Stream processing components:
Isabelle/HOL formalisation and case studies

Maria Spichkova
November 19, 2013

Abstract
This set of theories presents an Isabelle/HOL formalisation of stream processing components introduces 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 21
5 FOCUS streams: operators and lemmas 23
   5.1 Definition of the FOCUS stream types . . . . . . . . . 23
   5.2 Definitions of operators . . . . . . . . . . . . . . . . . 23
   5.3 Properties of operators . . . . . . . . . . . . . . . . . . 35
      5.3.1 Lemmas for concatenation operator . . . . . . . . 36
      5.3.2 Lemmas for operators $ts$ and $msg$ . . . . . . . 37
      5.3.3 Lemmas for $inf\_truncate$ . . . . . . . . . . . . 39
5.3.4 Lemmas for \textit{fin\_make\_untimed} .................................. 40
5.3.5 Lemmas for \textit{inf\_disj} and \textit{inf\_disjS} ................. 42

6 Properties of time-synchronous streams of types bool and bit 43

7 Changing time granularity of the streams 45
  7.1 Join time units ............................................. 45
  7.2 Split time units ............................................. 49
  7.3 Duality of the split and the join operators ................. 50

8 Steam Boiler System: Specification 51

9 Steam Boiler System: Verification 52
  9.1 Properties of the Boiler Component .......................... 52
  9.2 Properties of the Controller Component ......................... 54
  9.3 Properties of the Converter Component ......................... 65
  9.4 Properties of the System .................................... 65
  9.5 Proof of the Refinement Relation ............................. 68

10 FlexRay: Types 68

11 FlexRay: Specification 69
  11.1 Auxiliary predicates ....................................... 69
  11.2 Specifications of the FlexRay components ..................... 70

12 FlexRay: Verification 72
  12.1 Properties of the function Send ............................ 72
  12.2 Properties of the component Scheduler ......................... 73
  12.3 Disjoint Frames ............................................ 74
  12.4 Properties of the sheaf of channels nSend ..................... 76
  12.5 Properties of the sheaf of channels nGet ..................... 80
  12.6 Properties of the sheaf of channels nStore .................... 82
  12.7 Refinement Properties ..................................... 87

13 Gateway: Types 89

14 Gateway: Specification 89

15 Gateway: Verification 94
  15.1 Properties of the defined data types ......................... 94
  15.2 Properties of the Delay component ........................ 96
  15.3 Properties of the Loss component .......................... 98
  15.4 Properties of the composition of Delay and Loss components 100
  15.5 Auxiliary Lemmas ........................................ 100
15.6 Properties of the ServiceCenter component . . . . . . . . . . 122
15.7 General properties of stream values . . . . . . . . . . . . . . 122
15.8 Properties of the Gateway . . . . . . . . . . . . . . . . . . . . 126
15.9 Proof of the Refinement Relation for the Gateway Requirements 134
15.10 Lemmas about Gateway Requirements . . . . . . . . . . . . 134
15.11 Properties of the Gateway System . . . . . . . . . . . . . . . 135
15.12 Proof of the Refinement for the Gateway System . . . . . . . 141
1 Introduction

The set of theories presented in this paper is an extended and updated Isabelle/HOL 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.

All the operators defined in the presented theories are based on the standard Isabelle/HOL library.

1.2 Case Study 1: Steam Boiler System

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

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

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

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

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

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

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

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

\[
\llbracket *ControlSystemArch* \rrbracket \Rightarrow \llbracket *ControlSystem* \rrbracket
\]

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

**lemma** \(L0-\text{ControlSystem}: \llbracket *ControlSystemArch* \rrbracket \Rightarrow \llbracket *ControlSystem* \rrbracket \Rightarrow *ControlSystem* \)
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 CNI_Buffer component.

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

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

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

• FR\_types.thy – datatype definitions,
• FR.thy – specifications of the system components and auxiliary functions and predicates,
• FR.proof – proof of refinement relation between the requirements and the architecture specifications.

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

\[
\text{type Message} = \text{msg} (\text{message}_\text{id} : \mathbb{N}, \text{ftcdata} : \text{Data})
\]
\[
\text{type Frame} = \text{frm} (\text{slot} : \mathbb{N}, \text{data} : \text{Data})
\]
\[
\text{type Config} = \text{conf} (\text{schedule} : \mathbb{N}^*, \text{cycleLength} : \mathbb{N})
\]

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 `a. The types `a nFrame, `a nNat and `a 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 (return\(_1\), . . . ,return\(_n\)), (c\(_1\), . . . , c\(_n\)), (store\(_1\), . . . , store\(_n\)), (get\(_1\), . . . , get\(_n\)) and (send\(_1\), . . . , send\(_n\)). We also need to declare some constant, sN, for the number of specification replication and the corresponding number of channels in sheaves, as well as to define the list of sheaf upper bounds, sheafNumbers.

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

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

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

- the frame transmission by FlexRay must be correct in terms of the predicate FrameTransmission: if the time $t$ is equal modulo the length of the cycle (FlexRay communication round) to the element of the scheduler table of the node $k$, then this and only this node can send a data at the $t$th time interval;

- FlexRay component sends in every time interval at most one message to each node via channels get$_i$ and store$_i$, $1 \leq i \leq n$).

To show that the specified system fulfill the requirements we need to show that the specification FlexRayArch is a refinement of the specification FlexRay. It follows from the definition of behavioral refinement that in order to verify that $\text{FlexRay} \leadsto \text{FlexRayArch}$ 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:

$\text{FlexRayArch } n \ n\text{Return } nC \ n\text{Store } n\text{Get} \Rightarrow \text{FlexRay } n \ n\text{Return } nC \ n\text{Store } n\text{Get}$

### 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* .th - 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.

```plaintext
type Coordinates = N × N

type CollisionSpeed = N

type ECall_Info = ecall(coord ∈ Coordinates, speed ∈ CollisionSpeed)

type GatewayStatus = {init_state, call, connection_ok,
                      sending_data, voice_com}
```

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

The FOCUS specification of the general gateway system architecture is presented below:
The stream loss is specified to be a time-synchronous one (exactly one message each time interval). It represents the connection status: the message true at the time interval \( t \) corresponds to the connection failure at this time interval, the message false at the time interval \( t \) means that at this time interval no data loss on the gateway connection.

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

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

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

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

We specify the gateway requirements (GatewayReq) as follows:

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

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

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

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

\begin{center}
\begin{tikzpicture}

\node [state] (1) {\textit{Sample}};
\node [state] (2) [right of=1] {\textit{Delay}(d)};
\node [state] (3) [right of=2] {\textit{Loss}};
\node [state] (4) [above left of=1] {\textit{Gateway}(const \(d \in \mathbb{N}\))};

\draw [->] (1) -- node [above] {\textit{req}: \{\textit{init, send}\}} (4);
\draw [->] (2) -- node [above] {\textit{i1}: \textit{ECall\_info}} (4);
\draw [->] (3) -- node [above] {\textit{i2}: \textit{ECall\_info}} (4);
\draw [->] (4) -- node [above] {\textit{dt}: \textit{ECall\_info}} (1);
\draw [->] (1) -- node [above] {\textit{ack}: \textit{Gateway\_status}} (3);
\draw [->] (1) -- node [above] {\textit{vc}: \{\textit{vc\_com}\}} (3);
\draw [->] (1) -- node [above] {\textit{stop}: \{\textit{stop\_vc}\}} (3);
\draw [->] (2) -- node [above] {\textit{a1}: \{\textit{sc\_ack}\}} (4);
\draw [->] (3) -- node [above] {\textit{a2}: \{\textit{sc\_ack}\}} (4);
\draw [->] (3) -- node [above] {\textit{lose}: \textit{Bool}} (4);
\draw [->] (4) -- node [above] {\textit{a}: \{\textit{sc\_ack}\}} (2);
\end{tikzpicture}
\end{center}

The component \textit{Delay} models the communication delay. Its specification is parameterised one: it inherits the parameter of the component \textit{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 \textit{Loss} models the communication loss between the central station and the vehicle gateway: if during time interval \(t\) from the component \textit{Loss} no message about a lost connection comes, the messages come during time interval \(t\) via the input channels \(a\) and \(i2\) will be forwarded without any delay via channels \(a2\) and \(i\) respectively. Otherwise all messages come during time interval \(t\) will be lost.

The component \textit{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 \textit{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 \textit{Sample} we have the assumption, that the streams \textit{req}, \textit{a1}, and \textit{stop} can contain at every time interval at most one message, and also that the stream \textit{loss} must contain at every time interval exactly one message. This component uses local variables \textit{st} and \textit{buffer} (more precisely, a local variable \textit{buffer} and a state variable \textit{st}). The guarantee part of the component \textit{Sample} can be specified as a timed state transition diagram (TSTS) and an expression which says how the local variable \textit{buffer} is computed, or using the corresponding table representation, which is semantically equivalent to the TSTD.
To show that the specified gateway architecture fulfills the requirements we need to show that the specification \textit{Gateway} is a refinement of the specification \textit{GatewayReq}. Therefore, we need to define and to prove the following lemma:

\textbf{lemma} \textit{Gateway-L0}:  
\textit{Gateway Req dt a stop lose d ack i vc}  
\implies \textit{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 \textit{GatewaySystem} is a refinement of the specification \textit{GatewaySystemReq}. Therefore, we need to define and to prove the following lemma:

\textbf{lemma} \textit{GatewaySystem-L0}:  
\textit{GatewaySystem req dt stop lose d ack vc}  
\implies \textit{GatewaySystemReq req dt stop lose d ack vc}

Figure 1: Timed state transition diagram for the component Sample
2 Theory ArithExtras.thy

theory ArithExtras
imports Main
begin

datatype natInf = Fin nat | Infty

primrec nat2inat :: nat list ⇒ natInf list
where
  nat2inat [] = []
  nat2inat (x#xs) = (Fin x) # (nat2inat xs)
end

3 Auxiliary Theory ListExtras.thy

theory ListExtras
imports Main
begin

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

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

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

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

lemma mem-set-1:
  assumes h1: a mem l
  shows a ∈ set l
using assms
proof (induct l)
  case Nil
  from this show ?case
  by auto
next
  fix a la case (Cons a la)
  from Cons show ?case
  by auto
qed 

lemma mem-set-2:
  assumes h1: a ∈ set l
  shows a mem l
using assms
proof (induct l)
  case Nil
  from this show ?case
  by auto
next
  fix a la case (Cons a la)
  from Cons show ?case
  by auto
qed 

lemma set-inter-mem:
  assumes h1: x mem l1
     and h2: x mem l2
  shows set l1 ∩ set l2 ≠ {}
using assms
proof (induct l1)
  case Nil
  from this show ?case
  by auto
next
  fix a la case (Cons a la)
  from Cons show ?case
  by (auto, simp add: mem-set-1)
qed 

lemma mem-notdisjoint:
  assumes h1: x mem l1
     and h2: x mem l2
  shows ¬ disjoint l1 l2
proof
  assume \( sg0: \text{disjoint} \ l1 \ l2 \)
  from \( h1 \) and \( h2 \) have \( sg1: \text{set} \ l1 \cap \text{set} \ l2 \neq \{\} \)
    by (simp add: set-inter-mem)
  from \( h1 \) and \( h2 \) and \( sg1 \) and \( sg0 \) show False
    by (simp add: disjoint-def)
qed

lemma \textit{mem-notdisjoint2}:
  assumes \( h1: \text{disjoint} \ (\text{schedule} \ A) \ (\text{schedule} \ B) \)
  and \( h2: x \in \text{mem} \ (\text{schedule} \ A) \)
  shows \( \neg x \in \text{mem} \ (\text{schedule} \ B) \)
proof
  { assume \( a1: x \in \text{mem} \ (\text{schedule} \ B) \)
    from \( h2 \) and \( a1 \) have \( sg1: \neg \text{disjoint} \ (\text{schedule} \ A) \ (\text{schedule} \ B) \)
      by (simp add: mem-notdisjoint)
    from \( h1 \) and \( sg1 \) have False
      by simp
  }
  from this have \( sg2: \neg x \in \text{mem} \ (\text{schedule} \ B) \)
    by blast
  from this show ?thesis
    by simp
qed

lemma \textit{Add-Less}:
  assumes \( 0 < b \)
  shows \( (\text{Suc} \ a - b < \text{Suc} \ a) = \text{True} \)
using assms by arith

lemma \textit{list-length-hint1}:
  assumes \( l \sim = [] \)
  shows \( 0 < \text{length} \ l \)
using assms by simp

lemma \textit{list-length-hint1a}:
  assumes \( l \sim = [] \)
  shows \( 0 < \text{length} \ l \)
using assms by simp

lemma \textit{list-length-hint2}:
  assumes \( h1: \text{length} \ x = \text{Suc} \ 0 \)
  shows \( [\text{hd} \ x] = x \)
using assms
proof (induct \( x \))
  case Nil
  from this show ?case
    by auto
next
  fix \( a \ l a \) case (Cons \( a \ l a \))
  from Cons show ?case
    by auto
qed
lemma list-length-hint2a:
assumes h1:length l = Suc 0
shows tl l = []
using assms
proof (induct l)
  case Nil
  from this show ?case
  by auto
next
  fix a la case (Cons a la)
  from Cons show ?case
  by auto
qed

lemma list-length-hint3:
assumes length l = Suc 0
shows l ≠ []
using assms
proof (induct l)
  case Nil
  from this show ?case
  by auto
next
  fix a la case (Cons a la)
  from Cons show ?case
  by auto
qed

lemma list-length-hint4:
assumes h1:length x ≤ Suc 0
  and h2:x ≠ []
shows length x = Suc 0
using assms
proof (induct x)
  case Nil
  from this show ?case
  by auto
next
  fix a la case (Cons a la)
  from Cons show ?case
  by auto
qed

lemma length-nonempty:
assumes h1:x ≠ []
shows Suc 0 ≤ length x
using assms
proof (induct x)
  case Nil
from this show \(?\text{case}\) 
by auto
next
fix \(\text{a la case} (\text{Cons}\ \text{a la})\)
from \(\text{Cons}\) show \(?\text{case}\) 
by auto
qed

lemma last-nth-length:
assumes \(x \neq \[]\)
shows \(x ! (\text{length } x - \text{Suc 0}) = \text{last } x\)
using assms
proof (induct \(x\))
  case Nil
  from this show \(?\text{case}\) 
  by auto
next
fix \(\text{a la case} (\text{Cons}\ \text{a la})\)
from \(\text{Cons}\) show \(?\text{case}\) 
by auto
qed

lemma list-nth-append0:
assumes \(h1\):\(i < \text{length } x\)
shows \(x ! i = (x \bullet z) ! i\)
proof (cases \(i\))
  assume \(i=0\)
  with \(h1\) show \(?\text{thesis}\) by (simp add: nth-append)
next
fix \(\text{ii}\) assume \(i = \text{Suc } ii\)
with \(h1\) show \(?\text{thesis}\) by (simp add: nth-append)
qed

lemma list-nth-append1:
assumes \(h1\):\(i < \text{Suc} (\text{length } x)\)
shows \((b \# x) ! i = (b \# x \bullet a \# y) ! i\)
proof -
  from \(h1\) have \(sq1\):\(i < \text{length} (b \# x)\) by auto
  from this have \(sq2\):\((b \# x) ! i = ((b \# x) \bullet y) ! i\)
  by (rule list-nth-append0)
  from this show \(?\text{thesis}\) by simp
qed

lemma list-nth-append2:
assumes \(h1\):\(i < \text{Suc} (\text{length } x)\)
shows \((b \# x) ! i = (b \# x \bullet a \# y) ! i\)
proof -
  from \(h1\) have \(sq1\):\(i < \text{length} (b \# x)\) by auto
  from this have \(sq2\):\((b \# x) ! i = ((b \# x) \bullet (a \# y)) ! i\)
by (rule list-nth-append0)
from this show ?thesis by simp
qed

lemma list-nth-append3:
  assumes h1: i < Suc (length x)
  and h2: i - Suc (length x) < Suc (length y)
  shows (a # y) ! (i - Suc (length x)) = (b # x • a # y) ! i
proof (cases i)
  assume i=0
  with h1 show ?thesis by (simp add: nth-append)
next
  fix ii assume i = Suc ii
  with h1 show ?thesis by (simp add: nth-append)
qed

lemma list-nth-append4:
  assumes h1: i < Suc (length x + length y)
  and h2: i - Suc (length x) < Suc (length y)
  shows False
using assms by arith

lemma list-nth-append5:
  assumes h1: i - length x < Suc (length y)
  and h2: i - Suc (length x) < Suc (length y)
  shows i < Suc (length x + length y)
using assms by arith

lemma list-nth-append6:
  assumes h1: i - length x < Suc (length y)
  and h2: i - Suc (length x) < Suc (length y)
  shows i < Suc (Suc (length x + length y))
using assms by arith

lemma list-nth-append6a:
  assumes h1: i < Suc (length x + length y)
  and h2: i - length x < Suc (length y)
  shows False
using assms by arith

lemma list-nth-append7:
  assumes h1: i - length x < Suc (length y)
  and h2: i - Suc (length x) < Suc (length y)
  shows i < Suc (Suc (length x + length y))
using assms by arith

lemma list-nth-append8:
  assumes h1: i < Suc (length x + length y)
  and h2: i < Suc (Suc (length x + length y))
shows $i = \text{Suc} (\text{length } x + \text{length } y)$
using assms by arith

lemma list-nth-append9:
  assumes h1: $i = \text{Suc} (\text{length } x) < \text{Suc} (\text{length } y)$
  shows $i < \text{Suc} (\text{Suc} (\text{length } x + \text{length } y))$
using assms by arith

lemma list-nth-append10:
  assumes h1: $\neg i < \text{Suc} (\text{length } x)$
  and h2: $\neg i - \text{Suc} (\text{length } x) < \text{Suc} (\text{length } y)$
  shows $\neg i < \text{Suc} (\text{Suc} (\text{length } x + \text{length } y))$
using assms by arith

end

4 Auxiliary arithmetic lemmas

theory arith-hints
imports Main
begin

lemma arith-mod-neq:
  assumes h1: $a \mod n \neq b \mod n$
  shows $a \neq b$
using assms by auto

lemma arith-mod-nzero:
  fixes i :: nat
  assumes h1: $i < n$
  and h2: $0 < i$
  shows $0 < (n \ast t + i) \mod n$
proof -
  from h1 and h2 have sg1:$(i + n \ast t) \mod n = i$
    by (simp add: mod-mult-self2)
  also have sg2:$(n \ast t + i = i + n \ast t \text{ by simp}$
  from this and h1 and h2 show ?thesis
    by (simp (no-asm-simp))
qed

lemma arith-mult-neq-nzero1:
  fixes i :: nat
  assumes h1: $i < n$
  and h2: $0 < i$
  shows $i + n \ast t \neq n \ast q$
proof -
  from h1 and h2 have sg1:$(i + n \ast t) \mod n = i$
    by (simp add: mod-mult-self2)
  also have sg2:$(n \ast q) \mod n = 0 \text{ by simp}$

from this and h1 and h2 have \((i + n \cdot t) \mod n \neq (n \cdot q) \mod n\)  
by simp

from this show \(?thesis by (rule arith-mod-neq)\)

qed

lemma arith-mult-neq-nzero2:
fixes i::nat
assumes h1: \(i < n\)
and h2: \(0 < i\)
shows \(n \cdot t + i \neq n \cdot q\)
proof – 
from h1 and h2 have \(i + n \cdot t \neq n \cdot q\)  
by (rule arith-mult-neq-nzero1)
from this show \(?thesis by simp\)

qed

lemma arith-mult-neq-nzero3:
fixes i::nat
assumes h1: \(i < n\) and h2: \(0 < i\)
shows \(n + n \cdot t + i \neq n \cdot q\)
proof – 
from h1 and h2 have sg1: \(n \cdot (Suc t) + i \neq n \cdot q\)
by (rule arith-mult-neq-nzero2)
have sg2: \(n + n \cdot t + i = n \cdot (Suc t) + i\) by simp
from sg1 and sg2 show \(?thesis by arith\)

qed

lemma arith-modZero1:
\((t + n \cdot t) \mod Suc n = 0\)
proof – 
have \(((Suc n) \cdot t) \mod Suc n = 0\) by (rule mod-mult-self1-is-0)
from this show \(?thesis by simp\)

qed

lemma arith-modZero2:
\(Suc (n + (t + n \cdot t)) \mod Suc n = 0\)
proof – 
have \(((Suc n) \cdot (Suc t)) \mod Suc n = 0\) by (rule mod-mult-self1-is-0)
from this show \(?thesis by simp\)

qed

lemma arith1:
assumes h1:Suc n \* t = Suc n \* q
shows \(t = q\)
proof –
have Suc n \* t = Suc n \* q = (t = q \mid (Suc n) = (0::nat))
by (rule mult-cancel1)
from this and h1 show \(?thesis by simp\)

qed
lemma arith2:
  fixes t n q :: nat
  assumes h1: t + n * t = q + n * q
  shows t = q
proof –
  have sg1: t + n * t = (Suc n) * t by auto
  have sg2: q + n * q = (Suc n) * q by auto
  from h1 and sg1 and sg2 have Suc n * t = Suc n * q by arith
  from this show ?thesis by (rule arith1)
qed

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 iostream = 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 iostream — 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. \exists i. x = (\text{Fin} i) \land i < (\text{length} s) \}\n
— domain of a finite untimed stream
primrec
finU-dom :: 'a list ⇒ nat set
where
finU-dom [] = \{\}\ |
finU-dom (x#xs) = \{\text{length} \ xs\} \cup (\text{finU-dom} \ xs)

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

— range of a finite untimed stream
definition
finU-range :: 'a list ⇒ 'a set
where
finU-range x ≡ \text{set} \ x

— range of an infinite timed stream
definition
infT-range :: 'a istream ⇒ 'a set
where
infT-range s ≡ \{ y. \exists i::\text{nat}. y \ \text{mem} \ (s \ i) \}\n
— range of a finite untimed stream
definition
infU-range :: (nat ⇒ 'a) ⇒ 'a set
where
infU-range s ≡ \{ y. \exists i::\text{nat}. y = (s \ i) \}\n
— range of a (general) stream
definition
stream-range :: 'a stream ⇒ 'a set
where
stream-range s ≡ case s of
  FinT x ⇒ finT-range x
  | FinU x ⇒ finU-range x
  | InfT x ⇒ infT-range x
  | InfU x ⇒ infU-range x

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

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

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

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

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

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

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

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

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

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

27
(case s of (FinT y) ⇒ False
  | (FinU y) ⇒ False
  | (InfT y) ⇒ (∀ i. x i = y i)
  | (InfU y) ⇒ False)
  | (InfU x) ⇒
(case s of (FinT y) ⇒ False
  | (FinU y) ⇒ False
  | (InfT y) ⇒ False
  | (InfU y) ⇒ (∀ i. x i = y i))

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

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

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

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

— concatenation of a finite and an infinite stream
definition
fin-inf-append ::
  'a list ⇒ (nat ⇒ 'a) ⇒ (nat ⇒ 'a)
where

\begin{align*}
\text{fin-inf-append} ~ us ~ s \equiv \\
(\lambda i. \ ( \text{if} \ (i < (\text{length} ~ us)) \\
\text{then} \ (\text{nth} ~ us ~ i) \\
\text{else} ~ s \ (i - (\text{length} ~ us))))
\end{align*}

— insuring that the infinite timed stream is time-synchronous

\textbf{definition}

\begin{align*}
ts :: 'a \text{istream} \Rightarrow \text{bool}
\end{align*}

where

\begin{align*}
ts ~ s \equiv \forall \ i. \ (\text{length} ~ (s ~ i) = 1)
\end{align*}

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

\textbf{definition}

\begin{align*}
msg :: \text{nat} \Rightarrow 'a \text{istream} \Rightarrow \text{bool}
\end{align*}

where

\begin{align*}
msg ~ n ~ s \equiv \forall \ t. \ \text{length} ~ (s ~ t) \leq n
\end{align*}

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

\textbf{primrec}

\begin{align*}
\text{fin-msg} :: \text{nat} \Rightarrow 'a \text{list} \text{list} \Rightarrow \text{bool}
\end{align*}

where

\begin{align*}
\text{fin-msg} ~ n ~ [] = \text{True} \ |
\text{fin-msg} ~ n ~ (x \# xs) = (((\text{length} ~ x) \leq n) \land (\text{fin-msg} ~ n ~ xs))
\end{align*}

— making a finite timed stream to a finite untimed stream

\textbf{definition}

\begin{align*}
\text{fin-make-untimed} :: 'a \text{istream} \Rightarrow 'a \text{list}
\end{align*}

where

\begin{align*}
\text{fin-make-untimed} ~ x \equiv \text{concat} ~ x
\end{align*}

— making an infinite timed stream to an infinite untimed stream

— (auxiliary function)

\textbf{primrec}

\begin{align*}
\text{inf-make-untimed1} :: 'a \text{istream} \Rightarrow \text{nat} \Rightarrow 'a
\end{align*}

where

\begin{align*}
\text{inf-make-untimed1-0:} \\
\text{inf-make-untimed1} ~ s ~ 0 = \text{hd} \ (s \ (\text{LEAST} \ i. (s ~ i) \neq [])) \ | \\
\text{inf-make-untimed1-Suc:} \\
\text{inf-make-untimed1} ~ s ~ (\text{Suc} \ k) = \\
( \text{if} \ ((\text{Suc} ~ k) < \text{length} ~ (s ~ 0)) \\
\text{then} \ \text{nth} ~ (s ~ 0) ~ (\text{Suc} ~ k) \\
\text{else} \ (\text{if} \ (s ~ 0) = [] \\
\text{then} \ (\text{inf-make-untimed1} \ (\text{inf-tl} \ (\text{inf-drop} \\
(\text{LEAST} \ i. \ \forall \ j. \ j < i \rightarrow (s ~ j) = [])) \ k) \\
\text{else} \ \text{inf-make-untimed1} \ (\text{inf-tl} \ s) ~ k))
\end{align*}
— making an infinite timed stream to an infinite untimed stream
— (main function)

**definition**

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

**where**

\[\text{inf-make-untimed} \ s \equiv \lambda \ i. \ \text{inf-make-untimed}1 \ s \ i\]

— making a (general) stream untimed

**definition**

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

**where**

\[\text{make-untimed} \ s \equiv \text{case } s \text{ of} \]

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

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

**primrec**

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

**where**

\[\text{fin-tm} \ [] \ k = k \ |
\text{fin-tm} \ (x \#xs) \ k = \]

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

— auxiliary lemma for the definition of the truncate operator

**lemma** \[\text{inf-tm-hintI}:\]

**assumes**

\[i2 = \text{Suc} \ i - \text{length} \ a\]

**and**

\[\neg \ Suc \ i \leq \text{length} \ a\]

**and**

\[a \neq []\]

**shows**

\[i2 < \text{Suc} \ i\]

**using** \[\text{assms}\]

**by** \[\text{auto}\]

30
— filtering a finite timed stream

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

— filtering an infinite timed stream

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

— removing duplications from a finite timed stream

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

— removing duplications from an infinite timed stream

definition
infT-remdups :: 'a istream => 'a istream
where
infT-remdups s ≡ (λ i. (remdups (s i)))

— removing duplications from a time interval of a stream

primrec
fst-remdups :: 'a list => 'a list
where
fst-remdups [] = [] |
fst-remdups (x#xs) = (if xs = [] then [x] else (if x = (hd xs) then fst-remdups xs else (x#xs)))

— time interval operator

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

— insuring that a sheaf of channels is correctly defined

definition
CorrectSheaf :: nat ⇒ bool
where
CorrectSheaf n ≡ 0 < n
— insuring that all channels in a sheaf are disjunct
— indices in the sheaf are represented using an extra specified set

**definition**

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

**where**

\[ \text{inf-disjS} \text{ IdSet nS} \]

\[ \equiv \forall (t::\text{nat}) \ i \ j. (i::\text{IdSet}) \land (j::\text{IdSet}) \land ((nS i) \ t) \neq [] \rightarrow ((nS j) \ t) = [] \]

— insuring that all channels in a sheaf are disjunct
— indices in the sheaf are represented using natural numbers

**definition**

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

**where**

\[ \text{inf-disj} \text{ n nS} \]

\[ \equiv \forall (t::\text{nat}) \ i \ j. n < \text{length} x \land (i < n) \land (j < n) \land (i \neq j) \land ((nS i) \ t) \neq [] \rightarrow ((nS j) \ t) = [] \]

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

**fun** \[ \text{fin-get-prefix} :: ('a \text{ fstream} \times \text{nat}) \Rightarrow 'a \text{ fstream} \]

**where**

\[ \text{fin-get-prefix}(x, i) = \begin{cases} x \# \text{fin-get-prefix}(xs, i - (\text{length} x)) & \text{if } (\text{length} x) < i \\ \text{take i x} & \text{else} \end{cases} \]

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

**definition**

\[ \text{fin-get-prefix-plus} :: 'a \text{ fstream} \Rightarrow \text{natInf} \Rightarrow 'a \text{ fstream} \]

**where**

\[ \text{fin-get-prefix-plus} \text{ s n} \]

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

— auxiliary lemmas

**lemma** \[ \text{length-inf-drop-hint1} : \]

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

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

**using** \[ \text{assms} \]

**by** \[ \text{(auto simp: inf-drop-def)} \]
lemma length-inf-drop-hint2:
(s 0 ≠ []) → length (inf-drop 0 s 0) < Suc i
→ Suc i − length (inf-drop 0 s 0) < Suc i)
by (simp add: inf-drop-def list-length-hint1)

— taking the prefix of n data elements from an infinite timed stream
— (defined over natural numbers)
fun infT-get-prefix :: ('a istream × nat) ⇒ 'a fstream
where
infT-get-prefix(s, 0) = []

infT-get-prefix(s, Suc i) =
(if (s 0) = []
then (if (∀ i. s i = [])
then []
else (let
k = (LEAST k. s k ≠ [] ∧ (∀ i. i < k → s i = []));
s2 = inf-drop (k+1) s
in (if (length (s k)=0)
then []
else (if (length (s k) < (Suc i))
then s k ≠ infT-get-prefix (s2, Suc i − length (s k))
else [take (Suc i) (s k)]))
)
else
(if ((length (s 0)) < (Suc i))
then (s 0) # infT-get-prefix( inf-drop 1 s, (Suc i) − (length (s 0)))
else [take (Suc i) (s 0)]
)
)

— taking the prefix of n data elements from an infinite untimed stream
— (defined over natural numbers)
primrec
infU-get-prefix :: (nat ⇒ 'a) ⇒ nat ⇒ 'a list
where
infU-get-prefix s 0 = []
infU-get-prefix s (Suc i)
= (infU-get-prefix s i) • [s i]

— taking the prefix of n data elements from an infinite timed stream
— (defined over natural numbers enriched by Infinity)
definition
infT-get-prefix-plus :: 'a istream ⇒ natInf ⇒ 'a stream
where
infT-get-prefix-plus \ s \ n \\
≡ \ case \ n \ of \ (\text{Fin} \ i) \Rightarrow \text{FinT} \ (\text{infT-get-prefix} \ (s, \ i)) \\
| \infty \Rightarrow \text{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 :: (\text{nat} \Rightarrow 'a) \Rightarrow \text{natInf} \Rightarrow 'a \text{ stream}
where
infU-get-prefix-plus \ s \ n \\
≡ \ case \ n \ of \ (\text{Fin} \ i) \Rightarrow \text{FinU} \ (\text{infU-get-prefix} \ s \ i) \\
| \infty \Rightarrow \text{InfU} \ s

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

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

— merging time intervals of two finite timed streams

primrec
fin-merge-ti :: 'a \text{ fstream} \Rightarrow 'a \text{ fstream} \Rightarrow 'a \text{ fstream}
where
fin-merge-ti [] \ y = y \\
fin-merge-ti (x\#xs) \ y = \\
\ (\ case \ y \ of \ [] \Rightarrow (x\#xs) \\
\ | \ (z\#zs) \Rightarrow (x\#z) \# (\text{fin-merge-ti} \ xs \ zs))

— merging time intervals of two infinite timed streams
definition
inf-merge-ti :: 'a \text{ istream} \Rightarrow 'a \text{ istream} \Rightarrow 'a \text{ istream}
where
inf-merge-ti \( x \ y \)
\[
\equiv \\
\lambda \ i. (x \ i) \bullet (y \ i)
\]

— the last time interval of a finite timed stream

\textbf{primrec}
\[
\text{fin-last-ti} :: ('a list) list \Rightarrow \text{nat} \Rightarrow 'a list
\]

\textbf{where}
\[
\text{fin-last-ti} \ s \ \text{0} = \text{hd} \ s \\
\text{fin-last-ti} \ s \ \text{(Suc} \ i) = \\
\quad (\text{if} \ s!\text{(Suc} \ i) \neq [] \\
\qquad \text{then} \ s!\text{(Suc} \ i) \\
\qquad \quad \text{else} \ \text{fin-last-ti} \ s \ i)
\]

— the last nonempty time interval of a finite timed stream
— (can be applied to the streams which time intervals are empty from some moment)

\textbf{primrec}
\[
\text{inf-last-ti} :: 'a istream \Rightarrow \text{nat} \Rightarrow 'a list
\]

\textbf{where}
\[
\text{inf-last-ti} \ s \ \text{0} = s \ \text{0} \\
\text{inf-last-ti} \ s \ \text{(Suc} \ i) = \\
\quad (\text{if} \ s \text{(Suc} \ i) \neq [] \\
\qquad \text{then} \ s \text{(Suc} \ i) \\
\qquad \quad \text{else} \ \text{inf-last-ti} \ s \ i)
\]

\section{5.3 Properties of operators}

\textbf{lemma inf-last-ti-nonempty-k:}
\[
\text{assumes inf-last-ti dt t} \neq [] \\
\text{shows inf-last-ti dt (t + k)} \neq []
\]

\textbf{using assms by (induct k, auto)}

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

\textbf{using assms by (induct k, auto, induct t, auto)}

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

\textbf{by arith}

\textbf{lemma inf-last-ti-Suc2:}
\[
\text{assumes h1:dt (Suc} \ t) \neq [] \lor \text{dt (Suc} \text{(Suc} \ t)) \neq [] \\
\text{shows inf-last-ti dt (t + 2 + k)} \neq []
\]

\textbf{proof (cases dt (Suc} \ t) \neq [])}
\[
\text{assume a1:dt (Suc} \ t) \neq [] \\
\text{from a1 have sg2:inf-last-ti dt ((Suc} \ t) + (\text{Suc} \ k)) \neq []}
\]
by (rule inf-last-ti-nonempty)
from sg2 show ?thesis by (simp add: arith-sum-t2k)
next
assume a2::dt (Suc t) ≠ []
from a2 and h1 have sg1:dt (Suc (Suc t)) ≠ [] by simp
from sg1 have sg2:inf-last-ti dt (Suc (Suc t) + k) ≠ []
  by (rule inf-last-ti-nonempty)
from sg2 show ?thesis by auto
qed

5.3.1 Lemmas for concatenation operator

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

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

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

lemma correct-fin-inf-append2:
  fin-inf-append [x] s (Suc i) = s i
  by (simp add: fin-inf-append-def)

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

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

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

5.3.2 Lemmas for operators \(ts\) and \(msg\)

lemma \(ts\)-msg1:
  assumes \(ts\ \text{p}\)
  shows \(msg\ 1\ \text{p}\)
using assms by (simp add: ts-def msg-def)

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

lemma \(ts\)-length-hint1:
  assumes \(h1:\text{ts}\ \text{x}\)
  shows \(x\ i \neq []\)
proof
  from \(h1\) have \(sg1: length\ (x\ i) = Suc\ 0\) by (simp add: ts-def)
  thus \(?thesis\) by auto
qed

lemma \(ts\)-length-hint2:
  assumes \(h1:\text{ts}\ \text{x}\)
  shows \(length\ (x\ i) = Suc\ (0::nat)\)
using assms
by (simp add: ts-def)

lemma \(ts\)-Least-0:
  assumes \(h1:\text{ts}\ \text{x}\)
  shows \((LEAST\ i.\ (x\ i) \neq []\) = (0::nat)
using assms
proof
  from \(h1\) have \(sg1:x\ 0 \neq []\) by (rule ts-length-hint1)
  thus \(?thesis\)
  apply (simp add: Least-def)
  by auto
qed

lemma inf-tl-Suc:
  inf-tl\ \text{x} \ (i) = x\ (Suc\ i)
  by (simp add: inf-tl-def)
lemma ts-Least-Suc0:
  assumes h1: ts x
  shows \((\mathrm{LEAST} \ i. \ x \ (\mathrm{Suc} \ i) \neq \emptyset) = 0\) proof
    from h1 have sg1: x (Suc 0) \neq \emptyset by (simp add: ts-length-hint1)
    thus ?thesis by (simp add: Least-def, auto)
  qed

lemma ts-inf-make-untimed-inf-tl:
  assumes h1: ts x
  shows inf-make-untimed (inf-tl x) i = inf-make-untimed x (Suc i)
  using assms
  apply (simp add: inf-make-untimed-def)
  proof (induct i)
    case 0 from h1 show ?case by (simp add: ts-length-hint1 ts-length-hint2)
  next case (Suc i)
    from h1 show ?case by (simp add: ts-length-hint1 ts-length-hint2)
  qed

lemma ts-inf-make-untimed1-inf-tl:
  assumes h1: ts x
  shows inf-make-untimed1 (inf-tl x) i = inf-make-untimed1 x (Suc i)
  using assms
  proof (induct i)
    case 0 from h1 show ?case by (simp add: ts-length-hint1 ts-length-hint2)
  next case (Suc i)
    from h1 show ?case by (simp add: ts-length-hint1 ts-length-hint2)
  qed

lemma msg-nonempty1:
  assumes h1: \(\text{msg} \ (\text{Suc} \ 0) \ a\) and h2: \(a \ t = \text{aa} \neq \emptyset\)
  shows \(l = \emptyset\) proof
    from h1 have sg1: \((\text{length} \ (a \ t) \leq \text{Suc} \ 0)\) by (simp add: msg-def)
    from h2 and sg1 show ?thesis by auto
  qed
lemma msg-nonempty2:
  assumes h1: msg (Suc 0) a and h2: a t ≠ []
  shows length (a t) = (Suc 0)
proof -
  from h1 have sg1: length (a t) ≤ Suc 0 by (simp add: msg-def)
  from h2 have sg2: length (a t) ≠ 0 by auto
  from sg1 and sg2 show ?thesis by arith
qed

5.3.3 Lemmas for inf_truncate

lemma inf-truncate-nonempty:
  assumes h1: z i ≠ []
  shows inf-truncate z i ≠ []
proof (induct i)
  case 0
  from h1 show ?case by auto
next
  case (Suc i)
  from h1 show ?case by auto
qed

lemma concat-inf-truncate-nonempty:
  assumes h1: z i ≠ []
  shows concat (inf-truncate z i) ≠ []
using assms
proof (induct i)
  case 0
  thus ?case by auto
next
  case (Suc i)
  thus ?case by auto
qed

lemma concat-inf-truncate-nonempty-a:
  assumes h1: z i = [a]
  shows concat (inf-truncate z i) ≠ []
using assms
proof (induct i)
  case 0
  thus ?case by auto
next
  case (Suc i)
  thus ?case by auto
qed
lemma concat-inf-truncate-nonempty-el:
  assumes h1: z i ≠ []
  shows concat (inf-truncate z i) ≠ []
using assms
proof (induct i)
  case 0
  thus ?case by auto
next
  case (Suc i)
  thus ?case by auto
qed

lemma inf-truncate-append:
  (inf-truncate z i • [z (Suc i)]) = inf-truncate z (Suc i)
proof (induct i)
  case 0
  thus ?case by auto
next
  case (Suc i)
  thus ?case by auto
qed

5.3.4 Lemmas for fin_make_unptime

lemma fin-make-untimed-append:
  assumes h1: fin-make-untimed x ≠ []
  shows fin-make-untimed (x • y) ≠ []
using assms by (simp add: fin-make-untimed-def)

lemma fin-make-untimed-inf-truncate-Nonempty:
  assumes h1: z k ≠ []
    and h2: k ≤ i
  shows fin-make-untimed (inf-truncate z i) ≠ []
using assms
apply (simp add: fin-make-untimed-def)
proof (induct i)
  case 0
  thus ?case by auto
next
  case (Suc i)
  thus ?case
proof cases
  assume k ≤ i
  from Suc and this show ∃ xs ∈ set (inf-truncate z (Suc i)). xs ≠ []
    by auto

40
next
assume ¬ k ≤ i
from Suc and this have sg1:k = Suc i by arith
from Suc and this show ∃ xs∈ set (inf-truncate z (Suc i)). xs ≠ []
  by auto
qed
qed

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

lemma last-fin-make-untimed-inf-truncate:
assumes h1: z i = [a]
shows last (fin-make-untimed (inf-truncate z i)) = a
using assms
proof (induction i)
  case 0
  from this show ?case by (simp add: fin-make-untimed-def)
next
  case (Suc i)
  thus ?case by (simp add: fin-make-untimed-def)
qed

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

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

lemma fin-make-untimed-inf-truncate-Nonempty-all:
assumes h1: z k ≠ []
shows ∀ i. k ≤ i −→ fin-make-untimed (inf-truncate z i) ≠ []
using assms by (simp add: fin-make-untimed-inf-truncate-Nonempty)

lemma fin-make-untimed-inf-truncate-Nonempty-all0:
assumes h1: z 0 ≠ []
shows ∀ i. fin-make-untimed (inf-truncate z i) ≠ []
using assms by (simp add: fin-make-untimed-inf-truncate-Nonempty)
lemma fin-make-untimed-inf-truncate-Nonempty-all0a:
    assumes h1: z 0 = [a]
    shows \( \forall \, i. \, \text{fin-make-untimed} \, (\text{inf-truncate} \, z \, i) \neq [] \)
    using assms by (simp add: fin-make-untimed-inf-truncate-Nonempty-all0)

lemma fin-make-untimed-inf-truncate-Nonempty-all-app:
    assumes h1: z 0 = [a]
    shows \( \forall \, i. \, \text{fin-make-untimed} \, (\text{inf-truncate} \, z \, i \bullet [z \, (Suc \, i)]) \neq [] \)
    proof
      fix i
      from h1 have sg1: fin-make-untimed (inf-truncate z i) \( \neq [] \)
        by (simp add: fin-make-untimed-inf-truncate-Nonempty-all0a)
      from h1 and sg1 show fin-make-untimed (inf-truncate z i \bullet [z \, (Suc \, i)]) \( \neq [] \)
        by (simp add: fin-make-untimed-append)
    qed

lemma fin-make-untimed-nth-length:
    assumes h1: z i = [a]
    shows fin-make-untimed (inf-truncate z i) ! (length (fin-make-untimed (inf-truncate z i)) - Suc 0) = a
    proof
      from h1 have sg1: last (fin-make-untimed (inf-truncate z i)) = a
        by (simp add: last-fin-make-untimed-inf-truncate)
      from h1 have sg2: concat (inf-truncate z i) \( \neq [] \)
        by (rule concat-inf-truncate-nonempty-a)
      from h1 and sg1 and sg2 show \(?thesis \)
        by (simp add: fin-make-untimed-def last-nth-length)
    qed

5.3.5 Lemmas for inf_disj and inf_disjS

lemma inf-disj-index:
    assumes h1: inf-disj n nS
    and h2: nS k t \( \neq [] \)
    and h3: k < n
    shows (SOME i. i < n \( \land \) nS i t \( \neq [] \)) = k
    proof
      from h1 have \( \forall \, j. \, k < n \land j < n \land k \neq j \land nS \, k \, t \neq [] \, \rightarrow \, nS \, j \, t = [] \)
        by (simp add: inf-disj-def, auto)
      from this and assms show \(?thesis \) by auto
    qed
lemma \textit{inf-disjS-index}:
\begin{itemize}
\item assumes \( h1: \text{inf-disjS} \) \( \text{IdSet} \) \( nS \)
\item and \( h2: k: \text{IdSet} \)
\item and \( h3: nS \) \( k \) \( t \not\in [] \)
\end{itemize}
shows \( (\text{SOME} \ i. \ (i: \text{IdSet}) \land \text{nsEnd} \ i \ t \not\in []) = k \)
\begin{proof}
from \( h1 \) have \( \forall \ j. \ k \in \text{IdSet} \land j \in \text{IdSet} \land nS \) \( k \) \( t \not\in [] \longrightarrow nS \) \( j \) \( t = [] \)
by \( (\text{simp add: inf-disjS-def, auto}) \)
from this and \( \text{assms} \) show \( \text{thesis} \) by \( \text{auto} \)
\end{proof}
qed

end

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

theory \textit{BitBoolTS}
imports \textit{Main} \textit{stream}
begin

\textbf{datatype} \textit{bit} = \textit{Zero} | \textit{One}

\textbf{primrec}
\begin{itemize}
\item \textit{negation} :: \textit{bit} \Rightarrow \textit{bit}
\end{itemize}
\textbf{where}
\begin{itemize}
\item \textit{negation} \textit{Zero} = \textit{One} |
\item \textit{negation} \textit{One} = \textit{Zero}
\end{itemize}

\textbf{lemma} \textit{ts-bit-stream-One}:
\begin{itemize}
\item assumes \( h1: \text{ts} \) \( x \)
\item and \( h2: x \) \( i \not\in [\text{Zero}] \)
\end{itemize}
shows \( x \) \( i = [\text{One}] \)
\begin{proof}
from \( h1 \) have \( \text{sg1: length} \ (x \ i) = \text{Suc} \ 0 \)
by \( (\text{simp add: ts-def}) \)
from this and \( h2 \) show \( \text{thesis} \)
proof \( (\text{cases} \ x \ i) \)
\begin{itemize}
\item assume \( \text{Nil:} \ x \ i = [] \)
\item from this and \( \text{sg1} \) show \( \text{thesis} \) by \( \text{simp} \)
\end{itemize}
next
\begin{itemize}
\item fix \( a \ l \) assume \( \text{Cons:} \ x \ i = a \not\in l \)
\item from this and \( \text{sg1} \) and \( h2 \) show \( \text{thesis} \)
\item proof \( (\text{cases} \ a) \)
\end{itemize}
\begin{itemize}
\item assume \( a = \text{Zero} \)
\item from this and \( \text{sg1} \) and \( h2 \) and \( \text{Cons} \) show \( \text{thesis} \) by \( \text{auto} \)
\end{itemize}
\end{proof}
\end{document}
assume \( a = \text{One} \)
from this and \( sg1 \) and Cons show \( \text{thesis} \) by auto
qed
qed

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

lemma ts-bool-True:
assumes \( h1:ts \ x \)
and \( h2:x \ i \neq [\text{False}] \)
shows \( x \ i = [\text{True}] \)
proof
  from \( h1 \) have \( sg1:length \ (x \ i) = \text{Suc 0} \)
  by (simp add: ts-def)
from this and \( h2 \) show \( \text{thesis} \)
proof (cases \( x \ i \))
  assume Nil: \( x \ i = [] \)
  from this and \( sg1 \) show \( \text{thesis} \) by simp
next
fix \( a \ l \) assume Cons:\( x \ i = a \ # \ l \)
from this and \( sg1 \) have \( sg2:x \ i = [a] \) by simp
from this and \( h2 \) show \( \text{thesis} \) by auto
qed
qed
lemma ts-bool-False:
  assumes h1:ts x
  and h2:x i ≠ [True]
  shows x i = [False]
proof –
  from h1 have sg1:length (x i) = Suc 0
    by (simp add: ts-def)
  from this and h2 show ?thesis
  proof (cases x i)
    assume Nil:x i = []
    from this and sg1 show ?thesis by simp
  next
    fix a l assume Cons:x i = a # l
    from this and sg1 have sg2:x i = [a] by simp
  from this and h2 show ?thesis by auto
qed
qed

lemma ts-bool-True-False:
  fixes x::bool istream
  assumes h1:ts x
  shows x i = [True] ∨ x i = [False]
proof (cases x i = [True])
  assume x i = [True]
  from this and h1 show ?thesis by simp
next
  assume x i ≠ [True]
  from this and h1 show ?thesis by (simp add: ts-bool-False)
qed

end

7 Changing time granularity of the streams

theory JoinSplitTime
imports stream arith-hints
begin

7.1 Join time units

primrec join-ti :: 'a istream ⇒ nat ⇒ nat ⇒ 'a list
where
  join-ti-0: join-ti s x 0 = s x
  join-ti-Suc:
\[ \text{join-ti } s \ x \ (\text{Suc } i) = (\text{join-ti } s \ x \ i) \bullet (s \ (x + (\text{Suc } i))) \]

primrec
\[
\text{fin-join-ti} : 'a fstream \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \ list
\]
where
\[
\text{fin-join-ti-0:} \quad \text{fin-join-ti } s \ x \ 0 = \text{nth } s \ x \\
\text{fin-join-ti-Suc:} \quad \text{fin-join-ti } s \ x \ (\text{Suc } i) = (\text{fin-join-ti } s \ x \ i) \bullet (\text{nth } s \ (x + (\text{Suc } i)))
\]

definition
\[
\text{join-time} : 'a istream \Rightarrow \text{nat} \Rightarrow 'a \ istream
\]
where
\[
\text{join-time } s \ n \ t \equiv \\
\text{(case } n \text{ of} \\
\quad 0 \Rightarrow [] \\
\quad (\text{Suc } i) \Rightarrow \text{join-ti } s \ (n \ast \ i))
\]

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

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

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

lemma join-ti-empty-join:
\[
\text{assumes} \ h1 : i \leq n \\
\quad \text{and } h2 : \text{join-ti } s \ x \ n = [] \\
\text{shows} \quad s \ (x + i) = []
\]
using assms
proof (induct n)
\[
\text{case } 0 \\
\from \text{this show } \text{case by auto}
\]
next
\[
\text{case } (\text{Suc } n) \\
\from \text{this show } \text{case}
\]
proof (cases i = Suc n)
assume \( a1: i = \text{Suc} \ n \)
from \( a1 \) and \( \text{Suc} \) show \(?thesis\) by simp
next
assume \( a2: i \neq \text{Suc} \ n \)
from \( a2 \) and \( \text{Suc} \) show \(?thesis\) by simp
qed

lemma \( \text{join-ti-empty-ti} \): 
assumes \( \forall \ i \leq n. \ s \ (x+i) = [] \)
shows \( \text{join-ti} \ s \ x \ n = [] \)
using \( \text{assms} \) by (induct \( n \), auto)

lemma \( \text{join-ti-1nempty} \): 
assumes \( \forall \ i. \ 0 < i \land i < \text{Suc} \ n \longrightarrow s \ (x+i) = [] \)
shows \( \text{join-ti} \ s \ x \ n = s \ x \)
using \( \text{assms} \) by (induct \( n \), auto)

lemma \( \text{join-time11} \): \( \forall \ t. \ \text{join-time} \ s \ (1::\text{nat}) \ t = s \ t \)
by (simp add: join-time-def)

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

lemma \( \text{join-time-empty1} \): 
assumes \( h1: i < n \)
and \( h2: \text{join-time} \ s \ n \ t = [] \)
shows \( s \ (n*t + i) = [] \)
proof (cases \( n \))
assume \( a1: n = 0 \)
from \( \text{assms} \) and \( a1 \) show ?thesis by (simp add: join-time-def)
next
fix \( x \)
assume \( a2: n = \text{Suc} \ x \)
from \( \text{assms} \) and \( a2 \) have \( sg1: \text{join-ti} \ s \ (t + x \cdot t) \ x = [] \)
by (simp add: join-time-def)
from \( a2 \) and \( h1 \) have \( sg2: i \leq x \) by simp
from \( sg2 \) and \( sg1 \) and \( a2 \) show ?thesis by (simp add: join-ti-empty-join)
qed

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

47
lemma fin-join-ti-hint2:
assumes \( \text{fin-join-ti } s \ x \ (\text{Suc } i) = [] \)
shows \( \text{nth } s \ (x + (\text{Suc } i)) = [] \)
using assms by auto

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

lemma fin-join-ti-empty-join:
assumes h1: \( i \leq n \)
and h2: \( \text{fin-join-ti } s \ x \ n = [] \)
shows \( \text{nth } s \ (x + i) = [] \)
using assms
proof (induct n)
case 0
from this show ?case by auto
next
case (Suc n)
from this show ?case
proof (cases i = Suc n)
assume a1: \( i = \text{Suc } n \)
from Suc and a1 show ?thesis by simp
next
assume a2: \( i \neq \text{Suc } n \)
from Suc and a2 show ?thesis by simp
qed
qed

lemma fin-join-ti-empty-ti:
assumes \( \forall i \leq n. \text{nth } s \ (x+i) = [] \)
shows \( \text{fin-join-ti } s \ x \ n = [] \)
using assms by (induct n, auto)

lemma fin-join-ti-1nempty:
assumes \( \forall i. 0 < i \land i < \text{Suc } n \implies \text{nth } s \ (x+i) = [] \)
shows \( \text{fin-join-ti } s \ x \ n = \text{nth } s \ x \)
using assms by (induct n, auto)
7.2 Split time units

definition
  split-time :: 'a istream ⇒ nat ⇒ 'a istream

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

lemma split-time11: ∀ t. split-time s 1 t = s t
by (simp add: split-time-def)

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

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

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

lemma split-time-nempty-Suc:
  assumes h1: i < n and h2: 0 < i
  shows split-time s n (n * t + i) = []
proof
  have sg0: 0 < Suc n by simp
  from sg0 have sg1: split-time s (Suc n) ((Suc n) * t) = s t
    by (rule split-time-nempty)
  from assms have sg2: split-time s n (n * t) = s t
    by (rule split-time-nempty)
  from sg1 and sg2 show ?thesis by simp
qed

lemma split-time-empty:
  assumes h1: i < n and h2: 0 < i
  shows split-time s n (n * t + i) = []
proof
  from assms have sg1: 0 < (n * t + i) mod n by (simp add: arith-mod-nzero)
  from assms and sg1 show ?thesis by (simp add: split-time-def)
qed

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

from \( h1 \) have \( sg1: i < \text{Suc} \ n \) by simp 
from \( sg1 \) and \( h2 \) have \( sg2:\text{split-time} \ s \ (\text{Suc} \ n) \ (\text{Suc} \ n * t + i) = [] \) 
  by (rule split-time-empty) 
from assms have \( sg3:\text{split-time} \ s \ n \ (n * t + i) = [] \) 
  by (rule split-time-empty) 
from \( sg3 \) and \( sg2 \) show ?thesis by simp 
qed 

lemma split-time-hint1: 
  assumes \( n = \text{Suc} \ m \) 
  shows \( \text{split-time} \ s \ (\text{Suc} \ n) \ (i + n * i + n) = [] \) 
proof 
  have \( sg1:i + n * i + n = (\text{Suc} \ n) * i + n \) by simp 
  have \( sg2:n < \text{Suc} \ n \) by simp 
  from assms have \( sg3:0 < n \) by simp 
  from \( sg2 \) and \( sg3 \) have \( \forall j. \quad \text{split-time} \ s \ (\text{Suc} \ n) \ (\text{Suc} \ n * i + n) = [] \) 
    by (clarify, rule split-time-empty, auto) 
  from \( sg1 \) and \( sg4 \) show ?thesis by auto 
qed 

7.3 Duality of the split and the join operators 

lemma join-split-i: 
  assumes \( 0 < n \) 
  shows \( \text{join-time} \ (\text{split-time} \ s \ n) \ n i = s i \) 
proof (cases \( n \)) 
  assume \( a1:n = 0 \) 
  from this and assms show ?thesis by simp 
next 
  fix \( k \) 
  assume \( a2:n = \text{Suc} \ k \) 
  have \( sg0:0 < \text{Suc} \ k \) by simp 
  from \( sg0 \) have \( sg1: (\text{split-time} \ s \ (\text{Suc} \ k)) \ (\text{Suc} \ k * i) = s i \) 
    by (rule split-time-nempty) 
  have \( sg2:i + k * i = (\text{Suc} \ k) * i \) by simp 
  have \( sg3:\forall j. \quad 0 < j \land j < \text{Suc} \ k \quad \rightarrow \quad \text{split-time} \ s \ (\text{Suc} \ k) \ (\text{Suc} \ k * i + j) = [] \) 
    by (clarify, rule split-time-empty, auto) 
  from \( sg3 \) have \( \text{join-ti} \ (\text{split-time} \ s \ (\text{Suc} \ k)) \ ((\text{Suc} \ k) * i) \ k = \) 
    \( \text{split-time} \ s \ (\text{Suc} \ k)) \ (\text{Suc} \ k * i) \) 
    by (rule join-ti-Inempty) 
  from \( a2 \) and \( sg4 \) and \( sg1 \) show ?thesis by (simp add: join-time-def) 
qed 

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

end
8 Steam Boiler System: Specification

theory SteamBoiler
imports stream BitBoolTS
begin

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

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

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

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

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

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

9 Steam Boiler System: Verification

theory SteamBoiler-proof
imports SteamBoiler
begin

9.1 Properties of the Boiler Component

lemma L1-Boiler:
  assumes h1: SteamBoiler x s y
  and h2: ts x
  shows ts s
  using assms by (simp add: SteamBoiler-def)

lemma L2-Boiler:
  assumes h1: SteamBoiler x s y
Lemma L3-Boiler:
assumes h1:SteamBoiler x s y
and h2:ts x
shows 200 ≤ hd (s 0)
using assms by (simp add: SteamBoiler-def)

Lemma L4-Boiler:
assumes h1:SteamBoiler x s y
and h2:ts x
shows hd (s 0) ≤ 800
using assms by (simp add: SteamBoiler-def)

Lemma L5-Boiler:
assumes h1:SteamBoiler x s y
and h2:ts x
and h3:hd (x j) = Zero
shows (hd (s j)) ≤ hd (s (Suc j)) + (10::nat)
proof -
  from h1 and h2 obtain r where
  a1:r ≤ 10 and
  a2:hd (s (Suc j)) = (if hd (x j) = Zero then hd (s j) − r else hd (s j) + r)
  by (simp add: SteamBoiler-def, auto)
  from a2 and h3 have sg1:hd (s (Suc j)) = hd (s j) − r by simp
  from sg1 and a1 show ?thesis by auto
qed

Lemma L6-Boiler:
assumes h1:SteamBoiler x s y
and h2:ts x
and h3:hd (x j) = Zero
shows (hd (s j)) − (10::nat) ≤ hd (s (Suc j))
proof -
  from h1 and h2 obtain r where
  a1:r ≤ 10 and
  a2:hd (s (Suc j)) = (if hd (x j) = Zero then hd (s j) − r else hd (s j) + r)
  by (simp add: SteamBoiler-def, auto)
  from a2 and h3 have sg1:hd (s (Suc j)) = hd (s j) − r by simp
  from sg1 and a1 show ?thesis by auto
qed
lemma L7-Boiler:
  assumes h1:SteamBoiler x s y
  and h2:ts x
  and h3:hd (x j) ≠ Zero
  shows (hd (s j)) ≥ hd (s (Suc j)) - (10::nat)
using assms
proof –
  from h1 and h2 obtain r where
  a1:r ≤ 10  and
  a2:hd (s (Suc j)) = (if hd (x j) = Zero then hd (s j) - r else hd (s j) + r)
  by (simp add: SteamBoiler-def, auto)
from a2 and h3 have sg1:hd (s (Suc j)) = hd (s j) + r by simp
from sg1 and a1 show ?thesis by auto
qed

lemma L8-Boiler:
  assumes h1:SteamBoiler x s y
  and h2:ts x
  and h3:hd (x j) ≠ Zero
  shows (hd (s j)) + (10::nat) ≥ hd (s (Suc j))
proof –
  from h1 and h2 obtain r where
  a1:r ≤ 10  and
  a2:hd (s (Suc j)) = (if hd (x j) = Zero then hd (s j) - r else hd (s j) + r)
  by (simp add: SteamBoiler-def, auto)
from a2 and h3 have sg1:hd (s (Suc j)) = hd (s j) + r by simp
from sg1 and a1 show ?thesis by auto
qed

9.2 Properties of the Controller Component

lemma L1-Controller:
  assumes h1:Controller-L s (fin-inf-append [Zero] l) l z
  shows fin-make-untimed (inf-truncate z i) ≠ []
proof –
  from h1 have ∀ i. 0 ≤ i → fin-make-untimed (inf-truncate z i) ≠ []
  by (simp add: Controller-L-def fin-make-untimed-inf-truncate-Nonempty-all0a)
from this show ?thesis by simp
qed

lemma L2-Controller-Zero:
  assumes h1:Controller-L y (fin-inf-append [Zero] l) l z
  and h2:l t = Zero
  and h3:300 < hd (y (Suc t))
  shows z (Suc t) = []
proof –
  from h2 have sg1:fin-inf-append [Zero] l (Suc t) = Zero

54
by (simp add: correct-fin-inf-append1)
from h1 and sq1 and h3 show ?thesis by (simp add: Controller-L-def)
qed

lemma L2-Controller-One:
assumes h1:Controller-L y (fin-inf-append [Zero] l) l z
and h2:l t = One
and h3:hd (y (Suc t)) < 700
shows z (Suc t) = []
proof –
from h2 have sq1:fin-inf-append [Zero] l (Suc t) ≠ Zero
by (simp add: correct-fin-inf-append1)
from h1 and sq1 and h3 show ?thesis by (simp add: Controller-L-def)
qed

lemma L3-Controller-Zero:
assumes h1:Controller-L y (fin-inf-append [Zero] l) l z
and h2:l t = Zero
and h3:- 300 < hd (y (Suc t))
shows z (Suc t) = [One]
proof –
from h2 have sq1:fin-inf-append [Zero] l (Suc t) = Zero
by (simp add: correct-fin-inf-append1)
from h1 and sq1 and h3 show ?thesis by (simp add: Controller-L-def)
qed

lemma L3-Controller-One:
assumes h1:Controller-L y (fin-inf-append [Zero] l) l z
and h2:l t = One
and h3:- hd (y (Suc t)) < 700
shows z (Suc t) = [Zero]
proof –
from h2 have sq1:fin-inf-append [Zero] l (Suc t) ≠ Zero
by (simp add: correct-fin-inf-append1)
from h1 and sq1 and h3 show ?thesis by (simp add: Controller-L-def)
qed

lemma L4-Controller-Zero:
assumes h1:Controller-L y (fin-inf-append [Zero] l) l z
and h2:l (Suc t) = Zero
shows (z (Suc t) = [] ∧ l t = Zero) ∨ (z (Suc t) = [Zero] ∧ l t = One)
proof (cases l t)
assume a1:l t = Zero
from this and h1 and h2 show ?thesis
proof –

55
from a1 have sg1: fin-inf-append [Zero] l (Suc t) = Zero
  by (simp add: correct-fin-inf-append1)
from h1 and sg1 have sg2:
  if 300 < hd (y (Suc t))
  then z (Suc t) = [] ∧ l (Suc t) = Zero
  else z (Suc t) = [One] ∧ l (Suc t) = One
  by (simp add: Controller-L-def)
show ?thesis
proof (cases 300 < hd (y (Suc t)))
  assume a11: 300 < hd (y (Suc t))
  from a11 and sg2 have sg3: z (Suc t) = [] ∧ l (Suc t) = Zero by simp
  from this and a1 show ?thesis by simp
next
  assume a12:∀ 300 < hd (y (Suc t))
  from a12 and sg2 have sg4: z (Suc t) = [One] ∧ l (Suc t) = One by simp
  from this and h2 show ?thesis by simp
qed
qed

next
assume a2: l t = One
from this and h1 and h2 show ?thesis
proof -
  from a2 have sg5: fin-inf-append [Zero] l (Suc t) ≠ Zero
    by (simp add: correct-fin-inf-append1)
from h1 and sg5 have sg6:
  if hd (y (Suc t)) < 700
  then z (Suc t) = [] ∧ l (Suc t) = One
  else z (Suc t) = [Zero] ∧ l (Suc t) = Zero
  by (simp add: Controller-L-def)
show ?thesis
proof (cases hd (y (Suc t)) < 700)
  assume a21: hd (y (Suc t)) < 700
  from a21 and sg6 have sg7: z (Suc t) = [] ∧ l (Suc t) = One by simp
  from this and h2 show ?thesis by simp
next
  assume a22:∀ hd (y (Suc t)) < 700
  from a22 and sg6 have sg8: z (Suc t) = [Zero] ∧ l (Suc t) = Zero by simp
  from this and a2 show ?thesis by simp
qed
qed

lemma L4-Controller-One:
  assumes h1: Controller-L y (fin-inf-append [Zero] l) l z
    and h2: l (Suc t) = One
  shows (z (Suc t) = [] ∧ l t = One) ∨ (z (Suc t) = [One] ∧ l t = Zero)
proof (cases l t)
  assume a1: l t = Zero

56
from this and h1 and h2 show \( ?thesis \)
proof -
from \( a1 \) have \( sg1 : \text{fin-inf-append} [\text{Zero}] \ l \ (\text{Suc} \ t) = \text{Zero} \)
  by (simp add: correct-fin-inf-append1)
from h1 and sg1 have sg2:
  if 300 < hd \((y \ (\text{Suc} \ t))\)
    then \( z \ (\text{Suc} \ t) = \text{[]} \land l \ (\text{Suc} \ t) = \text{Zero} \)
    else \( z \ (\text{Suc} \ t) = \text{[One]} \land l \ (\text{Suc} \ t) = \text{One} \)
  by (simp add: Controller-L-def)
show \( ?thesis \)
proof (cases 300 < hd \((y \ (\text{Suc} \ t)))\))
assume a11:300 < hd \((y \ (\text{Suc} \ t))\)
from a11 and sg2 have sg3: \( z \ (\text{Suc} \ t) = \text{[]} \land l \ (\text{Suc} \ t) = \text{Zero} \)
  by simp
from this and h2 show \( ?thesis \) by simp
next
assume a12:¬ 300 < hd \((y \ (\text{Suc} \ t))\)
from a12 and sg2 have sg4: \( z \ (\text{Suc} \ t) = \text{[One]} \land l \ (\text{Suc} \ t) = \text{One} \)
  by simp
from this and a1 show \( ?thesis \) by simp
qed
qed
next
assume a2:l t = One
from this and h1 and h2 show \( ?thesis \)
proof -
from a2 have \( sg5 : \text{fin-inf-append} [\text{Zero}] \ l \ (\text{Suc} \ t) \neq \text{Zero} \)
  by (simp add: correct-fin-inf-append1)
from h1 and sg5 have sg6:
  if \( \text{hd} \ (y \ (\text{Suc} \ t)) < 700 \)
    then \( z \ (\text{Suc} \ t) = \text{[]} \land l \ (\text{Suc} \ t) = \text{One} \)
    else \( z \ (\text{Suc} \ t) = \text{[Zero]} \land l \ (\text{Suc} \ t) = \text{Zero} \)
  by (simp add: Controller-L-def)
show \( ?thesis \)
proof (cases \( \text{hd} \ (y \ (\text{Suc} \ t)) < 700 \))
assume a21:hd \((y \ (\text{Suc} \ t)) < 700 \)
from a21 and sg6 have sg7: \( z \ (\text{Suc} \ t) = \text{[]} \land l \ (\text{Suc} \ t) = \text{One} \)
  by simp
from this and a2 show \( ?thesis \) by simp
next
assume a22:¬ \( \text{hd} \ (y \ (\text{Suc} \ t)) < 700 \)
from a22 and sg6 have sg8: \( z \ (\text{Suc} \ t) = \text{[Zero]} \land l \ (\text{Suc} \ t) = \text{Zero} \)
  by simp
from this and h2 show \( ?thesis \) by simp
qed
qed

lemma L5-Controller-Zero:
  assumes h1:Controller-L \( y \ lIn \ lOut \ z \)
  and h2:|lOut t = Zero
  and h3:|z t = |lIn t = Zero
  shows lIn t = Zero
proof (cases \(\text{ln} \, t\))
  assume \(a1:\text{ln} \, t = \text{Zero}\)
  from this show \(?\text{thesis}\) by simp
next
  assume \(a2:\text{ln} \, t = \text{One}\)
  from \(a2\) and \(h1\) have \(sg1\):
    if \(\text{hd} \, (y \, t) < 700\)
    then \(z \, t = [\text{}] \land \text{loOut} \, t = \text{One}\)
    else \(z \, t = [\text{Zero}] \land \text{loOut} \, t = \text{Zero}\)
    by (simp add: Controller-L-def)
show \(?\text{thesis}\)
proof (cases \(\text{hd} \, (y \, t) < 700\))
  assume \(a3:\text{hd} \, (y \, t) < 700\)
  from \(a3\) and \(sg1\) have \(sg2\):
    if \(300 < \text{hd} \, (y \, t)\)
    then \(z \, t = [\text{}] \land \text{loOut} \, t = \text{Zero}\)
    else \(z \, t = [\text{One}] \land \text{loOut} \, t = \text{One}\)
    by (simp add: Controller-L-def)
  from this and \(h2\) show \(?\text{thesis}\) by simp
next
  assume \(a4:\neg \text{hd} \, (y \, t) < 700\)
  from \(a4\) and \(sg1\) have \(sg3\):
    if \(\neg \text{hd} \, (y \, t) < 300\)
    then \(z \, t = [\text{Zero}] \land \text{loOut} \, t = \text{Zero}\)
    else \(z \, t = [\text{One}] \land \text{loOut} \, t = \text{One}\)
    by (simp add: Controller-L-def)
  from this and \(h3\) show \(?\text{thesis}\) by simp
qed
next
  assume \(a2:\text{ln} \, t = \text{One}\)
  from this show \(?\text{thesis}\) by simp
qed

lemma \(\text{L5-Controller-One}\):
  assumes \(h1:\text{Controller-L} \, y \, \text{ln} \, \text{loOut} \, z\)
    and \(h2:\text{loOut} \, t = \text{One}\)
    and \(h3:\text{z} \, t = [\text{}]\)
  shows \(\text{ln} \, t = \text{One}\)
proof (cases \(\text{ln} \, t\))
  assume \(a1:\text{ln} \, t = \text{Zero}\)
  from \(a1\) and \(h1\) have \(sg1\):
    if \(300 < \text{hd} \, (y \, t)\)
    then \(z \, t = [\text{}] \land \text{loOut} \, t = \text{Zero}\)
    else \(z \, t = [\text{One}] \land \text{loOut} \, t = \text{One}\)
    by (simp add: Controller-L-def)
  from this and \(h2\) show \(?\text{thesis}\) by simp
next
  assume \(a4:\neg \text{hd} \, (y \, t) < 300\)
  from \(a4\) and \(sg1\) have \(sg2\):
    if \(\neg \text{hd} \, (y \, t) < 300\)
    then \(z \, t = [\text{}] \land \text{loOut} \, t = \text{Zero}\)
    else \(z \, t = [\text{One}] \land \text{loOut} \, t = \text{One}\)
    by (simp add: Controller-L-def)
  from this and \(h3\) show \(?\text{thesis}\) by simp
qed
next
  assume \(a2:\text{ln} \, t = \text{One}\)
  from this show \(?\text{thesis}\) by simp
qed

58
lemma L5-Controller:
assumes h1: Controller-L y lIn lOut z
and h2: lOut t = a
and h3: z t = []
shows lIn t = a
proof (cases a)
assume a = Zero
from this and h1 and h2 and h3 show ?thesis
  by (simp add: L5-Controller-Zero)
next
assume a = One
from this and h1 and h2 and h3 show ?thesis
  by (simp add: L5-Controller-One)
qed

lemma L6-Controller-Zero:
assumes h1: Controller-L y (fin-inf-append [Zero] l) l z
and h2: l (Suc t) = Zero
and h3: z (Suc t) = []
shows l t = Zero
proof (cases a)
  from h1 and h2 and h3 have (fin-inf-append [Zero] l) (Suc t) = Zero
  by (simp add: L5-Controller-Zero)
  from this show ?thesis
  by (simp add: correct-fin-inf-append2)
qed

lemma L6-Controller-One:
assumes h1: Controller-L y (fin-inf-append [Zero] l) l z
and h2: l (Suc t) = One
and h3: z (Suc t) = []
shows l t = One
proof (cases a)
  from h1 and h2 and h3 have (fin-inf-append [Zero] l) (Suc t) = One
  by (simp add: L5-Controller-One)
  from this show ?thesis
  by (simp add: correct-fin-inf-append2)
qed

lemma L6-Controller:
assumes h1: Controller-L y (fin-inf-append [Zero] l) l z
and h2: l (Suc t) = a
and h3: z (Suc t) = []
shows l t = a
proof (cases a)
  assume a = Zero
  from this and h1 and h2 and h3 show thesis
    by (simp add: L6-Controller-Zero)
next
  assume a = One
  from this and h1 and h2 and h3 show thesis
    by (simp add: L6-Controller-One)
qed

lemma L7-Controller-Zero:
  assumes h1:Controller-L y (fin-inf-append [Zero] l) l z
     and h2: l t = Zero
  shows last (fin-make-untimed (inf-truncate z t)) = Zero
using assms
proof (induct t)
  case 0
    from h1 have sg1: z 0 = [Zero] by (simp add: Controller-L-def)
  from this show ?case by (simp add: fin-make-untimed-def)
next
  fix t
  case (Suc t)
  from this show ?case
  proof (cases l t)
    assume a1: l t = Zero
    from Suc have sg1: z (Suc t) = [] \land l t = Zero \lor (z (Suc t) = [Zero] \land l t = One)
      by (simp add: L4-Controller-Zero)
    from this and a1 have sg2: z (Suc t) = []
      by simp
    from Suc and sg2 and a1 show ?case
      by (simp add: fin-make-untimed-append-empty)
  next
    assume a1: l t = One
    from Suc have sg1: z (Suc t) = [] \land l t = Zero \lor (z (Suc t) = [Zero] \land l t = One)
      by (simp add: L4-Controller-Zero)
    from this and a1 have sg2: z (Suc t) = [Zero]
      by simp
    from a1 and Suc and sg2 show ?case
      by (simp add: fin-make-untimed-def)
  qed
qed

lemma L7-Controller-One-l0:
  assumes h1:Controller-L y (fin-inf-append [Zero] l) l z
     and h2: y 0 = [500::nat]
  shows l 0 = Zero
proof (rule ccontr)
  assume a1: ¬l 0 = Zero
  from assms have sg1: z 0 = [Zero] by (simp add: Controller-L-def)
  have sg2: fin-inf-append [Zero] l (0::nat) = Zero by (simp add: fin-inf-append-def)
  from assms and a1 and sg1 and sg2 show False
  by (simp add: Controller-L-def)
qed

lemma L7-Controller-One:
  assumes h1: Controller-L y (fin-inf-append [Zero] l) l z
         and h2: l t = One
         and h3: y 0 = [500::nat]
  shows last (fin-make-untimed (inf-truncate z t)) = One
using assms
proof (induct t)
  case 0
  from h1 and h3 have sg0: l 0 = Zero by (simp add: L7-Controller-One-l0)
  from this and 0 show ?case by simp
next
  fix t
  case (Suc t)
  from this show ?case
proof (cases l t)
  assume a1: l t = Zero
  from Suc have
    sg1: z (Suc t) = [] ∧ l t = One) ∨ (z (Suc t) = [One] ∧ l t = Zero)
    by (simp add: L4-Controller-One)
  from this and a1 have sg2: z (Suc t) = [One]
    by simp
  from Suc and sg2 and a1 show ?case
    by (simp add: fin-make-untimed-def)
next
  assume a1: l t = One
  from Suc have
    sg1: z (Suc t) = [] ∧ l t = One) ∨ (z (Suc t) = [One] ∧ l t = Zero)
    by (simp add: L4-Controller-One)
  from this and a1 have sg2: z (Suc t) = []
    by simp
  from a1 and Suc and sg2 show ?case
    by (simp add: fin-make-untimed-def)
qed
qed

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

61
proof (cases l t)
  assume l t = Zero
  from this and h1 show thesis
    by (simp add: L7-Controller-Zero)
next
  assume l t = One
  from this and h1 and h2 show thesis
    by (simp add: L7-Controller-One)
qed

lemma L8-Controller:
  assumes h1: Controller-L y (fin-inf-append [Zero] l) l z
  shows z t = [] ∨ z t = [Zero] ∨ z t = [One]
proof (cases fin-inf-append [Zero] l t = Zero)
  assume a1: fin-inf-append [Zero] l t = Zero
  from a1 and h1 have sg1:
    if 300 < hd (y t)
    then z t = [] ∧ l t = Zero
    else z t = [One] ∧ l t = One
    by (simp add: Controller-L-def)
  show thesis
    proof (cases 300 < hd (y t))
      assume a11: 300 < hd (y t)
      from a11 and sg1 show thesis by simp
    next
      assume a12: ¬ 300 < hd (y t)
      from a12 and sg1 show thesis by simp
    qed
  qed
next
  assume a2: fin-inf-append [Zero] l t ≠ Zero
  from a2 and h1 have sg2:
    if hd (y t) < 700
    then z t = [] ∧ l t = One
    else z t = [Zero] ∧ l t = Zero
    by (simp add: Controller-L-def)
  show thesis
    proof (cases hd (y t) < 700)
      assume a21: hd (y t) < 700
      from a21 and sg2 show thesis by simp
    next
      assume a22: ¬ hd (y t) < 700
      from a22 and sg2 show thesis by simp
    qed
  qed
lemma L9-Controller:
assumes h1: Controller-L s (fin-inf-append [Zero] l) l z
  and h2: fin-make-untimed (inf-truncate z i)!
    (length (fin-make-untimed (inf-truncate z i)) - Suc 0) = Zero
  and h3: last (fin-make-untimed (inf-truncate z i)) = l i
  and h4: 200 ≤ hd (s i)
  and h5: hd (s (Suc i)) = hd (s i) - r
  and h6: fin-make-untimed (inf-truncate z i) ≠ []
  and h7: 0 < r
  and h8: r ≤ 10
shows 200 ≤ hd (s (Suc i))
proof
  from h6 and h2 and h3 have sg0: l i = Zero
  by (simp add: last-nth-length)
  show ?thesis
proof (cases fin-inf-append [Zero] l i = Zero)
  assume a1: fin-inf-append [Zero] l i = Zero
  from a1 and h1 have sg1:
    if 300 < hd (s i)
      then z i = [] ∧ l i = Zero
    else z i = [One] ∧ l i = One
    by (simp add: Controller-L-def)
  show ?thesis
proof (cases 300 < hd (s i))
  assume a11: 300 < hd (s i)
  from a11 and h5 and h8 show ?thesis by simp
next
  assume a12: ¬ 300 < hd (s i)
  from a12 and sg1 and sg0 show ?thesis by simp
qed
next
  assume a2: fin-inf-append [Zero] l i ≠ Zero
  from a2 and h1 have sg2:
    if hd (s i) < 700
      then z i = [] ∧ l i = One
    else z i = [Zero] ∧ l i = Zero
    by (simp add: Controller-L-def)
  show ?thesis
proof (cases hd (s i) < 700)
  assume a21: hd (s i) < 700
  from this and sg2 and sg0 show ?thesis by simp
next
  assume a22: ¬ hd (s i) < 700
  from this and h5 and h8 show ?thesis by simp
qed
qed
qed
lemma L10-Controller:

assumes h1: Controller-L s (fin-inf-append [Zero] l) l z
  and h2: fin-make-untimed (inf-truncate z i)!
    (length (fin-make-untimed (inf-truncate z i)) − Suc 0) ≠ Zero
  and h3: last (fin-make-untimed (inf-truncate z i)) = l i
  and h4: hd (s i) ≤ 800
  and h5: hd (s (Suc i)) = hd (s i) + r
  and h6: fin-make-untimed (inf-truncate z i) ≠ []
  and h7: 0 < r
  and h8: r ≤ 10

shows hd (s (Suc i)) ≤ 800

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

qed

64
9.3 Properties of the Converter Component

**Lemma L1-Converter:**

**Assumes**

- $h1$: $\text{Converter } z \ x$
- $h2$: $\text{fin-make-untimed} (\text{inf-truncate } z \ t) \neq []$

**Shows**

\[
hd (x \ t) = (\text{fin-make-untimed} (\text{inf-truncate } z \ t)) !
((\text{length} (\text{fin-make-untimed} (\text{inf-truncate } z \ t))) - (1 :: \text{nat}))
\]

**Using**

- $\text{assms}$

**By**

- $(\text{simp add: } \text{Converter-def})$

**Lemma L1a-Converter:**

**Assumes**

- $h1$: $\text{Converter } z \ x$
- $h2$: $\text{fin-make-untimed} (\text{inf-truncate } z \ t) \neq []$
- $h3$: $hd (x \ t) = \text{Zero}$

**Shows**

\[
(\text{fin-make-untimed} (\text{inf-truncate } z \ t)) !
((\text{length} (\text{fin-make-untimed} (\text{inf-truncate } z \ t))) - (1 :: \text{nat})) = \text{Zero}
\]

**Using**

- $\text{assms}$

**By**

- $(\text{simp add: L1-Converter})$

9.4 Properties of the System

**Lemma L1-ControlSystem:**

**Assumes**

- $h1$: $\text{ControlSystemArch } s$

**Shows**

- $ts s$

**Proof**

- From $h1$ obtain $x \ z \ y$
  - Where $a1$: $\text{Converter } z \ x$ and $a2$: $\text{SteamBoiler } x \ s \ y$
  - By $(\text{simp only: ControlSystemArch-def, auto})$

  - From this have $sg1: ts x$
    - By $(\text{simp add: Converter-def})$

  - From $a2$ and $sg1$ show $\text{?thesis}$ **by** (rule $L1$-$\text{Boiler}$)

**Qed**

**Lemma L2-ControlSystem:**

**Assumes**

- $h1$: $\text{ControlSystemArch } s$

**Shows**

- $(200 :: \text{nat}) \leq \text{hd} (s \ i)$

**Proof**

- From $h1$ obtain $x \ z \ y$
  - Where $a1$: $\text{Converter } z \ x$ and $a2$: $\text{SteamBoiler } x \ s \ y$ and $a3$: $\text{Controller } y \ z$
    - By $(\text{simp only: ControlSystemArch-def, auto})$

  - From this have $sg1: ts x$ **by** $(\text{simp add: Converter-def})$

  - From $sg1$ and $a2$ have $sg2: ts y$ **by** $(\text{simp add: L2-Boiler})$

  - From $sg1$ and $a2$ have $sg3: y = s$ **by** $(\text{simp add: SteamBoiler-def})$

  - From $a1$ and $a2$ and $a3$ and $sg1$ and $sg2$ and $sg3$ show $200 \leq \text{hd} (s \ i)$

**Proof (induction $i$)**

- **Case** 0

  - From this show $\text{?case}$ **by** $(\text{simp add: L3-Boiler})$

**Next**
fix i

case (Suc i)

from this obtain l
  where a4: Controller-L s (fin-inf-append [Zero] l) l z
  by (simp add: Controller-def, atomize, auto)

from Suc and a4 have sg4: fin-make-untimed (inf-truncate z i) ≠ []
  by (simp add: L1-Controller)

from a2 and sg1 have y0asm: y 0 = [500::nat] by (simp add: SteamBoiler-def)

from Suc and a4 and sg4 and y0asm have sg5: last (fin-make-untimed (inf-truncate z i)) = l i
  by (simp add: L7-Controller)

from a2 and sg1 obtain r where
  aa0: 0 < r and
  aa1: r ≤ 10 and
  aa2: hd (s (Suc i)) = (if hd (x i) = Zero then hd (s i) − r else hd (s i) + r)
  by (simp add: SteamBoiler-def, auto)

from Suc and a4 and sg4 and sg5 show ?case

proof (cases hd (x i) = Zero)
  assume aaZero: hd (x i) = Zero

  from a1 and sg4 and this have
    sg7: (fin-make-untimed (inf-truncate z i)) !
    ((length (fin-make-untimed (inf-truncate z i))) − Suc 0) = Zero
    by (simp add: L1-Converter)

  from aa2 and aaZero have sg8: hd (s (Suc i)) = hd (s i) − r by simp

  from Suc have sgSuc: 200 ≤ hd (s i) by simp

  from a4 and sg7 and sg5 and sgSuc and sg8 and sg4 and aa0 and aa1
  show ?thesis
    by (rule L9-Controller)

next
  assume aaOne: hd (x i) ≠ Zero

  from a1 and sg4 and this have
    sg7: (fin-make-untimed (inf-truncate z i)) !
    ((length (fin-make-untimed (inf-truncate z i))) − Suc 0) ≠ Zero
    by (simp add: L1-Converter)

  from aa2 and aaOne have sg9: hd (s (Suc i)) = hd (s i) + r by simp

  from Suc and this show ?thesis by simp

qed

qed

lemma L3-ControlSystem:
  assumes h1: ControlSystemArch s
  shows hd (s i) ≤ (800::nat)
proof –
  from h1 obtain x z y
    where a1: Converter z x and a2: SteamBoiler x s y and a3: Controller y z
    by (simp only: ControlSystemArch-def, auto)

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

lemma L0-ControlSystem:
assumes h1::ControlSystemArch s
shows ControlSystem s
  apply (simp add: ControlSystem-def)
  apply auto
proof
  from h1 show sg1::ts s by (rule L1-ControlSystem)
next
  fix j
  from h1 show sg2::(200::nat) ≤ hd (s j) by (rule L2-ControlSystem)
next
  fix j
  from h1 show sg3::hd (s j) ≤ (800:: nat) by (rule L3-ControlSystem)
qed

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 atn 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] ∧
(∀ j. j < n ∧ j ≠ k →
((nStore j) t) = ((nReturn k) t))

69
— The predicate Broadcast describes properties of FlexRay broadcast.

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

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

definition
Receive :: 'a Frame istream ⇒ 'a Frame istream ⇒ nat istream ⇒ bool
where
Receive recv store activation
≡ ∀ (t::nat).
( if (activation t) = []
then (store t) = (recv t)
else (store t) = [] )

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

definition
Send :: 'a Frame istream ⇒ 'a Frame istream ⇒ nat istream ⇒ nat istream ⇒ bool
where
Send return send get activation
≡ ∀ (t::nat).
( if (activation t) = []
then (get t) = [] ∧ (send t) = []
else (get t) = (activation t) ∧ (send t) = (return t) )

11.2 Specifications of the FlexRay components

definition
FlexRay :: nat ⇒ 'a nFrame ⇒ nConfig ⇒ 'a nFrame ⇒ nNat ⇒ bool
where
FlexRay n nReturn nC nStore nGet
≡ (CorrectSheaf n) ∧
((∀ (i::nat). i < n → (msg 1 (nReturn i))) ∧
(DisjointSchedules n nC) ∧ (IdenticalCycleLength n nC)
→
(\text{FrameTransmission\ n\ nStore\ nReturn\ nGet\ nC}) \land
(\forall\ (i:\!\!\!::\!\!\!nat).\ i < n \rightarrow (msg\ 1\ (nGet\ i)) \land (msg\ 1\ (nStore\ i))) \}

\textbf{definition}
\texttt{Cable \!::\!\!\!nat \!\rightarrow\! 'a\!\!\!nFrame \!\rightarrow\! 'a\!Frame\ istream \!\rightarrow\! bool}
\textbf{where}
\texttt{Cable\ n\ nSend\ recv}
\equiv
(\text{CorrectSheaf\ n})
\land
\((\text{inf-disj\ n\ nSend}) \rightarrow (\text{Broadcast\ n\ nSend\ recv}))

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

\textbf{definition}
\texttt{BusInterface \!::\! nat\ istream \!\rightarrow\! 'a\! Frame\ istream \!\rightarrow\! 'a\! Frame\ istream \!\rightarrow\! 'a\! Frame\ istream \!\rightarrow\! 'a\! Frame\ istream \!\rightarrow\! nat\ istream \!\rightarrow\! bool}
\textbf{where}
\texttt{BusInterface\ activation\ return\ recv\ store\ send\ get}
\equiv
(Receive\ recv\ store\ activation) \land
(Send\ return\ send\ get\ activation)

\textbf{definition}
\texttt{FlexRayController \!::\! 'a\! Frame\ istream \!\rightarrow\! 'a\! Frame\ istream \!\rightarrow\! Config \!\rightarrow\! 'a\! Frame\ istream \!\rightarrow\! 'a\! Frame\ istream \!\rightarrow\! nat\ istream \!\rightarrow\! bool}
\textbf{where}
\texttt{FlexRayController\ return\ recv\ c\ store\ send\ get}
\equiv
(\exists\ activation.\ (Scheduler\ c\ activation) \land
(BusInterface\ activation\ return\ recv\ store\ send\ get)))
definition

\[ \text{FlexRayArchitecture} :\]
\[ \text{nat} \Rightarrow 'a \text{ nFrame} \Rightarrow \text{nConfig} \Rightarrow 'a \text{ nFrame} \Rightarrow \text{nNat} \Rightarrow \text{bool} \]

where

\[ \text{FlexRayArchitecture} \ n \ \text{nReturn} \ nC \ \text{nStore} \ \text{nGet} \]
\[ \equiv \]
\[ (\text{CorrectSheaf} \ n) \land \]
\[ (\exists \ n\text{Send} \ \text{recv}). \]
\[ (\text{Cable} \ n \ n\text{Send} \ \text{recv}) \land \]
\[ (\forall \ (i::\text{nat}). \ i < n \rightarrow \]
\[ \text{FlexRayController} \ (\text{nReturn} \ i) \ \text{recv} \ (\text{nC} \ i) \]
\[ (\text{nStore} \ i) \ (\text{nSend} \ i) \ (\text{nGet} \ i)) ) \]

definition

\[ \text{FlexRayArch} :\]
\[ \text{nat} \Rightarrow 'a \text{ nFrame} \Rightarrow \text{nConfig} \Rightarrow 'a \text{ nFrame} \Rightarrow \text{nNat} \Rightarrow \text{bool} \]

where

\[ \text{FlexRayArch} \ n \ \text{nReturn} \ nC \ \text{nStore} \ \text{nGet} \]
\[ \equiv \]
\[ (\text{CorrectSheaf} \ n) \land \]
\[ (\forall \ (i::\text{nat}). \ i < n \rightarrow \text{msg} \ 1 \ (\text{nReturn} \ i)) \land \]
\[ (\text{DisjointSchedules} \ n \ nC) \land (\text{IdenticCycleLength} \ n \ nC) \]
\[ \rightarrow \]
\[ (\text{FlexRayArchitecture} \ n \ \text{nReturn} \ nC \ \text{nStore} \ \text{nGet}) ) \]

end

12 FlexRay: Verification

theory FR-proof
imports FR
begin

12.1 Properties of the function Send

lemma Send-L1:
\[ \text{assumes} \ h1: \text{Send return send get activation} \]
\[ \text{and} \ h2: \text{send} \ t \neq [] \]
\[ \text{shows} \ (\text{activation} \ t) \neq [] \]
\[ \text{using assms by} \ (\text{simp add: Send-def, auto}) \]

lemma Send-L2:
\[ \text{assumes} \ h1: \text{Send return send get activation} \]
\[ \text{and} \ h2: (\text{activation} \ t) \neq [] \]
\[ \text{and} \ h3: \text{return} \ t \neq [] \]
\[ \text{shows} \ (\text{send} \ t) \neq [] \]
\[ \text{using assms by} \ (\text{simp add: Send-def}) \]
12.2 Properties of the component Scheduler

**lemma** Scheduler-L1:

**assumes** h1: Scheduler C activation

and h2: (activation t) ≠ []

**shows** (t mod (cycleLength C)) mem (schedule C)

**using** assms

**proof** –

{ assume a1: ¬ t mod cycleLength C mem schedule C

from h1 have sg1:

if t mod cycleLength C mem schedule C

then activation t = [t mod cycleLength C]

else activation t = []

by (simp add: Scheduler-def Let-def)

from a1 and sg1 have sg2: activation t = [] by simp

from sg2 and h2 have sg3: False by simp

} from this have sg4: (t mod (cycleLength C)) mem (schedule C) by blast

from this show ?thesis by simp

qed

**lemma** Scheduler-L2:

**assumes** h1: Scheduler C activation

and h2: ¬ (t mod cycleLength C) mem (schedule C)

**shows** activation t = []

**using** assms by (simp add: Scheduler-def Let-def)

**lemma** Scheduler-L3:

**assumes** h1: Scheduler C activation

and h2: (t mod cycleLength C) mem (schedule C)

**shows** activation t ≠ []

**using** assms by (simp add: Scheduler-def Let-def)

**lemma** Scheduler-L4:

**assumes** h1: Scheduler C activation

and h2: (t mod cycleLength C) mem (schedule C)

**shows** activation t = [t mod cycleLength C]

**using** assms by (simp add: Scheduler-def Let-def)

**lemma** correct-DisjointSchedules1:

**assumes** h1: DisjointSchedules n nC

and h2: IdentcCycleLength n nC

and h3: (t mod cycleLength (nC i)) mem schedule (nC i)

and h4: i < n

and h5: j < n

and h6: i ≠ j

**shows** ¬ (t mod cycleLength (nC j) mem schedule (nC j))
proof –
  from h1 and h4 and h5 and h6 have sg1: disjoint (schedule (nC i)) (schedule (nC j))
    by (simp add: DisjointSchedules-def)
  from h2 and h4 and h5 have sg2: cycleLength (nC i) = cycleLength (nC j)
    by (simp only: IdenticCycleLength-def, blast)
  from sg1 and h3 have sg3: ¬(t mod (cycleLength (nC i))) mem (schedule (nC j))
    by (simp add: mem-notdisjoint2)
  from sg2 and sg3 show ?thesis by simp
qed

12.3 Disjoint Frames

lemma disjointFrame-L1:
assumes h1: DisjointSchedules n nC and h2: IdenticCycleLength n nC and h3: ∀i<n. FlexRayController (nReturn i) rcv (nC i) (nStore i) (nSend i) (nGet i)
and h4: nSend i t ≠ [] and h5: i < n and h6: j < n and h7: i ≠ j
shows nSend j t = []
proof –
  from h3 and h5 have sg1:
    FlexRayController (nReturn i) rcv (nC i) (nStore i) (nSend i) (nGet i)
    by auto
  from h3 and h6 have sg2:
    FlexRayController (nReturn j) rcv (nC j) (nStore j) (nSend j) (nGet j)
    by auto
  from sg1 obtain activation1 where
    a1: Scheduler (nC i) activation1 and
    a2: BusInterface activation1 (nReturn i) rcv (nStore i) (nSend i) (nGet i)
    by (simp add: FlexRayController-def, auto)
  from sg2 obtain activation2 where
    a3: Scheduler (nC j) activation2 and
    a4: BusInterface activation2 (nReturn j) rcv (nStore j) (nSend j) (nGet j)
    by (simp add: FlexRayController-def, auto)
  from h1 and h5 and h6 and h7 have sg3: disjoint (schedule (nC i)) (schedule (nC j))
    by (simp add: DisjointSchedules-def)
  from a2 have sg4a: Send (nReturn i) (nSend i) (nGet i) activation1
    by (simp add: BusInterface-def)
  from sg4a and h4 have sg5: activation1 t ≠ [] by (simp add: Send-L1)
  from a1 and sg5 have sg6: (t mod (cycleLength (nC i))) mem (schedule (nC i))
    by (simp add: Scheduler-L1)
  from h2 and h5 and h6 have sg7: cycleLength (nC i) = cycleLength (nC j)
by (simp only: IdenticCycleLength-def, blast)
from sg3 and sg6 have sg8:¬ (t mod (cycleLength (nC i))) mem (schedule (nC j))
  by (simp add: mem-notdisjoint2)
from sg8 and sg7 have sg9:¬ (t mod (cycleLength (nC j))) mem (schedule (nC j))
  by simp
from sg9 and a3 have sg10:activation2 t = [] by (simp add: Scheduler-L2)
from a4 have sg11:Send (nReturn j) (nSend j) (nGet j) activation2
  by (simp add: BusInterface-def)
from sg11 and sg10 show thesis by (simp add: Send-def)
qed

lemma disjointFrame-L2:
assumes h1:DisjointSchedules n nC
  and h2:IdenticCycleLength n nC
  and h3:∀ i < n. FlexRayController (nReturn i) rev (nC i) (nStore i) (nSend i) (nGet i)
shows inf-disj n nSend
using assms
  apply (simp add: inf-disj-def, clarify)
  by (rule disjointFrame-L1, auto)

lemma disjointFrame-L3:
assumes h1:DisjointSchedules n nC
  and h2:IdenticCycleLength n nC
  and h3:∀ i < n. FlexRayController (nReturn i) rev (nC i) (nStore i) (nSend i) (nGet i)
  and h4:t mod cycleLength (nC i) mem schedule (nC i)
  and h5:i < n
  and h6:j < n
  and h7:i ≠ j
shows nSend j t = []
proof –
  from h2 and h5 and h6 have sg1:cycleLength (nC i) = cycleLength (nC j)
    by (simp only: IdenticCycleLength-def, blast)
  from h1 and h5 and h6 and h7 have sg2:disjoint (schedule (nC i)) (schedule (nC j))
    by (simp add: DisjointSchedules-def)
  from sg2 and h4 have sg3:¬ (t mod (cycleLength (nC i))) mem (schedule (nC j))
    by (simp add: mem-notdisjoint2)
  from sg1 and sg3 have sg4:¬ (t mod (cycleLength (nC j))) mem (schedule (nC j))
    by simp
  from h3 and h6 have sg5:
    FlexRayController (nReturn j) rev (nC j) (nStore j) (nSend j) (nGet j)
by auto

from sg5 obtain activation2 where
  a1: Scheduler (nC j) activation2 and
  a2: BusInterface activation2 (nReturn j) recv (nStore j) (nSend j) (nGet j)
  by (simp add: FlexRayController-def, auto)

from sg4 and a1 have sg6: activation2 t = [] by (simp add: Scheduler-L2)
from a2 have sg7: Send (nReturn j) (nSend j) (nGet j) activation2
  by (simp add: BusInterface-def)
from sg7 and sg6 show ?thesis by (simp add: Send-def)
qed

12.4 Properties of the sheaf of channels nSend

lemma fr-Send1:
  assumes h1: FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
  and h2: ¬ (t mod cycleLength (nC i) mem schedule (nC i))
  shows (nSend i) t = []
proof –
  from h1 obtain activation where
    a1: Scheduler (nC i) activation and
    a2: BusInterface activation (nReturn i) recv (nStore i) (nSend i) (nGet i)
    by (simp add: FlexRayController-def, auto)
  from a1 and h2 have sg1: activation t = [] by (simp add: Scheduler-L2)
  from a2 have sg2: Send (nReturn i) (nSend i) (nGet i) activation
    by (simp add: BusInterface-def)
  from sg2 and sg1 show ?thesis by (simp add: Send-def)
qed

lemma fr-Send2:
  assumes h1: ∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
  and h2: DisjointSchedules n nC
  and h3: IdenticCycleLength n nC
  and h4: t mod cycleLength (nC k) mem schedule (nC k)
  and h5: k < n
  shows nSend k t = nReturn k t
using assms
proof –
  from h1 and h5 have sg1:
    FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)
    by auto
  from sg1 obtain activation where
    a1: Scheduler (nC k) activation and
    a2: BusInterface activation (nReturn k) recv (nStore k) (nSend k) (nGet k)
    by (simp add: FlexRayController-def, auto)
  from a1 and h4 have sg3: activation t ≠ [] by (simp add: Scheduler-L3)
  from a2 have sg4: Send (nReturn k) (nSend k) (nGet k) activation
lemma fr-Send3:
assumes h1:∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
and h2:DisjointSchedules n nC
and h3:IdenticCycleLength n nC
and h4:t mod cycleLength (nC k) mem schedule (nC k)
and h5:k < n
and h6:nReturn k t ´ []
shows nSend k t ´ []
using assms by (simp add: fr-Send2)

lemma fr-Send4:
assumes h1:∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
and h2:DisjointSchedules n nC
and h3:IdenticCycleLength n nC
and h4:t mod cycleLength (nC k) mem schedule (nC k)
and h5:k < n
and h6:nReturn k t ´ []
shows ∃ k. k < n −→ nSend k t ´ []
proof
from assms show k < n −→ nSend k t ´ [] by (simp add: fr-Send3)
qed

lemma fr-Send5:
assumes h1:∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
and h2:DisjointSchedules n nC
and h3:IdenticCycleLength n nC
and h4:t mod cycleLength (nC k) mem schedule (nC k)
and h5:k < n
and h6:nReturn k t ´ []
and h7:∀ k<n. nSend k t = []
shows False
proof
from h1 and h2 and h3 and h4 and h5 and h6 have sg1:nSend k t ´ []
by (simp add: fr-Send2)
from h7 and h5 have sg2:nSend k t = [] by blast
from sg1 and sg2 show thesis by simp
qed
lemma fr-Send6:
assumes h1:∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
  and h2:DisjointSchedules n nC
  and h3:IdentieCycleLength n nC
  and h4:t mod cycleLength (nC k) mem schedule (nC k)
  and h5:k<n
  and h6:nReturn k t ≠ []
shows ∃k<n. nSend k t ≠ []
proof (rule ccontr)
  assume ¬ (∃k<n. nSend k t ≠ [])
  from this and assms show False
  apply auto
  by (rule fr-Send5, auto)
qed

lemma fr-Send7:
assumes h1:∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
  and h2:DisjointSchedules n nC
  and h3:IdentieCycleLength n nC
  and h4:t mod cycleLength (nC k) mem schedule (nC k)
  and h5:k<n
  and h6:j<n
  and h6:nReturn k t = []
shows ¬ (∃k<n. nSend k t ≠ [])
proof (cases j = k)
  assume a1: j = k
  from assms have sg1: nSend k t = nReturn k t by (simp add: fr-Send2)
  from sg1 and a1 and h6 show ?thesis by simp
next
  assume a2:j ≠ k
  from assms and a2 show ?thesis by (simp add: disjointFrame-L3)
qed

lemma fr-Send8:
assumes h1:∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
  and h2:DisjointSchedules n nC
  and h3:IdentieCycleLength n nC
  and h4:t mod cycleLength (nC k) mem schedule (nC k)
  and h5:k<n
  and h6:nReturn k t = []
shows ¬ (∃k<n. nSend k t ≠ [])
using assms by (auto, simp add: fr-Send7)
lemma fr-nC-Send:
assumes h1:\forall i < n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
  and h2:k < n
  and h3:DisjointSchedules n nC
  and h4:IdentCycLeN ecycleLength n nC
  and h5:t mod cycleLength (nC k) mem schedule (nC k)
shows \forall j. j < n \land j \neq k \rightarrow (nSend j) t = []
using assms by (clarify, simp add: disjointFrame-L3)

lemma length-nSend:
assumes h1:BusInterface activation (nReturn i) recv (nStore i) (nSend i) (nGet i)
  and h2:\forall t. length (nReturn i t) \leq Suc 0
shows length (nSend i t) \leq Suc 0
proof –
from h1 have sg1:Send (nReturn i) (nSend i) (nGet i) activation
  by (simp add: BusInterface-def)
from sg1 have sg2:
  if activation t = [] then nGet i t = [] \land nSend i t = []
  else nGet i t = activation t \land nSend i t = nReturn i t
  by (simp add: Send-def)
show ?thesis
proof (cases activation t = [])
  assume a1:activation t = []
  from sg2 and a1 show ?thesis by simp
next
  assume a2:activation t \neq []
  from h2 have sg3:length (nReturn i t) \leq Suc 0 by auto
  from sg2 and a2 and sg3 show ?thesis by simp
qed
qed

lemma msg-nSend:
assumes h1:BusInterface activation (nReturn i) recv (nStore i) (nSend i) (nGet i)
  and h2:msg (Suc 0) (nReturn i)
shows msg (Suc 0) (nSend i)
using assms by (simp add: msg-def, clarify, simp add: length-nSend)

lemma Broadcast-nSend-empty1:
assumes h1:Broadcast n nSend recv
  and h2:\forall k < n. nSend k t = []
shows recv t = []
proof –
from h1 have sg1:
if \( \exists k < n. \) nSend \( k \neq [] \)
then recv \( t = nSend \) \((\text{SOME } k. \ k < n \land nSend \ k t \neq [])\) t
else recv \( t = [] \)
by (simp add: \text{Broadcast-def})
from sg1 and h2 show \?thesis by simp
qed

12.5 Properties of the sheaf of channels \( nGet \)

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

lemma fr-nGet1:
assumes h1: \( \forall i < n. \) FlexRayController \((\text{nReturn } i) \ (\text{recv } (nC \ i) \ (nStore \ i) \ (nSend \ i) \ (nGet \ i))\)
and h2: \( t \mod \text{cycleLength} \ (nC \ k) \ mem \text{schedule} \ (nC \ k) \)
and h3: \( k < n \)
shows \( nGet \ k t = [t \mod \text{cycleLength} \ (nC \ k)] \)
proof –
from h1 and h3 have sg1:
FlexRayController \((\text{nReturn } k) \ (\text{recv } (nC \ k) \ (nStore \ k) \ (nSend \ k) \ (nGet \ k))\)
by auto
from sg1 and h2 show \?thesis by (rule fr-nGet1a)
qed

lemma fr-nGet2a:
assumes h1: FlexRayController \((\text{nReturn } k) \ (\text{recv } (nC \ k) \ (nStore \ k) \ (nSend \ k) \ (nGet \ k))\)

80
and h2:¬ (t mod cycleLength (nC k) mem schedule (nC k))
shows nGet k t = []

proof –
  from h1 obtain activation1 where
  a1:Scheduler (nC k) activation1 and
  a2:BusInterface activation1 (nReturn k) recv (nStore k) (nSend k) (nGet k)
  by (simp add: FlexRayController-def, auto)
from a2 have sg2:Send (nReturn k) (nSend k) (nGet k) activation1
  by (simp add: BusInterface-def)
from sg2 have sg3:
  if activation1 t = [] then nGet k t = [] ∧ nSend k t = []
  else nGet k t = activation1 t ∧ nSend k t = nReturn k t
  by (simp add: Send-def)
from sg3 have sg4:activation1 t = []
  by (simp add: Scheduler-L2)
from sg3 and sg4 show ?thesis by simp
qed

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

proof –
  from h1 and h3 have sg1:
    FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)
    by auto
from sg1 and h2 show ?thesis by (rule fr-nGet2a)
qed

lemma length-nGet1:
  assumes h1:FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)
  (nC k)
shows length (nGet k t) ≤ Suc 0
proof (cases t mod cycleLength (nC k) mem schedule (nC k))
  assume a1:t mod cycleLength (nC k) mem schedule (nC k)
  from h1 and a1 have sg1:nGet k t = [t mod cycleLength (nC k)]
    by (rule fr-nGet1a)
from sg1 show ?thesis by auto
next
  assume a2:¬ (t mod cycleLength (nC k) mem schedule (nC k))
  from h1 and a2 have sg2:nGet k t = [] by (rule fr-nGet2a)
from sg2 show ?thesis by auto
qed
lemma msg-nGet1:
  assumes h1: FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)
  shows msg (Suc 0) (nGet k)
  using assms by (simp add: msg-def, auto, rule length-nGet1)

lemma msg-nGet2:
  assumes h1: \( \forall i < n. \text{FlexRayController} \ (nReturn i) \ recv \ (nC i) \ (nStore i) \ (nSend i) \ (nGet i) \)
    and h2: \( k < n \)
  shows msg (Suc 0) (nGet k)
proof
  from h1 and h2 have sg1:
    FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)
    by auto
  from sg1 show \(?thesis\) by (rule msg-nGet1)
qed

12.6 Properties of the sheaf of channels nStore

lemma fr-nStore-nReturn1:
  assumes h0: Broadcast n nSend recv
    and h1: \( \forall i < n \). FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
    and h3: \( \text{DisjointSchedules} \ n \ nC \)
    and h4: \( \text{IdenticCycleLength} \ n \ nC \)
    and h5: \( t \mod \text{cycleLength} \ (nC k) \ \text{mem schedule} \ (nC k) \)
    and h6: \( k < n \)
    and h7: \( j < n \)
    and h8: \( j \neq k \)
  shows nStore j t = nReturn k t
proof
  from h2 and h6 have sg1:
    FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)
    by auto
  from h2 and h7 have sg2:
    FlexRayController (nReturn j) recv (nC j) (nStore j) (nSend j) (nGet j)
    by auto
  from sg1 obtain activation1 where
    a1: Scheduler (nC k) activation1 and
    a2: BusInterface activation1 (nReturn k) recv (nStore k) (nSend k) (nGet k)
    by (simp add: FlexRayController-def, auto)
  from sg2 obtain activation2 where
    a3: Scheduler (nC j) activation2 and
    a4: BusInterface activation2 (nReturn j) recv (nStore j) (nSend j) (nGet j)
    by (simp add: FlexRayController-def, auto)
  from a4 have sg3: Receive recv (nStore j) activation2
by (simp add: BusInterface-def)
from this have sg4:
  if activation2 t = [] then nStore j t = recv t else nStore j t = []
  by (simp add: Receive-def)
from a1 and h5 have sg5: activation1 t ≠ []
  by (simp add: Scheduler-L3)
from h4 and h6 and h7 have sg6: cycleLength (nC k) = cycleLength (nC j)
  by (simp only: IdenticCycleLength-def, blast)
from h3 and h6 and h7 and h8 have sg7: disjoint (schedule (nC k)) (schedule (nC j))
  by (simp add: DisjointSchedules-def)
from sg7 and h5 have sg8:
  by (simp add: mem-notdisjoint2)
from sg6 and sg8 have sg9:
  by simp
from this and a3 have sg10: activation2 t = []
  by (simp add: Scheduler-L2)
from sg10 and sg4 have sg11:
  by simp
from h0 have sg15:
  if ∃ k < n. nSend k t ≠ []
  then recv t = nSend (SOME k. k < n ∧ nSend k t ≠ []) t
  else recv t = []
  by (simp add: Broadcast-def)
show ?thesis
proof (cases nReturn k t = [])
  assume a5: nReturn k t = []
  from h2 and h3 and h4 and h5 and h6 and a5 have sg16:
    by (simp add: fr-Send8)
  from sg16 and sg15 have sg17: recv t = [] by simp
  from sg11 and sg17 have sg18: nStore j t = [] by simp
  from this and a5 show ?thesis by simp
next
  assume a6: nReturn k t ≠ []
  from h2 and h3 and h4 and h5 and h6 and a6 have sg19:
    ∃ k < n. nSend k t ≠ []
    by (simp add: fr-Send6)
  from h2 and h3 and h4 and h5 and h6 and a6 have sg20: nSend k t ≠ []
    by (simp add: fr-Send3)
  from h1 and sg20 and h6 have sg21: (SOME k. k < n ∧ nSend k t ≠ []) = k
    by (simp add: inf-disj-index)
  from sg15 and sg19 have sg22:
    recv t = nSend (SOME k. k < n ∧ nSend k t ≠ []) t
    by simp
  from sg22 and sg21 have sg23: recv t = nSend k t by simp
  from h2 and h3 and h4 and h5 and h6 have sg24: nSend k t = nReturn k t
    by (simp add: fr-Send2)
  from sg11 and sg23 and sg24 show ?thesis by simp
lemma fr-nStore-nReturn2:
  assumes h1: Cable n nSend recv
  and h2: \( \forall i < n \). FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
  and h3: DisjointSchedules n nC
  and h4: IdenticCycleLength n nC
  and h5: \( t \mod \text{cycleLength} (nC k) \) mem schedule (nC k)
  and h6: \( k < n \)
  and h7: \( j < n \)
  and h8: \( j \neq k \)
  shows \( nStore j t = nReturn k t \)
proof -
  from h2 and h6 have sg1: inf-disj n nSend \( \longrightarrow \) Broadcast n nSend recv
    by (simp add: Cable-def)
  from h3 and h4 and h2 have sg2: inf-disj n nSend
    by (simp add: disjointFrame-L2)
  from sg1 and sg2 have sg3: Broadcast n nSend recv by simp
  from sg3 and sg2 and assms show ?thesis by (simp add: fr-nStore-nReturn1)
qed

lemma fr-nStore-empty1:
  assumes h1: Cable n nSend recv
  and h2: \( \forall i < n \). FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
  and h3: DisjointSchedules n nC
  and h4: IdenticCycleLength n nC
  and h5: \( t \mod \text{cycleLength} (nC k) \) mem schedule (nC k)
  and h6: \( k < n \)
  shows \( nStore k t = [] \)
proof -
  from h2 and h6 have sg1: FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)
    by auto
  from sg1 obtain activation1 where
    a1: Scheduler (nC k) activation1 and
    a2: BusInterface activation1 (nReturn k) recv (nStore k) (nSend k) (nGet k)
    by (simp add: FlexRayController-def, auto)
  from a2 have sg2: Receive recv (nStore k) activation1
    by (simp add: BusInterface-def)
  from this have sg3:
    if activation1 t = [] then nStore k t = recv t else nStore k t = []
    by (simp add: Receive-def)
  from a1 and h5 have sg4: activation1 t \( \neq [] \)
    by (simp add: Scheduler-L3)
  from sg3 and sg4 show ?thesis by simp
qed
lemma fr-nStore-nReturn3:
assumes h1: Cable n nSend recv
  and h2: ∀ i < n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
  and h3: DisjointSchedules n nC
  and h4: IdenticalCycleLength n nC
  and h5: t mod cycleLength (nC k) mem schedule (nC k)
  and h6: k < n
shows ∀ j. j < n ∧ j ≠ k → nStore j t = nReturn k t
using assms
by (clarify, simp add: fr-nStore-nReturn2)

lemma length-nStore:
assumes h1: ∀ i < n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
  and h2: DisjointSchedules n nC
  and h3: IdenticalCycleLength n nC
  and h4: inf-disj n nSend
  and h5: i < n
  and h6: ∀ i < n. msg (Suc 0) (nReturn i)
  and h7: Broadcast n nSend recv
shows length (nStore i t) ≤ Suc 0
proof
  from h7 have sg1:
    if ∃ k < n. nSend k t ≠ []
    then recv t = nSend (SOME k. k < n ∧ nSend k t ≠ []) t
    else recv t = []
    by (simp add: Broadcast-def)
  show ?thesis
proof (cases ∃ k < n. nSend k t ≠ [])
  assume a1: ∃ k < n. nSend k t ≠ []
  from a1 obtain k where a2: k < n and a3: nSend k t ≠ [] by auto
  from h1 and a2 have sg4:
    FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)
    by auto
  from sg4 obtain activation1 where
    a4: Scheduler (nC k) activation1 and
    a5: BusInterface activation1 (nReturn k) recv (nStore k) (nSend k) (nGet k)
    by (simp add: FlexRayController-def, auto)
  from a5 have sg5: Send (nReturn k) (nSend k) (nGet k) activation1
    by (simp add: BusInterface-def)
  from a5 have sg6: Receive recv (nStore k) activation1
    by (simp add: BusInterface-def)
  from sg5 and a3 have sg7: (activation1 t) ≠ [] by (simp add: Send-L1)
  from sg6 have sg8:
    if activation1 t = []
then nStore k t = recv t else nStore k t = []
by (simp add: Receive-def)
from sg5 and sg7 have sg9:nStore k t = [] by simp
from a4 and sg7 have sg10:(t mod (cycleLength (nC k))) mem (schedule (nC k))
by (simp add: Scheduler-L1)
show ?thesis
proof (cases i = k)
assume aa1: i = k
from sg9 and aa1 show ?thesis by simp
next
assume aa2:i ≠ k
from h7 and h4 and h1 and h3 and sg10 and a2 and h5 and aa2 have sg11:
  nStore i t = nReturn k t
by (simp add: fr-nStore-nReturn1)
from h6 and a2 have sg12:msg (Suc 0) (nReturn k) by auto
from a2 and h6 have sg13:length (nReturn k t) ≤ Suc 0
by (simp add: msg-def)
from sg11 and sg13 show ?thesis by simp
qed
next
assume a10:⇒ (∃k<n. nSend k t ≠ [])
from h7 and a10 have sg14:recv t = [] by (simp add: Broadcast-nSend-empty1)
from h1 and h5 have sg15:
  FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
by auto
from sg15 obtain activation2 where
  a11:Scheduler (nC i) activation2 and
  a12:BusInterface activation2 (nReturn i) recv (nStore i) (nSend i) (nGet i)
by (simp add: FlexRayController-def, auto)
from a12 have sg16:Receive recv (nStore i) activation2
by (simp add: BusInterface-def)
from sg16 have sg17:
  if activation2 t = []
  then nStore i t = recv t else nStore i t = []
by (simp add: Receive-def)
show ?thesis
proof (cases activation2 t = [])
assume aa3:activation2 t = []
from sg17 and aa3 and sg14 have sg18:nStore i t = [] by simp
from this show ?thesis by simp
next
assume aa4:activation2 t ≠ []
from sg17 and aa4 have sg18:nStore i t = [] by simp
from this show ?thesis by simp
qed
qed

86
lemma msg-nStore:
assumes h1: ∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
    and h2: DisjointSchedules n nC
    and h3: IdenticCycleLength n nC
    and h4: inf-disj n nSend
    and h5: i < n
    and h6: ∀ i<n. msg (Suc 0) (nReturn i)
    and h7: Cable n nSend recv
shows msg (Suc 0) (nStore i)
using assms
apply (simp (no-asem) add: msg-def, simp add: Cable-def, clarify)
by (simp add: length-nStore)

12.7 Refinement Properties

lemma fr-refinement-FrameTransmission:
assumes h1: Cable n nSend recv
    and h2: ∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
    and h3: DisjointSchedules n nC
    and h4: IdenticCycleLength n nC
shows FrameTransmission n nStore nReturn nGet nC
using assms
apply (simp add: FrameTransmission-def Let-def, auto)
apply (simp add: fr-nGet1)
by (simp add: fr-nStore-nReturn3)

lemma FlexRayArch-CorrectSheaf:
assumes h1: FlexRayArch n nReturn nC nStore nGet
shows CorrectSheaf n
using assms by (simp add: FlexRayArch-def)

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

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

theorem main-fr-refinement:
assumes h1: FlexRayArch n nReturn nC nStore nGet
shows FlexRay n nReturn nC nStore nGet
using assms
  by (simp add: FlexRay-def
  FlexRayArch-CorrectSheaf
  FlexRayArch-FrameTransmission
  FlexRayArch-nGet
  FlexRayArch-nStore)
end
13 Gateway: Types

theory Gateway-types imports stream begin

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

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

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

datatype reqType = init | send
datatype stopType = stop-vc
datatype vcType = vc-com
datatype aType = sc-ack

end

14 Gateway: Specification

theory Gateway imports Gateway-types begin

definition ServiceCenter ::
  ECall-Info istream ⇒ aType istream ⇒ bool
where
ServiceCenter i a ≡
  ∀ (t::nat).
  a 0 = [] ∧ 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
\[
\text{Loss lose a i2 a2 i}
\]
\[\equiv\]
\[
\forall (t::\text{nat}).
(\text{if lose t} = [\text{False}]
\quad \text{then a2 t} = a t \land i t = i2 t
\quad \text{else a2 t} = [] \land i t = [])
\]

definition
\[
\text{Delay} ::
\text{aType istream} \Rightarrow \text{ECall-Info istream} \Rightarrow \text{nat} \Rightarrow \text{aType istream} \Rightarrow \text{ECall-Info istream} \Rightarrow \text{bool}
\]
where
\[
\text{Delay a2 i1 d a1 i2}
\]
\[\equiv\]
\[
\forall (t::\text{nat})
(t < d \rightarrow a1 t = [] \land i2 t = [] \land
(t \geq d \rightarrow (a1 t = a2 (t-d)) \land (i2 t = i1 (t-d)))
\]
definition
\[
\text{tiTable-SampleT} ::
\text{reqType istream} \Rightarrow \text{aType istream} \Rightarrow
\text{stopType istream} \Rightarrow \text{bool istream} \Rightarrow
(\text{nat} \Rightarrow \text{GatewayStatus}) \Rightarrow
(\text{nat} \Rightarrow \text{ECall-Info list}) \Rightarrow
\text{GatewayStatus istream} \Rightarrow \text{ECall-Info istream} \Rightarrow
\text{vcType istream} \Rightarrow
(\text{nat} \Rightarrow \text{GatewayStatus}) \Rightarrow \text{bool}
\]
where
\[
\text{tiTable-SampleT req a1 stop lose st-in buffer-in ack i1 vc st-out}
\]
\[\equiv\]
\[
\forall (t::\text{nat})
(\text{r::reqType list}) (\text{x::aType list})
(\text{y::stopType list}) (\text{z::bool list})
(\ast 1\ast)
(\text{sta-in t} = \text{init-state} \land \text{req t} = [\text{init}]
\rightarrow \text{ack t} = [\text{call}] \land i1 t = [] \land vc t = []
\land \text{sta-out t} = \text{call})
\land
(\ast 2\ast)
(\text{sta-in t} = \text{init-state} \land \text{req t} \neq [\text{init}]
\rightarrow \text{ack t} = [\text{init-state}] \land i1 t = [] \land vc t = []
\land \text{sta-out t} = \text{init-state})
\land
(\ast 3\ast)
((\text{sta-in t} = \text{call} \lor (\text{sta-in t} = \text{connection-ok} \land r \neq [\text{send}])) \land
\text{req t} = r \land \text{lose t} = [\text{False}]
\rightarrow \text{ack t} = [\text{connection-ok}] \land i1 t = [] \land vc t = []
\land \text{sta-out t} = \text{connection-ok})
\land
(\ast 4\ast)
\]
90
\[\begin{align*}
\text{definition} \\
\text{Sample-L ::} \\
\text{reqType istream} & \Rightarrow \text{ECall-Info istream} \Rightarrow \text{aType istream} \Rightarrow \\
\text{stopType istream} & \Rightarrow \text{bool istream} \Rightarrow \\
\text{(nat} & \Rightarrow \text{GatewayStatus}) \Rightarrow \text{(nat} & \Rightarrow \text{ECall-Info list}) \\
\text{GatewayStatus istream} & \Rightarrow \text{ECall-Info istream} \Rightarrow \text{vcType istream} \\
\Rightarrow \text{(nat} & \Rightarrow \text{GatewayStatus}) \Rightarrow \text{(nat} & \Rightarrow \text{ECall-Info list}) \\
\Rightarrow \text{bool} \\
\text{where} \\
\text{Sample-L req dt a1 stop lose st-in buffer-in} \\
\text{ack i1 vc st-out buffer-out}
\end{align*}\]
(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 ack i1 vc
≡
(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])
  ∧
  ( ∀ (k::nat). k ≤ (d+1) → lose (t+k) = [False])
  → i ((Suc t) + d) = inf-last-ti dt t
  ∧ ack (Suc t) = [sending-data]
  ∧
  ( ∀ (k::nat). k ≤ (d+1) → lose (t+k) = [False])
  → vc ((Suc t) + d) = [vc-com])

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

end
15 Gateway: Verification

theory Gateway-proof-aux
imports Gateway BitBoolTS
begin

15.1 Properties of the defined data types

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

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

lemma aType-lemma:
  assumes h1:msg (Suc 0) a
  shows a t = [] ∨ a t = [sc-ack]
using assms
  apply auto
  by (simp add: aType-empty)
lemma stopType-empty:
  assumes h1:msg (Suc 0) a
  and h2:a t ≠ [stop-vc]
  shows a t = []
proof (cases a t)
  assume a1:a t = []
  from this show ?thesis by simp
next
  fix aa l
  assume a2:a t = aa ≠ l
  show ?thesis
    proof (cases aa)
      assume a3:aa = stop-vc
      from h1 have sq1:length (a t) ≤ Suc 0 by (simp add: msg-def)
      from this and h1 and h2 and a2 and a3 show ?thesis by auto
    qed
qed

def stopType-nonenmp:
  assumes h1:msg (Suc 0) a
  and h2:a t ≠ []
  shows a t = [stop-vc]
proof (cases a t)
  assume a1:a t = []
  from this and h2 show ?thesis by simp
next
  fix aa l
  assume a2:a t = aa ≠ l
  show ?thesis
    proof (cases aa)
      assume a3:aa = stop-vc
      from h1 have sq1:length (a t) ≤ Suc 0 by (simp add: msg-def)
      from this and h1 and h2 and a2 and a3 show ?thesis by auto
    qed
qed

lemma stopType-lemma:
  assumes h1:msg (Suc 0) a
  shows a t = [] ∨ a t = [stop-vc]
using assms
apply auto
by (simp add: stopType-empty)

lemma vcType-empty:
  assumes h1:msg (Suc 0) a
  and h2:a t ≠ [vc-com]
  shows a t = []
proof (cases a t)
assume \( a \vdash a \vdash t = [] \)
from this show \( ?thesis \) by simp

next
fix \( aa \)
assume \( a2: aa \vdash t = aa \# t \)
show \( ?thesis \)
proof (cases \( aa \))
assume \( a3: \vdash aa = vc-com \)
from \( h1 \) have \( sg1: length (a t) \leq Suc 0 \) by (simp add: msg-def)
from this and \( h1 \) and \( h2 \) and \( a2 \) and \( a3 \) show \( ?thesis \) by auto
qed

lemma \( vcType-lemma \):
assumes \( h1: \vdash msg (Suc 0) a \)
shows \( \vdash a t = [] \lor a t = [vc-com] \)
using assms
apply auto
by (simp add: vcType-empty)

15.2 Properties of the Delay component

lemma \( Delay-L1 \):
assumes \( h1: \forall t1 < t. i1 t1 = [] \)
and \( h2: \vdash Delay y i1 d x i2 \)
and \( h3: i2 t2 = [] \)
shows \( \forall t2 < t + d. i2 t2 = [] \)
proof (cases \( t2 < d \))
assume \( a1: \vdash t2 < d \)
from \( h2 \) have \( sg1: i2 t2 < d \longrightarrow i2 t2 = [] \)
by (simp add: Delay-def)
from \( sg1 \) and \( a1 \) show \( ?thesis \) by simp
next
assume \( a2: \neg t2 < d \)
from \( h2 \) have \( sg2: d \leq t2 \longrightarrow i2 t2 = i1 (t2 - d) \)
by (simp add: Delay-def)
from \( a2 \) and \( sg2 \) have \( sg3: i2 t2 = i1 (t2 - d) \) by simp
from \( h1 \) and \( a2 \) and \( h3 \) and \( sg3 \) show \( ?thesis \) by auto
qed

lemma \( Delay-L2 \):
assumes \( h1: \forall t1 < t. i1 t1 = [] \)
and \( h2: \vdash Delay y i1 d x i2 \)
shows \( \forall t2 < t + d. i2 t2 = [] \)
using assms by (clarify, rule Delay-L1, auto)

lemma \( Delay-L3 \):
\[\forall t_1 \leq t. \hspace{1em} \text{y} t_1 = []\]
\[\text{and} h_2: \hspace{1em} \text{Delay} \hspace{1em} \text{y} \hspace{1em} i_1 \hspace{1em} d \hspace{1em} \text{x} \hspace{1em} i_2\]
\[\text{and} \hspace{1em} h_3: t_2 \leq t + d\]
\[\text{shows} x \hspace{-1em} t_2 = []\]

**proof** (cases \(t_2 < d\))
- **assume** \(a_1: \hspace{1em} t_2 < d\)
  - from \(h_2\) have \(sg_1: \hspace{1em} t_2 < d \rightarrow x \hspace{-1em} t_2 = []\)
    - by (simp add: Delay-def)
  - from \(sg_1\) and \(a_1\) show \(?\text{thesis}\) by simp
- **next**
  - **assume** \(a_2: \neg \hspace{1em} t_2 < d\)
    - from \(h_2\) have \(sg_2: \hspace{1em} d \leq t_2 \rightarrow x \hspace{-1em} t_2 = y \hspace{1em} (t_2 - d)\)
      - by (simp add: Delay-def)
    - from \(a_2\) and \(sg_2\) have \(sg_3: \hspace{1em} x \hspace{-1em} t_2 = y \hspace{1em} (t_2 - d)\) by simp
    - from \(h_1\) and \(a_2\) and \(h_3\) and \(sg_3\) show \(?\text{thesis}\) by auto

**qed**

**lemma** Delay-L4:
- **assumes** \(h_1: \forall t_1 \leq t. \hspace{1em} \text{y} t_1 = []\)
  - **and** \(h_2: \hspace{1em} \text{Delay} \hspace{1em} \text{y} \hspace{1em} i_1 \hspace{1em} d \hspace{1em} \text{x} \hspace{1em} i_2\)
  - **shows** \(\forall t_2 \leq t + d. \hspace{1em} x \hspace{-1em} t_2 = []\)
- **using** \(\text{assms by (clarify, rule Delay-L3, auto)}\)

**lemma** Delay-lengthOut1:
- **assumes** \(h_1: \forall t. \hspace{1em} \text{length} \hspace{1em} (x \hspace{-1em} t) \leq \text{Suc} \hspace{1em} 0\)
  - **and** \(h_2: \hspace{1em} \text{Delay} \hspace{1em} x \hspace{1em} i_1 \hspace{1em} d \hspace{1em} y \hspace{1em} i_2\)
  - **shows** \(\text{length} \hspace{1em} (y \hspace{-1em} t) \leq \text{Suc} \hspace{1em} 0\)
- **proof** (cases \(t < d\))
  - **assume** \(a_1: \hspace{1em} t < d\)
    - from \(h_2\) have \(sg_1: \hspace{1em} t < d \rightarrow y \hspace{-1em} t = []\)
      - by (simp add: Delay-def)
    - from \(a_1\) and \(sg_1\) show \(?\text{thesis}\) by auto
  - **next**
    - **assume** \(a_2: \neg \hspace{1em} t < d\)
      - from \(h_2\) have \(sg_2: \hspace{1em} t \geq d \rightarrow (y \hspace{-1em} t = x \hspace{1em} (t - d))\)
        - by (simp add: Delay-def)
      - from \(a_2\) and \(sg_2\) and \(h_1\) show \(?\text{thesis}\) by auto

**qed**

**lemma** Delay-msg1:
- **assumes** \(h_1: \hspace{1em} \text{msg} \hspace{1em} \text{Suc} \hspace{1em} 0 \hspace{1em} x\)
  - **and** \(h_2: \hspace{1em} \text{Delay} \hspace{1em} x \hspace{1em} i_1 \hspace{1em} d \hspace{1em} y \hspace{1em} i_2\)
  - **shows** \(\text{msg} \hspace{1em} \text{Suc} \hspace{1em} 0 \hspace{1em} y\)
- **using** \(\text{assms by (simp add: msg-def Delay-lengthOut1)}\)
15.3 Properties of the Loss component

lemma Loss-L1:
assumes h1:∀ t2 < t. i2 t2 = []
and h2:Loss lose a i2 y i
and h3:t2 < t
and h4:ts lose
shows i t2 = []
proof (cases lose t2 = [False])
assume a1:lose t2 = [False]
from assms and a1 show ?thesis by (simp add: Loss-def)
next
assume a2:lose t2 ≠ [False]
from a2 and h4 have sg1:lose t2 = [True] by (simp add: ts-bool-True)
from assms and sg1 show ?thesis by (simp add: Loss-def)
qed

lemma Loss-L2:
assumes h1:∀ t2 < t. i t2 = []
and h2:Loss lose a i2 y i
and h3:ts lose
shows ∀ t2 < t. i t2 = []
using assms
apply clarify
by (rule Loss-L1, auto)

lemma Loss-L3:
assumes h1:∀ t2 < t. a t2 = []
and h2:Loss lose a i2 y i
and h3:t2 < t
and h4:ts lose
shows y t2 = []
proof (cases lose t2 = [False])
assume a1:lose t2 = [False]
from assms and a1 show ?thesis by (simp add: Loss-def)
next
assume a2:lose t2 ≠ [False]
from a2 and h4 have sg1:lose t2 = [True] by (simp add: ts-bool-True)
from assms and sg1 show ?thesis by (simp add: Loss-def)
qed

lemma Loss-L4:
assumes h1:∀ t2 < t. a t2 = []
and h2:Loss lose a i2 y i
and h3:ts lose
shows ∀ t2 < t. y t2 = []
using assms
apply clarify
by (rule Loss-L3, auto)
lemma Loss-L5:
assumes h1:∀ t1 ≤ t. a t1 = []
and h2:Loss lose a i2 y i
and h3:t2 ≤ t
and h4:ts lose
shows y t2 = []
proof (cases lose t2 = [False])
  assume a1:lose t2 = [False]
  from assms and a1 show ?thesis by (simp add: Loss-def)
next
  assume a2:lose t2 ≠ [False]
  from a2 and h4 have sg1:lose t2 = [True] by (simp add: ts-bool-True)
  from assms and sg1 show ?thesis by (simp add: Loss-def)
qed

lemma Loss-L5Suc:
assumes h1:∀ j ≤ d. a (t + Suc j) = []
and h2:Loss lose a i2 y i
and h3:Suc j ≤ d
and h4:ts lose
shows y (t + Suc j) = []
proof (cases lose (t + Suc j) = [False])
  assume a1:lose (t + Suc j) = [False]
  from a1 and h2 have sg1:lose (t + Suc j) = [True] by (simp add: ts-bool-True)
  from assms and sg1 show ?thesis by (simp add: Loss-def)
next
  assume a2:lose (t + Suc j) ≠ [False]
  from a2 and h4 have sg1:lose (t + Suc j) = [True] by (simp add: ts-bool-True)
  from assms and sg1 show ?thesis by (simp add: Loss-def)
qed

lemma Loss-L6:
assumes h1:∀ t2 ≤ t. a t2 = []
and h2:Loss lose a i2 y i
and h3:ts lose
shows ∀ t2 ≤ t. y t2 = []
using assms
apply clarify
by (rule Loss-L5, auto)

lemma Loss-lengthOut1:
assumes h1:∀ t. length (a t) ≤ Suc 0
and h2:Loss lose a i2 x i
and h3:ts lose
shows length (x t) ≤ Suc 0
proof (cases lose t = [False])
  assume a1:lose t = [False]
  from a1 and h2 have sg1:lose x t = a t by (simp add: Loss-def)
  from h1 have sg2:length (a t) ≤ Suc 0 by auto
  from sg1 and sg2 show ?thesis by simp
next
assume a2:lose t ≠ [False]
from a2 and h2 have sg2:x t = [] by (simp add: Loss-def)
from sg2 show ?thesis by simp
qed

lemma Loss-lengthOut2:
  assumes h1:∀ t. length (a t) ≤ Suc 0
  and h2:Loss lose a i2 x i
  shows ∀ t. length (x t) ≤ Suc 0
using assms
by (simp add: Loss-lengthOut1)

lemma Loss-msg1:
  assumes h1:msg (Suc 0) a
  and h2:Loss lose a i2 x i
  shows msg (Suc 0) x
using assms
by (simp add: msg-def Loss-def Loss-lengthOut1)

15.4 Properties of the composition of Delay and Loss components

lemma Loss-Delay-length-y:
  assumes h1:∀ t. length (a t) ≤ Suc 0
  and h2:Delay x i1 d y i2
  and h3:Loss lose a i2 x i
  shows length (y t) ≤ Suc 0
proof –
  from h1 and h3 have sg1:∀ t. length (x t) ≤ Suc 0
  by (simp add: Loss-lengthOut2)
  from this and h2 show ?thesis
  by (simp add: Delay-lengthOut1)
qed

lemma Loss-Delay-msg-a:
  assumes h1:msg (Suc 0) a
  and h2:Delay x i1 d y i2
  and h3:Loss lose a i2 x i
  shows msg (Suc 0) y
using assms
by (simp add: msg-def Loss-Delay-length-y)

15.5 Auxiliary Lemmas

lemma inf-last-ti2:
  assumes h1:inf-last-ti dt (Suc (Suc t)) ≠ []
  shows inf-last-ti dt (Suc (Suc (t + k))) ≠ []
using assms
proof (induct k)
case 0
  from this show ?case by auto
next
  case Suc
  from this show ?case by auto
qed

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

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

lemma aux-lemma-lose-2:
  assumes h1:∀ j≤((2::nat) * d + (4::nat) + k). lose (t + j) = [False]
  shows ∀ x≤d + (1::nat). lose (t + x) = [False]
  using assms by auto

lemma aux-lemma-lose-3a:
  assumes h1:∀ j≤2 * d + (4 + k). lose (t + j) = [False]
  and h2:ka ≤ Suc d

101
shows \( \text{lose } (d + (t + (3 + k)) + ka) = [\text{False}] \)

proof –
from \( h_2 \) have \( sg_1: (d + 3 + k + ka) \leq 2 * d + (4 + k) \)
by arith
from \( h_1 \) and \( h_2 \) and \( sg_1 \) obtain \( j \) where \( a1:j = (d + 3 + k + ka) \) and  
\( a2: \text{lose } (t + j) = [\text{False}] \)
by simp
from \( h_2 \) and \( sg_1 \) have \( sg_2: (t + (d + 3 + k + ka)) = (d + (t + (3 + k)) + ka) \)
by arith
from \( h_1 \) and \( h_2 \) and \( a1 \) and \( a2 \) and \( sg_2 \) show ?thesis 
by simp
qed

lemma aux-lemma-lose-3:
assumes \( h_1: \forall j \leq 2 * d + (4 + k). \text{lose } (t + j) = [\text{False}] \)
shows \( \forall ka \leq \text{Suc } d. \text{lose } (d + (t + (3 + k)) + ka) = [\text{False}] \)
using assms 
by (auto, simp add: aux-lemma-lose-3a)

lemma aux-arith1-Gateway7:
assumes \( h_1: t_2 - t \leq (2::nat) * d + (t + ((4::nat) + k)) \)
and \( h_2: t_2 < t + (3::nat) + k + d \)
and \( h_3: \neg t_2 - d < (0::nat) \)
shows \( t_2 - d < t + (3::nat) + k \)
using assms by arith

lemma ts-lose-ack-st1ts:
assumes \( h_1: ts \text{ lose } \)
and \( h_2: \text{lose } t = [\text{True}] \longrightarrow \text{ack } t = [x] \land \text{st-out } t = x \)
and \( h_3: \text{lose } t = [\text{False}] \longrightarrow \text{ack } t = [y] \land \text{st-out } t = y \)
shows \( \text{ack } t = [\text{st-out } t] \)
proof (cases lose t = [\text{False}])
assume \( a1: \text{lose } t = [\text{False}] \)
from this and \( h_3 \) show ?thesis by simp
next
assume \( a2: \text{lose } t \neq [\text{False}] \)
from this and \( h_1 \) have \( ag1: \text{lose } t = [\text{True}] \) by (simp add: ts-bool-True)
from this and \( a2 \) and \( h_2 \) show ?thesis by simp
qed

lemma ts-lose-ack-st1:
assumes \( h_1: \text{lose } t = [\text{True}] \lor \text{lose } t = [\text{False}] \)
and \( h_2: \text{lose } t = [\text{True}] \longrightarrow \text{ack } t = [x] \land \text{st-out } t = x \)
and \( h_3: \text{lose } t = [\text{False}] \longrightarrow \text{ack } t = [y] \land \text{st-out } t = y \)
shows $\text{ack } t = [\text{st-out } t]$

proof (cases $\text{lose } t = [\text{False}]$)
  assume $a1 : \text{lose } t = [\text{False}]$
  from this and $h3$ show $?\text{thesis}$ by simp
next
  assume $a2 : \text{lose } t \neq [\text{False}]$
  from this and $h1$ have $ag1 : \text{lose } t = [\text{True}]$ by (simp add: ts-bool-True)
  from this and $a2$ and $h2$ show $?\text{thesis}$ by simp
qed

lemma ts-lose-ack-st2ts:
  assumes $h1 : \text{ts lose}$
  and $h2 : \text{lose } t = [\text{True}] \rightarrow$
  $\text{ack } t = [x] \land i1 t = [] \land vc t = [] \land \text{st-out } t = x$
  and $h3 : \text{lose } t = [\text{False}] \rightarrow$
  $\text{ack } t = [y] \land i1 t = [] \land vc t = [] \land \text{st-out } t = y$
  shows $\text{ack } t = [\text{st-out } t]$

proof (cases $\text{lose } t = [\text{False}]$)
  assume $a1 : \text{lose } t = [\text{False}]$
  from this and $h3$ show $?\text{thesis}$ by simp
next
  assume $a2 : \text{lose } t \neq [\text{False}]$
  from this and $h1$ have $ag1 : \text{lose } t = [\text{True}]$ by (simp add: ts-bool-True)
  from this and $a2$ and $h2$ show $?\text{thesis}$ by simp
qed

lemma ts-lose-ack-st2vc-com:
  assumes $h1 : \text{lose } t = [\text{True}] \lor \text{lose } t = [\text{False}]$
  and $h2 : \text{lose } t = [\text{True}] \rightarrow$
  $\text{ack } t = [x] \land i1 t = [] \land vc t = [] \land \text{st-out } t = x$
  and $h3 : \text{lose } t = [\text{False}] \rightarrow$
  $\text{ack } t = [y] \land i1 t = [] \land vc t = [] \land \text{st-out } t = y$
  shows $\text{ack } t = [\text{st-out } t]$

proof (cases $\text{lose } t = [\text{False}]$)
  assume $a1 : \text{lose } t = [\text{False}]$
  from this and $h3$ show $?\text{thesis}$ by simp
next
  assume $a2 : \text{lose } t \neq [\text{False}]$
  from this and $h1$ have $ag1 : \text{lose } t = [\text{True}]$ by (simp add: ts-bool-True)
  from this and $a2$ and $h2$ show $?\text{thesis}$ by simp
qed
\[ \text{ack } t = [y] \land i_1 t = \emptyset \land vc t = [vc-com] \land st-out t = y \]
shows \( \text{ack } t = [st-out t] \)

**proof** (cases lose t = [False])
assume \( a_1: \text{lose } t = [False] \)
from this and \( h_3 \) show \( \text{thesis by simp} \)
next
assume \( a_2: \text{lose } t \neq [False] \)
from this and \( h_1 \) have \( ag_1: \text{lose } t = [True] \) by (simp add: ts-bool-True)
from this and \( a_2 \) and \( h_2 \) show \( \text{thesis by simp} \)
qed

**lemma** ts-lose-ack-st2send:
assumes \( h_1: \text{lose } t = [True] \lor \text{lose } t = [False] \)
and \( h_2: \text{lose } t = [True] \longrightarrow \)
\( \text{ack } t = [x] \land i_1 t = \emptyset \land vc t = \emptyset \land st-out t = x \)
and \( h_3: \text{lose } t = [False] \longrightarrow \)
\( \text{ack } t = [y] \land i_1 t = b t \land vc t = \emptyset \land st-out t = y \)
shows \( \text{ack } t = [st-out t] \)

**proof** (cases lose t = [False])
assume \( a_1: \text{lose } t = [False] \)
from this and \( h_3 \) show \( \text{thesis by simp} \)
next
assume \( a_2: \text{lose } t \neq [False] \)
from this and \( h_1 \) have \( ag_1: \text{lose } t = [True] \) by (simp add: ts-bool-True)
from this and \( a_2 \) and \( h_2 \) show \( \text{thesis by simp} \)
qed

**lemma** tiTable-ack-st-splitten:
assumes \( h_1: \text{ts lose} \)
and \( h_2: \text{msg } (\text{Suc } 0) \) \( a_1 \)
and \( h_3: \text{msg } (\text{Suc } 0) \) \( \text{stop} \)
and \( h_4: \text{st-in } t = \text{init-state} \land \text{req } t = [\text{init}] \longrightarrow \)
\( \text{ack } t = [\text{call}] \land i_1 t = \emptyset \land vc t = \emptyset \land st-out t = \text{call} \)
and \( h_5: \text{st-in } t = \text{init-state} \land \text{req } t \neq [\text{init}] \longrightarrow \)
\( \text{ack } t = [\text{init-state}] \land i_1 t = \emptyset \land vc t = \emptyset \land st-out t = \text{init-state} \)
and \( h_6: \text{st-in } t = \text{call} \lor \text{st-in } t = \text{connection-ok} \land \text{req } t \neq [\text{send}] \) \land \( \text{lose } t = [\text{False}] \longrightarrow \)
\( \text{ack } t = [\text{connection-ok}] \land i_1 t = \emptyset \land vc t = \emptyset \land st-out t = \text{connection-ok} \)
and \( h_7: \text{st-in } t = \text{call} \lor \text{st-in } t = \text{connection-ok} \lor \text{st-in } t = \text{sending-data} \land\)
\( \text{lose } t = [\text{True}] \longrightarrow \)
\( \text{ack } t = [\text{init-state}] \land i_1 t = \emptyset \land vc t = \emptyset \land st-out t = \text{init-state} \)
and \( h_8: \text{st-in } t = \text{connection-ok} \land \text{req } t = [\text{send}] \land \text{lose } t = [\text{False}] \longrightarrow \)
\( \text{ack } t = [\text{sending-data}] \land i_1 t = b t \land vc t = \emptyset \land st-out t = \text{sending-data} \)
and \( h_9: \text{st-in } t = \text{sending-data} \land a_1 t = \emptyset \land \text{lose } t = [\text{False}] \longrightarrow \)
\( \text{ack } t = [\text{voice-com}] \land i_1 t = \emptyset \land vc t = [vc-com] \land st-out t = \text{voice-com} \)
and h11: st-in t = voice-com ∧ stop t = [] ∧ lose t = [False] →
  ack t = [voice-com] ∧ i1 t = [] ∧ vc t = [vc-com] ∧ st-out t = voice-com
and h12: st-in t = voice-com ∧ stop t = [] ∧ lose t = [True] →
  ack t = [voice-com] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = voice-com
and h13: st-in t = voice-com ∧ stop t = [stop-vc] →
  ack t = [init-state] ∧ i1 t = [] ∧ vc t = [] ∧ st-out t = init-state
shows ack t = [st-out t]
proof
from h1 and h6 and h7 have sg1: lose t = [True] ∨ lose t = [False]
  by (simp add: ts-bool-True-False)
show ?thesis
proof (cases st-in t)
assume a1: st-in t = init-state
from a1 and h4 and h5 show ?thesis
  proof (cases req t = [init])
  assume a11: req t = [init]
  from a11 and a1 and h4 and h5 show ?thesis by simp
next
  assume a12: req t ≠ [init]
  from a12 and a1 and h4 and h5 show ?thesis by simp
qed
next
assume a2: st-in t = call
from a2 and sg1 and h6 and h7 show ?thesis
  apply simp
  by (rule ts-lose-ack-st2, assumption+)
next
assume a3: st-in t = connection-ok
from a3 and h6 and h7 and h8 show ?thesis apply simp
  proof (cases req t = [send])
  assume a31: req t = [send]
  from this and a3 and h6 and h7 and h8 and sg1 show ?thesis
  apply simp
  by (rule ts-lose-ack-st2send, assumption+)
next
  assume a32: req t ≠ [send]
  from this and a3 and h6 and h7 and h8 and sg1 show ?thesis
  apply simp
  by (rule ts-lose-ack-st2, assumption+)
qed
next
assume a4: st-in t = sending-data
from sg1 and a4 and h7 and h9 and h10 show ?thesis apply simp
  proof (cases a1 t = [])
  assume a41: a1 t = []
  from this and a4 and sg1 and h7 and h9 and h10 show ?thesis
  apply simp
  by (rule ts-lose-ack-st2, assumption+)
next
assume $a_4^2: a_1 t \neq []$

from this and $h_2$ have $a_1 t = [sc\text{-ack}]$ by (simp add: $a\text{Type-nonempty}$)

from this and $a_4$ and $a_4^2$ and $sg_1$ and $h_7$ and $h_9$ and $h_{10}$ show $?\text{thesis}$
apply simp
by (rule $ts\text{-lose-ack-st2vc\text{-com}}$, assumption+)

qed

next

assume $a_5: st\text{-in} t = voice\text{-com}$

from $a_5$ and $h_{11}$ and $h_{12}$ and $h_{13}$ show $?\text{thesis}$ apply simp

proof (cases $\text{stop} t = []$

assume $a_{51}: \text{stop} t = []$

from this and $a_5$ and $h_{11}$ and $h_{12}$ and $h_{13}$ and $sg_1$ show $?\text{thesis}$
apply simp
by (rule $ts\text{-lose-ack-st2vc\text{-com}}$, assumption+)

next

assume $a_{52}: \text{stop} t \neq []$

from this and $h_3$ have $sg_7:\text{stop} t = [\text{stop\text{-vc}]}
by (simp add: $\text{stop\text{Type-nonempty}}$

from this and $a_5$ and $a_{52}$ and $h_{13}$ show $?\text{thesis}$ by simp

qed

qed

qed

lemma $ti\text{Table-ack-st}$:

assumes $h_1: ti\text{Table-SampleT req} a_1 \text{ stop lose st\text{-in b ack i1 vc st\text{-out}}$
and $h_2: ts\text{ lose}$
and $h_3: \text{msg} (\text{Suc 0}) a_1$
and $h_4: \text{msg} (\text{Suc 0}) \text{ stop}$

shows $\text{ack} t = [\text{st\text{-out} t}]$

proof –

from assms have $sg_1$:
$st\text{-in} t = \text{init\text{-state} } \land \text{ req} t = [\text{init}] \quad\longrightarrow$
$\text{ack} t = [\text{call}] \land i_1 t = [] \land \text{ vc} t = [] \land \text{ st\text{-out} t = call}$
by (simp add: $ti\text{Table-SampleT-def}$)

from assms have $sg_2$:
$st\text{-in} t = \text{init\text{-state} } \land \text{ req} t \neq [\text{init}] \quad\longrightarrow$
$\text{ack} t = [\text{init\text{-state} } ] \land i_1 t = [] \land \text{ vc} t = [] \land \text{ st\text{-out} t = init\text{-state} }$
by (simp add: $ti\text{Table-SampleT-def}$)

from assms have $sg_3$:
$(st\text{-in} t = \text{call } \lor st\text{-in} t = \text{ connection\text{-ok} } \land \text{ req} t \neq [\text{send}]) \land$
$\text{ lose} t = [\text{False} ] \quad\longrightarrow$
$\text{ack} t = [\text{connection\text{-ok} } ] \land i_1 t = [] \land \text{ vc} t = [] \land \text{ st\text{-out} t = connection\text{-ok} }$
by (simp add: $ti\text{Table-SampleT-def}$)

from assms have $sg_4$:
$(st\text{-in} t = \text{call } \lor st\text{-in} t = \text{ connection\text{-ok} } \lor st\text{-in} t = \text{ sending\text{-data} } ) \land$
$\text{ lose} t = [\text{True} ] \quad\longrightarrow$
$\text{ack} t = [\text{init\text{-state} } ] \land i_1 t = [] \land \text{ vc} t = [] \land \text{ st\text{-out} t = init\text{-state} }$
by (simp add: $ti\text{Table-SampleT-def}$)

from assms have $sg_5$:
\[\begin{align*}
st\text{-in } t &= \text{connection-ok} \land req t = [\text{send}] \land lose t = [\text{False}] \quad \rightarrow \\
ack t &= [\text{sending-data}] \land i1 t = b t \land vc t = [] \land st\text{-out } t = \text{sending-data} \\
&\text{by (simp add: tiTable-SampleT-def)} \\
\end{align*}\]

```plaintext
from hasms have ss6:
  st\text{-in } t = \text{sending-data} \land a1 t = [] \land lose t = [\text{False}] \quad \rightarrow \\
  ack t = [\text{sending-data}] \land i1 t = [] \land vc t = [] \land st\text{-out } t = \text{sending-data} \\
  by (simp add: tiTable-SampleT-def)
```

```plaintext
from hasms have ss7:
  st\text{-in } t = \text{voice-com} \land stop t = [] \land lose t = [\text{True}] \quad \rightarrow \\
  ack t = [\text{voice-com}] \land i1 t = [] \land vc t = [\text{vc-com}] \land st\text{-out } t = \text{voice-com} \\
  by (simp add: tiTable-SampleT-def)
```

```plaintext
from hasms have ss8:
  st\text{-in } t = \text{voice-com} \land stop t = [\text{stop-vc}] \quad \rightarrow \\
  ack t = [\text{init-state}] \land i1 t = [] \land vc t = [] \land st\text{-out } t = \text{init-state} \\
  by (simp add: tiTable-SampleT-def)
```

```plaintext
from h2 and h3 and h4 and ss1 and ss2 and ss3 and ss4 and ss5 and ss6 and ss7 and ss8 and ss9 and ss10 show \(?\)thesis \\
by (rule tiTable-ack-st-splitten)
```

```plaintext
qed
```

```plaintext
lemma \text{tiTable-ack-st-hd:}
  assumes h1: \text{tiTable-SampleT } req a1 stop lose st-in b ack i1 vc st-out \\
  and h2: ts lose \\
  and h3: msg (Suc 0) a1 \\
  and h4: msg (Suc 0) stop \\
  shows st-out t = \text{hd} (ack t) \\
  using hasms by (simp add: tiTable-ack-st)
```

```plaintext
lemma \text{tiTable-ack-connection-ok:}
  assumes h1: \text{tiTable-SampleT } req x stop lose st-in b ack i1 vc st-out \\
  and h2: ack t = [\text{connection-ok}] \\
  and h3: msg (Suc 0) x \\
  and h4: ts lose \\
  and h5: msg (Suc 0) stop \\
  shows (st-in t = call \lor st-in t = \text{connection-ok} \land req t \neq [\text{send}] ) \land \\
  lose t = [\text{False}] \\
proof \\
  from h1 and h4 have ss1: lose t = [\text{True}] \lor lose t = [\text{False}] \\
  by (simp add: ts-bool-True-False)
from h1 and h3 have ss2: x t = [] \lor x t = [\text{sc-ack}] \\
  by (simp add: aType-lemma)
```

107
from $h1$ and $h5$ have $\text{sg3:stop } t = [] \lor \text{stop } t = \text{[stop-vc]}$

by (simp add: stopType-lemma) show ?thesis

proof (cases $\text{st-in } t$)
  assume $a1: \text{st-in } t = \text{init-state}$
  show ?thesis
  proof (cases $\text{req } t = [\text{init}]$)
    assume $a11: \text{req } t = [\text{init}]$
    from $h1$ and $a1$ and $a11$ and $h2$ show ?thesis by (simp add: tiTable-SampleT-def)
  next
    assume $a12: \text{req } t \neq [\text{init}]$
    from $h1$ and $a1$ and $a12$ and $h2$ show ?thesis by (simp add: tiTable-SampleT-def)
  qed

next
  assume $a2: \text{st-in } t = \text{call}$
  show ?thesis
  proof (cases $\text{lose } t = [\text{True}]$)
    assume $a21: \text{lose } t = [\text{True}]$
    from $h1$ and $a2$ and $a21$ and $h2$ show ?thesis by (simp add: tiTable-SampleT-def)
  next
    assume $a22: \text{lose } t \neq [\text{True}]$
    from this and $h4$ have $a22a: \text{lose } t = [\text{False}]$ by (simp add: ts-bool-False)
    from $h1$ have
      $(\text{st-in } t = \text{call } \lor \text{st-in } t = \text{connection-ok } \land \text{req } t \neq [\text{send}]) \land$
      $\text{lose } t = [\text{False}] \longrightarrow$
      $\text{ack } t = [\text{connection-ok } ] \land \text{i1 } t = [] \land \text{vc } t = [] \land \text{st-out } t = \text{connection-ok}$
    by (simp add: tiTable-SampleT-def)
    from this and $a2$ and $a22a$ and $h2$ show ?thesis by simp
  qed

next
  assume $a3: \text{st-in } t = \text{connection-ok}$
  show ?thesis
  proof (cases $\text{lose } t = [\text{True}]$)
    assume $a31: \text{lose } t = [\text{True}]$
    from $h1$ have
      $(\text{st-in } t = \text{call } \lor \text{st-in } t = \text{connection-ok } \lor \text{st-in } t = \text{sending-data}) \land$
      $\text{lose } t = [\text{True}] \longrightarrow$
      $\text{ack } t = [\text{init-state } ] \land \text{i1 } t = [] \land \text{vc } t = [] \land \text{st-out } t = \text{init-state}$
    by (simp add: tiTable-SampleT-def)
    from this and $a3$ and $a31$ and $h2$ show ?thesis by simp
  qed

next
  assume $a32: \text{lose } t \neq [\text{True}]$
  from this and $h4$ have $a32a: \text{lose } t = [\text{False}]$ by (simp add: ts-bool-False)
  show ?thesis
  proof (cases $\text{req } t = [\text{send}]$)
    assume $a321: \text{req } t = [\text{send}]$
    from $h1$ and $a3$ and $a32a$ and $a321$ and $h2$ show ?thesis
    by (simp add: tiTable-SampleT-def)
  next
    assume $a322: \text{req } t \neq [\text{send}]$

108
from h1 and a3 and a32a and a322 and h2 show \(?thesis\)
  by (simp add: tiTable-SampleT-def)
qed
qed

next
assume a4: st-in t = sending-data
show \(?thesis\)
proof (cases lose t = [True])
  assume a41: lose t = [True]
  from h1 and a4 and a41 and h2 show \(?thesis\) by (simp add: tiTable-SampleT-def)
next
assume a42: lose t ≠ [True]
from this and h4 have a42a: lose t = [False] by (simp add: ts-bool-False)
show \(?thesis\)
proof (cases x t = [sc-ack])
  assume a421: x t = [sc-ack]
  from h1 and a4 and a42a and a421 and h2 show \(?thesis\)
    by (simp add: tiTable-SampleT-def)
next
assume a422: x t ≠ [sc-ack]
from this and h3 have a422a: x t = [] by (simp add: aType-empty)
from h1 and a4 and a42a and a422a and h2 show \(?thesis\)
  by (simp add: tiTable-SampleT-def)
qed
qed

next
assume a5: st-in t = voice-com
show \(?thesis\)
proof (cases stop t = [stop-vc])
  assume a51: stop t = [stop-vc]
  from h1 and a5 and a51 and h2 show \(?thesis\)
    by (simp add: tiTable-SampleT-def)
next
assume a52: stop t ≠ [stop-vc]
from this and h5 have a52a: stop t = [] by (simp add: stopType-empty)
show \(?thesis\)
proof (cases lose t = [True])
  assume a521: lose t = [True]
  from h1 and a5 and a52a and a521 and h2 show \(?thesis\)
    by (simp add: tiTable-SampleT-def)
next
assume a522: lose t ≠ [True]
from this and h4 have a522a: lose t = [False] by (simp add: ts-bool-False)
from h1 and a5 and a52a and a522a and h2 show \(?thesis\)
  by (simp add: tiTable-SampleT-def)
qed
qed
qed
lemma \textit{tiTable-i1-1}:
\begin{itemize}
  \item \textbf{assumes} \( h1 : \text{tiTable-SampleT} \) \( \text{req} \ x \ \text{stop} \ \text{lose} \ \text{st-in} \ b \ \text{ack} \ \text{i1} \ \text{vc} \ \text{st-out} \)
  \begin{itemize}
    \item and \( h2 : \text{ts} \ \text{lose} \)
    \item and \( h3 : \text{msg} \ (\text{Suc} \ 0) \ x \)
    \item and \( h4 : \text{msg} \ (\text{Suc} \ 0) \ \text{stop} \)
    \item and \( h5 : \text{ack} \ t = \text{[connection-ok]} \)
  \end{itemize}
  \textbf{shows} \( \text{i1} \ t = [] \)
\end{itemize}
\textbf{proof} –
from assms have \( sg1 : \) 
\begin{itemize}
  \item \((\text{st-in} \ t = \text{call} \lor \text{st-in} \ t = \text{connection-ok} \land \text{req} \ t \neq [\text{send}]) \land \text{lose} \ t = [\text{False}] \)
  \item by (simp add: \text{tiTable-ack-connection-ok})
\end{itemize}
from this and \( h1 \) show \(?\text{thesis}?) \ by (simp add: \text{tiTable-SampleT-def})
\textbf{qed}

lemma \textit{tiTable-ack-call}:
\begin{itemize}
  \item \textbf{assumes} \( h1 : \text{tiTable-SampleT} \) \( \text{req} \ x \ \text{stop} \ \text{lose} \ \text{st-in} \ b \ \text{ack} \ \text{i1} \ \text{vc} \ \text{st-out} \)
  \begin{itemize}
    \item and \( h2 : \text{ack} \ t = \text{[call]} \)
    \item and \( h3 : \text{msg} \ (\text{Suc} \ 0) \ x \)
    \item and \( h4 : \text{ts} \ \text{lose} \)
    \item and \( h5 : \text{msg} \ (\text{Suc} \ 0) \ \text{stop} \)
  \end{itemize}
  \textbf{shows} \( \text{st-in} \ t = \text{init-state} \land \text{req} \ t = [\text{init}] \)
\end{itemize}
\textbf{proof} –
from \( h1 \) and \( h4 \) have \( sg1 : \text{lose} \ t = [\text{True}] \lor \text{lose} \ t = [\text{False}] \)
  \item by (simp add: \text{ts-bool-True-False})
from \( h1 \) and \( h3 \) have \( sg2 : x \ t = [] \lor x \ t = [\text{sc-ack}] \)
  \item by (simp add: \text{aType-lemma})
from \( h1 \) and \( h5 \) have \( sg3 : \text{stop} \ t = [] \lor \text{stop} \ t = [\text{stop-vc}] \)
  \item by (simp add: \text{stopType-lemma})
show \(?\text{thesis}?)
\textbf{proof} (cases \text{st-in} \ t)
assume \( a1 : \text{st-in} \ t = \text{init-state} \)
show \(?\text{thesis}?)
\textbf{proof} (cases \text{req} \ t = [\text{init}])
assume \( a11 : \text{req} \ t = [\text{init}] \)
from \( h1 \) and \( a1 \) and \( a11 \) and \( h2 \) show \(?\text{thesis}?)
  \item by (simp add: \text{tiTable-SampleT-def})
next
assume \( a12 : \text{req} \ t \neq [\text{init}] \)
from \( h1 \) and \( a1 \) and \( a12 \) and \( h2 \) show \(?\text{thesis}?)
  \item by (simp add: \text{tiTable-SampleT-def})
\textbf{qed}
next
assume \( a2 : \text{st-in} \ t = \text{call} \)
show \(?\text{thesis}?)
\textbf{proof} (cases \text{lose} \ t = [\text{True}])
assume \( a21 : \text{lose} \ t = [\text{True}] \)
from \( h1 \) and \( a2 \) and \( a21 \) and \( h2 \) show \(?\text{thesis}?)
\end{itemize}
by (simp add: tiTable-SampleT-def)

next
assume a22:lose t \not= [True]
from this and h4 have a22a:lose t = [False] by (simp add: ts-bool-False)
from h1 and a2 and a22a and h2 show \(\)thesis
  by (simp add: tiTable-SampleT-def)
qed

next
assume a3:st-in t = connection-ok
show \(\)thesis
proof (cases lose t = [True])
  assume a31:lose t = [True]
  from h1 and a3 and a31 and h2 show \(\)thesis by (simp add: tiTable-SampleT-def)
next
assume a32:lose t \not= [True]
from this and h4 have a32a:lose t = [False] by (simp add: ts-bool-False)
show \(\)thesis
proof (cases req t = [send])
  assume a321:req t = [send]
  from h1 and a3 and a32a and a321 and h2 show \(\)thesis
    by (simp add: tiTable-SampleT-def)
next
assume a322:req t \not= [send]
from h1 and a3 and a32a and a322 and h2 show \(\)thesis
  by (simp add: tiTable-SampleT-def)
qed
qed

next
assume a4:st-in t = sending-data
show \(\)thesis
proof (cases lose t = [True])
  assume a41:lose t = [True]
  from h1 and a4 and a41 and h2 show \(\)thesis
    by (simp add: tiTable-SampleT-def)
next
assume a42:lose t \not= [True]
from this and h4 have a42a:lose t = [False] by (simp add: ts-bool-False)
show \(\)thesis
proof (cases x t = [sc-ack])
  assume a421:x t = [sc-ack]
  from h1 and a4 and a42a and a421 and h2 show \(\)thesis
    by (simp add: tiTable-SampleT-def)
next
assume a422: x t \not= [sc-ack]
from this and h3 have a422a:x t = [] by (simp add: aType-empty)
from h1 and a4 and a42a and a422a and h2 show \(\)thesis
  by (simp add: tiTable-SampleT-def)
qed
qed
next
  assume a5: st-in t = voice-com
  show thesis
    proof (cases stop t = [stop-vc])
      assume a51: stop t = [stop-vc]
      from h1 and a5 and a51 and h2 show thesis
        by (simp add: tiTable-SampleT-def)
    next
      assume a52: stop t ≠ [stop-vc]
      from this and h5 have a52a: stop t = []
        by (simp add: stopType-empty)
      show thesis
        proof (cases lose t = [True])
          assume a521: lose t = [True]
          from h1 and a5 and a52a and a521 and h2 show thesis
            by (simp add: tiTable-SampleT-def)
        next
          assume a522: lose t ≠ [True]
          from this and h4 have a522a: lose t = [False]
            by (simp add: ts-bool-False)
          from h1 and a5 and a52a and a522a and h2 show thesis
            by (simp add: tiTable-SampleT-def)
        qed
    qed
  qed

lemma tiTable-i1-2:
  assumes h1: tiTable-SampleT req a1 stop lose st-in b ack i1 vc st-out
    and h2: ts lose
    and h3: msg (Suc 0) a1 and h4: msg (Suc 0) stop
    and h5: ack t = [call]
  shows i1 t = []
  proof -
    from assms have sg1: st-in t = init-state ∧ req t = [init]
      by (simp add: tiTable-ack-call)
    from this and h1 show thesis
      by (simp add: tiTable-SampleT-def)
  qed

lemma tiTable-ack-init0:
  assumes h1: tiTable-SampleT req a1 stop lose
    (fin-inf-append [init-state] st)
    b ack i1 vc st
    and h2: req 0 = []
  shows ack 0 = [init-state]
  proof -
    have sg1: (fin-inf-append [init-state] st) (0::nat) = init-state
      by (simp add: fin-inf-append-def)
    from h1 and sg1 and h2 show thesis
      by (simp add: tiTable-SampleT-def)
  qed
lemma \textit{tiTable-ack-init}:  
\textbf{assumes} h1: tiTable-SampleT req a1 stop lose  
\hspace{2em} (fin-inf-append [init-state] st)  
\hspace{2em} b ack i1 vc st  
\hspace{2em} \text{and} h2: ts lose  
\hspace{2em} \text{and} h3: msg (Suc 0) a1  
\hspace{2em} \text{and} h4: msg (Suc 0) stop  
\hspace{2em} \text{and} h5: \forall t1 \leq t. req t1 = []  
\textbf{shows} ack t = [init-state]  
\textbf{using assms}  
\textbf{proof (induction t)}  
\hspace{2em} \text{case} 0  
\hspace{4em} \text{from this show} \ ?\text{case}  
\hspace{5em} \text{by (simp add: tiTable-ack-init0)}  
\hspace{2em} \text{next}  
\hspace{2em} \text{case} (Suc t)  
\hspace{4em} \text{from Suc} \ \text{have} \ sg1: st t = hd (ack t)  
\hspace{5em} \text{by (simp add: tiTable-ack-st-hd)}  
\hspace{4em} \text{from Suc and sg1} \ \text{have} \ sg2:  
\hspace{5em} \hspace{2em} \text{(fin-inf-append [init-state] st) (Suc t) = init-state}  
\hspace{5em} \hspace{4em} \text{by (simp add: correct-fin-inf-append2)}  
\hspace{4em} \text{from Suc and sg1 and sg2 show} \ ?\text{case}  
\hspace{5em} \hspace{4em} \text{by (simp add: tiTable-SampleT-def)}  
\textbf{qed}

\begin{verbatim}
lemma tiTable-i1-3:  
\textbf{assumes} h1: tiTable-SampleT req x stop lose  
\hspace{2em} (fin-inf-append [init-state] st)  
\hspace{2em} b ack i1 vc st  
\hspace{2em} \text{and} h2: ts lose  
\hspace{2em} \text{and} h3: msg (Suc 0) x  
\hspace{2em} \text{and} h4: msg (Suc 0) stop  
\hspace{2em} \text{and} h5: \forall t1 \leq t. req t1 = []  
\textbf{shows} i1 t = []  
\textbf{proof }  
\hspace{2em} \text{from assms have} \ sg1: ack t = [init-state]  
\hspace{3em} \text{by (simp add: tiTable-ack-init)}  
\hspace{2em} \text{from assms have} \ sg2: st t = hd (ack t)  
\hspace{3em} \text{by (simp add: tiTable-ack-st-hd)}  
\hspace{2em} \text{from sg1 and sg2 have} \ sg3:  
\hspace{3em} \hspace{2em} \text{(fin-inf-append [init-state] st) (Suc t) = init-state}  
\hspace{3em} \hspace{3em} \text{by (simp add: correct-fin-inf-append2)}  
\hspace{3em} \text{from h1 and h2 have} \ sg4: lose t = [True] \lor lose t = [False]  
\hspace{3em} \hspace{3em} \text{by (simp add: ts-bool-True-False)}  
\hspace{3em} \text{from h1 and h3 have} \ sg5: x t = [] \lor x t = [sc-ack]  
\hspace{3em} \hspace{3em} \text{by (simp add: aType-lemma)}
\end{verbatim}

\textbf{113}
from \( h1 \) and \( h4 \) have \( \text{sg6:stop t} = [] \lor \text{stop t} = [\text{stop-vc}] \)

by (simp add: stopType-lemma)

show \( \text{thesis} \)

proof (cases fin-inf-append \( [\text{init-state}] \) st t)
  assume a1:fin-inf-append \( [\text{init-state}] \) st t = init-state
  from assms and \( \text{sg1 and sg2 and sg3 and a1} \) show \( \text{thesis} \)
  by (simp add: tiTable-SampleT-def)

next
  assume a2:fin-inf-append \( [\text{init-state}] \) st t = call
  show \( \text{thesis} \)
  proof (cases lose t = [True])
    assume a21:lose t = [True]
    from \( h1 \) and \( a2 \) and \( a21 \) show \( \text{thesis} \)
    by (simp add: tiTable-SampleT-def)
  next
    assume a22:lose t \( \neq \) [True]
    from this and \( h2 \) have a22a:lose t = [False] by (simp add: ts-bool-False)
    from \( h1 \) and \( a2 \) and \( a22a \) show \( \text{thesis} \)
    by (simp add: tiTable-SampleT-def)
  qed

next
  assume a3:fin-inf-append \( [\text{init-state}] \) st t = connection-ok
  show \( \text{thesis} \)
  proof (cases lose t = [True])
    assume a31:lose t = [True]
    from \( h1 \) and \( a3 \) and \( a31 \) show \( \text{thesis} \)
    by (simp add: tiTable-SampleT-def)
  next
    assume a32:lose t \( \neq \) [True]
    from this and \( h2 \) have a32a:lose t = [False] by (simp add: ts-bool-False)
    from \( h1 \) and \( a3 \) and \( a32a \) and \( a322 \) show \( \text{thesis} \)
    by (simp add: tiTable-SampleT-def)
  qed

next
  assume a4:fin-inf-append \( [\text{init-state}] \) st t = sending-data
  show \( \text{thesis} \)
  proof (cases lose t = [True])
    assume a41:lose t = [True]
    from \( h1 \) and \( a4 \) and \( a41 \) show \( \text{thesis} \)
    by (simp add: tiTable-SampleT-def)
  next
    assume a42:lose t \( \neq \) [True]
    from this and \( h2 \) have a42a:lose t = [False] by (simp add: ts-bool-False)
    show \( \text{thesis} \)
    proof (cases x t = [sc-ack])
      assume a421:x t = [sc-ack]
      from \( h1 \) and \( a4 \) and \( a42a \) and \( a421 \) and \( h2 \) show \( \text{thesis} \)
      by (simp add: tiTable-SampleT-def)
    next
      assume a422: x t \( \neq \) [sc-ack]
      from this and \( h3 \) have a422a:x t = [] by (simp add: aType-empty)
 qed
from h1 and a5 and a42a and a422a and h2 show \(?thesis
by (simp add: tiTable-SampleT-def)
qed
qed
next
assume a5:fin-inf-append \[init-state\] st t = voice-com
show \(?thesis
proof (cases stop t = \[stop-vc\])
  assume a51:stop t = \[stop-vc\]
  from h1 and a5 and a51 and h2 show \(?thesis by (simp add: tiTable-SampleT-def)
next
  assume a52:stop t \neq \[stop-vc\]
  from this and h4 have a52a:stop t = [] by (simp add: stopType-empty)
  show \(?thesis
  proof (cases lose t = \[True\])
    assume a521:lose t = \[True\]
    from h1 and a5 and a52a and a521 and h2 show \(?thesis
      by (simp add: tiTable-SampleT-def)
  next
    assume a522:lose t \neq \[True\]
    from this and h2 have a522a:lose t = \[False\] by (simp add: ts-bool-False)
    from h1 and h2 have a522a:lose t = \[False\] by (simp add: tiTable-SampleT-def)
qed
qed
qed

lemma tiTable-st-call-ok:
assumes h1:tiTable-SampleT req x stop lose
  (fin-inf-append \[init-state\] st)
  b ack it vc st
and h2:ts lose
and h3:\(\forall m \leq k.\) ack (Suc (Suc (t + m))) = \[connection-ok\]
and h4:st (Suc t) = call
shows st (Suc (Suc t)) = \[connection-ok\]
proof
  from h4 have sg1:
    (fin-inf-append \[init-state\] st) (Suc (Suc t)) = call
    by (simp add: correct-fin-inf-append2)
  from h1 and h2 have sg2:lose (Suc (Suc t)) = \[True\] \(\lor\) lose (Suc (Suc t)) = \[False\]
    by (simp add: ts-bool-True-False)
  show \(?thesis
  proof (cases lose (Suc (Suc t)) = \[False\])
    assume a1:lose (Suc (Suc t)) = \[False\]
    from h1 and a1 and sg1 show \(?thesis
      by (simp add: tiTable-SampleT-def)
  qed
next

\textbf{assume} \ a2\!:\text{lose} \ (\text{Suc} \ (\text{Suc} \ t)) \neq [\text{False}]
\textbf{from} \ \text{h3} \ \textbf{have} \ \text{sg3}:\text{ack} \ (\text{Suc} \ (\text{Suc} \ t)) = [\text{connection-ok}] \ \textbf{by} \ \text{auto}
\textbf{from} \ \text{h1} \ \text{and} \ \text{a2} \ \text{and} \ \text{sg1} \ \text{and} \ \text{sg2} \ \text{and} \ \text{sg3} \ \textbf{show} \ \text{\textit{thesis}}
\textbf{by} \ (\text{simp add: tiTable-SampleT-def})
\textbf{qed}

\text{qed}

\textbf{lemma} \ tiTable-i1-4b:
\textbf{assumes} \ \text{h1}:\text{tiTable-SampleT req x stop lose}
\ (\text{fin-inf-append} \ [\text{init-state}] \ st)
\ b \ \text{ack i1 vc st}
\ \text{and} \ \text{h2}:ts \ \text{lose}
\ \text{and} \ \text{h3}:\text{msg} \ (\text{Suc} \ 0) \ x
\ \text{and} \ \text{h4}:\text{msg} \ (\text{Suc} \ 0) \ \text{stop}
\ \text{and} \ \text{h5}:\forall \ t1 \leq t. \ \text{req t1} = []
\ \text{and} \ \text{h6}:\text{req} \ (\text{Suc} \ t) = [\text{init}]
\ \text{and} \ \text{h7}:\forall \ m < k + 3. \ \text{req} \ (t + m) \neq [\text{send}]
\ \text{and} \ \text{h7}:\forall \ m \leq k. \ \text{ack} \ (\text{Suc} \ (\text{Suc} \ (t + m))) = [\text{connection-ok}]
\ \text{and} \ \text{h8}:\forall \ j \leq k + 3. \ \text{lose} \ (t + j) = [\text{False}]
\ \text{and} \ \text{h9}:t2 < (t + 3 + k)
\ \text{shows} \ i1 \ t2 = []
\ \textbf{proof} \ (\text{cases} \ t2 \leq t)
\ \textbf{assume} \ a1: \text{t2} \leq t
\ \textbf{from} \ \text{assms} \ \text{and} \ \text{a1} \ \textbf{show} \ \text{\textit{thesis}} \ \textbf{by} \ (\text{simp add: tiTable-i1-3})
\ \textbf{next}
\ \textbf{assume} \ a2:\neg \ t2 \leq t
\ \textbf{from} \ \text{assms} \ \textbf{have} \ \text{sg1}:\text{ack} \ t = [\text{init-state}] \ \textbf{by} \ (\text{simp add: tiTable-ack-init})
\ \textbf{from} \ \text{assms} \ \textbf{have} \ \text{sg2}:st \ t = \text{hd} \ (\text{ack} \ t) \ \textbf{by} \ (\text{simp add: tiTable-ack-st-hd})
\ \textbf{from} \ \text{sg1} \ \text{and} \ \text{sg2} \ \textbf{have} \ \text{sg3}:
\ (\text{fin-inf-append} \ [\text{init-state}] \ st) \ (\text{Suc} \ t) = \text{init-state}
\ \textbf{by} \ (\text{simp add: correct-fin-inf-append2})
\ \textbf{from} \ \text{assms} \ \text{and} \ \text{sg3} \ \textbf{have} \ \text{sg4}:st \ (\text{Suc} \ t) = \text{call}
\ \textbf{by} \ (\text{simp add: tiTable-SampleT-def})
\ \textbf{show} \ \text{\textit{thesis}}
\ \textbf{proof} \ (\text{cases} \ t2 = \text{Suc} \ t)
\ \textbf{assume} \ a3:t2 = \text{Suc} \ t
\ \textbf{from} \ \text{assms} \ \text{and} \ \text{sg3} \ \text{and} \ \text{a3} \ \textbf{show} \ \text{\textit{thesis}}
\ \textbf{by} \ (\text{simp add: tiTable-SampleT-def})
\ \textbf{next}
\ \textbf{assume} \ a4:t2 \neq \text{Suc} \ t
\ \textbf{from} \ \text{assms} \ \text{and} \ \text{sg4} \ \text{and} \ \text{a4} \ \text{and} \ \text{a2} \ \textbf{have} \ \text{sg7}:st \ (\text{Suc} \ (\text{Suc} \ t)) = \text{connection-ok}
\ \textbf{by} \ (\text{simp add: tiTable-st-call-ok})
\ \textbf{from} \ \text{assms} \ \textbf{have} \ \text{sg8}:\text{ack} \ (\text{Suc} \ (\text{Suc} \ t)) = \text{[st} \ (\text{Suc} \ (\text{Suc} \ t))]\]
\ \textbf{by} \ (\text{simp add: tiTable-ack-st})
\ \textbf{show} \ \text{\textit{thesis}}
\ \textbf{proof} \ (\text{cases} \ t2 = \text{Suc} \ (\text{Suc} \ t))
\ \textbf{assume} \ a5:t2 = \text{Suc} \ (\text{Suc} \ t)

116
from h7 and h9 and a5 have sg9:ack t2 = [connection-ok] by auto
from assms and sg9 show ?thesis by (simp add: tiTable-i1-1)
next
assume a6:t2 ≠ Suc (Suc t)
from a6 and a4 and a2 have sg10:Suc (Suc t) < t2 by arith
from h7 and h9 and sg10 have sg11:ack t2 = [connection-ok]
by (simp add: aux-ack-t2)
from assms and a6 and sg7 and sg8 and sg11 show ?thesis
by (simp add: tiTable-i1-1)
qed
qed
qed

lemma tiTable-i1-4:
assumes h1:tiTable-SampleT req a1 stop lose
(fin-inf-append [init-state] st)
  b ack i1 vc st
and h2:ts lose
and h3:msg (Suc 0) a1
and h4:msg (Suc 0) stop
and h5:∀ t1 ≤ t. req t1 = []
and h6:req (Suc t) = [init]
and h7:∀ m < k + 3. req (t + m) ≠ [send]
and h8:∀ m ≤ k. ack (Suc (Suc (t + m))) = [connection-ok]
and h9:∀ j ≤ k + 3. lose (t + j) = [False]
shows ∀ t2 < (t + 3 + k). i1 t2 = []
using assms by (simp add: tiTable-i1-4b)

lemma tiTable-ack-ok:
assumes h1:∀ j ≤ d + 2. lose (t + j) = [False]
and h2:ts lose
and h4:msg (Suc 0) stop
and h5:msg (Suc 0) a1
and h6:req (Suc t) ≠ [send]
and h7:ack t = [connection-ok]
and h8:tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st) b ack i1 vc st
shows ack (Suc t) = [connection-ok]

proof –
from h8 and h2 and h5 and h4 have sg1:st t = hd (ack t)
  by (simp add: tiTable-ack-st-hd)
from sg1 and h7 have sg2:
  (fin-inf-append [init-state] st) (Suc t) = connection-ok
  by (simp add: correct-fin-inf-append2)
  have sg3a:Suc 0 ≤ d + 2 by arith
from h1 and sg3a have sg3:lose (t + Suc 0) = [False] by auto
from sg2 and sg3 and h6 and h8 show ?thesis
by \((\text{simp add: tiTable-SampleT-def})\)

\[\text{qed}\]

lemma \textit{Gateway-L7a}:

assumes \(h1\): \textit{Gateway req dt a stop lose d ack i vc}

and \(h2\): \textit{msg (Suc 0) a}

and \(h3\): \textit{msg (Suc 0) stop}

and \(h4\): \textit{msg (Suc 0) req}

and \(h5\): \textit{ts lose}

and \(h6\): \(\forall j \leq d + 2. \text{lose} (t + j) = [False]\)

and \(h7\): \textit{req (Suc t) \neq [send]}

and \(h8\): \textit{ack (t) = [connection-ok]}

shows \textit{ack (Suc t) = [connection-ok]}

proof –

from \(h1\) and \(h3\) and \(h4\) and \(h7\) obtain \(i1 i2 a1 a2\) where

ah1: \textit{Sample req dt a1 stop lose ack i1 vc and}

ah2: \textit{Delay a2 i1 d a1 i2 and}

ah3: \textit{Loss lose a i2 a2 i}

by \((\text{simp add: Gateway-def, auto})\)

from \(ah2\) and \(ah3\) and \(h2\) have \(sg1\): \textit{msg (Suc 0) a1}

by \((\text{simp add: Loss-Delay-msg-a})\)

from \(ah1\) and \(sg1\) and \(h3\) and \(h4\) obtain \(st buffer\) where

ah4: \textit{Sample-L req dt a1 stop lose (fin-inf-append \([\text{init-state}]\) st)}

\((\text{fin-inf-append \(\text{buffer}\)}\))

\(ack i1 vc st buffer\)

by \((\text{simp add: Sample-L-def, auto})\)

from \(ah4\) have \(sg2\):

\textit{tiTable-SampleT req a1 stop lose (fin-inf-append \([\text{init-state}]\) st)}

\((\text{fin-inf-append \(\text{buffer}\)}\))

\(ack i1 vc st buffer\)

by \((\text{simp add: Sample-L-def})\)

from \(h6\) and \(h5\) and \(h3\) and \(sg1\) and \(h7\) and \(h8\) and \(sg2\) show \(?\text{thesis}\)

by \((\text{simp add: tiTable-ack-ok})\)

\[\text{qed}\]

lemma \textit{Sample-L-buffer}:

assumes \(h1\):

\textit{Sample-L req dt a1 stop lose (fin-inf-append \([\text{init-state}]\) st)}

\((\text{fin-inf-append \(\text{buffer}\)}\))

\(ack i1 vc st buffer\)

shows \textit{buffer t = inf-last-ti dt t}

proof –

from \(h1\) have \(sg1\):

\(\forall t. \text{buffer t} =\)

\(\text{if dt t} = [] \text{ then fin-inf-append \(\text{buffer}\) else dt t}\)

by \((\text{simp add: Sample-L-def})\)

from \(sg1\) show \(?\text{thesis}\)

\[\text{118}\]
proof (induct t)

  case 0
    from this show ?case
    by (simp add: fin-inf-append-def)

next
  fix t
  case (Suc t)
    from this show ?case
    proof (cases dt t = [])
      assume a1:dt t = []
      from a1 and Suc show ?thesis
      by (simp add: correct-fin-inf-append1)
    next
      assume a2:dt t ≠ []
      from a2 and Suc show ?thesis
      by (simp add: correct-fin-inf-append1)
    qed
  qed

lemma tiTable-SampleT-i1-buffer:
assumes h1:ack t = [connection-ok]
  and h2:req (Suc t) = [send]
  and h3:∀ k≤Suc d. lose (t + k) = [False]
  and h4:buffer t = inf-last-ti dt t
  and h6:tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st) (fin-inf-append [[]] buffer) ack i1 vc st
    and h7:st t = hd (ack t)
    and h8:fin-inf-append [init-state] st (Suc t) = connection-ok
shows i1 (Suc t) = inf-last-ti dt t
proof –
  have sg1:Suc 0 ≤ Suc d by arith
  from h3 and sg1 have sg2:lose (Suc t) = [False] by auto
  from h6 have
    fin-inf-append [init-state] st (Suc t) = connection-ok ∧
    req (Suc t) = [send] ∧
    lose (Suc t) = [False] −→
    ack (Suc t) = [sending-data] ∧
    i1 (Suc t) = (fin-inf-append [[]] buffer) (Suc t) ∧
    vc (Suc t) = [] ∧ st (Suc t) = sending-data
    by (simp add: tiTable-SampleT-def)
  from this and h8 and h2 and sg2 have
    i1 (Suc t) = (fin-inf-append [[]] buffer) (Suc t) by simp
  from this and h4 show ?thesis by (simp add: correct-fin-inf-append1)
  qed

qed
lemma Sample-L-i1-buffer:
assumes h1: msg (Suc 0) req 
  and h2: msg (Suc 0) a 
  and h3: msg (Suc 0) stop 
  and h4: msg (Suc 0) a1 
  and h5: ts lose 
  and h6: ack t = [connection-ok] 
and h7: req (Suc t) = [send] 
and h8: \( \forall k \leq \text{Suc} \ d. \text{lose} (t + k) = [\text{False}] \) 
and h9: Sample-L req dt a1 stop lose (fin-inf-append [init-state] st) (fin-inf-append [] buffer) ack i1 vc st buffer 
shows i1 (Suc t) = buffer t 
proof – 
from h9 have sg1: buffer t = inf-last-ti dt t 
  by (simp add: Sample-L-buffer) 
from h9 have sg2: 
  \( \forall t. \text{buffer} t = (\text{if dt} t = [] \text{then fin-inf-append} [\text{[]}] \text{buffer} t \text{else dt} t) \) 
  by (simp add: Sample-L-def) 
from h9 have sg3: 
  tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st) (fin-inf-append [] buffer) ack i1 vc st 
  by (simp add: Sample-L-def) 
from sg3 and h5 and h4 and h3 have sg4: st t = hd (ack t) 
  by (simp add: tiTable-ack-st-hd) 
from h6 and sg4 have sg5: 
  (fin-inf-append [init-state] st) (Suc t) = connection-ok 
  by (simp add: correct-fin-inf-append1) 
from h6 and h7 and h8 and sg1 and sg3 and sg4 and sg5 have sg6: 
  i1 (Suc t) = inf-last-ti dt t 
  by (simp add: tiTable-SampleT-i1-buffer) 
from this and sg1 show ?thesis by simp 
qed 

lemma tiTable-SampleT-sending-data: 
assumes h1: tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st) (fin-inf-append [] buffer) ack i1 vc st 
  and h2: \( \forall j \leq 2 \ast d. \text{lose} (t + j) = [\text{False}] \) 
and h3: \( \forall t4 \leq t + d + d. \text{a1} t4 = [] \) 
and h4: ack (t + x) = [sending-data] 
and h5: fin-inf-append [init-state] st (Suc (t + x)) = sending-data 
and h6: Suc (t + x) \( \leq 2 \ast d + t \) 
shows ack (Suc (t + x)) = [sending-data] 
proof – 
from h6 have Suc x \( \leq 2 \ast d \) by arith 
from this and h2 have sg1: lose (t + Suc x) = [False] by auto
from \( h6 \) have \( \text{Suc} \ (t + x) \leq t + d + d \) by \( \text{arith} \)
from \( \text{this} \) and \( h3 \) have \( \text{sg2:a1} \ (\text{Suc} \ (t + x)) = \[] \) by \( \text{auto} \)
from \( h1 \) and \( \text{sg1} \) and \( \text{sg2} \) and \( h5 \) show \( ?\text{thesis} \)
by (simp add: \text{tiTable-SampleT-def})

qed

lemma Sample-sending-data:
assumes \( h1: \text{msg} \ (\text{Suc} \ 0) \) \( \text{stop} \)
  and \( h2: \text{ts} \) \( \text{lose} \)
  and \( h3: \text{msg} \ (\text{Suc} \ 0) \) \( \text{req} \)
  and \( h4: \text{msg} \ (\text{Suc} \ 0) \) \( \text{a1} \)
  and \( h5: \forall j \leq 2 \ast d. \text{lose} \ (t + j) = [\text{False}] \)
  and \( h6: \text{ack} \ t = [\text{sending-data}] \)
  and \( h8: x \leq d + d \)
  and \( h9: \forall t4 \leq t + d + d. \text{a1} \ t4 = [] \)
shows \( \text{ack} \ (t + x) = [\text{sending-data}] \)
using \( \text{assms} \)

proof
  --
from \( h1 \) and \( h2 \) and \( h3 \) and \( h4 \) and \( h7 \) obtain \( \text{st buffer} \) where \( \text{a1}: \)
  \( \text{Sample-L req dt a1 stop lose} \ (\text{fin-inf-append} \ [\text{init-state}] \text{ st}) \)
  \( \{\text{fin-inf-append} \ [\text{buffer}] \text{ ack} \)
  \( \text{if vt st buffer} \)
  by (simp add: \text{Sample-def, auto})

from \( \text{a1} \) have \( \text{sg1}: \)
  \( \text{tiTable-SampleT req a1 stop lose} \ (\text{fin-inf-append} \ [\text{init-state}] \text{ st}) \)
  \( \{\text{fin-inf-append} \ [\text{buffer}] \text{ ack} \)
  \( \text{if vt st buffer} \)
  by (simp add: \text{Sample-L-def})

from \( \text{a1} \) have \( \text{sg2}: \)
  \( \forall t. \text{buffer t} = (\text{if dt t} = [] \text{ then fin-inf-append} \ [\text{buffer}] \text{ t else dt t}) \)
  by (simp add: \text{Sample-L-def})

from \( h1 \) and \( h2 \) and \( h3 \) and \( h5 \) and \( h8 \) and \( \text{sg1} \) and \( \text{sg2} \) show \( ?\text{thesis} \)
proof (induct \( x \))
  case 0
  from this show \( ?\text{case} \) by simp
next
  fix \( x \)
  case (Suc \( x \))
  from this have \( \text{sg2:st} \ (t + x) = \text{hd} \ (\text{ack} \ (t + x)) \)
  by (simp add: \text{tiTable-ack-st-hd})
  from \( \text{Suc} \) have \( \text{sg4:x} \leq d + d \) by \( \text{arith} \)
  from \( \text{Suc} \) and \( \text{sg3} \) and \( \text{sg4} \) have \( \text{sg5}: \)
  \( \{\text{fin-inf-append} \ [\text{init-state}] \text{ st} \} \ (\text{Suc} \ (t + x)) = \text{sending-data} \)
  by (simp add: \text{fin-inf-append-def})
  from \( \text{Suc} \) have \( \text{sg6:Suc} \ (t + x) \leq 2 \ast d + t \) by \( \text{simp} \)
  from \( \text{Suc} \) have \( \text{sg7:ack} \ (t + x) = [\text{sending-data}] \) by \( \text{simp} \)
  from \( \text{sg1} \) and \( h5 \) and \( h9 \) and \( \text{sg7} \) and \( \text{sg5} \) and \( \text{sg6} \) have \( \text{sg7}: \)

121
ack (Suc (t + x)) = [sending-data]
by (simp add: tiTable-SampleT-sending-data)
from this show ?case by simp
qed

15.6 Properties of the ServiceCenter component

lemma ServiceCenter-a-l:
  assumes h1: ServiceCenter i a
  shows length (a t) ≤ (Suc 0)
proof (cases t)
  case 0
  from this and h1 show ?thesis by (simp add: ServiceCenter-def)
next
  fix m assume Suc: t = Suc m
  from this and h1 show ?thesis by (simp add: ServiceCenter-def)
qed

lemma ServiceCenter-a-msg:
  assumes h1: ServiceCenter i a
  shows msg (Suc 0) a
using assms by (simp add: msg-def ServiceCenter-a-l)

lemma ServiceCenter-L1:
  assumes h1: ∀ t2 < x. i t2 = []
  and h2: ServiceCenter i a
  and h3: t ≤ x
  shows a t = []
using assms
proof (induct t)
  case 0
  from this show ?case by (simp add: ServiceCenter-def)
next
  case (Suc t)
  from this show ?case by (simp add: ServiceCenter-def)
qed

lemma ServiceCenter-L2:
  assumes h1: ∀ t2 < x. i t2 = []
  and h2: ServiceCenter i a
  shows ∀ t3 ≤ x. a t3 = []
using assms by (clarify, simp add: ServiceCenter-L1)

15.7 General properties of stream values

lemma streamValue1:
  assumes h1: ∀ j ≤ D + (z::nat). str (t + j) = x
  and h2: j ≤ D
  shows str (t + j + z) = x
proof –
  from \( h_2 \) have \( sg_1: j + z \leq D + z \) by arith
  have \( sg_2: t + j + z = t + (j + z) \) by arith
  from \( h_1 \) and \( sg_1 \) and \( sg_2 \) show \( \text{thesis} \) by (simp (no-asmp-simp))
qed

lemma streamValue2:
  assumes \( h_1: \forall j \leq D. \, \text{str} (t + j) = x \)
  shows \( \forall j \leq D. \, \text{str} (t + j + z) = x \)
  using \( \text{assms} \) by (clarify, simp add: streamValue1)

lemma streamValue3:
  assumes \( h_1: \forall j \leq D. \, \text{str} (t + j + (\text{Suc} \, y)) = x \)
  and \( h_2: j \leq D \)
  and \( h_3: \text{str} (t + y) = x \)
  shows \( \text{str} (t + j + y) = x \)
  using \( \text{assms} \)
  proof (induct \( j \))
    case 0
    from \( h_3 \) show \( \text{case} \) by simp
  next
    case (Suc \( j \))
    from this show \( \text{case} \) by auto
  qed

lemma streamValue4:
  assumes \( h_1: \forall j \leq D. \, \text{str} (t + j + (\text{Suc} \, y)) = x \)
  and \( h_3: \text{str} (t + y) = x \)
  shows \( \forall j \leq D. \, \text{str} (t + j + y) = x \)
  using \( \text{assms} \) by (clarify, simp add: streamValue3)

lemma streamValue5:
  assumes \( h_1: \forall j \leq D. \, \text{str} (t + j + ((i::nat) + k)) = x \)
  and \( h_2: j \leq D \)
  shows \( \text{str} (t + i + k + j) = x \)
  proof –
    have \( sg_1: t + i + k + j = t + j + (i + k) \) by arith
    from \( \text{assms} \) and \( sg_1 \) show \( \text{thesis} \) by (simp (no-asmp-simp))
  qed

lemma streamValue6:
  assumes \( h_1: \forall j \leq D. \, \text{str} (t + j + ((i::nat) + k)) = x \)
  shows \( \forall j \leq D. \, \text{str} (t + (i::nat) + k + j) = x \)
  using \( \text{assms} \) by (clarify, simp add: streamValue5)

lemma streamValue7:
  assumes \( h_1: \forall j \leq D. \, \text{str} (t + i + k + d + \text{Suc} \, j) = x \)
  and \( h_2: \text{str} (t + i + k + d) = x \)

123
and \( h3 : j \leq \text{Suc} \ d \)
shows \( \text{str} \ (t + i + k + d + j) = x \)
proof
- from \( h1 \) have \( \text{sg1} : \text{str} \ (t + i + k + d + \text{Suc} \ d) = x \)
  by (simp (no-asm-simp), simp)
from \( \text{assms} \) show \( \text{?thesis} \)
  proof (cases \( j = \text{Suc} \ d \))
    assume \( a1 : j = \text{Suc} \ d \)
    from \( a1 \) and \( \text{sg1} \) show \( \text{?thesis} \) by simp
  next
    assume \( a2 : j \neq \text{Suc} \ d \)
    from \( a2 \) and \( h3 \) have \( \text{sg2} : j \leq d \) by auto
    from \( \text{assms} \) and \( \text{sg2} \) show \( \text{?thesis} \)
      proof (cases \( j > 0 \))
        assume \( a3 : 0 < j \)
        from \( a3 \) and \( h3 \) have \( \text{sg3} : j - (\text{Suc} \ (j - (\text{Suc} \ \text{nat}))) = j \) by arith
        from \( \text{sg3} \) and \( h1 \) and \( \text{sg4} \) have \( \text{sg5} : \text{str} \ (t + i + k + d + j) = x \) by auto
        from \( \text{sg5} \) show \( \text{?thesis} \) by simp
      next
        assume \( a4 : \neg 0 < j \)
        from \( a4 \) have \( \text{sg6} : j = 0 \) by simp
        from \( h2 \) and \( \text{sg6} \) show \( \text{?thesis} \) by simp
      qed
qed

lemma \( \text{streamValue8} \):
  assumes \( h1 : \forall j \leq d. \ \text{str} \ (t + i + k + d + \text{Suc} \ j) = x \)
  and \( h2 : \text{str} \ (t + i + k + d) = x \)
shows \( \forall j \leq \text{Suc} \ d. \ \text{str} \ (t + i + k + d + j) = x \)
using \( \text{assms} \) by (clarify, simp add: \( \text{streamValue7} \))

lemma \( \text{arith-streamValue9aux} \):
\begin{align*}
\text{Suc} \ (t + (j + d) + (i + k)) & = \text{Suc} \ (t + i + k + d + j) \\
\text{by arith}
\end{align*}

lemma \( \text{streamValue9} \):
  assumes \( h1 : \forall j \leq 2 \ast d. \ \text{str} \ (t + j + \text{Suc} \ (i + k)) = x \)
  and \( h2 : j \leq d \)
shows \( \text{str} \ (t + i + k + d + \text{Suc} \ j) = x \)
proof
- from \( h2 \) have \( (j + d) \leq 2 \ast d \) by arith
  from \( h1 \) and \( \text{this} \) have \( \text{str} \ (t + (j + d) + \text{Suc} \ (i + k)) = x \) by auto
  from \( \text{this} \) show \( \text{?thesis} \) by (simp add: \( \text{arith-streamValue9aux} \))
qed
lemma streamValue10:  
  assumes h1:∀ j≤ 2 * d. lose (t + j + Suc (i + k)) = x  
  and h2: j≤ Suc d  
  shows lose (t + 2 + k + j) = x  
proof  
  from h2 have sg1: 2 + k + j ≤ 2 * d + (4 + k) by arith  
  have sg2: Suc (Suc (t + k + j)) = Suc (Suc (t + (k + j))) by arith  
  from sg1 and h1 have lose (t + (2 + k + j)) = x by blast  
  from this and sg2 show ?thesis by (simp add: arith-sum2) 
qed

lemma streamValue12:  
  assumes h1:∀ j≤ 2 * d + (4 + k). lose (t + j) = x  
  shows ∀ j≤ Suc d. lose (t + 2 + k + j) = x  
using assms  
  apply clarify by (rule streamValue11, auto)

lemma streamValue43:  
  assumes h1:∀ j≤ 2 * d + ((4::nat) + k). lose (t + j) = [False]  
  shows ∀ j≤ 2 * d. lose ((t + (3::nat) + k) + j) = [False]  
proof  
  from h1 have sg1:∀ j≤ 2 * d. lose (t + j + (4 + k)) = [False]  
    by (simp add: streamValue2)  
  have sg2: Suc (3 + k) = (4 + k) by arith  
  from sg1 and sg2 have sg3:∀ j≤ 2 * d. lose (t + j + Suc (3 + k)) = [False]  
    by (simp (no_asm-simp))  
  from h1 have sg4: lose (t + (3 + k)) = [False] by auto  
  from sg3 and sg4 have sg5:∀ j≤ 2 * d. lose (t + j + (3 + k)) = [False]  
    by (rule streamValue4)  
  from sg5 show ?thesis by (rule streamValue6) 
qed

end
theory Gateway-proof
imports Gateway-proof-aux
begin

15.8 Properties of the Gateway

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

proof –
from h1 obtain i1 i2 x y
where a1: Sample req dt x stop lose ack i1 vc
and a2: Delay y i1 d x i2
and a3: Loss lose a i2 y i
by (simp only: Gateway-def, auto)
from a2 and a3 and h3 have sg1: msg (Suc 0) x
by (simp add: Loss-Delay-msg-a)
from a1 and h2 and h4 and sg1 obtain st buffer where a4:
tiTable-SampleT req x stop lose
(fin-inf-append [init-state] st) (fin-inf-append [] buffer) ack i1 vc st
by (simp add: Sample-def Sample-L-def, auto)
from a4 and h5 and sg1 and h4 have sg2: st t = hd (ack t)
by (simp add: tiTable-ack-st-hd)
from h6 and sg1 and sg2 and h4 have sg3:
(fin-inf-append [init-state] st) (Suc t) = init-state
by (simp add: correct-fin-inf-append1)
from a4 and h7 and sg3 have sg4: st (Suc t) = call
by (simp add: tiTable-SampleT-def)
from sg4 have sg5: (fin-inf-append [init-state] st) (Suc (Suc t)) = call
by (simp add: correct-fin-inf-append1)
from a4 and sg5 and assms show thesis
by (simp add: tiTable-SampleT-def)

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

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

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

from sg5 and sg4 and h7 have sg6: x (Suc (t + d)) = [sc-ack] by simp

from h8 have sg7: lose (Suc (t + d)) = [False] by auto

from sg6 and a4 and h2 and sg1 and h4 and h5 and sg7 and sg3 show thesis
  by (simp add: tiTable-SampleT-def)

qed

lemma Gateway-L5:
  assumes h1: Gateway req dt a stop lose d ack i vc
  and h2: msg (Suc 0) req
  and h3: msg (Suc 0) a
  and h4: msg (Suc 0) stop
  and h5: ts lose
  and h6: ack (t + d) = [sending-data]
  and h7: \( \forall j \leq Suc d \). a (t+j) = []
  and h8:\( \forall k \leq (d + d) \). lose (t + k) = [False]

  shows j \leq d \longrightarrow ack (t+d+j) = [sending-data]

proof
  from h1 obtain i1 i2 x y
    where a1: Sample req dt x stop lose d ack i1 vc
    and a2: Delay y i1 d x i2
    and a3: Loss lose a i2 y i
    by (simp only: Gateway-def, auto)

  from a2 and a3 and h3 have sg1: msg (Suc 0) x
    by (simp add: Loss-Delay-msg-a)

  from a1 and h2 and h4 and sg1 obtain st buffer where a4:
    tiTable-SampleT req x stop lose
    (fin-inf-append [init-state] st) (fin-inf-append [] buffer) ack i1 vc st
    by (simp add: Sample-def Sample-L-def, auto)

  from assms and a2 and a3 and sg1 and a4 show thesis

proof (induct j)
  case 0
  from 0 show \( \text{thesis} \) by simp

next
  case (Suc j)
  from Suc show \( \text{thesis} \)

proof (cases Suc j \leq d)
  assume \( \neg \) Suc j \leq d from this show \( \text{thesis} \) by simp

next
  assume a0: Suc j \leq d

  from a0 have sg2: d + Suc j \leq d + d by arith
  from sg2 have sg3: Suc (d + j) \leq d + d by arith

  from a4 and h2 and sg1 and h4 and h5 have sg4:
    st (t+d+j) = hd (ack (t+d+j))
    by (simp add: tiTable-ack-st-hd)

  from Suc and a0 and sg4 have sg5:
    (fin-inf-append [init-state] st) (Suc (t+d+j)) = sending-data
by (simp add: correct-fin-inf-append1)
from \( h_7 \) and \( a_0 \) have \( sg_6 : \forall j \leq d. \ a \ (t + Suc \ j) = [] \) by auto
from \( sg_6 \) and \( a_3 \) and \( a_0 \) and \( h_5 \) have \( sg_7 : y \ (t + (Suc \ j)) = [] \)
  by (rule Loss-L5Suc)
from \( sg_8 \) and \( a_2 \) have \( sg_8 \ (Suc \ (t + d + j)) = [] \) by simp
have \( sg_9 : Suc \ (t + d + j) = Suc \ (t + (d + j)) \) by arith
from \( a_4 \) have \( sg_{10} : \)
  \( \text{fin-inf-append} \ [\text{init-state}] \ st \ (Suc \ (t + d + j)) = \text{sending-data} \land \)
  \( x \ (Suc \ (t + d + j)) = [] \land \)
  \( \text{lose} \ (Suc \ (t + d + j)) = \false \rightarrow \)
  \( \text{ack} \ (Suc \ (t + d + j)) = [\text{sending-data}] \)
  by (simp add: tiTable-SampleT-def)
from \( h_8 \) and \( sg_3 \) have \( sg_{11} : \text{lose} \ (Suc \ (d + j)) = \false \) by blast
have \( sg_{12} : Suc \ (t + d + j) = t + Suc \ (d + j) \) by arith
from \( sg_{12} \) and \( sg_{11} \) have \( sg_{13} : \text{lose} \ (Suc \ (t + d + j)) = \false \)
  by (simp (no-asn-simp), simp)
from \( sg_{10} \) and \( sg_5 \) and \( sg_8a \) and \( sg_{13} \) show \( ?\text{thesis} \) by simp
qed
qed

lemma Gateway-L6-induction:
assumes \( h_1 : \text{msg} \ (Suc \ 0) \ \text{req} \)
  and \( h_2 : \text{msg} \ (Suc \ 0) \ x \)
  and \( h_3 : \text{msg} \ (Suc \ 0) \ \text{stop} \)
  and \( h_4 : ts \ \text{lose} \)
  and \( h_5 : \forall j \leq k. \ \text{lose} \ (t + j) = \false \)
  and \( h_6 : \forall m \leq k. \ \text{req} \ (t + m) \neq [\text{send}] \)
  and \( h_7 : \text{ack} \ t = [\text{connection-ok}] \)
  and \( h_8 : \text{Sample} \ \text{req} \ dt \ x_1 \ \text{stop} \ \text{lose} \ \text{ack} \ i_1 \ \text{vc} \)
  and \( h_9 : \text{Delay} \ x_2 \ i_1 \ d \ x_1 \ i_2 \)
  and \( h_{10} : \text{Loss} \ \text{lose} \ x \ i_2 \ x_2 \ i \)
  and \( h_{11} : m \leq k \)
shows \( \text{ack} \ (t + m) = [\text{connection-ok}] \)
using assms
proof (induct \( m \))
case \( 0 \) from this show \( ?\text{case} \) by simp
next
case \( \text{Suc} \ m \)
from \( \text{Suc} \) have \( sg_1 : \text{msg} \ (Suc \ 0) \ x_1 \) by (simp add: Loss-Delay-msg-a)
from \( \text{Suc} \) and \( sg_1 \) obtain \( st \ \text{buffer} \ \text{where} \)
  \( a_1 : \text{tiTable-SampleT} \ \text{req} \ x_1 \ \text{stop} \ \text{lose} \ \text{(fin-inf-append} \ [\text{init-state}] \ st) \)
  \( \text{(fin-inf-append} \ [] \ \text{buffer}) \ \text{ack} \ i_1 \ \text{vc} \ st \ \text{and} \)
  \( a_2 : \forall t. \ \text{buffer} \ t = (\text{if} \ dt \ t = [] \ \text{then} \ \text{fin-inf-append} \ [] \ \text{buffer} \ t \ \text{else} \ dt \ t) \)
  by (simp add: Sample-def Sample-L-def, auto)
from \( a_1 \) and \( sg_1 \) and \( h_3 \) and \( h_4 \) have \( sg_2 : st \ (t + m) = \ \text{hd} \ (\text{ack} \ (t + m)) \)
  by (simp add: tiTable-ack-st-hd)
from Suc have sg3:ack (t + m) = [connection-ok] by simp

from a1 and sg2 and sg3 have sg4:
  (fin-inf-append [init-state] st) (Suc (t + m)) = connection-ok
  by (simp add: fin-inf-append-def)

from Suc have sg5:Suc m ≤ k by simp

from sg5 and h5 have sg6:lose (Suc (t + m)) = [False] by auto

from a1 and sg3 and sg4 and sg5 and sg6 and sg7 show ?case
  by (simp add: tiTable-SampleT-def)
qed

lemma Gateway-L6:
  assumes h1:Gateway req dt a stop lose d ack i vc
  and h2:∀ m ≤ k. req (t + m) ≠ [send]
  and h3:∀ j ≤ k. lose (t + j) = [False]
  and h4:ack t = [connection-ok]
  and h5:msg (Suc 0) req
  and h6:msg (Suc 0) stop
  and h7:msg (Suc 0) a
  and h8:ts lose
  shows ∀ m ≤ k. ack (t + m) = [connection-ok]
  using assms by (simp add: Gateway-def, clarify, simp add: Gateway-L6-induction)

lemma Gateway-L6a:
  assumes h1:Gateway req dt a stop lose d ack i vc
  and h2:∀ m ≤ k. req (t + 2 + m) ≠ [send]
  and h3:∀ j ≤ k. lose (t + 2 + j) = [False]
  and h4:ack (t + 2) = [connection-ok]
  and h5:msg (Suc 0) req
  and h6:msg (Suc 0) stop
  and h7:msg (Suc 0) a
  and h8:ts lose
  shows ∀ m ≤ k. ack (t + 2 + m) = [connection-ok]
  using assms by (rule Gateway-L6)

lemma aux-k3req:
  assumes h1:∀ m < k + 3. req (t + m) ≠ [send] and h2:m ≤ k
  shows req (Suc (Suc (t + m))) ≠ [send]
  proof –
    from h2 have m + 2 < k + 3 by arith
    from h1 and this have req (t + (m + 2)) ≠ [send] by blast
  from this show ?thesis by simp
qed

lemma aux3lose:
  assumes h1:∀ j ≤ k + d + 3. lose (t + j) = [False]
  and h2:j ≤ k
  shows lose (Suc (Suc (t + j))) = [False]
proof –

from $h_2$ have $(j + 2) \leq k + d + 3$ by arith
from $h_1$ and this have lose $(t + (j + 2)) = [\text{False}]$ by blast
from this show ?thesis by simp

qed

lemma $\text{Gateway-L7}$:

assumes $h_1$: $\text{Gateway req dt a stop lose d ack i vc}$
and $h_2$: $\text{ts lose}$
and $h_3$: $\text{msg (Suc 0) a}$
and $h_4$: $\text{msg (Suc 0) stop}$
and $h_5$: $\text{msg (Suc 0) req}$
and $h_6$: $\text{req (Suc t) = [init]}$
and $h_7$: $\forall m < (k + 3). \text{req (t + m) ≠ [send]}$
and $h_8$: $\text{req (t + 3 + k) = [send]}$
and $h_9$: $\text{ack t = [init-state]}$
and $h_{10}$: $\forall j \leq k + d + 3. \text{lose (t + j) = [False]}$
and $h_{11}$: $\forall t \leq t. \text{req t1 = []}$

shows $\forall t_2 < (t + 3 + k + d). i t_2 = []$

proof –

have $\text{Suc 0} \leq k + d + 3$ by arith
from $h_{10}$ and this have lose $(t + \text{Suc 0}) = [\text{False}]$ by blast
from this have $\text{sg1:lose (Suc t) = [False]}$ by simp
have $\text{Suc (Suc 0)} \leq k + d + 3$ by arith
from $h_{10}$ and this have lose $(t + \text{Suc (Suc 0)}) = [\text{False}]$ by blast
from this have $\text{sg2:lose (Suc (Suc t)) = [False]}$ by simp
from $h_1$ and $h_2$ and $h_3$ and $h_4$ and $h_5$ and $h_6$ and $h_9$ and $sg_1$ and $sg_2$

have $sg_3$:

ack $(t + 2) = [\text{connection-ok}]$

by (simp add: $\text{Gateway-L1}$)

from $h_7$ and this have $\text{sg4:} \forall m \leq k. \text{req ((t + 2) + m) ≠ [send]}$
by (auto, simp add: aux-k3req)
from $h_{10}$ have $\text{sg5:} \forall j \leq k. \text{lose ((t + 2) + j) = [False]}$
by (auto, simp add: aux-k3lose)
from $h_1$ and $sg_4$ and $sg_5$ and $sg_3$ and $h_5$ and $h_4$ and $h_3$ and $h_2$ have $sg_6$:

$\forall m \leq k. \text{ack ((t + 2) + m) = [connection-ok]}$

by (rule $\text{Gateway-L6a}$)
from $sg_6$ have $\text{sg7:ack (t + 2 + k) = [connection-ok]}$ by auto
from $h_1$ obtain $i_1$ $i_2$ $x$ $y$ where

a1: $\text{Sample req dt x stop lose ack i1 vc and}$
a2: $\text{Delay y i1 d x i2 and}$
a3: $\text{Loss lose a i2 y i}$

by (simp add: $\text{Gateway-def, auto}$)
from $h_3$ and $a_2$ and $a_3$ have $\text{sg8:msg (Suc 0) x}$
by (simp add: $\text{Loss-Delay-msg-a}$)
from $a_1$ and $sg_8$ and $h_4$ and $h_5$ obtain $st$ $buffer$ where

a4: $\text{tiTable-SampleT req x stop lose (fin-inf-append [init-state] st)}$

(fin-inf-append [] buffer) ack i1 vc st and
a5: $\forall t. \text{buffer t = (if dt t = [] then fin-inf-append [] buffer t else dt t)}$
by (simp add: Sample-def Sample-L-def, auto)
from a1 and h2 and sg8 and h4 and h11 and h6 and h7 and sg6 and h10
have sg9: ∀ t1 < (t + 3 + k). i1 t1 = []
  by (simp add: tiTable-i1-4)
from sg9 and a2 have sg10: ∀ t2 < (t + 3 + k + d). i2 t2 = []
  by (rule Delay-L2)
from sg10 and a3 and h2 show ?thesis by (rule Loss-L2)
qed

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

lemma Gateway-L8:
assumes h1: Gateway req dt a stop lose d ack i vc
  and h2: msg (Suc 0) req
  and h3: msg (Suc 0) stop
  and h4: msg (Suc 0) a
  and h5: ts lose
  and h6: ∀ j ≤ 2 * d. lose (t + j) = [False]
  and h7: ack t = [sending-data]
  and h8: ∀ t3 ≤ t + d. a t3 = []
shows ∀ x ≤ d + d. ack (t + x) = [sending-data]
using assms
by (simp add: Gateway-L8a)
15.9 Proof of the Refinement Relation for the Gateway Requirements

**Lemma** Gateway-L0:

**Assume** \( h_1 : \text{Gateway req dt a stop lose d ack i vc} \)

**Show** \( \text{GatewayReq req dt a stop lose d ack i vc} \)

**Using** \( \text{assms by (simp add: GatewayReq-def Gateway-L1 Gateway-L2 Gateway-L3 Gateway-L4)} \)

15.10 Lemmas about Gateway Requirements

**Lemma** GatewayReq-L1:

**Assume** \( h_1 : \text{msg (Suc 0) req} \)

**And** \( h_2 : \text{msg (Suc 0) stop} \)

**And** \( h_3 : \text{msg (Suc 0) a} \)

**And** \( h_4 : \text{ts lose} \)

**And** \( h_6 : \text{req (t + 3 + k) = [send]} \)

**And** \( h_7 : \forall j \leq 2 * d + (4 + k). \text{lose (t + j) = [False]} \)

**And** \( h_9 : \forall m \leq k. \text{ack (t + 2 + m) = [connection-ok]} \)

**And** \( h_{10} : \text{GatewayReq req dt a stop lose d ack i vc} \)

**Show** \( \text{ack (t + 3 + k) = [sending-data]} \)

**Proof**

- From \( h_9 \) have \( \text{sg1: ack (Suc (Suc (Suc (Suc (t + k))))) = [connection-ok] by auto} \)

- From \( h_7 \) have \( \text{sg2: } \forall k a \leq \text{Suc d}. \text{lose (Suc (Suc (Suc (Suc (t + ka))))) = [False]} \)

- By \( \text{(simp add: aux-lemma-lose-1)} \)

- From \( h_1 \) and \( h_2 \) and \( h_3 \) and \( h_4 \) and \( h_6 \) and \( h_{10} \) and \( \text{sg1 and sg2 have sg3:} \)

  \[ \text{ack (t + 2 + k) = [connection-ok] \land} \]

  \[ \text{req (Suc (t + 2 + k)) = [send] \land (\forall k \leq \text{Suc d}. \text{lose (t + k) = [False]}) \rightarrow} \]

  \[ \text{ack (Suc (t + 2 + k)) = [sending-data]} \]

  **By** \( \text{(simp add: GatewayReq-def)} \)

- Have \( \text{sg4: t + 3 + k = Suc (Suc (Suc (Suc (t + k))))) by arith} \)

- From \( \text{sg3 and sg1 and h6 and h7 and sg4 show ?thesis} \)

  **By** \( \text{(simp add: eval-nat-numeral)} \)

**Qed**

**Lemma** GatewayReq-L2:

**Assume** \( h_1 : \text{msg (Suc 0) req} \)

**And** \( h_2 : \text{msg (Suc 0) stop} \)

**And** \( h_3 : \text{msg (Suc 0) a} \)

**And** \( h_4 : \text{ts lose} \)

**And** \( h_5 : \text{GatewayReq req dt a stop lose d ack i vc} \)

**And** \( h_6 : \text{req (t + 3 + k) = [send]} \)

**And** \( h_7 : \text{inf-last-ti dt t} \neq [] \)

**And** \( h_8 : \forall j \leq 2 * d + (4 + k). \text{lose (t + j) = [False]} \)

**And** \( h_9 : \forall m \leq k. \text{ack (t + 2 + m) = [connection-ok]} \)

**Show** \( \text{i (t + 3 + k + d) \neq []} \)

**Proof**

- From \( h_8 \) have \( \text{sg1: (\forall (x::nat). x \leq (d+1) \rightarrow lose (t+x) = [False])} \)

  **By** \( \text{(simp add: aux-lemma-lose-2)} \)

134
15.11 Properties of the Gateway System

**lemma** GatewaySystem-L1aux:

assumes h1:msg (Suc 0) req
and h2:msg (Suc 0) stop
and h3:msg (Suc 0) a
and h4:ts lose
and h5:msg (Suc 0) req ∧ msg (Suc 0) a ∧ msg (Suc 0) stop ∧ ts lose → (∀ t. (ack t = [init-state] ∧ req (Suc t) = [init] ∧ lose (Suc t) = [False] ∧ lose (Suc t) = [False] →
ack (Suc (Suc t)) = [connection-ok] ∧ (ack t = [connection-ok] ∧ req (Suc t) = [send] ∧
(∀ k≤Suc d. lose (t + k) = [False]) →
i (Suc (t + d)) = inf-last-ti dt t ∧ ack (Suc t) = [sending-data] ∧
(ack (t + d) = [send-data] ∧ a (Suc t) = [sc-ack] ∧
(∀ k≤Suc d. lose (t + k) = [False]) →
vc (Suc (t + d)) = [vc-com]))
shows ack (t + 3 + k + d + d) = [sending-data] ∧
a (Suc (t + 3 + k + d + d)) = [sc-ack] ∧
(∀ ka≤Suc d. lose (t + 3 + k + d + d + ka) = [False]) →
vc (Suc (t + 3 + k + d + d)) = [vc-com]
using assms by blast

**lemma** GatewaySystem-L3aux:

assumes h1:msg (Suc 0) req
and h2:msg (Suc 0) stop
and h3:msg (Suc 0) a
and h4:ts lose
and h5:msg (Suc 0) req ∧ msg (Suc 0) a ∧ msg (Suc 0) stop ∧ ts lose → (∀ t. (ack t = [init-state] ∧
req (Suc t) = [init] ∧ lose (Suc t) = [False] ∧
(∀ k≤Suc d. lose (t + k) = [False]) →
vc (Suc (t + 3 + k + d + d)) = [vc-com]
\[
\begin{align*}
\text{lose } (\text{Suc } (\text{Suc } t)) &= \text{[False]} \rightarrow \\
\text{ack } (\text{Suc } (\text{Suc } t)) &= \text{[connection-ok]} \land \\
(\text{ack } t = [\text{connection-ok}] \land \text{req } (\text{Suc } t) = \text{[send]} \land \\
(\forall k \leq \text{Suc } d. \text{ lose } (t + k) = \text{[False]})) \rightarrow \\
i (\text{Suc } (t + d)) &= \text{inf-last-ti } dt \rightarrow \text{ack } (\text{Suc } t) = \text{[sending-data]} \land \\
(\text{ack } (t + d) = \text{[sending-data]} \land a (\text{Suc } t) = [\text{sc-ack}] \land \\
(\forall k \leq \text{Suc } d. \text{ lose } (t + k) = \text{[False]})) \rightarrow \\
\text{vc } (\text{Suc } (t + d)) &= [\text{vc-com}]) \\
\text{shows } \text{ack } (t + 2 + k) &= [\text{connection-ok}] \land \\
\text{req } (\text{Suc } (t + 2 + k)) &= [\text{send}] \land \\
(\forall j \leq \text{Suc } d. \text{ lose } (t + 2 + k + j) = \text{[False]})) \rightarrow \\
i (\text{Suc } (t + 2 + k + d)) &= \text{inf-last-ti } dt (t + 2 + k)
\end{align*}
\]

using \texttt{assms} by blast

\textbf{lemma} \texttt{GatewaySystem-L1:}

\textbf{assumes} \texttt{h2:ServiceCenter \ i \ a}
and \texttt{h3:GatewayReq \ req \ dt \ a \ stop \ lose \ d \ ack \ i \ vc}
and \texttt{h4:msg} (Suc 0) req
and \texttt{h5:msg} (Suc 0) stop
and \texttt{h6:msg} (Suc 0) a
and \texttt{h7:ts \ lose}
and \texttt{h9:∀j≤2 \ast \ d \ + \ (4 + k). \ lose \ (t + j) = \text{[False]}}
and \texttt{h11:i \ (t + 3 + k + d) ≠ []}
and \texttt{h14:∀x ≤ d + d. \ ack \ (t + 3 + k + x) = \text{[sending-data]}}
\textbf{shows} \texttt{vc} (2 \ast \ d + (t + (4 + k))) = [\text{vc-com}]

\textbf{proof} –

\textbf{from} \texttt{h2} \textbf{have} ∀t. a (Suc t) = (if \ i \ t = [] \ then [] \ else \ [\text{sc-ack}])
by (simp add:ServiceCenter-def)
\textbf{from} \texttt{this \ have} \texttt{sq1:}
a (Suc (t + 3 + k + d)) = (if \ i \ (t + 3 + k + d) = [] \ then [] \ else \ [\text{sc-ack}])
by blast
\textbf{from} \texttt{sq1 \ and \ h11 \ have} \texttt{sq2:a} (Suc (t + 3 + k + d)) = [\text{sc-ack}] \textbf{by} auto
\textbf{from} \texttt{h14 \ have} \texttt{sq3:ack} (t + 3 + k + 2\ast d) = [\text{sending-data}] \textbf{by} simp
\textbf{from} \texttt{h4 \ and \ h5 \ and \ h6 \ and \ h7 \ and \ h3 \ have} \texttt{sq4:}
ack (t + 3 + k + d + d) = [\text{sending-data}] ∧ a (Suc (t + 3 + k + d)) = [\text{sc-ack}] ∧
(∀ka≤Suc \ d. \ lose \ (t + 3 + k + d + ka) = \text{[False]}) \rightarrow \\
vc (Suc (t + 3 + k + d + d)) = [\text{vc-com}]
\textbf{apply} (simp \ only: \texttt{GatewayReq-def})
by (rule GatewaySystem-L1aux, auto)
\textbf{from} \texttt{h0 \ have} \texttt{sq5:∀ka≤Suc \ d. \ lose \ (d + (t + (3 + k)) + ka) = \text{[False]}}
by (simp \ add: aux-lemma-lose-3)
\textbf{from} \texttt{h} \texttt{5a} \textbf{have} \texttt{sq5a:d + (t + (3 + k)) = t + 3 + k + d \ by} arith
\textbf{from} \texttt{sq5 \ and \ sq5a \ have} \texttt{sq5b:∀ka≤Suc \ d. \ lose \ (t + 3 + k + d + ka) = \text{[False]}}
by auto
\textbf{have} \texttt{sq6:(t + 3 + k + 2\ast d) = (2\ast d + (t + (3 + k))) \ by} arith
\textbf{have} \texttt{sq7:Suc \ (Suc \ (Suc \ (Suc \ t + k + d + d))) = Suc \ (Suc \ (Suc \ (Suc \ t + k + d + d)))} \textbf{by} arith

136
have \( sg8 : \text{Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc ((t + k + d + d)))))))))} = \text{Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc ((d + d + (t + k))))))))))}) \) by arith
from \( sg4 \) and \( sg3 \) and \( sg2 \) and \( sg5b \) and \( sg6 \) and \( sg7 \) and \( sg8 \) show \( \text{thesis} \) by \( \text{(simp add: eval-nat-numeral)} \)
qed

lemma \( \text{aux4lose1} \):
assumes \( h1 : \forall j \leq 2 * d + (4 + k). \text{lose} (t + j) = [\text{False}] \)
and \( h2 : j \leq k \)
shows \( \text{lose} (t + (2 :: \text{nat}) + j) = [\text{False}] \)
proof –
from \( h1 \) have \( (2 :: \text{nat}) + j \leq (2 :: \text{nat}) * d + (4 + k) \) by arith
from \( h1 \) and this have \( \text{lose} (t + (2 + j)) = [\text{False}] \) by blast
from this show \( \text{thesis} \) by simp
qed

lemma \( \text{aux4lose2} \):
assumes \( h1 : \forall j \leq 2 * d + (4 + k). \text{lose} (t + j) = [\text{False}] \)
and \( h2 : 3 + k + d \leq 2 * d + (4 + k) \)
shows \( \text{lose} (t + (3 :: \text{nat}) + k + d) = [\text{False}] \)
proof –
from \( \text{assms} \) have \( \text{lose} (t + ((3 :: \text{nat}) + k + d)) = [\text{False}] \) by blast
from this show \( \text{thesis} \) by \( \text{(simp add: arith-sum1)} \)
qed

lemma \( \text{aux4req} \):
assumes \( h1 : \forall (m :: \text{nat}) \leq k + 2. \text{req} (t + m) \neq [\text{send}] \)
and \( h2 : m \leq k \)
and \( h3 : \text{req} (t + 2 + m) = [\text{send}] \)
shows \( \text{False} \)
proof –
from \( h2 \) have \( (2 :: \text{nat}) + m \leq k + (2 :: \text{nat}) \) by arith
from \( h1 \) and this have \( \text{req} (t + (2 + m)) \neq [\text{send}] \) by blast
from this and \( h3 \) show \( \text{thesis} \) by simp
qed

lemma \( \text{GatewaySystem-L2} \):
assumes \( h1 : \text{Gateway req dt a stop lose d ack i vc} \)
and \( h2 : \text{ServiceCenter i a} \)
and \( h3 : \text{GatewayReq req dt a stop lose d ack i vc} \)
and \( h4 : \text{msg (Suc 0) req} \)
and \( h5 : \text{msg (Suc 0) stop} \)
and \( h6 : \text{msg (Suc 0) a} \)
and \( h7 : \text{ts lose} \)
and \( h8 : \text{ack t = [init-state]} \)
and \( h9 : \text{req (Suc t) = [init]} \)
and \( h10 : \forall t. \text{req t1} = [] \)
and h11 : \forall m \leq k + 2. \text{req} (t + m) \neq [\text{send}]
and h12 : \text{req} (t + 3 + k) = [\text{send}]
and h13 : \text{inf}-last-li dt t \neq []
and h14 : \forall j \leq 2 * d + (4 + k). \text{lose} (t + j) = [\text{False}]
shows \text{vc} (2 * d + (t + (4 + k))) = [\text{vc-com}]

proof –

have Suc 0 \leq 2 * d + (4 + k) by arith
from h14 and this have lose (t + Suc 0) = [False] by blast
from this have sg1:lose (Suc t) = [False] by simp
have Suc (Suc 0) \leq 2 * d + (4 + k) by arith
from h14 and this have lose (t + Suc (Suc 0)) = [False] by blast
from this have sg2:lose (Suc (Suc t)) = [False] by simp
from h3 and h4 and h5 and h6 and h7 and h8 and h9 and sg1 and sg2 have sg3:

ack (t + 2) = [\text{connection-ok}]
by (simp add: GatewayReq-def)
from h14 have sg4 : \forall j \leq k. lose (t + 2 + j) = [False]
by (clarify, rule aux4lose1, simp)
from h11 have sg5 : \forall m \leq k. req (t + 2 + m) \neq [send]
by (clarify, rule aux4req, auto)

from h3 and sg5 and sg4 and sg3 and h4 and h5 and h6 and h7 have sg6:
\forall m \leq k. ack (t + 2 + m) = [\text{connection-ok}]
by (rule Gateway-L6)

from h3 and h4 and h5 and h6 and h7 and h12 and h14 and sg6 have sg10:

ack (t + 3 + k) = [\text{sending-data}]
by (simp add: GatewayReq-L1)
from h11 have sg12 : \forall m < k + 3. \text{req} (t + m) \neq [\text{send}] by auto
from h14 have sg13 : \forall j \leq k + d + 3. \text{lose} (t + j) = [\text{False}] by auto
from h1 and h7 and h6 and h5 and h4 and h9 and sg12 and h12 and h8 and sg13 and h10 have sg11:
i (t + 3 + k + d) \neq []
by (simp add: GatewayReq-L2)

from h11 have sg14 : \forall t2 < (t + 3 + k + d). i t2 = []
by (simp add: Gateway-L7)
from sg14 and h2 have sg15 : \forall t3 \leq (t + 3 + k + d). a t3 = []
by (simp add: ServiceCenter-L2)
from h14 have sg18 : \forall j \leq 2 * d. lose ((t + 3 + k) + j) = [\text{False}]
by (simp add: streamValue43)
from h14 have sg16a : \forall j \leq 2 * d. lose (t + j + (4 + k)) = [\text{False}]
by (simp add: streamValue2)
have sg16b : Suc (3 + k) = (4 + k) by arith
from sg16a and sg16b have sg16 : \forall j \leq 2 * d. lose (t + j + Suc (3 + k)) = [\text{False}]

138
by (simp (no-asmp-simp))
from h1 and h4 and h5 and h6 and h7 and sg18 and sg10 and sg15 have sg19:
\[ \forall x \leq d + d. \text{ack} (t + 3 + k + x) = [\text{sending-data}] \]
by (simp add: Gateway-L8)
from sg19 have sg19a:ack (t + 3 + k + d + d) = [sending-data] by auto
from sg16 have sg20a:\(\forall j \leq d. \text{lose} (t + 3 + k + d + (Suc j)) = [False] \]
by (rule streamValue10)
have sg20b:3 + k + d \leq 2 \ast d + (4 + k) by arith
from h14 and sg20b have sg20c:lose (t + 3 + k + d) = [False]
by (rule aux4(lose2)
from sg20a and sg20c have sg20:\(\forall j \leq Suc d. \text{lose} (t + 3 + k + d + j) = [False] \]
by (rule streamValue8)
from h4 and h5 and h6 and h7 and h3 have sg21:
\[ \text{ack} (t + 3 + k + d + d) = [\text{sending-data}] \land \]
\[ a (Suc (t + 3 + k + d + d)) = [sc-ack] \land \]
\[ (\forall j \leq Suc d. \text{lose} (t + 3 + k + d + j) = [False]) \rightarrow \]
\[ \text{vc} (Suc (t + 3 + k + d + d)) = [vc-com] \]
apply (simp only: GatewayReq-def)
by (rule GatewaySystem-L1aux, auto)
from h2 and sg11 have sg22:a (Suc (t + 3 + k + d)) = [sc-ack]
by (simp only: ServiceCenter-def, auto)
from sg21 and sg19a and sg22 and sg20 have sg23:
\[ \text{vc} (Suc (t + 3 + k + d + d)) = [vc-com] by simp \]
have sg24:2 \ast d + (t + (4 + k)) = (Suc (t + 3 + k + d + d)) by arith
from sg23 and sg24 show ?thesis
by (simp (no-asmp-simp), simp)
qed

lemma GatewaySystem-L3:
assumes h1:Gateway req dt a stop lose d ack i vc
and h2:ServiceCenter i a
and h3:GatewayReq req dt a stop lose d ack i vc
and h4:msg (Suc 0) req
and h5:msg (Suc 0) stop
and h6:msg (Suc 0) a
and h7:ts lose
and h8: dt (Suc t) \neq [] \land dt (Suc (Suc t)) \neq []
and h9: ack t = [init-state]
and h10:req (Suc t) = [init]
and h11:Suc t1 \leq t. req t1 = []
and h12:Suc m \leq k + 2. req (t + m) \neq [send]
and h13:req (t + 3 + k) = [send]
and h14:Suc j \leq 2 \ast d + (4 + k). lose (t + j) = [False]
shows vc (2 \ast d + (t + (4 + k))) = [vc-com]
proof -
have Suc 0 \leq 2 \ast d + (4 + k) by arith
from h14 and this have lose (t + Suc 0) = [False] by blast
from this have sg1:lose (Suc t) = [False] by simp
have Suc (Suc 0) ≤ 2 * d + (4 + k) by arith
from h14 and this have lose (t + Suc (Suc 0)) = [False] by blast
from this have sg2:lose (Suc (Suc t)) = [False] by simp
from h3 and h4 and h5 and h6 and h7 and h10 and h9 and sg1 and sg2
have sg3:
  ack (t + 2) = [connection-ok]
  by (simp add: GatewayReq-def)
from h14 have sg4: ∀ j ≤ k. lose (t + 2 + j) = [False]
  by (clarify, rule aux4lose1, simp)
from h12 have sg5: ∀ m ≤ k. req (t + 2 + m) ≠ [send]
  by (clarify, rule aux4req, auto)
from h1 and sg5 and sg4 and sg3 and h4 and h5 and h6 and h7 have sg6:
  ∀ m ≤ k. ack (t + 2 + m) = [connection-ok]
  by (rule Gateway-L6)
from sg6 have sg6a: ack (t + 2 + k) = [connection-ok] by simp
from h3 and h4 and h5 and h6 and h7 and h13 and h14 and sg6 have
sg10:
  ack (t + 3 + k) = [sending-data]
  by (simp add: GatewayReq-L1)
from h3 and h4 and h5 and h6 and h7 have sg11a:
  ack (t + 2 + k) = [connection-ok] ∧
  req (Suc (t + 2 + k)) = [send] ∧
  (∀ j ≤ Suc d. lose ((t + 2 + k) + j) = [False]) →
  i (Suc (t + (2::nat) + k + d)) = inf-last-ti dt (t + 2 + k)
  apply (simp only: GatewayReq-def)
  by (rule GatewaySystem-L3aux, auto)
have sg12: Suc (t + 2 + k) = t + 3 + k by arith
from h13 and sg12 have sg12a: req (Suc (t + 2 + k)) = [send]
  by (simp add: eval-nat-numeral)
from h14 have sg13: ∀ j ≤ Suc d. lose ((t + 2 + k) + j) = [False]
  by (rule streamValue12)
from sg11a and sg6a and h13 and sg12a and sg13 have sg14:
  i (Suc ((t + (2::nat) + k + d)) = inf-last-ti dt (t + 2 + k) by simp
from h5 have sg15: inf-last-ti dt (t + 2 + k) ≠ []
  by (rule inf-last-ti-Suc2)
from sg14 and sg15 have sg16: i (t + 3 + k + d) ≠ []
  by (simp add: arith-sum4)
from h14 have sg17: ∀ j ≤ k + d + 3. lose (t + j) = [False] by auto
from h12 have sg18: ∀ m < (k + 3). req (t + m) ≠ [send] by auto
from h1 and h4 and h5 and h6 and h7 and h10 and sg18 and h13 and h9
and sg17 and h11
  have sg19: ∀ t2 < (t + 3 + k + d). i t2 = []
  by (simp add: Gateway-L7)
from h2 and sg19 have sg20: ∀ t3 ≤ (t + 3 + k + d). a t3 = []
by (simp add: ServiceCenter-L2)
from h14 have sg21:\forall j \leq 2 \ast d. lose (t + 3 + k + j) = [False]
  by (simp add: streamValue43)
from h1 and h4 and h5 and h6 and h7 and sg21 and sg10 and sg20 have
sg22:
  \forall x \leq d + d. ack (t + 3 + k + x) = [sending-data]
  by (simp add: Gateway-L8)
from h2 and h3 and h4 and h5 and h6 and h7 and h14 and sg16 and sg22
show \?thesis
  by (simp add: GatewaySystem-L1)
qed

15.12 Proof of the Refinement for the Gateway System

lemma GatewaySystem-L0:
  assumes h1:GatewaySystem req dt stop lose d ack vc
  shows GatewaySystemReq req dt stop lose d ack vc
proof –
  from h1 obtain x i where
  a1:Gateway req dt stop lose d ack i vc and
  a2:ServiceCenter i x
  by (simp add: GatewaySystem-def, auto)
from a1 have sg1:GatewayReq req dt x stop lose d ack i vc
  by (simp add: Gateway-L0)
from a2 have sg2:msg (Suc 0) x
  by (simp add: ServiceCenter-a-msg)
from h1 and a1 and a2 and sg1 and sg2 show \?thesis
  apply (simp add: GatewaySystemReq-def, auto)
  apply (simp add: GatewaySystem-L3)
  apply (simp add: GatewaySystem-L3)
  apply (simp add: GatewaySystem-L3)
  by (simp add: GatewaySystem-L2)
qed

end
References


