransmission}:
\begin{itemize}
\item \textbf{assumes} \textit{Cable} \textit{n} \textit{nSend} \textit{recv}
\item \textbf{and} \forall i < n. \textit{FlexRayController} (\textit{nReturn} i) \textit{recv} (\textit{nC} i) (\textit{nStore} i) (\textit{nSend} i) (\textit{nGet} i)
\item \textbf{and} \textit{DisjointSchedules} \textit{n} \textit{nC}
\item \textbf{and} \textit{IdenticCycleLength} \textit{n} \textit{nC}
\end{itemize}
\textbf{shows} \textit{FrameTransmission} \textit{n} \textit{nStore} \textit{nReturn} \textit{nGet} \textit{nC}
\hfill (proof)

lemma \textit{FlexRayArch-CorrectSheaf}:
\begin{itemize}
\item \textbf{assumes} \textit{FlexRayArch} \textit{n} \textit{nReturn} \textit{nC} \textit{nStore} \textit{nGet}
\item \textbf{shows} \textit{CorrectSheaf} \textit{n}
\end{itemize}
\hfill (proof)

lemma \textit{FlexRayArch-FrameTransmission}:
\begin{itemize}
\item \textbf{assumes} \textit{h1}:\textit{FlexRayArch} \textit{n} \textit{nReturn} \textit{nC} \textit{nStore} \textit{nGet}
\item \textbf{and} \textit{h2}:\forall i < n. \textit{msg} (\textit{Suc} 0) (\textit{nReturn} i)
\item \textbf{and} \textit{h3}:\textit{DisjointSchedules} \textit{n} \textit{nC}
\item \textbf{and} \textit{h4}:\textit{IdenticCycleLength} \textit{n} \textit{nC}
\end{itemize}
\textbf{shows} \textit{FrameTransmission} \textit{n} \textit{nStore} \textit{nReturn} \textit{nGet} \textit{nC}
\hfill (proof)

lemma \textit{FlexRayArch-nGet}:
\begin{itemize}
\item \textbf{assumes} \textit{h1}:\textit{FlexRayArch} \textit{n} \textit{nReturn} \textit{nC} \textit{nStore} \textit{nGet}
\item \textbf{and} \textit{h2}:\forall i < n. \textit{msg} (\textit{Suc} 0) (\textit{nReturn} i)
\end{itemize}
and \( h3 : \text{DisjointSchedules} \ n \ nC \)
and \( h4 : \text{IdentCycleLength} \ n \ nC \)
and \( h5 : i < n \)
shows \( \langle \text{proof} \rangle \)

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

theorem main-fr-refinement:
assumes \( \text{FlexRayArch} \ n \ n\text{Return} \ nC \ n\text{Store} \ n\text{Get} \)
shows \( \text{FlexRay} \ n \ n\text{Return} \ nC \ n\text{Store} \ n\text{Get} \)
\( \langle \text{proof} \rangle \)

end

13 Gateway: Types

theory Gateway-types
imports stream
begin

type-synonym Coordinates = nat × 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

∀ (t::nat)
  (r::reqType list) (x::aType list)
  (y::stopType list) (z::bool list).
  1:
  ( st-in t = init-state ∧ req t = [init]
    → ack t = [call] ∧ i1 t = [] ∧ vc t = []
    ∧ st-out t = call )
  ∧
  2:
  ( st-in t = init-state ∧ req t ≠ [init]
    → ack t = [init-state] ∧ i1 t = [] ∧ vc t = []
    ∧ st-out t = init-state )
  ∧
  3:
  ( (st-in t = call ∨ (st-in t = connection-ok ∧ r ≠ [send])) ∧
    req t = r ∧ lose t = [False]
    → ack t = [connection-ok] ∧ i1 t = [] ∧ vc t = []
    ∧ st-out t = connection-ok )
  ∧
  4:
  ( (st-in t = call ∨ st-in t = connection-ok ∨ st-in t = sending-data)
    ∧ lose t = [True]
    → ack t = [init-state] ∧ i1 t = [] ∧ vc t = []
    ∧ st-out t = init-state )
  ∧
  5:
  ( st-in t = connection-ok ∧ req t = [send] ∧ lose t = [False]
    → ack t = [sending-data] ∧ i1 t = buffer-in t ∧ vc t = []
    ∧ st-out t = sending-data )
  ∧
  6:
  ( st-in t = sending-data ∧ a1 t = [] ∧ lose t = [False]
    → ack t = [sending-data] ∧ i1 t = [] ∧ vc t = []
    ∧ st-out t = sending-data )
  ∧
  7:
  ( st-in t = sending-data ∧ a1 t = [sc-ack] ∧ lose t = [False]
    → ack t = [voice-com] ∧ i1 t = [] ∧ vc t = [vc-com]
    ∧ st-out t = voice-com )
  ∧
  8:
  ( st-in t = voice-com ∧ stop t = [] ∧ lose t = [False]
    → ack t = [voice-com] ∧ i1 t = [] ∧ vc t = [vc-com]
    ∧ st-out t = voice-com )
∧
— 9:
( st-in t = voice-com ∧ stop t = [] ∧ lose t = [True]
→ ack t = [voice-com] ∧ i1 t = [] ∧ vc t = []
∧ st-out t = voice-com )
∧
— 10:
( st-in t = voice-com ∧ stop t = [stop-vc]
→ ack t = [init-state] ∧ i1 t = [] ∧ vc t = []
∧ st-out t = init-state )
definition
Sample-L ::
reqType istream ⇒ ECall-Info istream ⇒ aType istream ⇒
stopType istream ⇒ bool istream ⇒
(nat ⇒ GatewayStatus) ⇒ (nat ⇒ ECall-Info list) ⇒
GatewayStatus istream ⇒ ECall-Info istream ⇒ vcType istream ⇒
⇒ (nat ⇒ GatewayStatus) ⇒ (nat ⇒ ECall-Info list) ⇒
⇒ bool
where
Sample-L req dt a1 stop lose st-in buffer-in
ack i1 vc st-out buffer-out
≡
(∀ (t::nat).
buffer-out t =
(if dt t = [] then buffer-in t else dt t )
∧
(tiTable-SampleT req a1 stop lose st-in buffer-in
ack i1 vc st-out)
definition
Sample ::
reqType istream ⇒ ECall-Info istream ⇒ aType istream ⇒
stopType istream ⇒ bool istream ⇒
GatewayStatus istream ⇒ ECall-Info istream ⇒ vcType istream ⇒
⇒ bool
where
Sample req dt a1 stop lose ack i1 vc
≡
((msg (1::nat) req) ∧
(msg (1::nat) a1) ∧
(msg (1::nat) stop))
→
(∃ st buffer.
(Sample-L req dt a1 stop lose
(fin-inf-append [init-state] st)
(fin-inf-append [] buffer)
ack i1 vc st buffer) )
definition
Gateway ::
  reqType istream ⇒ ECall-Info istream ⇒ aType istream ⇒
  stopType istream ⇒ bool istream ⇒ nat ⇒
  GatewayStatus istream ⇒ ECall-Info istream ⇒ vcType istream
  ⇒ bool
where
Gateway req dt a stop lose d ack i vc
≡ ∃ i1 i2 x y.
  (Sample req dt x stop lose ack i1 vc) ∧
  (Delay y i1 d x i2) ∧
  (Loss lose a i2 y i)

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

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

63
(ack (t+d) = [sending-data] ∧ a (Suc t) = [sc-ack] ∧
(∀ (k::nat). k ≤ (d+1) → lose (t+k) = [False])
→ vc ((Suc t) + d) = [vc-com]) )

definition
GatewaySystemReq ::
reqType istream ⇒ ECall-Info istream ⇒
stopType istream ⇒ bool istream ⇒ nat ⇒
GatewayStatus istream ⇒ vcType istream ⇒
⇒ bool

where
GatewaySystemReq req dt stop lose d ack vc
≡
((msg (1::nat) req) ∧ (msg (1::nat) stop) ∧ (ts lose))
→
(∀ (t::nat) (k::nat). (ack t = [init-state] ∧ req (Suc t) = [init]
∧ (∀ t1. t1 ≤ t → req t1 = []))
∧ req (t+2) = []
∧ (∀ m. m < k + 3 → req (t + m) ≠ [send])
∧ req (t+3+k) = [send] ∧ inf-last-ti dt (t+2) ≠ []
∧ (∀ j::nat).
. j ≤ (4 + k + d + d) → lose (t+j) = [False])
→ vc (t + 4 + k + d + d) = [vc-com]) )

end

15 Gateway: Verification

theory Gateway-proof-aux
imports Gateway BitBoolTS
begin

15.1 Properties of the defined data types

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

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

lemma aType-lemma:
assumes msg (Suc 0) a
\textbf{shows} \quad a \ t = [] \lor a \ t = [\text{sc-ack}]
\langle \text{proof} \rangle

\textbf{lemma} \ stopType-empty:
\textbf{assumes} \ msg \ (\text{Suc} \ 0) \ a \\
\text{and} \ a \ t \neq [] \\
\textbf{shows} \ a \ t = []
\langle \text{proof} \rangle

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

\textbf{lemma} \ stopType-lemma:
\textbf{assumes} \ msg \ (\text{Suc} \ 0) \ a \\
\textbf{shows} \ a \ t = [] \lor a \ t = [\text{stop-vc}]
\langle \text{proof} \rangle

\textbf{lemma} \ vcType-empty:
\textbf{assumes} \ msg \ (\text{Suc} \ 0) \ a \\
\text{and} \ a \ t \neq [] \\
\textbf{shows} \ a \ t = []
\langle \text{proof} \rangle

\textbf{lemma} \ vcType-lemma:
\textbf{assumes} \ msg \ (\text{Suc} \ 0) \ a \\
\textbf{shows} \ a \ t = [] \lor a \ t = [\text{vc-com}]
\langle \text{proof} \rangle

\textbf{15.2 Properties of the Delay component}

\textbf{lemma} \ Delay-L1:
\textbf{assumes} \ h1: \forall \ t1 < t. \ i1 \ t1 = [] \\
\text{and} \ h2: \text{Delay} \ y \ i1 \ d \ x \ i2 \\
\text{and} \ h3: \ t2 < t + d \\
\textbf{shows} \ i2 \ t2 = []
\langle \text{proof} \rangle

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

\textbf{lemma} \ Delay-L3:
\textbf{assumes} \ h1: \forall \ t1 \leq t. \ y \ t1 = [] \\
\text{and} \ h2: \text{Delay} \ y \ i1 \ d \ x \ i2
and $h3 : t2 \leq t + d$
shows $x t2 = []$

(\textit{proof})

\textbf{lemma Delay-L4}:
\begin{itemize}
  \item \textbf{assumes} $\forall t1 \leq t. \ y t1 = []$
  \item and $\text{Delay} \ y \ i1 \ d \ x \ i2$
\end{itemize}
shows $\forall t2 \leq t + d. \ x t2 = []$

(\textit{proof})

\textbf{lemma Delay-lengthOut1}:
\begin{itemize}
  \item \textbf{assumes} $h1 : \forall t. \ \text{length} \ (x \ t) \leq \text{Suc} \ 0$
  \item and $h2 : \text{Delay} \ x \ i1 \ d \ y \ i2$
\end{itemize}
shows $\text{length} \ (y \ t) \leq \text{Suc} \ 0$

(\textit{proof})

\textbf{lemma Delay-msg1}:
\begin{itemize}
  \item \textbf{assumes} $\text{msg} \ (\text{Suc} \ 0) \ x$
  \item and $\text{Delay} \ x \ i1 \ d \ y \ i2$
\end{itemize}
shows $\text{msg} \ (\text{Suc} \ 0) \ y$

(\textit{proof})

\section*{15.3 Properties of the Loss component}

\textbf{lemma Loss-L1}:
\begin{itemize}
  \item \textbf{assumes} $\forall t2 < t. \ i2 \ t2 = []$
  \item and $\text{Loss} \ \text{lose} \ a \ i2 \ y \ i$
  \item and $t2 < t$
  \item and $\text{ts} \ \text{lose}$
\end{itemize}
shows $i \ t2 = []$

(\textit{proof})

\textbf{lemma Loss-L2}:
\begin{itemize}
  \item \textbf{assumes} $\forall t2 < t. \ i2 \ t2 = []$
  \item and $\text{Loss} \ \text{lose} \ a \ i2 \ y \ i$
  \item and $\text{ts} \ \text{lose}$
\end{itemize}
shows $\forall t2 < t. \ i \ t2 = []$

(\textit{proof})

\textbf{lemma Loss-L3}:
\begin{itemize}
  \item \textbf{assumes} $\forall t2 < t. \ a \ t2 = []$
  \item and $\text{Loss} \ \text{lose} \ a \ i2 \ y \ i$
  \item and $t2 < t$
  \item and $\text{ts} \ \text{lose}$
\end{itemize}
shows $y \ t2 = []$

(\textit{proof})

\textbf{lemma Loss-L4}:
\begin{itemize}
  \item \textbf{assumes} $\forall t2 < t. \ a \ t2 = []$
\end{itemize}

66
and Loss lose a i2 y i
and ts lose
shows ∀ t2 < t. y t2 = []
(proof)

lemma Loss-L5:
assumes ∀ t1 ≤ t. a t1 = []
and Loss lose a i2 y i
and t2 ≤ t
and ts lose
shows y t2 = []
(proof)

lemma Loss-L5Suc:
assumes ∀ j ≤ d. a (t + Suc j) = []
and Loss lose a i2 y i
and Suc j ≤ d
and tsLose : ts lose
shows y (t + Suc j) = []
(proof)

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

lemma Loss-lengthOut1:
assumes h1: ∀ t. length (a t) ≤ Suc 0
and h2: Loss lose a i2 x i
shows length (x t) ≤ Suc 0
(proof)

lemma Loss-lengthOut2:
assumes ∀ t. length (a t) ≤ Suc 0
and Loss lose a i2 x i
shows ∀ t. length (x t) ≤ 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 \( \forall t. \text{length}(a t) \leq \text{Suc} \ 0 \)
and \( \text{Delay} x i1 d y i2 \)
and \( \text{Loss} \ lose a i2 x i \)
shows \( \text{length}(y t) \leq \text{Suc} \ 0 \)
(proof)

**Lemma** Loss-Delay-msg-a:
assumes \( \text{msg}(\text{Suc} \ 0) \ a \)
and \( \text{Delay} x i1 d y i2 \)
and \( \text{Loss} \ lose a i2 x i \)
shows \( \text{msg}(\text{Suc} \ 0) \ y \)
(proof)

15.5 Auxiliary Lemmas

**Lemma** inf-last-ti2:
assumes \( \text{inf-last-ti dt} (\text{Suc} (\text{Suc} t)) \neq \left[\right] \)
shows \( \text{inf-last-ti dt} (\text{Suc} (\text{Suc} (\text{Suc} t + k))) \neq \left[\right] \)
(proof)

**Lemma** aux-ack-t2:
assumes \( h1: \forall m \leq k. \text{ack}(\text{Suc}(\text{Suc}(t + m))) = \left[\text{connection-ok}\right] \)
and \( h2: \text{Suc}(\text{Suc}(t)) < t2 \)
and \( h3: t2 < t + 3 + k \)
shows \( \text{ack} t2 = \left[\text{connection-ok}\right] \)
(proof)

**Lemma** aux-lemma-lose-1:
assumes \( h1: \forall j \leq ((2::\text{nat}) \ast d + (4::\text{nat}) + k). \text{lose}(t + j) = x \)
and \( h2: k a \leq \text{Suc} \ d \)
shows \( \text{lose}(\text{Suc}(t + k + ka)) = x \)
(proof)

**Lemma** aux-lemma-lose-2:
assumes \( \forall j \leq (2::\text{nat}) \ast d + ((4::\text{nat}) + k). \text{lose}(t + j) = \left[\text{False}\right] \)
shows \( \forall x \leq d + (1::\text{nat}). \text{lose}(t + x) = \left[\text{False}\right] \)
(proof)

**Lemma** aux-lemma-lose-3a:
assumes \( h1: \forall j \leq 2 \ast d + (4 + k). \text{lose}(t + j) = \left[\text{False}\right] \)
and \( h2: k a \leq \text{Suc} \ d \)
shows \( \text{lose}(d + (t + (3 + k)) + ka) = \left[\text{False}\right] \)
(proof)

**Lemma** aux-lemma-lose-3:
assumes \( \forall j \leq 2 \ast d + (4 + k). \text{lose}(t + j) = \left[\text{False}\right] \)
∀ka≤Succ 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] −→ ack t = [x] ∧ st-out t = x
and lose t = [False] −→ ack t = [y] ∧ st-out t = y
shows ack t = [st-out t]
(proof)

lemma ts-lose-ack-st1:
assumes h1: lose t = [True] ∨ lose t = [False]
and h2: lose t = [True] −→ ack t = [x] ∧ st-out t = x
and h3: lose t = [False] −→ ack t = [y] ∧ st-out t = y
shows ack t = [st-out t]
(proof)

lemma ts-lose-ack-st2ts:
assumes ts lose
and lose t = [True] −→
  ack t = [x] ∧ i1 t = [] ∧ st-out t = x
and lose t = [False] −→
  ack t = [y] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = y
shows ack t = [st-out t]
(proof)

lemma ts-lose-ack-st2:
assumes h1: lose t = [True] ∨ lose t = [False]
and h2: lose t = [True] −→
  ack t = [x] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = x
and h3: lose t = [False] −→
  ack t = [y] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = y
shows ack t = [st-out t]
(proof)

lemma ts-lose-ack-st2vc-com:
assumes h1: lose t = [True] ∨ lose t = [False]
and h2: lose t = [True] −→
  ack t = [x] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = x
and h3: lose t = [False] −→
  ack t = [y] ∧ i1 t = [] ∧ vc t = [vc-com] ∧ st-out t = y
shows ack t = [st-out t]
\langle proof \rangle

**lemma** ts-lose-ack-st2send:

assumes \( h1: \text{lose } t = [\text{True}] \lor \text{lose } t = [\text{False}] \)
and \( h2: \text{lose } t = [\text{True}] \rightarrow \) 
\( \text{ack } t = [x] \land \text{i1 } t = [] \land \text{vc } t = [] \land \text{st-out } t = x \)
and \( h3: \text{lose } t = [\text{False}] \rightarrow \) 
\( \text{ack } t = [y] \land \text{i1 } t = b \land \text{vc } t = [] \land \text{st-out } t = y \)
shows \( \text{ack } t = [\text{st-out } t] \)
\langle proof \rangle

**lemma** tiTable-ack-st-splitten:

assumes \( h1: \text{ts lose} \)
and \( h2: \text{msg } (\text{Suc } 0) \) a1
and \( h3: \text{msg } (\text{Suc } 0) \) stop
and \( h4: \text{st-in } t = \text{init-state } \land \text{req } t = [\text{init}] \rightarrow \) 
\( \text{ack } t = [\text{call}] \land \text{i1 } t = [] \land \text{vc } t = [] \land \text{st-out } t = \text{call} \)
and \( h5: \text{st-in } t = \text{init-state } \land \text{req } t \neq [\text{init}] \rightarrow \) 
\( \text{ack } t = [\text{init-state}] \land \text{i1 } t = [] \land \text{vc } t = [] \land \text{st-out } t = \text{init-state} \)
and \( h6: \langle \text{st-in } t = \text{call } \lor \text{st-in } t = \text{connection-ok } \land \text{req } t \neq [\text{send}] \rangle \land \text{lose } t = [\text{False}] \rightarrow \) 
\( \text{ack } t = [\text{connection-ok}] \land \text{i1 } t = [] \land \text{vc } t = [] \land \text{st-out } t = \text{connection-ok} \)
and \( h7: \langle \text{st-in } t = \text{call } \lor \text{st-in } t = \text{connection-ok } \land \text{st-in } t = \text{sending-data} \rangle \land \text{lose } t = [\text{True}] \rightarrow \) 
\( \text{ack } t = [\text{init-state}] \land \text{i1 } t = [] \land \text{vc } t = [] \land \text{st-out } t = \text{init-state} \)
and \( h8: \langle \text{st-in } t = \text{connection-ok } \land \text{req } t = [\text{send}] \land \text{lose } t = [\text{False}] \rightarrow \) 
\( \text{ack } t = [\text{sending-data}] \land \text{i1 } t = b \land \text{vc } t = [] \land \text{st-out } t = \text{sending-data} \)
and \( h9: \langle \text{st-in } t = \text{sending-data } \land \text{a1 } t = [] \land \text{lose } t = [\text{False}] \rightarrow \) 
\( \text{ack } t = [\text{voice-com}] \land \text{i1 } t = [] \land \text{vc } t = [] \land \text{st-out } t = \text{voice-com} \)
and \( h10: \langle \text{st-in } t = \text{voice-com } \land \text{stop } t = [] \land \text{lose } t = [\text{False}] \rightarrow \) 
\( \text{ack } t = [\text{voice-com}] \land \text{i1 } t = [] \land \text{vc } t = [] \land \text{st-out } t = \text{voice-com} \)
and \( h11: \langle \text{st-in } t = \text{voice-com } \land \text{stop } t = [] \land \text{lose } t = [\text{True}] \rightarrow \) 
\( \text{ack } t = [\text{voice-com}] \land \text{i1 } t = [] \land \text{vc } t = [] \land \text{st-out } t = \text{voice-com} \)
and \( h12: \langle \text{st-in } t = \text{voice-com } \land \text{stop } t = [] \land \text{lose } t = [\text{True}] \rightarrow \) 
\( \text{ack } t = [\text{voice-com}] \land \text{i1 } t = [] \land \text{vc } t = [] \land \text{st-out } t = \text{voice-com} \)
and \( h13: \langle \text{st-in } t = \text{voice-com } \land \text{stop } t = [\text{stop-vc}] \rightarrow \) 
\( \text{ack } t = [\text{init-state}] \land \text{i1 } t = [] \land \text{vc } t = [] \land \text{st-out } t = \text{init-state} \)
shows \( \text{ack } t = [\text{st-out } t] \)
\langle proof \rangle

**lemma** tiTable-ack-st-st:

assumes \( \text{tiTable-SampleT } \text{req } a1 \) stop lose st-in b ack i1 vc st-out
and \( \text{tsLose:ts lose} \)
and \( \text{a1Msg1:msg } (\text{Suc } 0) \) a1
and \( \text{stopMsg1:msg } (\text{Suc } 0) \) stop
shows \( \text{ack } t = [\text{st-out } t] \)
\langle proof \rangle

**lemma** tiTable-ack-st-td:

70
\textbf{lemma} \textit{tiTable-ack-connection-ok}:
\begin{itemize}
\item \textbf{assumes} \text{tbl}: \text{tiTable-SampleT} \ \text{req} \ x \ \text{stop} \ \text{lose} \ \text{st-in} \ b \ \text{ack} \ i1 \ \text{vc} \ \text{st-out} \\
\hspace{1cm} \text{and} \ ts \ \text{lose} \\
\hspace{1cm} \text{and} \ \text{msg} \ (\text{Suc} \ 0) \ a1 \\
\hspace{1cm} \text{and} \ \text{msg} \ (\text{Suc} \ 0) \ \text{stop} \\
\item \textbf{shows} \ \text{st-out} \ t \ = \ \text{hd} \ (\text{ack} \ t) \\
\end{itemize}
\hspace{1cm} \langle \text{proof} \rangle 

\begin{itemize}
\item \textbf{lemma} \textit{tiTable-i1-1}:
\begin{itemize}
\item \textbf{assumes} \text{tbl}: \text{tiTable-SampleT} \ \text{req} \ x \ \text{stop} \ \text{lose} \ \text{st-in} \ b \ \text{ack} \ i1 \ \text{vc} \ \text{st-out} \\
\hspace{1cm} \text{and} \ ts \ \text{lose} \\
\hspace{1cm} \text{and} \ \text{msg} \ (\text{Suc} \ 0) \ x \\
\hspace{1cm} \text{and} \ \text{msg} \ (\text{Suc} \ 0) \ \text{stop} \\
\hspace{1cm} \text{and} \ \text{ack} \ t \ = \ [\text{connection-ok}] \\
\item \textbf{shows} \ i1 \ t \ = \ [] \\
\end{itemize}
\hspace{1cm} \langle \text{proof} \rangle 

\begin{itemize}
\item \textbf{lemma} \textit{tiTable-ack-call}:
\begin{itemize}
\item \textbf{assumes} \text{tbl}: \text{tiTable-SampleT} \ \text{req} \ x \ \text{stop} \ \text{lose} \ \text{st-in} \ b \ \text{ack} \ i1 \ \text{vc} \ \text{st-out} \\
\hspace{1cm} \text{and} \ ts \ \text{lose} \\
\hspace{1cm} \text{and} \ \text{msg} \ (\text{Suc} \ 0) \ x \\
\hspace{1cm} \text{and} \ \text{msg} \ (\text{Suc} \ 0) \ \text{stop} \\
\hspace{1cm} \text{and} \ \text{ack} \ t \ = \ [\text{call}] \\
\item \textbf{shows} \ \text{st-in} \ t \ = \ \text{init-state} \ \land \ \text{req} \ t \ = \ [\text{init}] \\
\end{itemize}
\hspace{1cm} \langle \text{proof} \rangle 

\begin{itemize}
\item \textbf{lemma} \textit{tiTable-i1-2}:
\begin{itemize}
\item \textbf{assumes} \text{tbl}: \text{tiTable-SampleT} \ \text{req} \ a1 \ \text{stop} \ \text{lose} \ \text{st-in} \ b \ \text{ack} \ i1 \ \text{vc} \ \text{st-out} \\
\hspace{1cm} \text{and} \ ts \ \text{lose} \\
\hspace{1cm} \text{and} \ \text{msg} \ (\text{Suc} \ 0) \ a1 \\
\hspace{1cm} \text{and} \ \text{msg} \ (\text{Suc} \ 0) \ \text{stop} \\
\hspace{1cm} \text{and} \ \text{ack} \ t \ = \ [\text{call}] \\
\item \textbf{shows} \ i1 \ t \ = \ [] \\
\end{itemize}
\hspace{1cm} \langle \text{proof} \rangle 

\begin{itemize}
\item \textbf{lemma} \textit{tiTable-ack-init0}:
\begin{itemize}
\item \textbf{assumes} \text{tbl}: \text{tiTable-SampleT} \ \text{req} \ a1 \ \text{stop} \ \text{lose} \\
\hspace{1cm} (\text{fin-inf-append} \ [\text{init-state}] \ \text{st}) \\
\hspace{1cm} b \ \text{ack} \ i1 \ \text{vc} \ \text{st} \\
\hspace{1cm} \text{and} \ \text{req0} \ : \ \text{req} \ 0 \ = \ [] \\
\end{itemize}
\end{itemize}

71
shows $\text{ack } 0 = \text{[init-state]}$

(proof)

lemma $\text{tiTable-ack-init}$:
assumes $\text{tiTable-SampleT req a1 stop lose}$
\[
\begin{align*}
\text{and } & \text{ts lose} \\
\text{and } & \text{msg (Suc 0) a1} \\
\text{and } & \text{msg (Suc 0) stop} \\
\text{and } & \forall t1 \leq t. \text{req t1} = []
\end{align*}
\]
shows $\text{ack t} = \text{[init-state]}$

(proof)

lemma $\text{tiTable-i1-3}$:
assumes $\text{tbl:tiTable-SampleT req x stop lose}$
\[
\begin{align*}
\text{and } & \text{tslose:ts lose} \\
\text{and } & \text{xMsg1:msg (Suc 0) x} \\
\text{and } & \text{stopMsg1:msg (Suc 0) stop} \\
\text{and } & \forall t1 \leq t. \text{req t1} = []
\end{align*}
\]
shows $\text{i1 t} = []$

(proof)

lemma $\text{tiTable-st-call-ok}$:
assumes $\text{tbl:tiTable-SampleT req x stop lose}$
\[
\begin{align*}
\text{and } & \text{tslose:ts lose} \\
\text{and } & \text{h3:}\forall m \leq k. \text{ack (Suc (Suc (t + m)))} = \text{[connection-ok]} \\
\text{and } & \text{h4:st (Suc t) = call}
\end{align*}
\]
shows $\text{st (Suc (Suc t)) = connection-ok}$

(proof)

lemma $\text{tiTable-i1-4b}$:
assumes $\text{tiTable-SampleT req x stop lose}$
\[
\begin{align*}
\text{and } & \text{ts lose} \\
\text{and } & \text{msg (Suc 0) x} \\
\text{and } & \text{msg (Suc 0) stop} \\
\text{and } & \forall t1 \leq t. \text{req t1} = [] \\
\text{and } & \text{req (Suc t) = [init]} \\
\text{and } & \forall m < k + 3. \text{req (t + m) \neq [send]} \\
\text{and } & \text{h7:}\forall m \leq k. \text{ack (Suc (Suc (t + m)))} = \text{[connection-ok]} \\
\text{and } & \forall j \leq k + 3. \text{lose (t + j) = [False]} \\
\text{and } & \text{h9:t2 < (t + 3 + k)}
\end{align*}
\]
shows $\text{i1 t2} = []$

(proof)
lemma \textit{tiTable-\textit{i1-\textit{4}}:}
\textbf{assumes} \textit{tiTable-SampleT \textit{req a1 stop lose}}
\begin{itemize}
\item \textit{(fin-inf-append [init-state] st)} \textit{b \textit{ack i1 vc st}}
\item \textit{ts lose}
\item \textit{msg (Suc 0) a1}
\item \textit{msg (Suc 0) stop}
\item \textit{\(\forall t \leq t. \text{req } t = []\)}
\item \textit{\textit{req} (Suc t) = [\textit{init}]}\end{itemize}
\textbf{and} \textit{\(\forall m < k + 3. \text{req } (t + m) \neq [\textit{send}]\)}
\textbf{and} \textit{\(\forall m \leq k. \text{ack } (\text{Suc } (t + m)) = [\textit{connection-ok}]\)}
\textbf{and} \textit{\(\forall j \leq k + 3. \text{lose } (t + j) = [\textit{False}]\)}
\textbf{shows} \textit{\(\forall t2 < (t + 3 + k). \text{\it{i1 t2} = []}\)}
\begin{proof}
\end{proof}

\textbf{lemma} \textit{tiTable-ack-ok:}
\textbf{assumes} \textit{h1:}\textit{\(\forall j \leq d + 2. \text{\it{lose} } (t + j) = [\textit{False}]\)}
\textbf{and} \textit{tsLose:ts lose}
\textbf{and} \textit{stopMsg1:msg (Suc 0) stop}
\textbf{and} \textit{a1Msg1:msg (Suc 0) a1}
\textbf{and} \textit{reqNsend:req (Suc t) \neq [\textit{send}]}
\textbf{and} \textit{ackCon:ack t = [\textit{connection-ok}]}
\textbf{and} \textit{tbl:tiTable-SampleT \textit{req a1 stop lose \textit{(fin-inf-append [init-state] st)}} b \textit{ack i1 vc st}}
\textbf{shows} \textit{\textit{ack} (Suc t) = [\textit{connection-ok}]}
\begin{proof}
\end{proof}

\textbf{lemma} \textit{Gateway-L7a:}
\textbf{assumes} \textit{gw:Gateway \textit{req dt a stop lose d \textit{ack} i vc}}
\textbf{and} \textit{aMsg1:msg (Suc 0) a}
\textbf{and} \textit{stopMsg1:msg (Suc 0) stop}
\textbf{and} \textit{reqMsg1:msg (Suc 0) req}
\textbf{and} \textit{tsLose:ts lose}
\textbf{and} \textit{loseFalse:\(\forall j \leq d + 2. \text{\it{lose} } (t + j) = [\textit{False}]\)}
\textbf{and} \textit{nsend:req (Suc t) \neq [\textit{send}]}
\textbf{and} \textit{ackNCon:ack t = [\textit{connection-ok}]}
\textbf{shows} \textit{\textit{ack} (Suc t) = [\textit{connection-ok}]}
\begin{proof}
\end{proof}

\textbf{lemma} \textit{Sample-L-buffer:}
\textbf{assumes} \textit{Sample-L \textit{req dt a1 stop lose (\textit{fin-inf-append [init-state] st})}}
\textbf{and} \textit{i1 vc st buffer}
\textbf{shows} \textit{buffer t = \textit{inf-last-ti dt t}}
\begin{proof}
\end{proof}

\textbf{lemma} \textit{tiTable-SampleT-i1-buffer:}
\textbf{assumes} \textit{ack t = [\textit{connection-ok}]}\textbf{and} \textit{reqSend:req (Suc t) = [\textit{send}]}

73
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 zdd: 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 \( \text{length} \ (a \ t) \leq (\text{Suc} 0) \)
  ⟨proof⟩

lemma ServiceCenter-a-msg:
  assumes ServiceCenter \( i \) \( a \)
  shows \( \text{msg} \ (\text{Suc} 0) \) \( a \)
  ⟨proof⟩

lemma ServiceCenter-L1:
  assumes \( \forall t2 < x. \ i t2 = \[] \)
    and ServiceCenter \( i \) \( a \)
    and \( t \leq x \)
  shows \( a \ t = \[] \)
  ⟨proof⟩

lemma ServiceCenter-L2:
  assumes \( \forall t2 < x. \ i t2 = \[] \)
    and ServiceCenter \( i \) \( a \)
  shows \( \forall t3 \leq x. \ a t3 = \[] \)
  ⟨proof⟩

15.7 General properties of stream values

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

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

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

lemma streamValue4:
  assumes \( \forall \ j \leq D. \ \text{str} \ (t + j + (\text{Suc} y)) = x \)
    and \( \text{str} \ (t + y) = x \)
  shows \( \forall \ j \leq D. \ \text{str} \ (t + j + y) = x \)
  ⟨proof⟩
lemma streamValue5:
assumes $\forall j \leq D. \text{str} (t + j + ((i::nat) + k)) = x$
and $j \leq D$
shows $\text{str} (t + i + k + j) = x$
(\proof)

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

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

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

lemma arith-streamValue9aux:
Suc $(t + (j + d)) + (i + k)) = \text{Suc} \, (t + i + k + d + j)$
(\proof)

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

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

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

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

lemma arith-sum4: $t + 3 + k + d = \text{Suc} \, (t + (2::nat) + k + d)$
(\proof)
lemma \textit{streamValue11}: \\
\begin{align*}
\text{assumes } & h1: \forall j \leq 2 \ast d + (4 + k). \text{lose } (t + j) = x \\
\text{and } & h2: j \leq \text{Suc } d \\
\text{shows } & \text{lose } (t + 2 + k + j) = x \\
\langle \text{proof} \rangle
\end{align*}

lemma \textit{streamValue12}: \\
\begin{align*}
\text{assumes } & \forall j \leq 2 \ast d + (4 + k). \text{lose } (t + j) = x \\
\text{shows } & \forall j \leq \text{Suc } d. \text{lose } (t + 2 + k + j) = x \\
\langle \text{proof} \rangle
\end{align*}

lemma \textit{streamValue43}: \\
\begin{align*}
\text{assumes } & \forall j \leq 2 \ast d + ((4::\text{nat}) + k). \text{lose } (t + j) = [\text{False}] \\
\text{shows } & \forall j \leq 2 \ast d. \text{lose } ((t + (3::\text{nat}) + k) + j) = [\text{False}] \\
\langle \text{proof} \rangle
\end{align*}

end

theory \textit{Gateway-proof} \\
imports \textit{Gateway-proof-aux} \\
begin

15.8 Properties of the Gateway

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

lemma \textit{Gateway-L2}: \\
\begin{align*}
\text{assumes } & h1: \text{Gateway req dt a stop lose d ack i vc} \\
\text{and } & h2: \text{msg } (\text{Suc } 0) \text{ req} \\
\text{and } & h3: \text{msg } (\text{Suc } 0) \text{ a} \\
\text{and } & h4: \text{msg } (\text{Suc } 0) \text{ stop} \\
\text{and } & h5: \text{ts lose} \\
\text{and } & h6: \text{ack } t = [\text{connection-ok}] \\
\text{and } & h7: \text{req } (\text{Suc } t) = [\text{send}] \\
\text{and } & h8: \forall k \leq \text{Suc } d. \text{lose } (t + k) = [\text{False}] \\
\text{shows } & i \ (\text{Suc } (t + d)) = \text{inf-last-ti dt t} \\
\langle \text{proof} \rangle
\end{align*}
lemma \textit{Gateway-L3}:
\begin{itemize}
  \item \textbf{assumes} \( h1: \text{Gateway req dt a stop lose d ack i vc} \)
  \item \( h2: \text{msg (Suc 0) req} \)
  \item \( h3: \text{msg (Suc 0) a} \)
  \item \( h4: \text{msg (Suc 0) stop} \)
  \item \( h5: \text{ts lose} \)
  \item \( h6: \text{ack t = [connection-ok]} \)
  \item \( h7: \text{req (Suc t) = [send]} \)
  \item \( h8: \forall k \leq \text{Suc d}. \text{lose (t + k) = [False]} \)
\end{itemize}
\textbf{shows} \( \text{ack (Suc t) = [sending-data]} \)

(\textbf{proof})

lemma \textit{Gateway-L4}:
\begin{itemize}
  \item \textbf{assumes} \( h1: \text{Gateway req dt a stop lose d ack i vc} \)
  \item \( h2: \text{msg (Suc 0) req} \)
  \item \( h3: \text{msg (Suc 0) a} \)
  \item \( h4: \text{msg (Suc 0) stop} \)
  \item \( h5: \text{ts lose} \)
  \item \( h6: \text{ack (t + d) = [sending-data]} \)
  \item \( h7: \text{a (Suc t) = [sc-ack]} \)
  \item \( h8: \forall k \leq \text{Suc d}. \text{lose (t + k) = [False]} \)
\end{itemize}
\textbf{shows} \( \text{vc (Suc (t + d)) = [vc-com]} \)

(\textbf{proof})

lemma \textit{Gateway-L5}:
\begin{itemize}
  \item \textbf{assumes} \( h1: \text{Gateway req dt a stop lose d ack i vc} \)
  \item \( h2: \text{msg (Suc 0) req} \)
  \item \( h3: \text{msg (Suc 0) a} \)
  \item \( h4: \text{msg (Suc 0) stop} \)
  \item \( h5: \text{ts lose} \)
  \item \( h6: \text{ack (t + d) = [sending-data]} \)
  \item \( h7: \forall j \leq \text{Suc d}. \text{a (t+j) = []} \)
  \item \( h8: \forall k \leq (d + d). \text{lose (t + k) = [False]} \)
\end{itemize}
\textbf{shows} \( j \leq d \rightarrow \text{ack (t+d+j) = [sending-data]} \)

(\textbf{proof})

lemma \textit{Gateway-L6-induction}:
\begin{itemize}
  \item \textbf{assumes} \( h1: \text{msg (Suc 0) req} \)
  \item \( h2: \text{msg (Suc 0) x} \)
  \item \( h3: \text{msg (Suc 0) stop} \)
  \item \( h4: \text{ts lose} \)
  \item \( h5: \forall j \leq k. \text{lose (t + j) = [False]} \)
  \item \( h6: \forall m \leq k. \text{req (t + m) \neq [send]} \)
  \item \( h7: \text{ack t = [connection-ok]} \)
  \item \( h8: \text{Sample req dt x1 stop lose ack i1 vc} \)
  \item \( h9: \text{Delay x2 i1 d x1 i2} \)
  \item \( h10: \text{Loss lose x i2 x2 i} \)
\end{itemize}
and \( h11: m \leq k \)
shows \( \text{ack} \ (t + m) = \text{[connection-ok]} \)

(\textit{proof})

\textbf{lemma} Gateway-L6:
\textbf{assumes} Gateway req dt a stop lose d ack i vc
\ \ \ \ and \ \ \ \ \forall m \leq k. \ \text{req} \ (t + m) \neq \text{[send]}
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ and \ \ \ \ \forall j \leq k. \ \text{lose} \ (t + j) = \text{[False]}
\ \ \ \ \ and \ \ \ \ \text{ack} \ t = \text{[connection-ok]}
\ \ \ \ \and \ \ \ \ \text{msg} \ (\text{Suc} \ 0) \ \text{req}
\ \ \ \ \and \ \ \ \ \text{msg} \ (\text{Suc} \ 0) \ \text{stop}
\ \ \ \ \and \ \ \ \ \text{msg} \ (\text{Suc} \ 0) \ a
\and \ \ \ \ ts \ \text{lose}
shows \ \forall m \leq k. \ \text{ack} \ (t + m) = \text{[connection-ok]}
(\textit{proof})

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

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

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

\textbf{lemma} Gateway-L7:
\textbf{assumes} \( h1: \) Gateway req dt a stop lose d ack i vc
\and \( h2: ts \ \text{lose} \)
\and \( h3: \text{msg} \ (\text{Suc} \ 0) \ a \)
\and \( h4: \text{msg} \ (\text{Suc} \ 0) \ \text{stop} \)
\and \( h5: \text{msg} \ (\text{Suc} \ 0) \ \text{req} \)
\and \( h6: \text{req} \ (\text{Suc} \ t) = \text{[init]} \)
\and \( h7: \forall m < (k + 3). \ \text{req} \ (t + m) \neq \text{[send]} \)

79
and \(h_8:\text{req} (t + 3 + k) = [\text{send}]\)
and \(h_9:\text{ack} t = [\text{init-state}]\)
and \(h_{10}: \forall j \leq k + d + 3. \text{lose} (t + j) = [\text{False}]\)
and \(h_{11}: \forall t1 \leq t. \text{req} t1 = []\)
shows \(\forall t2 < (t + 3 + k + d), i t2 = []\)

(\text{proof})

\textbf{lemma} Gateway-L8a:
\textbf{assumes} \(h_{11}:\text{Gateway req dt a stop lose d ack i vc}\)
\textbf{and} \(h_{21}:\text{msg} (\text{Suc 0}) \text{ req}\)
\textbf{and} \(h_{31}:\text{msg} (\text{Suc 0}) \text{ stop}\)
\textbf{and} \(h_{41}:\text{msg} (\text{Suc 0}) \text{ a}\)
\textbf{and} \(h_{51}:\text{ts lose}\)
\textbf{and} \(h_{61}: \forall j \leq 2 * d. \text{lose} (t + j) = [\text{False}]\)
\textbf{and} \(h_{71}:\text{ack} t = [\text{sending-data}]\)
\textbf{and} \(h_{81}:\forall t3 \leq t + d. a t3 = []\)
\textbf{and} \(h_{91}:x \leq d + d\)
\textbf{shows} \(\text{ack} (t + x) = [\text{sending-data}]\)
(\text{proof})

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

\textbf{15.9 Proof of the Refinement Relation for the Gateway Requirements}

\textbf{lemma} Gateway-L0:
\textbf{assumes} \(\text{Gateway req dt a stop lose d ack i vc}\)
\textbf{shows} \(\text{GatewayReq req dt a stop lose d ack i vc}\)
(\text{proof})

\textbf{15.10 Lemmas about Gateway Requirements}

\textbf{lemma} GatewayReq-L1:
\textbf{assumes} \(h_{11}:\text{msg} (\text{Suc 0}) \text{ req}\)
\textbf{and} \(h_{21}:\text{msg} (\text{Suc 0}) \text{ stop}\)
\textbf{and} \(h_{31}:\text{msg} (\text{Suc 0}) \text{ a}\)
\textbf{and} \(h_{41}:\text{ts lose}\)
\textbf{and} \(h_{61}:\text{req} (t + 3 + k) = [\text{send}]\)
and \( h7: \forall j \leq 2 \cdot d + (4 + k).\) lose \((t + j) = [\text{False}] \)
and \( h9: \forall m \leq k.\) ack \((t + 2 + m) = [\text{connection-ok}] \)
and \( h10: \text{GatewayReq req dt a stop lose d ack i vc} \)
shows ack \((t + 3 + k) = [\text{sending-data}] \)

\[ \text{proof} \]

\textbf{lemma} \text{GatewayReq-L2:}
\textbf{assumes} \( h1: \text{msg (Suc 0) req} \)
and \( h2: \text{msg (Suc 0) stop} \)
and \( h3: \text{msg (Suc 0) a} \)
and \( h4: \text{ts lose} \)
and \( h5: \text{GatewayReq req dt a stop lose d ack i vc} \)
and \( h6: \text{req (t + 3 + k) = [send]} \)
and \( h7: \text{inf-last-ti dt t } \neq [] \)
and \( h8: \forall j \leq 2 \cdot d + (4 + k).\) lose \((t + j) = [\text{False}] \)
and \( h9: \forall m \leq k.\) ack \((t + 2 + m) = [\text{connection-ok}] \)
shows \( i (t + 3 + k + d) \neq [] \)
\[ \text{proof} \]

\[ \text{15.11 Properties of the Gateway System} \]
\textbf{lemma} \text{GatewaySystem-L1aux:}
\textbf{assumes} \( \text{msg (Suc 0) req} \)
and \( \text{msg (Suc 0) stop} \)
and \( \text{msg (Suc 0) a} \)
and \( \text{ts lose} \)
and \( \text{msg (Suc 0) req } \land \text{msg (Suc 0) a } \land \text{msg (Suc 0) stop } \land \text{ts lose } \rightarrow \)
\((\forall t. \text{ack t } = [\text{init-state}] \land\)
\text{req (Suc t) } = [\text{init}] \land \text{lose (Suc t) } = [\text{False}] \land\)
\text{ack (Suc (Suc t)) } = [\text{connection-ok}] \land\)
\((\forall k \leq \text{Suc d}.\) lose \((t + k) = [\text{False}] \rightarrow\)
\text{i (Suc (t + d)) } = [\text{inf-last-ti dt t } \land \text{ack (Suc t) } = [\text{sending-data}] \land\)
\text{ack (t + d) } = [\text{sending-data }] \land \text{a (Suc t) } = [\text{sc-ack}] \land\)
\((\forall k \leq \text{Suc d}.\) lose \((t + k) = [\text{False}] \rightarrow\)
\text{vc (Suc (t + d)) } = [\text{vc-com}])
\textbf{shows} \( \text{ack (t + 3 + k + d + d) } = [\text{sending-data}] \land\)
\text{a (Suc (t + 3 + k + d + d)) } = [\text{sc-ack}] \land\)
\((\forall ka \leq \text{Suc d}.\) lose \((t + 3 + k + d + ka) = [\text{False}] \rightarrow\)
\text{vc (Suc (t + 3 + k + d + d)) } = [\text{vc-com}]
\[ \text{proof} \]

\textbf{lemma} \text{GatewaySystem-L3aux:}
\textbf{assumes} \( \text{msg (Suc 0) req} \)
and \( \text{msg (Suc 0) stop} \)
and \( \text{msg (Suc 0) a} \)
and \( \text{ts lose} \)
and \( \text{msg (Suc 0) req } \land \text{msg (Suc 0) a } \land \text{msg (Suc 0) stop } \land \text{ts lose } \rightarrow \)

81
\( \forall t. \, (\text{ack } t = \text{init-state}) \land \\
\text{req } (\text{Suc } t) = \text{init} \land \text{lose } (\text{Suc } t) = \text{False} \land \\
\text{ack } (\text{Suc } t) = \text{connection-ok} \land \\
(\text{ack } t = \text{connection-ok} \land \text{req } (\text{Suc } t) = \text{send} \land \\
(\forall k \leq \text{Suc } d. \, \text{lose } (t + k) = \text{False}) \rightarrow \\
i \, (\text{Suc } (t + d)) = \text{inf-last-ti dt } t \land \text{ack } (\text{Suc } t) = \text{sending-data} \land \\
(\text{ack } (t + d) = \text{[sending-data]} \land \text{a } (\text{Suc } t) = \text{sc-ack} \land \\
(\forall k \leq \text{Suc } d. \, \text{lose } (t + k) = \text{False}) \rightarrow \\
\text{vc } (\text{Suc } (t + d)) = \text{[vc-com]}) \\
\text{shows } \text{ack } (t + 2 + k) = \text{connection-ok} \land \\
\text{req } (\text{Suc } (t + 2 + k)) = \text{send} \land \\
(\forall j \leq \text{Suc } d. \, \text{lose } (t + 2 + k + j) = \text{False}) \rightarrow \\
i \, (\text{Suc } (t + 2 + k + d)) = \text{inf-last-ti dt } (t + 2 + k) \\
\langle \text{proof} \rangle \\
\text{lemma } \text{GatewaySystem-L1:} \\
\text{assumes } h2: \text{ServiceCenter } i \ a \\
\text{and } h3: \text{GatewayReq } \text{req } \text{dt } a \text{ stop } lose \ d \ \text{ack } i \ \text{vc} \\
\text{and } h4: \text{msg } (\text{Suc } 0) \ \text{req} \\
\text{and } h5: \text{msg } (\text{Suc } 0) \ \text{stop} \\
\text{and } h6: \text{msg } (\text{Suc } 0) \ a \\
\text{and } h7: ts \ \text{lose} \\
\text{and } h9: \forall j \leq 2 * d + (4 + k). \, \text{lose } (t + j) = \text{False} \\
\text{and } h11: i \, (t + 3 + k + d) \neq [] \\
\text{and } h14: \forall x \leq d + d. \, \text{ack } (t + 3 + k + x) = \text{[sending-data]} \\
\text{shows } \text{vc } (2 * d + (t + (4 + k))) = \text{[vc-com]} \\
\langle \text{proof} \rangle \\
\text{lemma } \text{auxlose1:} \\
\text{assumes } h1: \forall j \leq 2 * d + (4 + k). \, \text{lose } (t + j) = \text{False} \\
\text{and } h2: j \leq k \\
\text{shows } \text{lose } (t + (2::\text{nat} + j)) = \text{False} \\
\langle \text{proof} \rangle \\
\text{lemma } \text{auxlose2:} \\
\text{assumes } \forall j \leq 2 * d + (4 + k). \, \text{lose } (t + j) = \text{False} \\
\text{and } 3 + k + d \leq 2 * d + (4 + k) \\
\text{shows } \text{lose } (t + (3::\text{nat} + k + d)) = \text{False} \\
\langle \text{proof} \rangle \\
\text{lemma } \text{auxreq:} \\
\text{assumes } h1: \forall \, (m::\text{nat}) \leq k + 2. \, \text{req } (t + m) \neq \text{send} \\
\text{and } h2: m \leq k \\
\text{and } h3: \text{req } (t + 2 + m) = \text{send} \text{ shows False} \\
\langle \text{proof} \rangle \\
\text{lemma } \text{GatewaySystem-L2:} \\
\text{assumes } h1: \text{Gateway req dt a stop lose d ack i vc}
Lemma GatewaySystem-L3:
Assumes
\begin{align*}
& h1: \text{Gateway req dt a stop lose d ack vc} \\
& h2: \text{ServiceCenter i a} \\
& h3: \text{GatewayReq req dt a stop lose d ack i vc} \\
& h4: \text{msg (Suc 0) req} \\
& h5: \text{msg (Suc 0) stop} \\
& h6: \text{msg (Suc 0) a} \\
& h7: dt lose \\
& h8: \text{ack t} = \text{[init-state]} \\
& h9: \text{req (Suc t)} = \text{[init]} \\
& h10: \forall t \leq t. \text{req t = [init]} \\
& h11: \forall m \leq k + 2. \text{req (t + m) \neq [send]} \\
& h12: \text{req (t + 3 + k)} = \text{[send]} \\
& h13: \text{inf-last-ti dt t} \neq [] \\
& h14: \forall j \leq 2 * d + (4 + k). \text{lose (t + j) = [False]} \\
\end{align*}

Shows \( \forall j \leq 2 * d + (t + (4 + k)) \) = \text{[vc-com]} 

⟨proof⟩

15.12 Proof of the Refinement for the Gateway System

Lemma GatewaySystem-L0:
Assumes
\begin{align*}
& \text{GatewaySystem req dt stop lose d ack vc} \\
\end{align*}

Shows
\begin{align*}
& \text{GatewaySystemReq req dt stop lose d ack vc} \\
\end{align*}

⟨proof⟩

End

References


