Stream processing components:
Isabelle/HOL formalisation and case studies

Maria Spichkova

May 28, 2015

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

Contents

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

2 Theory ArithExtras.thy 15

3 Auxiliary Theory ListExtras.thy 15

4 Auxiliary arithmetic lemmas 19

5 FOCUS streams: operators and lemmas 21
  5.1 Definition of the FOCUS stream types . . . . . . . . . . . . . . 21
  5.2 Definitions of operators . . . . . . . . . . . . . . . . . . . . . 21
  5.3 Properties of operators . . . . . . . . . . . . . . . . . . . . . . 33
    5.3.1 Lemmas for concatenation operator . . . . . . . . . . . . 33
    5.3.2 Lemmas for operators ts and msg . . . . . . . . . . . . . 34
    5.3.3 Lemmas for inf_truncate . . . . . . . . . . . . . . . . . . 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.4</td>
<td>Lemmas for <code>fin_make_untimed</code></td>
<td>37</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Lemmas for <code>inf_disj</code> and <code>inf_disjS</code></td>
<td>39</td>
</tr>
<tr>
<td>6</td>
<td>Properties of time-synchronous streams of types <code>bool</code> and <code>bit</code></td>
<td>39</td>
</tr>
<tr>
<td>7</td>
<td>Changing time granularity of the streams</td>
<td>42</td>
</tr>
<tr>
<td>7.1</td>
<td>Join time units</td>
<td>42</td>
</tr>
<tr>
<td>7.2</td>
<td>Split time units</td>
<td>45</td>
</tr>
<tr>
<td>7.3</td>
<td>Duality of the split and the join operators</td>
<td>46</td>
</tr>
<tr>
<td>8</td>
<td>Steam Boiler System: Specification</td>
<td>47</td>
</tr>
<tr>
<td>9</td>
<td>Steam Boiler System: Verification</td>
<td>48</td>
</tr>
<tr>
<td>9.1</td>
<td>Properties of the Boiler Component</td>
<td>48</td>
</tr>
<tr>
<td>9.2</td>
<td>Properties of the Controller Component</td>
<td>50</td>
</tr>
<tr>
<td>9.3</td>
<td>Properties of the Converter Component</td>
<td>59</td>
</tr>
<tr>
<td>9.4</td>
<td>Properties of the System</td>
<td>59</td>
</tr>
<tr>
<td>9.5</td>
<td>Proof of the Refinement Relation</td>
<td>62</td>
</tr>
<tr>
<td>10</td>
<td>FlexRay: Types</td>
<td>62</td>
</tr>
<tr>
<td>11</td>
<td>FlexRay: Specification</td>
<td>63</td>
</tr>
<tr>
<td>11.1</td>
<td>Auxiliary predicates</td>
<td>63</td>
</tr>
<tr>
<td>11.2</td>
<td>Specifications of the FlexRay components</td>
<td>65</td>
</tr>
<tr>
<td>12</td>
<td>FlexRay: Verification</td>
<td>66</td>
</tr>
<tr>
<td>12.1</td>
<td>Properties of the function Send</td>
<td>67</td>
</tr>
<tr>
<td>12.2</td>
<td>Properties of the component Scheduler</td>
<td>67</td>
</tr>
<tr>
<td>12.3</td>
<td>Disjoint Frames</td>
<td>68</td>
</tr>
<tr>
<td>12.4</td>
<td>Properties of the sheaf of channels <code>nSend</code></td>
<td>70</td>
</tr>
<tr>
<td>12.5</td>
<td>Properties of the sheaf of channels <code>nGet</code></td>
<td>73</td>
</tr>
<tr>
<td>12.6</td>
<td>Properties of the sheaf of channels <code>nStore</code></td>
<td>75</td>
</tr>
<tr>
<td>12.7</td>
<td>Refinement Properties</td>
<td>80</td>
</tr>
<tr>
<td>13</td>
<td>Gateway: Types</td>
<td>82</td>
</tr>
<tr>
<td>14</td>
<td>Gateway: Specification</td>
<td>82</td>
</tr>
<tr>
<td>15</td>
<td>Gateway: Verification</td>
<td>87</td>
</tr>
<tr>
<td>15.1</td>
<td>Properties of the defined data types</td>
<td>87</td>
</tr>
<tr>
<td>15.2</td>
<td>Properties of the Delay component</td>
<td>89</td>
</tr>
<tr>
<td>15.3</td>
<td>Properties of the Loss component</td>
<td>90</td>
</tr>
<tr>
<td>15.4</td>
<td>Properties of the composition of Delay and Loss components</td>
<td>92</td>
</tr>
<tr>
<td>15.5</td>
<td>Auxiliary Lemmas</td>
<td>93</td>
</tr>
</tbody>
</table>
15.6 Properties of the ServiceCenter component . . . . . . . . . . 113
15.7 General properties of stream values . . . . . . . . . . . . . . 114
15.8 Properties of the Gateway . . . . . . . . . . . . . . . . . . . . 117
15.9 Proof of the Refinement Relation for the Gateway Requirements125
15.10 Lemmas about Gateway Requirements . . . . . . . . . . . . 125
15.11 Properties of the Gateway System . . . . . . . . . . . . . . 126
15.12 Proof of the Refinement for the Gateway System . . . . . . . 132
1 Introduction

The set of theories presented in this paper is an extended and updated Isabelle/HOL\cite{5} formalisation of stream processing components elaborated within the methodology “Focus on Isabelle” \cite{6}. This paper is organised as follows: in the first section we give a general introduction to the Focus stream processing components \cite{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:

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

A timed stream, as suggested in our previous work \cite{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:

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

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

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

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

1.2 Case Study 1: Steam Boiler System

A steam boiler control system can be represented 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 \textit{ControlSystemArch} is a refinement of the specification \textit{ControlSystem}. It follows from the definition of behavioral refinement that in order to verify that \textit{ControlSystem} \(\Rightarrow\) \textit{ControlSystemArch} it is enough to prove that

\[
\llbracket \textit{ControlSystemArch} s \rrbracket \Rightarrow \llbracket \textit{ControlSystem} \rrbracket
\]

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

\textbf{lemma} \textit{L0-ControlSystem}: \(\llbracket \textit{ControlSystemArch} s \rrbracket \Rightarrow \textit{ControlSystem} s\)
1.3 Case Study 2: FlexRay Communication Protocol

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The $\text{Focus}$ specification $\text{FlexRayController}$ represent the controller component for a single node of the system. It consists of the components $\text{Scheduler}$ and $\text{BusInterface}$. The $\text{Scheduler}$ signals the $\text{BusInterface}$, 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 $\text{Scheduler}$ describes the communication scheduler. It sends at every time $t$ interval, which is equal modulo the length of the
communication cycle to some FlexRay frame identifier (that corresponds to the number of the slot in the communication round) from the scheduler table, this frame identifier.

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

- the frame transmission by FlexRay must be correct in terms of the predicate FrameTransmission: if the time <i>t</i> is equal modulo the length of the cycle (FlexRay communication round) to the element of the scheduler table of the node <i>k</i>, then this and only this node can send a data at <i>t</i>th time interval;
- FlexRay component sends in every time interval at most one message to each node via channels get<sub>i</sub> and store<sub>i</sub>, 1 ≤ i ≤ n).

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

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

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

**lemma** main-fr-refinement:

FlexRayArch n nReturn nC nStore nGet ⇒ FlexRay n nReturn nC nStore nGet

### 1.4 Case Study 3: Automotive-Gateway

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

1[http://www.verisoft.de]
• **GatewaySystem** (gateway system architecture),
• **GatewaySystemReq** (gateway system requirements),
• **ServiceCenter** (Emergency Service Center),
• **Gateway** (gateway architecture),
• **GatewayReq** (gateway requirements),
• **Sample** (the main component describing its logic),
• **Delay** (the component modelling the communication delay), and
• **Loss** (the component modelling the communication loss).

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

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

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

```isar
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* + 2*d*,

then at time interval *t* + 4 + *k* + 2*d* 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* ∈ ℕ denotes the communication delay between the
central station and a vehicle. This component consists of three subcomponents: Sample, Delay, and Loss:

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 \( i \) will be forwarded without any delay via channels \( a_2 \) and \( i_2 \) 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 an 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 an 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 \( st \) and \textit{buffer} (more precisely, a local variable \textit{buffer} and a state variable \( st \)). The guarantee part of the component \textit{Sample} can be specified as a timed state transition diagram (TSTD) 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 fulfils 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
Figure 1: Timed state transition diagram for the component Sample

lemma:

**lemma** *Gateway-L0*:

Gateway req dt a stop lose d ack i vc  
⇒ GatewayReq req dt a stop lose d ack i vc

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

**lemma** *GatewaySystem-L0*:

GatewaySystem req dt stop lose d ack i vc  
⇒ GatewaySystemReq req dt stop lose d ack i vc
2 Theory ArithExtras.thy

theory ArithExtras
imports Main
begin

datatype natInf = Fin nat
             | Infty (∞)

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

3 Auxiliary Theory ListExtras.thy

theory ListExtras
imports Main
begin

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

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

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

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

lemma mem-set-1:
assumes a mem l
shows \( a \in \text{set } l \)
using assms by (metis memS-def mem-memS-eq)

lemma mem-set-2:
assumes \( a \in \text{set } l \)
shows \( a \in l \)
using assms by (metis (full-types) memS-def mem-memS-eq)

lemma set-inter-mem:
assumes \( x \in l_1 \)
and \( x \in l_2 \)
shows \( l_1 \cap l_2 \neq \{\} \)
using assms by (metis IntI empty-iff mem-set-1)

lemma mem-notdisjoint:
assumes \( x \in l_1 \)
and \( x \in l_2 \)
shows \( \neg \text{disjoint } l_1 l_2 \)
using assms by (metis disjoint-def set-inter-mem)

lemma mem-notdisjoint2:
assumes \( h_1: \text{disjoint } (\text{schedule } A) (\text{schedule } B) \)
and \( h_2: x \in \text{schedule } A \)
shows \( \neg x \in \text{schedule } B \)
proof –
  \{ assume \( x \in \text{schedule } B \)
  from \( h_2 \) and this have \( \neg \text{disjoint } (\text{schedule } A) (\text{schedule } B) \)
  by (simp add: mem-notdisjoint)
  from \( h_1 \) and this have False by simp
  \} then have \( \neg x \in \text{schedule } B \) by blast
then show \( \neg \text{thesis} \) by simp
qed

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

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

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

lemma list-length-hint2:
assumes \( \text{length } x = \text{Suc } 0 \)
shows \( \text{hd } x = x \)
using assms
by (metis Zero-neq-Suc list.sel(1) length-Suc-conv neq-Nil-conv)

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

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

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

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

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

lemma list-nth-append0:
assumes \( i < \text{length } x \)
shows \( x ! i = (x \cdot z) ! i \)
using assms
by (metis nth-append)

lemma list-nth-append1:
assumes \( i < \text{length } x \)
shows \( (b \# x) ! i = (b \# x \cdot y) ! i \)
proof
- from assms have \( i < \text{length } (b \# x) \) by auto
then have \( \text{sg2: (b \# x) ! i = ((b \# x) \cdot y) ! i} \)
  by (rule list-nth-append0)
then show ?thesis by simp
qed

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

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

lemma list-nth-append4:
assumes \( i < \text{Suc} \ (\text{length} \ x + \text{length} \ y) \)
and \( \neg i - \text{Suc} \ (\text{length} \ x) < \text{Suc} \ (\text{length} \ y) \)
shows False
using assms by arith

lemma list-nth-append5:
assumes \( i - \text{length} \ x < \text{Suc} \ (\text{length} \ y) \)
and \( \neg i - \text{Suc} \ (\text{length} \ x) < \text{Suc} \ (\text{length} \ y) \)
shows \( \neg i < \text{Suc} \ (\text{length} \ x + \text{length} \ y) \)
using assms by arith

lemma list-nth-append6:
assumes \( \neg i - \text{length} \ x < \text{Suc} \ (\text{length} \ y) \)
and \( \neg i - \text{Suc} \ (\text{length} \ x) < \text{Suc} \ (\text{length} \ y) \)
shows \( \neg i < \text{Suc} \ (\text{length} \ x + \text{length} \ y) \)
using assms by arith

lemma list-nth-append6a:
assumes \( i < \text{Suc} \ (\text{length} \ x + \text{length} \ y) \)
and \( \neg i - \text{length} \ x < \text{Suc} \ (\text{length} \ y) \)
shows False
using assms by arith

lemma list-nth-append7:
assumes \( i - \text{length} \ x < \text{Suc} \ (\text{length} \ y) \)
and \( i - \text{Suc} \ (\text{length} \ x) < \text{Suc} \ (\text{length} \ y) \)
shows \( i < \text{Suc} \ (\text{Suc} \ (\text{length} \ x + \text{length} \ y)) \)
lemmas list-nth-append8:
assumes ¬ i < Suc (Suc (length x + length y))
and i < Suc (Suc (length x + length y))
shows i = Suc (Suc (length x + length y))
using assms by arith

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

lemmas list-nth-append10:
assumes ¬ i < Suc (Suc (length x))
and ¬ i − Suc (Suc (length x)) < Suc (length y)
shows ¬ i < Suc (Suc (Suc (length x + length y)))
using assms by arith

end

4 Auxiliary arithmetic lemmas

theory arith-hints
import Main
begin

lemmas arith-mod-neq:
assumes a mod n ≠ b mod n
shows a ≠ b
using assms by auto

lemmas arith-mod-nzero:
fixes :nat
assumes i < n
and o < i
shows o < (n * t + i) mod n
using assms
by (metis Divides.mod-less mod-mult2 add.commute)

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

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

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

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

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

lemma arith1:
  assumes h1:Suc n * t = Suc n * q
  shows t = q
using assms
by (metis mult-cancel2 mult.commute neg0-conv zero-less-Suc)

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

end
5 FOCUS streams: operators and lemmas

theory stream
  imports ListExtras ArithExtras
begin

5.1 Definition of the FOCUS stream types

— Finite timed FOCUS stream
type-synonym 'a fstream = 'a list list

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

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

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

5.2 Definitions of operators

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

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

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

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

— range of a finite untimed stream
definition
\[ \text{finU-range :: 'a \,list \,\Rightarrow \,'a \,set} \]
\[ \text{where} \]
\[ \text{finU-range \,x \equiv set \,x} \]

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

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

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

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

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

— removing \( i \) first elements from an infinite stream \( s \)
— in the case of an untimed stream: removing \( i \) first data elements
— in the case of a timed stream: removing \( i \) first time intervals
definition
inf-drop :: nat ⇒ (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
\[ \text{fin-length} \begin{array}{l} \begin{array}{l} [ ] = 0 \\ \{ x \neq [] \} = (\text{length } x) + (\text{fin-length } x) \end{array} \end{array} \]

length of a (general) stream

\textbf{definition}

\begin{align*}
\text{stream-length} :: \quad & \quad \text{\textquote{\text{\textbackslash \text{a}} stream} \Rightarrow \text{natInf} \\
\text{where} \\
\text{stream-length } s & \equiv \\
& \begin{cases} \\
\text{(FinT } x) & \Rightarrow \text{Fin (fin-length } x) \\
\text{(FinU } x) & \Rightarrow \text{Fin (length } x) \\
\text{(InfT } x) & \Rightarrow \infty \\
\text{(InfU } x) & \Rightarrow \infty \\
\end{cases} 
\end{align*}

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

\textbf{axiomatization}

\begin{align*}
\text{fin-nth} :: \quad & \quad \text{\textquote{\text{\textbackslash \text{a}} fstream} \Rightarrow \text{nat} \Rightarrow \text{\textbackslash \text{a}} \\
\text{where} \\
\text{fin-nth-Cons:} \\
\text{fin-nth} (\text{hds} \neq \text{tls}) \quad k = \\
\begin{cases} \\
\text{if } \text{hds} = [] & \text{then fin-nth tls} \quad k \\
\text{else} & \begin{cases} \\
\text{if} \quad (k < (\text{length } \text{hds})) & \text{then nth} \quad \text{hds} \quad k \\
\text{else fin-nth tls} \quad (k - \text{length hds}) \\
\end{cases} \\
\end{cases} 
\end{align*}

removing \( i \) first data elements from an infinite timed stream \( s \)

\textbf{primrec}

\begin{align*}
\text{inf-nth} :: \quad & \quad \text{\textquote{\text{\textbackslash \text{a}} istream} \Rightarrow \text{nat} \Rightarrow \text{\textbackslash \text{a}} \\
\text{where} \\
\text{inf-nth } s \quad 0 & = \text{hd} \quad (\text{LEAST } i. (s \ i) \neq []) \\
\text{inf-nth } s \quad (\text{Suc } k) = \\
\begin{cases} \\
\text{if} \quad ((\text{Suc } k) < (\text{length } (s \ 0))) & \text{then nth} \quad (s \ 0) \quad (\text{Suc } k) \\
\text{else} & \begin{cases} \\
\text{if} \quad (s \ 0) = [] & \text{then (inf-nth (inf-tl (inf-drop} \\
\quad (\text{LEAST } i. (s \ i) \neq [])) \ 0) \\
\text{else inf-nth (inf-tl } s \quad k) \\
\end{cases} \\
\end{cases} 
\end{align*}

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

\textbf{definition}

\begin{align*}
\text{stream-nth} :: \quad & \quad \text{\textquote{\text{\textbackslash \text{a}} stream} \Rightarrow \text{nat} \Rightarrow \text{\textbackslash \text{a}} \\
\text{where} \\
\text{stream-nth } s \quad k & \equiv \\
\begin{cases} \\
\text{case } s \quad \text{of} \quad \text{(FinT } x) & \Rightarrow \text{fin-nth } x \quad k \\
\quad \text{(FinU } x) & \Rightarrow \text{nth } x \quad k \\
\quad \text{(InfT } x) & \Rightarrow \text{inf-nth } x \quad k \\
\quad \text{(InfU } x) & \Rightarrow x \quad k \\
\end{cases} 
\end{align*}
— prefix of an infinite stream

**primrec**

`inf-prefix :: 'a list ⇒ (nat ⇒ 'a) ⇒ nat ⇒ bool`

**where**

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

— prefix of a finite stream

**primrec**

`fin-prefix :: 'a list ⇒ 'a list ⇒ bool`

**where**

`fin-prefix [] s = True |`
`fin-prefix (x#xs) s =
  (if (s = [])
   then False
   else (x = (hd s)) ∧ (fin-prefix xs s) )`

— prefix of a (general) stream

**definition**

`stream-prefix :: 'a stream ⇒ 'a stream ⇒ bool`

**where**

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

— truncating a finite stream after the n-th element

**primrec**

`fin-truncate :: 'a list ⇒ nat ⇒ 'a list`

**where**

`fin-truncate [] n = [] |`
\( \text{fin-truncate}(x\#xs) \ i = \)
\[
\begin{align*}
\text{case } i \text{ of } 0 & \Rightarrow [] \\
& \mid (\text{Suc } n) \Rightarrow x \# (\text{fin-truncate } xs \ n)
\end{align*}
\]— truncating a finite stream after the \( n \)-th element
— \( n \) is of type of natural numbers enriched by Infinity

**definition**

\( \text{fin-truncate-plus} :: \text{'}a\ \text{list} \Rightarrow \text{natInf} \Rightarrow \text{'}a\ \text{list} \)

**where**

\( \text{fin-truncate-plus} \ s \ n \equiv \)
\[
\begin{align*}
\text{case } n \text{ of } (\text{Fin } i) & \Rightarrow \text{fin-truncate } s \ i \\
& \mid \infty \Rightarrow s
\end{align*}
\]— truncating an infinite stream after the \( n \)-th element

**primrec**

\( \text{inf-truncate} :: (\text{nat} \Rightarrow \text{'}a) \Rightarrow \text{nat} \Rightarrow \text{'}a\ \text{list} \)

**where**

\( \text{inf-truncate} \ s \ 0 = [s \ 0] \mid \text{inf-truncate} \ s \ (\text{Suc } k) = (\text{inf-truncate} \ s \ k) \bullet [s \ (\text{Suc } k)] \)

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

**definition**

\( \text{inf-truncate-plus} :: \text{'}a\ \text{istream} \Rightarrow \text{natInf} \Rightarrow \text{'}a\ \text{stream} \)

**where**

\( \text{inf-truncate-plus} \ s \ n \equiv \)
\[
\begin{align*}
\text{case } n \text{ of } (\text{Fin } i) & \Rightarrow \text{FinT} \ (\text{inf-truncate} \ s \ i) \\
& \mid \infty \Rightarrow \text{InfT} \ s
\end{align*}
\]— concatenation of a finite and an infinite stream

**definition**

\( \text{fin-inf-append} :: \text{'}a\ \text{list} \Rightarrow (\text{nat} \Rightarrow \text{'}a) \Rightarrow (\text{nat} \Rightarrow \text{'}a) \)

**where**

\( \text{fin-inf-append} \ us \ s \equiv \)
\[
(\lambda \ i. \ (\text{if } (i < (\text{length } us)) \ \text{then } (\text{nth } us \ i) \ \text{else } s \ (i - (\text{length } us))))
\]
— insuring that the infinite timed stream is time-synchronous

**definition**

\( \text{ts} :: \text{'}a\ \text{istream} \Rightarrow \text{bool} \)

**where**

\( \text{ts} \ s \equiv \forall \ i. \ (\text{length } (s \ i) = 1) \)
— insuring that each time interval of an infinite timed stream contains at most \( n \) data elements
**definition**  
\[ \text{msg} :: \text{nat} \Rightarrow 'a \text{istream} \Rightarrow \text{bool} \]

**where**  
\[ \text{msg} n s \equiv \forall t. \text{length} (s t) \leq n \]

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

**primrec**  
\[ \text{fin-msg} :: \text{nat} \Rightarrow 'a \text{list list} \Rightarrow \text{bool} \]

**where**  
\[ \text{fin-msg} n [] = \text{True} \mid \text{fin-msg} n (x\#xs) = ((\text{length} x) \leq n) \land (\text{fin-msg} n xs) \]

— making a finite timed stream to a finite untimed stream

**definition**  
\[ \text{fin-make-untimed} :: 'a \text{fstream} \Rightarrow 'a \text{list} \]

**where**  
\[ \text{fin-make-untimed} x \equiv \text{concat} x \]

— making an infinite timed stream to an infinite untimed stream

— (auxiliary function)

**primrec**  
\[ \text{inf-make-untimed1} :: 'a \text{istream} \Rightarrow \text{nat} \Rightarrow 'a \]

**where**  
\[ \text{inf-make-untimed1-0:} \text{inf-make-untimed1} s 0 = \text{hd} (s (\text{LEAST} i.(s i) \neq [])) \mid \text{inf-make-untimed1-Suc:} \text{inf-make-untimed1} s (\text{Suc} k) = \]
\[ \quad (\text{if} ((\text{Suc} k) < \text{length} (s 0)) \]
\[ \quad \quad \text{then} \text{nth} (s 0) (\text{Suc} k) \]
\[ \quad \quad \text{else} (\text{if} (s 0) = []) \]
\[ \quad \quad \text{then} (\text{inf-make-untimed1} (\text{inf-tl} (\text{inf-drop} \]
\[ \quad \quad \quad (\text{LEAST} i. \forall j. j < i \rightarrow (s j) = []) \]
\[ \quad \quad \quad s)) k) \]
\[ \quad \quad \text{else} \text{inf-make-untimed1} (\text{inf-tl} s) k) \]

— 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**  
\[ \text{inf-make-untimed} s \equiv \lambda i. \text{inf-make-untimed1} s i \]

— making a (general) stream untimed

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

**where**
make-untimed s ≡
case s of
  (FinT x) ⇒ FinU (fin-make-untimed x)
  | (FinU x) ⇒ FinU x
  | (InfT x) ⇒
    (if (∃ i. ∀ j. i < j → (x j) = [])
    then FinU (fin-make-untimed (inf-truncate x
    (LEAST i. ∀ j. i < j → (x j) = [])))
    else InfU (inf-make-untimed x))
  | (InfU x) ⇒ InfU x

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

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

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

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

— removing duplications from a finite timed stream
definition
finT-remdups :: 'a fstream ⇒ 'a fstream
where
\[ \text{finT-remdups } s \equiv \text{map (} \lambda \text{. } \text{remdups } s \text{)} \text{ s} \]
— removing duplications from an infinite timed stream
definition
infT-remdups :: 'a istream => 'a istream
where
infT-remdups s \equiv (\lambda i. \text{ remdups } (s \text{ i})))
— removing duplications from a time interval of a stream
primrec
fst-remdups :: 'a list \Rightarrow 'a list
where
\[
\text{fst-remdups } [] = [] \mid
\text{fst-remdups } (x \# xs) =
\begin{cases} 
[x] & \text{if } xs = [] \\
\text{else } \text{if } x = (hd \text{ xs}) \\
\text{then } \text{fst-remdups } xs \\
\text{else } (x \# xs) 
\end{cases}
\]
— time interval operator
definition
ti :: 'a fstream \Rightarrow nat \Rightarrow 'a list
where
ti s i \equiv
\begin{cases} 
[] & \text{if } s = [] \\
\text{nth } s \text{ i} & \text{else}
\end{cases}
— insuring that a sheaf of channels is correctly defined
definition
CorrectSheaf :: nat \Rightarrow bool
where
CorrectSheaf n \equiv 0 < n
— insuring that all channels in a sheaf are disjunct
— indices in the sheaf are represented using an extra specified set
definition
inf-disjS :: 'b set \Rightarrow ('b \Rightarrow 'a istream) \Rightarrow bool
where
inf-disjS IdSet nS
\equiv \forall (t::nat) \ i \ j. (i:IdSet) \land (j: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
inf-disj :: nat ⇒ (nat ⇒ 'a istream) ⇒ bool

where
inf-disj n nS
≡ ∀ (t::nat) (i::nat) (j::nat).
  i < n ∧ j < n ∧ i ≠ j ∧ ((nS i) t) ≠ [] →
  ((nS j) t) = []

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

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

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

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

— taking the prefix of n data elements from an infinite timed stream
— (defined over natural numbers)
fun infT-get-prefix :: ('a istream × nat) ⇒ 'a fstream
where
  infT-get-prefix(s, 0) = [] |
  infT-get-prefix(s, Suc i) =
    ( if (s 0) = []
        then ( if (∀ i. s i = [])
       |
then \[\]\[else (let
k = (LEAST k. s k \neq [] \land (\forall i. i < k \rightarrow 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 \neq 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) \neq 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 \Rightarrow 'a) \Rightarrow nat \Rightarrow '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 \Rightarrow natInf \Rightarrow 'a stream
where
infT-get-prefix-plus s n
≡
case n of (Fin i) \Rightarrow FinT (infT-get-prefix(s, i))
| \infty \Rightarrow InfT s

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

— taking the prefix of n data elements from an infinite stream
— (defined over natural numbers enriched by Infinity)
definition
take-plus :: natInf ⇒ 'a list ⇒ 'a list

where
take-plus n s
≡
case n of (Fin i) ⇒ (take i s)
| ∞ ⇒ s

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

definition
get-prefix :: 'a stream ⇒ natInf ⇒ 'a stream

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

— merging time intervals of two finite timed streams

primrec
fin-merge-ti :: 'a fstream ⇒ 'a fstream ⇒ 'a fstream

where
fin-merge-ti [] y = y |
fin-merge-ti (x#xs) y =
( case y of [] ⇒ (x#xs)
| (z#zs) ⇒ (x•z) # (fin-merge-ti xs zs))

— merging time intervals of two infinite timed streams

definition
inf-merge-ti :: 'a istream ⇒ 'a istream ⇒ 'a istream

where
inf-merge-ti x y
≡
λ i. (x i)•(y i)

— the last time interval of a finite timed stream

primrec
fin-last-ti :: ('a list) list ⇒ nat ⇒ 'a list

where
fin-last-ti s 0 = hd s |
fin-last-ti s (Suc i) =
( if s!(Suc i) ≠ []
  then s!(Suc i)
  else fin-last-ti s i)

— the last nonempty time interval of a finite timed stream
— (can be applied to the streams which time intervals are empty from some moment)
inf-last-ti :: 'a istream ⇒ nat ⇒ 'a list

where
inf-last-ti s 0 = s 0 |
inf-last-ti s (Suc i) =
  ( if s (Suc i) ≠ []
    then s (Suc i)
    else inf-last-ti s i)

5.3 Properties of operators

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

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

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

lemma inf-last-ti-Suc2:
assumes dt (Suc t) ≠ [] ∨ dt (Suc (Suc t)) ≠ []
shows inf-last-ti dt (t + 2 + k) ≠ []
proof (cases dt (Suc t) ≠ [])
assume a1: dt (Suc t) ≠ []
from a1 have sg2: inf-last-ti dt ((Suc t) + (Suc k)) ≠ []
  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 assms 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)

33
lemma correct-fin-inf-append1:
  assumes s1 = fin-inf-append [x] s
  shows  s1 (Suc i) = s i
  using assms
  by (simp add: fin-inf-append-def)

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

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

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

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

5.3.2 Lemmas for operators ts and msg

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

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

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

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

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

lemma \( inf\text{-}tl\text{-}Suc \): \( inf\text{-}tl \ x \ i = x \ (Suc \ i) \)
by \((simp \ add: \ inf\text{-}tl\text{-}def)\)

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

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

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

using assms
by (metis inf-make-untimed-def ts-inf-make-untimed-inf-tl)

**Lemma** msg-nonempty1:
- assumes \( h1:msg(Suc\;0)\;a \)
  - and \( h2:a\;t\;=\;aa\;\#\;l \)
- shows \( l = [] \)

**Proof** –
- from \( h1 \) have \( \text{length (a t) \leq Suc\;0} \) by \( \text{(simp add: msg-def)} \)
- from \( h2 \) and this show \( \text{thesis by auto} \)

**qed**

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

**Proof** –
- from \( h1 \) have \( \text{sg1:length (a t) \leq Suc\;0} \) by \( \text{(simp add: msg-def)} \)
- from \( h2 \) have \( \text{sg2:length (a t) \neq 0} \) by \( \text{auto} \)
- from \( \text{sg1 and sg2 show \( \text{thesis by arith} \)} \)

**qed**

**5.3.3 Lemmas for inf_truncate**

**Lemma** inf-truncate-nonempty:
- assumes \( z\;i\;\neq\;[] \)
- shows \( \text{inf-truncate z i \neq \[]} \)

**Proof (induct i)**
- case \( 0 \)
  - from assms show \( \text{case by auto} \)
- next
  - case \( (Suc\;i) \)
  - from assms show \( \text{case by auto} \)

**qed**

**Lemma** concat-inf-truncate-nonempty:
- assumes \( z\;i\;\neq\;[] \)
- shows \( \text{concat (inf-truncate z i) \neq \[]} \)

**Using assms**
**Proof (induct i)**
- case \( 0 \)
  - thus \( \text{case by auto} \)
- next
  - case \( (Suc\;i) \)
  - thus \( \text{case by auto} \)

**qed**
lemma concat-inf-truncate-nonempty-a:
assumes \( z \cdot i = [a] \)
shows \( \text{concat (inf-truncate z i)} \neq [] \)
using assms
by (metis concat-inf-truncate-nonempty list.distinct(1))

lemma concat-inf-truncate-nonempty-el:
assumes \( z \cdot i \neq [] \)
shows \( \text{concat (inf-truncate z i)} \neq [] \)
using assms
by (metis concat-inf-truncate-nonempty)

lemma inf-truncate-append:
\( (\text{inf-truncate z i} \bullet [z \cdot (\text{Suc i})]) = \text{inf-truncate z (Suc i)} \)
using assms
by (metis inf-truncate.simps(2))

5.3.4 Lemmas for fin_make_untimed

lemma fin-make-untimed-append:
assumes \( \text{fin-make-untimed} x \neq [] \)
shows \( \text{fin-make-untimed} (x \cdot y) \neq [] \)
using assms
by (simp add: fin-make-untimed_def)

lemma fin-make-untimed-inf-truncate-Nonempty:
assumes \( z \cdot k \neq [] \)
and \( k \leq i \)
shows \( \text{fin-make-untimed (inf-truncate z i)} \neq [] \)
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 \( k \leq i \)
  from Suc and this show \( \exists xs \in \text{set (inf-truncate z (Suc i))}. xs \neq [] \)
  by auto
next
  assume \( \neg k \leq i \)
  from Suc and this have \( k = \text{Suc i} \) by arith
  from Suc and this show \( \exists xs \in \text{set (inf-truncate z (Suc i))}. xs \neq [] \)
  by auto
qed
qed

37
lemma last-fin-make-untimed-append:
last (fin-make-untimed (z • [[a]])) = a
by (simp add: fin-make-untimed-def)

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

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

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

lemma fin-make-untimed-inf-truncate-Nonempty-all:
assumes z k ≠ []
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 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 z 0 = [a]
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-all-app:
assumes z 0 = [a]
shows ∀ i. fin-make-untimed (inf-truncate z i • [z (Suc i)]) ≠ []
proof
fix i
from assms have fin-make-untimed (inf-truncate z i) ≠ []
  by (simp add: fin-make-untimed-inf-truncate-Nonempty-all0a)
from assms and this show
  fin-make-untimed (inf-truncate z i • [z (Suc i)]) ≠ []
  by (simp add: fin-make-untimed-append)
qed
lemma fin-make-untimed-nth-length:
  assumes z i = [a]
  shows fin-make-untimed (inf-truncate z i)! (length (fin-make-untimed (inf-truncate z i)) - Suc 0) = a
proof -
from assms have sq1: last (fin-make-untimed (inf-truncate z i)) = a
  by (simp add: last-fin-make-untimed-inf-truncate)
from assms have sq2: concat (inf-truncate z i) ≠ []
  by (rule concat-inf-truncate-nonempty-a)
from assms and sq1 and sq2 show ?thesis
  by (simp add: fin-make-untimed-def last-nth-length)
qed

5.3.5 Lemmas for inf_disj and inf_disjS

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

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

end

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

theory BitBoolTS
imports Main stream
begin
datatype bit = Zero | One

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

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

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

from this and $sg1$ and $\text{Cons}$ show ?thesis by auto

next

assume $a = \text{One}$

from this and $sg1$ and $h2$ and $\text{Cons}$ show ?thesis by auto

qed

qed

lemma $\text{ts-bool-True}$:

assumes $h1$: $\text{ts \, x}$

and $h2$: $x \cdot i \neq [\text{False}]$

shows $x \cdot i = [\text{True}]$

proof

from $h1$ have $sg1$: $\text{length} \ (x \cdot i) = \text{Suc} \ 0$

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

from this and $h2$ show ?thesis

proof (cases $x \cdot i$)

assume $\text{Nil}$: $x \cdot i = []$

from this and $sg1$ show ?thesis by simp

next

fix $a \ l$ assume $\text{Cons}$: $x \cdot i = a \ # \ l$

from this and $sg1$ have $x \cdot i = [a]$ by simp

from this and $h2$ show ?thesis by auto

qed

qed

lemma $\text{ts-bool-False}$:

assumes $h1$: $\text{ts \, x}$

and $h2$: $x \cdot i \neq [\text{True}]$

shows $x \cdot i = [\text{False}]$

proof

from $h1$ have $sg1$: $\text{length} \ (x \cdot i) = \text{Suc} \ 0$

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

from this and $h2$ show ?thesis

proof (cases $x \cdot i$)

assume $\text{Nil}$: $x \cdot i = []$

from this and $sg1$ show ?thesis by simp

next

fix $a \ l$ assume $\text{Cons}$: $x \cdot i = a \ # \ l$

from this and $sg1$ have $x \cdot i = [a]$ by simp

from this and $h2$ show ?thesis by auto

qed

qed

lemma $\text{ts-bool-True-False}$:

fixes $x$: bool istream

assumes $\text{ts \, x}$

shows $x \cdot i = [\text{True}] \lor x \cdot i = [\text{False}]$
proof (cases x i = [True])
  assume x i = [True]
  from this and assms show ?thesis by simp
next
  assume x i ≠ [True]
  from this and assms show ?thesis by (simp add: ts-bool-False)
qed

7 Changing time granularity of the streams

theory JoinSplitTime
imports stream arith-hints
begin

7.1 Join time units

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

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

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

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

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

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

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

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

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

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

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

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

43
from $a_2$ and $h1$ have $i \leq x$ by simp
from this and $sg1$ and $a2$ show $\theta$thesis by (simp add: join-ti-empty-join)
qed

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

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 $i \leq n$
  and $\text{fin-join-ti } s \ x \ n = []$
  shows $\text{nth } s \ (x+i) = []$
  using assms
  proof (induct $n$)
    case 0 then show $\theta$case by auto
  next
    case (Suc $n$) then show $\theta$case
      proof (cases $i = \text{Suc } n$)
        assume $i = \text{Suc } n$
        from Suc and this show $\theta$thesis by simp
      next
        assume $i \neq \text{Suc } n$
        from Suc and this show $\theta$thesis by simp
      qed
  qed

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

lemma fin-join-ti-1nempty:
  assumes $\forall \ i. \ 0 < i \wedge i < \text{Suc } n \rightarrow \text{nth } s \ (x+i) = []$
  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-time1; \∀ t. split-time s 1 t = s t
by (simp add: split-time-def)

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

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

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

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

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

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

lemma split-time-hint1:
  assumes n = Suc m
  shows split-time s (Suc n) (n * i + i + n) = []
proof –
  have sg1: i + n * i + n = (Suc n) * i + n by simp
  have sg2: n < Suc n by simp
  from assms have sg3: 0 < n by simp
  from sg2 and sg3 have sg4: split-time s (Suc n) (Suc n * i + n) = []
    by (rule split-time-empty)
  from sg1 and sg4 show ?thesis by auto
qed

7.3 Duality of the split and the join operators

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

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

definition
ControlSystem :: nat istream ⇒ bool
where
ControlSystem s ≡
(t 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
tns y ∧ tns s ∧ y = s ∧
(∀ (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 istream ⇒ bit istream ⇒ bit istream ⇒ bool

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

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

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

9 Steam Boiler System: Verification

theory SteamBoiler-proof
imports SteamBoiler
begin

9.1 Properties of the Boiler Component

lemma L1-Boiler:
  assumes SteamBoiler x s y
  and ts x
  shows ts s
using assms by (simp add: SteamBoiler-def)
lemma L2-Boiler:
assumes \( \text{SteamBoiler } x s y \)
and \( ts x \)
shows \( ts y \)
using assms by (simp add: SteamBoiler-def)

lemma L3-Boiler:
assumes \( \text{SteamBoiler } x s y \)
and \( ts x \)
shows \( 200 \leq \text{hd}(s 0) \)
using assms by (simp add: SteamBoiler-def)

lemma L4-Boiler:
assumes \( \text{SteamBoiler } x s y \)
and \( ts x \)
shows \( \text{hd}(s 0) \leq 800 \)
using assms by (simp add: SteamBoiler-def)

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

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

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

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

9.2 Properties of the Controller Component

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

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

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

50
lemma L3-Controller-Zero:
  assumes Controller-L y (fin-inf-append [Zero] l) l z
  and \( l \cdot t = \text{Zero} \)
  and \( \neg 300 < \text{hd} \ (y \ (\text{Suc} \ t)) \)
  shows \( z \ (\text{Suc} \ t) = [\text{One}] \)
using assms
by (metis Controller-L-def correct-fin-inf-append1)

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

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

next
assume \( a22 : \neg (y \text{Suc} t) < 700 \)
from \( a22 \) and \( sg6 \)
have \( sg8 : z (\text{Suc} t) = [\text{Zero}] \land l (\text{Suc} t) = \text{Zero} \)
by simp
from this and \( a2 \) show \(?thesis\) by simp

qed

qed

lemma \(L4\text{-Controller-One} :\)

assumes \( h1 : \text{Controller-L} y (\text{fin-inf-append} [\text{Zero}] l) z \)
and \( h2 : l (\text{Suc} t) = \text{One} \)

shows \( (z (\text{Suc} t) = [\text{Zero}] \land l t = \text{One}) \lor (z (\text{Suc} t) = [\text{One}] \land l t = \text{Zero}) \)

proof (cases \( l t \))
assume \( a1 : l t = \text{Zero} \)
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 \( 300 < \text{hd} (y \text{Suc} t) \)
then \( z (\text{Suc} t) = [\text{Zero}] \land l (\text{Suc} t) = \text{Zero} \)
else \( z (\text{Suc} t) = [\text{One}] \land l (\text{Suc} t) = \text{One} \)
by (simp add: Controller-L-def)

show \(?thesis\)

proof (cases \( 300 < \text{hd} (y \text{Suc} t) \))
assume \( a11 : 300 < \text{hd} (y \text{Suc} t) \)
from \( a11 \) and \( sg2 \)
have \( sg3 : z (\text{Suc} t) = [\text{Zero}] \land l (\text{Suc} t) = \text{Zero} \)
by simp
from this and \( h2 \) show \(?thesis\) by simp

next
assume \( a12 : \neg 300 < \text{hd} (y \text{Suc} t) \)
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 \( a2 : l t = \text{One} \)
from this and \( h1 \) and \( h2 \) show \(?thesis\)

proof –
from \( a2 \)
have \( sg5 : \text{fin-inf-append} [\text{Zero}] l (\text{Suc} t) \neq \text{Zero} \)

52
by (simp add: correct-fin-inf-append1)

from h1 and sg5 have sg6:
  if hd (y (Suc t)) < 700
    then z (Suc t) = [] ∧ l (Suc t) = One
    else z (Suc t) = [Zero] ∧ l (Suc t) = Zero
  by (simp add: Controller-L-def)

show ?thesis
proof (cases hd (y (Suc t)) < 700)
  assume a21:hd (y (Suc t)) < 700
  from a21 and sg6 have sg7:z (Suc t) = [] ∧ l (Suc t) = One by simp
  from this and a2 show ?thesis by simp

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

qed

lemma L5-Controller-Zero:
  assumes h1:Controller-L y lIn lOut z
      and h2:lOut t = Zero
      and h3:z t = []
  shows lIn t = Zero
proof (cases lIn t)
  assume a1:lIn t = Zero
  from this show ?thesis by simp

next
  assume a2:lIn t = One
  from a2 and h1 have sg1:
    if hd (y t) < 700
      then z t = [] ∧ lOut t = One
      else z t = [Zero] ∧ lOut t = Zero
    by (simp add: Controller-L-def)

  show ?thesis
  proof (cases hd (y t) < 700)
    assume a3:hd (y t) < 700
    from a3 and sg1 have z t = [] ∧ lOut t = One by simp
    from this and h2 show ?thesis by simp

next
  assume a4:¬ hd (y t) < 700
  from a4 and sg1 have z t = [Zero] ∧ lOut t = Zero by simp
  from this and h3 show ?thesis by simp

qed

lemma L5-Controller-One:
  assumes h1:Controller-L y lIn lOut z
      and h2:lOut t = One
  shows lIn t = One

qed
and \( h3 \): \( t = [] \)

shows \( lIn \ t = \)

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

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

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

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

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

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

lemma L7-Controller-One-0:
  assumes \( Controller-L y \) \((\text{fin-inf-append} [\text{Zero}] l) \) \( l z \)
  and \( y \ 0 = [500::\text{nat}] \)
shows \( l 0 = \text{Zero} \)
proof (rule ccontr)
  assume a1: \( \neg l 0 = \text{Zero} \)
  from assms have sg1: \( z 0 = [\text{Zero}] \)
    by (simp add: Controller-L-def)
  have sg2: \( \text{fin-inf-append} [\text{Zero}] l (0::\text{nat}) = \text{Zero} \)
    by (simp add: fin-inf-append-def)

55
lemma L7-Controller-One:
  assumes h1: Controller-L y (fin-inf-append [Zero] l) l z
  and h2: l t = One
  and h3: y 0 = [500::nat]
  shows last (fin-make-untimed (inf-truncate z t)) = One
using assms
proof (induct t)
case 0
  from h1 and h3 have l 0 = Zero
  by (simp add: L7-Controller-One-l0)
from this and 0 show ?case by simp
next
  fix t
  case (Suc t)
  from this show ?case
proof (cases l t)
  assume a1: l t = Zero
  from Suc have
    (z (Suc t) = [] ∧ l t = One) ∨ (z (Suc t) = [One] ∧ l t = Zero)
  by (simp add: L4-Controller-One)
  from this and a1 have z (Suc t) = [One]
  by simp
  from Suc and this and a1 show ?case
  by (simp add: fin-make-untimed-def)
next
  assume a1: l t = One
  from Suc have
    (z (Suc t) = [] ∧ l t = One) ∨ (z (Suc t) = [One] ∧ l t = Zero)
  by (simp add: L4-Controller-One)
  from this and a1 have z (Suc t) = []
  by simp
  from a1 and Suc and this show ?case
  by (simp add: fin-make-untimed-def)
qed

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

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

proof (cases fin-inf-append [Zero] \( lt = \text{Zero} \))
assume a1: fin-inf-append [Zero] \( lt = \text{Zero} \)

from a1 and assms have sg1:
if 300 < hd (\( y t \))
then $z t = \[] \land l t = \text{Zero}$
else $z t = [\text{One}] \land l t = \text{One}$
by (simp add: Controller-L-def)
show ?thesis

proof (cases 300 < hd (\( y t \)))
assume a11: 300 < hd (\( y t \))
from a11 and sg1 show ?thesis by simp
next
assume a12: ¬ 300 < hd (\( y t \))
from a12 and sg1 show ?thesis by simp
qed

next
assume a2: fin-inf-append [Zero] \( lt \neq \text{Zero} \)
from a2 and assms have sg2:
if hd (\( y t \)) < 700
then $z t = \[] \land l t = \text{One}$
else $z t = [\text{Zero}] \land l t = \text{Zero}$
by (simp add: Controller-L-def)
show ?thesis

proof (cases hd (\( y t \)) < 700)
assume a21: hd (\( y t \)) < 700
from a21 and sg2 show ?thesis by simp
next
assume a22: ¬ hd (\( y t \)) < 700
from a22 and sg2 show ?thesis by simp
qed

qed

lemma L9-Controller:
assumes h1: Controller-L $s$ (fin-inf-append [Zero] $l$) $l z$
and h2: fin-make-untimed (inf-truncate $z$ i)!
  (length (fin-make-untimed (inf-truncate $z$ i)) − Suc 0) = Zero
and h3: last (fin-make-untimed (inf-truncate $z$ i)) = l i
and h5: hd (s (Suc i)) = hd (s i) − r
and h6: fin-make-untimed (inf-truncate $z$ i) \( \neq \[] \)
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] \( li = \text{Zero} \))
assume a1: fin-inf-append [Zero] \( li = \text{Zero} \)
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 fin-inf-append [Zero] l i = Zero)
assume a1: fin-inf-append [Zero] l i = Zero
from a1 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)
lemma L10-Controller:
  assumes h1: Controller-L s (fin-inf-append [Zero] l) l z
          and h2: fin-make-untimed (inf-truncate z i) !
          (length (fin-make-untimed (inf-truncate z i)) = Suc 0) ≠ Zero
          and h3: last (fin-make-untimed (inf-truncate z i)) = l i
          and h5: hd (s (Suc i)) = hd (s i) + r
          and h6: fin-make-untimed (inf-truncate z i) ≠ []
          and h8: r ≤ 10
  shows hd (s (Suc i)) ≤ 800
proof
  from h6 and h2 and h3 have sg0: l i ≠ Zero
    by (simp add: last-nth-length)
  show thesis
  proof (cases fin-inf-append [Zero] l i = Zero)
    assume a1: fin-inf-append [Zero] l i = Zero
    from a1 and h1 have sg1:
      if 300 < hd (s i)
        then z i = [] ∧ l i = Zero
        else z i = [One] ∧ l i = One
    by (simp add: Controller-L-def)
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

9.3 Properties of the Converter Component

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

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

9.4 Properties of the System

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

lemma L2-ControlSystem:
assumes ControlSystemArch \( s \)
shows \((200::\text{nat}) \leq \text{hd} \ (s \ i)\)
proof –
from assms obtain \( x z y \)
  where \( a1: \text{Controller} \ y z \ \text{and} \ a2: \text{SteamBoiler} \ x s y \ \text{and} \ a3: \text{Controller} \ y z \)
  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 \( a1 \) 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 (simp add: L3-Boiler)
next
fix \( i \)
case (Suc \( i \))
from this obtain \( l \)
  where \( a4: \text{Controller-L} \ s \ (\text{fin-inf-append} \ [\text{Zero}] \ l) \ l z \)
  by (simp add: Controller-def, atomize, auto)
from Suc and \( a4 \) have \( sg4: \text{fin-make-untimed} \ (\text{inf-truncate} \ z \ i) \neq [] \)
  by (simp add: L1-Controller)
from \( a2 \) and \( sg1 \) have \( y0asm: y = [500::\text{nat}] \)
  by (simp add: SteamBoiler-def)
from Suc and \( a4 \) and \( sg4 \) and \( y0asm \) have \( sg5: \text{last} \ (\text{fin-make-untimed} \ (\text{inf-truncate} \ z \ i)) = l i \)
  by (simp add: L7-Controller)
from \( a2 \) and \( sg1 \) obtain \( r \) where
  \begin{align*}
  a00: & 0 < r \text{ and} \\
  a01: & r \leq 10 \text{ and} \\
  a02: & \text{hd} \ (s \ (\text{Suc} \ i)) = (\text{if} \ \text{hd} \ (x \ i) = \text{Zero} \ \text{then} \ \text{hd} \ (s \ i) - r \ \text{else} \ \text{hd} \ (s \ i) + r) \text{ by (simp add: SteamBoiler-def, auto)}
  \end{align*}
from Suc and \( a4 \) and \( sg4 \) and \( sg5 \) show \( \text{?case} \)
proof (cases \( \text{hd} \ (x \ i) = \text{Zero} \))
assume \( a0Zero: \text{hd} \ (x \ i) = \text{Zero} \)
from \( a1 \) and \( sg4 \) and this have
  \( sg7: (\text{fin-make-untimed} \ (\text{inf-truncate} \ z \ i)) = ((\text{length} \ (\text{fin-make-untimed} \ (\text{inf-truncate} \ z \ i))) - \text{Suc} \ 0) = \text{Zero} \)
by (simp add: L1-Converter)
from aa2 and aaZero have sg8: hd (s (Suc i)) = hd (s i) − r by simp
from a4 and sg7 and sg5 and sg8 and sg4 and aa1 show ?thesis
  by (rule L9-Controller)
next
  assume aaOne: hd (x i) ≠ Zero
  from a1 and sg4 and this have
    sg7: (fin-make-untimed (inf-truncate z i)) !
    ((length (fin-make-untimed (inf-truncate z i))) - Suc 0) ≠ Zero
    by (simp add: L1-Controller)
from aa2 and aaOne have sg9: hd (s (Suc i)) = hd (s i) + r by simp
from Suc and this show ?thesis by simp
qed
qed
qed

lemma L3-ControlSystem:
  assumes ControlSystemArch s
  shows hd (s i) ≤ (800::nat)
proof
  from assms obtain x z y
    where a1: Converter z x and a2: Boiler 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: Boiler-def)
from sg1 and a2 have sg3: y = s by (simp add: Boiler-def)
from a1 and a2 and a3 and sg1 and sg2 and sg3 show hd (s i) ≤ (800::nat)
proof (induction i)
  case 0
  from this show ?case by (simp add: L4-Boiler)
next
  fix i
  case (Suc i)
  from this obtain l
    where a4: Controller-L s (fin-inf-append [Zero] l) l z
      by (simp add: Controller-def, atomize, auto)
from a4 have sg4: fin-make-untimed (inf-truncate z i) ≠ []
  by (simp add: L1-Controller)
from a2 and sg1 have y0asm: y 0 = [500::nat] by (simp add: Boiler-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: Boiler-def, auto)

from this and Suc and a4 and sg4 and sg5 show ?case
proof (cases hd (x i) = Zero)
  assume aaZero: hd (x i) = Zero
  from a1 and sg4 and this have
    sg7: (fin-make-untimed (inf-truncate z i)) !
    ((length (fin-make-untimed (inf-truncate z i))) - Suc 0) = Zero
    by (simp add: L1-Converter)
  from aa2 and aaZero have sg8: hd (s (Suc i)) = hd (s i) - r by simp
  from this and Suc show ?thesis by simp
next
  assume aaOne: hd (x i) ≠ Zero
  from a1 and sg4 and this have
    sg7: (fin-make-untimed (inf-truncate z i)) !
    ((length (fin-make-untimed (inf-truncate z i))) - Suc 0) ≠ Zero
    by (simp add: L1-Converter)
  from aa2 and aaOne have sg9: hd (s (Suc i)) = hd (s i) + r by simp
  from a4 and sg7 and sg5 and sg4 and aa1 show ?thesis
  by (rule L10-Controller)
qed
qed
qed

9.5 Proof of the Refinement Relation

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

10 FlexRay: Types

theory FR-types
imports stream
begin

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

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

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

62
type-synonym 'a nFrame = nat ⇒ ('a Frame) istream

type-synonym nNat = nat ⇒ nat istream

type-synonym nConfig = nat ⇒ Config

conds 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)) ))

— 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)
→
(FrameTransmission n nStore nReturn nGet nC) ∧
(∀ (i::nat). i < n → (msg 1 (nGet i)) ∧ (msg 1 (nStore i))))

definition
Cable :: nat ⇒ 'a nFrame ⇒ 'a Frame istream ⇒ bool
where
Cable n nSend recv
≡
(CorrectSheaf n)
∧
(((inf-disj n nSend) → (Broadcast n nSend recv))

definition
Scheduler :: Config ⇒ nat istream ⇒ bool
where
Scheduler c activation
≡
∀ (t::nat).
(let s = (t mod (cycleLength c))
in
( if (s mem (schedule c))
then (activation t) = [s]
else (activation t) = [] )

definition
BusInterface ::
nat istream ⇒ 'a Frame istream ⇒ 'a Frame istream ⇒
'a Frame istream ⇒ 'a Frame istream ⇒ nat istream ⇒ bool
where
BusInterface activation return recv store send get
≡
(Receive recv store activation) ∧
(Send return send get activation)

**definition**

*FlexRayController*::

'a Frame stream ⇒ 'a Frame stream ⇒ Config ⇒
'a Frame stream ⇒ 'a Frame stream ⇒ nat stream ⇒ bool

**where**

*FlexRayController* return recv c store send get

≡

(∃ activation.

(Scheduler c activation) ∧

(BusInterface activation return recv store send get))

**definition**

*FlexRayArchitecture*::

nat ⇒ 'a nFrame ⇒ nConfig ⇒ 'a nFrame ⇒ nNat ⇒ bool

**where**

*FlexRayArchitecture* n nReturn nC nStore nGet

≡

(CorrectSheaf n) ∧

(∃ nSend recv.

(Cable n nSend recv) ∧

(∀ (i:nat). i < n →

  *FlexRayController* (nReturn i) recec (nC i)

  (nStore i) (nSend i) (nGet i)))

**definition**

*FlexRayArch*::

nat ⇒ 'a nFrame ⇒ nConfig ⇒ 'a nFrame ⇒ nNat ⇒ bool

**where**

*FlexRayArch* n nReturn nC nStore nGet

≡

(CorrectSheaf n) ∧

((∀ (i:nat). i < n → msg 1 (nReturn i)) ∧

(DisjointSchedules n nC) ∧ (IdenticalCycleLength n nC)

→

(*FlexRayArchitecture* n nReturn nC nStore nGet))

**end**

12 **FlexRay: Verification**

theory *FR-proof*

imports *FR*

begin
12.1 Properties of the function Send

`lemma Send-L1:`

`assumes` `Send return send get activation`

`and` `send t ≠ []`

`shows` `(activation t) ≠ []`

`using` `assms` `by (simp add: Send-def, auto)`

`lemma Send-L2:`

`assumes` `Send return send get activation`

`and` `(activation t) ≠ []`

`and` `return t ≠ []`

`shows` `(send t) ≠ []`

`using` `assms` `by (simp add: Send-def)

12.2 Properties of the component Scheduler

`lemma Scheduler-L1:`

`assumes` `h1: Scheduler C activation`

`and` `h2: (activation t) ≠ []`

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

`using` `assms` `proof –`

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

`from` `h1` `have`

`if` `t mod cycleLength C mem schedule C`

`then` `activation t = [t mod cycleLength C]`

`else` `activation t = []`

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

`from` `a1` `and` `this` `have` `activation t = []` `by` `simp`

`from` `this` `and` `h2` `have` `sg3:False` `by` `simp`

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

`from` `this` `show` `?thesis` `by` `simp`

`qed`

`lemma Scheduler-L2:`

`assumes` `Scheduler C activation`

`and` `¬ (t mod cycleLength C) mem (schedule C)`

`shows` `activation t = []`

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

`lemma Scheduler-L3:`

`assumes` `Scheduler C activation`

`and` `(t mod cycleLength C) mem (schedule C)`

`shows` `activation t ≠ []`

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

`lemma Scheduler-L4:`

`assumes` `Scheduler C activation`

`and` `(t mod cycleLength C) mem (schedule C)`

67
shows activation t = [t mod cycleLength C]
using assms by (simp add: Scheduler-def Let-def)

lemma correct-DisjointSchedules1:
assumes h1:DisjointSchedules n nC
and h2:IdenticCycleLength n nC
and h3:(t mod cycleLength (nC i)) mem schedule (nC i)
and h4:i < n
and h5:j < n
and h6:i ≠ j
shows ¬(t mod cycleLength (nC j) mem schedule (nC j))
proof –
from h1 and h4 and h5 and h6 have sg1:disjoint (schedule (nC i)) (schedule (nC j))
  by (simp add: DisjointSchedules-def)
from h2 and h4 and h5 have sg2:cycleLength (nC i) = cycleLength (nC j)
  by (metis IdenticCycleLength-def)
from sg1 and h3 have sg3:¬(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: BusInterface-def)

from sg3 and sg6 have sg7: cycleLength (nC i) = cycleLength (nC j)
  by (metis IdenticCycleLength-def)

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 a1 have sg11:Send (nReturn j) (nSend j) (nGet j) activation2
  by (simp add: BusInterface-def)

from sg11 and sg10 show ?thesis by (simp add: Send-def)

qed

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

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

12.4 Properties of the sheaf of channels nSend

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

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

lemma fr-Send3:
assumes ∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i)
(nGet i)
  and DisjointSchedules n nC
  and IdenticCycleLength n nC
  and t mod cycleLength (nC k) mem schedule (nC k)
  and k < n
  and nReturn k t ≠ []
shows nSend k t ≠ []
using assms by (simp add: fr-Send2)

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

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

qed

lemma fr-Send6:
assumes \( \forall i < n. \text{FlexRayController} (n \text{Return } i) \text{ recv} (n \text{C } i) (n \text{Store } i) (n \text{Send } i) (n \text{Get } i) \)
  and \( \text{DisjointSchedules } n \text{C} n \text{C} \)
  and \( \text{IdenticCycleLength } n \text{C } n \text{C} \)
  and \( t \text{ mod cycleLength} (n \text{C } k) \text{ mem schedule} (n \text{C } k) \)
  and \( k < n \)
  and \( \text{nReturn } k t \neq [] \)
shows \( \exists k < n. n \text{Send } k t \neq [] \)
using assms
by (metis fr-Send3)

lemma fr-Send7:
assumes \( \forall i < n. \text{FlexRayController} (n \text{Return } i) \text{ recv} (n \text{C } i) (n \text{Store } i) (n \text{Send } i) (n \text{Get } i) \)
  and \( \text{DisjointSchedules } n \text{C } n \text{C} \)
  and \( \text{IdenticCycleLength } n \text{C } n \text{C} \)
  and \( t \text{ mod cycleLength} (n \text{C } k) \text{ mem schedule} (n \text{C } k) \)
  and \( k < n \)
  and \( j < n \)
  and \( \text{nReturn } k t = [] \)
shows \( n \text{Send } j t = [] \)
using assms
by (metis (full-types) disjointFrame-L3 fr-Send2)

lemma fr-Send8:
assumes \( \forall i < n. \text{FlexRayController} (n \text{Return } i) \text{ recv} (n \text{C } i) (n \text{Store } i) (n \text{Send } i) (n \text{Get } i) \)
  and \( \text{DisjointSchedules } n \text{C } n \text{C} \)
  and \( \text{IdenticCycleLength } n \text{C } n \text{C} \)
  and \( t \text{ mod cycleLength} (n \text{C } k) \text{ mem schedule} (n \text{C } k) \)
  and \( k < n \)
  and \( \text{nReturn } k t = [] \)
shows \( \neg (\exists k < n. n \text{Send } k t \neq []) \)
using assms by (auto, simp add: fr-Send7)

lemma fr-nC-Send:
assumes \( \forall i < n. \text{FlexRayController} (n \text{Return } i) \text{ recv} (n \text{C } i) (n \text{Store } i) (n \text{Send } i) (n \text{Get } i) \)
  and \( k < n \)
  and \( \text{DisjointSchedules } n \text{C } n \text{C} \)
  and \( \text{IdenticCycleLength } n \text{C } n \text{C} \)
  and \( t \text{ mod cycleLength} (n \text{C } k) \text{ mem schedule} (n \text{C } k) \)
shows \( \forall j. j < n \land j \neq k \rightarrow (n \text{Send } j) t = [] \)
using assms by (clarify, simp add: disjointFrame-L3)
lemma \textit{length-nSend}:  
assumes \( h1: \text{BusInterface activation (nReturn i) recv (nStore i) (nSend i) (nGet i)} \)  
and \( h2: \forall t. \text{length (nReturn i t)} \leq \text{Suc 0} \)  
shows \( \text{length (nSend i t)} \leq \text{Suc 0} \)  
proof  
from \( h1 \) have \( sg1: \text{Send (nReturn i) (nSend i) (nGet i) activation} \)  
by (simp add: BusInterface-def)  
from \( sg1 \) have \( sg2: \)  
if activation \( t = [] \) then \( nGet i t = [] \) \( \land \text{nSend i t} = [] \)  
else \( nGet i t = \text{activation t} \) \( \land \text{nSend i t} = \text{nReturn i t} \)  
by (simp add: Send-def)  
show ?thesis  
proof (cases activation \( t = [] \))  
assume a1: \( \text{activation t} = [] \)  
from \( sg2 \) and a1 show ?thesis by simp  
next  
assume a2: \( \text{activation t} \neq [] \)  
from \( h2 \) have \( sg3: \text{length (nReturn i t)} \leq \text{Suc 0} \) by auto  
from \( sg2 \) and a2 and \( sg3 \) show ?thesis by simp  
qed  
qed

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

lemma \textit{Broadcast-nSend-empty1}:  
assumes \( \text{Broadcast n nSend recv} \)  
and \( \text{h2: \forall k < n. nSend k t} = [] \)  
shows \( \text{recv t} = [] \)  
using \( \text{assms by (metis Broadcast-def)} \)

12.5 Properties of the sheaf of channels nGet

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

lemma fr-nGet1:
assumes \( \forall i < n. \) FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
  and \( t \mod \text{cycleLength} (nC k) \) mem schedule (nC k)
  and \( k < n \)
shows nGet k t = [t mod cycleLength (nC k)]
using assms
by (metis fr-nGet1a)

lemma fr-nGet2a:
assumes h1: FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)
  and h2: ¬ (t mod cycleLength (nC k) mem schedule (nC k))
shows nGet k t = []
proof –
  from h1 and h3 have sg1: FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
    (nGet i)
    and \( t \mod \text{cycleLength} (nC k) \) mem schedule (nC k)
    and \( k < n \)
  shows nGet k t = []
  proof –
  from h1 obtain activation1 where
    a1: Scheduler (nC k) activation1 and
    a2: BusInterface activation1 (nReturn k) recv (nStore k) (nSend k) (nGet k)
    by (simp add: FlexRayController-def, auto)
from a2 have sg2: Send (nReturn k) (nSend k) (nGet k) activation1
  by (simp add: BusInterface-def)
from sg2 have sg3:
  if activation1 t = [] then nGet k t = [] ∧ nSend k t = []
  else nGet k t = activation1 t ∧ nSend k t = nReturn k t
  by (simp add: Send-def)
from a1 and h2 have sg4: activation1 t = []
  by (simp add: Scheduler-L2)
from sg3 and sg4 show ?thesis by simp
qed

lemma fr-nGet2:
assumes h1: \( \forall i < n. \) 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 i) recv (nC i) (nStore i) (nSend i) (nGet i)
  by (simp add: BusInterface-def)
by auto

from sg1 and h2 show ?thesis by (rule fr-nGet2a)
qed

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

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

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

12.6 Properties of the sheaf of channels nStore

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

from \( h2 \) and \( h3 \) and \( h4 \) and \( h5 \) and \( h6 \) and \( a6 \) have \( sq19: \exists k < n. \ \text{nSend} \ k \ t \neq [] \)

by (simp add: fr-Send6)

from \( h2 \) and \( h3 \) and \( h4 \) and \( h5 \) and \( h6 \) and \( a6 \) have \( sq20: \text{nSend} \ k \ t \neq [] \)

by (simp add: fr-Send3)

from \( sq15 \) and \( sq16 \) have \( \text{recv} \ t = \text{nSend} \ ((\exists k. k < n \land \text{nSend} \ k \ t \neq [])) = k \)

by (simp add: inf-disj-index)

from \( sq15 \) and \( sq16 \) have \( \text{recv} \ t = \text{nSend} \ (\exists k. k < n \land \text{nSend} \ k \ t \neq []) \)

t by simp

from \( sq22 \) and \( sq21 \) have \( \text{recv} \ t = \text{nSend} \ k \ t \)

by simp

from \( h2 \) and \( h3 \) and \( h4 \) and \( h5 \) and \( h6 \)

have \( \text{nSend} \ k \ t = \text{nReturn} \ k \ t \)

by (simp add: fr-Send2)

lemma fr-nStore-nReturn2:

assumes \( h1 \): \( \text{Cable} \ n \ \text{nSend} \ \text{recv} \)

and \( h2 \): \( \forall i < n. \ \text{FlexRayController} \ (\text{nReturn} \ i) \ \text{recv} \ (\text{nC} \ i) \ (\text{nStore} \ i) \ (\text{nSend} \ i) \ (\text{nGet} \ i) \)

and \( h3 \): \( \text{DisjointSchedules} \ n \ \text{nC} \)

and \( h4 \): \( \text{IdenticalCycleLength} \ n \ \text{nC} \)

and \( h5 \): \( t \mod \text{cycleLength} \ (\text{nC} \ k) \ \text{mem} \ \text{schedule} \ (\text{nC} \ k) \)

and \( h6 \): \( k < n \)

and \( h7 \): \( j < n \)

and \( h8 \): \( j \neq k \)

shows \( \text{nStore} \ j \ t = \text{nReturn} \ k \ t \)

proof –

from \( h1 \) have \( sq1: \text{inf-disj} \ n \ \text{nSend} \ \rightarrow \text{Broadcast} \ n \ \text{nSend} \ \text{recv} \)

by (simp add: Cable-def)

from \( h3 \) and \( h4 \) and \( h2 \) have \( sq2: \text{inf-disj} \ n \ \text{nSend} \)

by (simp add: disjointFrame-L2)

from \( sq1 \) and \( sq2 \) have \( \text{sq3:Broadcast} \ n \ \text{nSend} \ \text{recv} \) by simp

from \( sq3 \) and \( sq2 \) and \( \text{assms} \) show \( \text{thesis} \) by (simp add: fr-nStore-nReturn1)

qed

lemma fr-nStore-empty1:

assumes \( h1 \): \( \text{Cable} \ n \ \text{nSend} \ \text{recv} \)

and \( h2 \): \( \forall i < n. \ \text{FlexRayController} \ (\text{nReturn} \ i) \ \text{recv} \ (\text{nC} \ i) \ (\text{nStore} \ i) \ (\text{nSend} \ i) \ (\text{nGet} \ i) \)

and \( h3 \): \( \text{DisjointSchedules} \ n \ \text{nC} \)

and \( h4 \): \( \text{IdenticalCycleLength} \ n \ \text{nC} \)

and \( h5 \): \( (t \mod \text{cycleLength} \ (\text{nC} \ k) \ \text{mem} \ \text{schedule} \ (\text{nC} \ k)) \)

and \( h6 \): \( k < n \)

shows \( \text{nStore} \ k \ t = [] \)

proof –

77
from h2 and h6 have sg1:
FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)
bys auto
from sg1 obtain activation1 where
a1: Scheduler (nC k) activation1 and
a2: BusInterface activation1 (nReturn k) recv (nStore k) (nSend k) (nGet k)
bys simp add: FlexRayController-def, auto
from a2 have sg2: Receive recv (nStore k) activation1
bys simp add: BusInterface-def
from this have sg3:
if activation1 t = [] then nStore k t = recv t else nStore k t = []
bys simp add: Receive-def
from a1 and h5 have sg4: activation1 t ≠ []
bys simp add: Scheduler-L3
from sg3 and sg4 show ?thesis bys simp
qed

lemma fr-nStore-nReturn3:
assumes Cable n nSend recv
and ∀ i<n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
and DisjointSchedules n nC
and IdenticalCycleLength n nC
and t mod cycleLength (nC k) mem schedule (nC k)
and k < n
shows ∀ j. j < n ∧ j ≠ k −→ nStore j t = nReturn k t
using assms
bys (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 = []
bys simp add: Broadcast-def
show ?thesis
proof (cases ∀ i<n. nSend k t ≠ [])
assumes ∀ i<n. nSend k t ≠ []
from this obtain k where a2:k<n and a3:nSend k t ≠ [] b ys auto
from h1 and a2 have
  FlexRayController (nReturn k) recv (nC k) (nStore k) (nSend k) (nGet k)
  by auto
then obtain activation1 where
  a4:Scheduler (nC k) activation1 and
  a5:BusInterface activation1 (nReturn k) recv (nStore k) (nSend k) (nGet k)
  by (simp add: FlexRayController-def, auto)
from a5 have sg5:Send (nReturn k) (nSend k) (nGet k) activation1
  by (simp add: BusInterface-def)
from a5 have sg6:Receive recv (nStore k) activation1
  by (simp add: BusInterface-def)
from sg5 and a3 have sg7:(activation1 t) ≠ [] by (simp add: Send-L1)
from sg6 have sg8:
  if activation1 t = []
    then nStore k t = recv t else nStore k t = []
  by (simp add: Receive-def)
from sg8 and sg7 have sg9:nStore k t = [] by simp
from a4 and sg7 have sg10:(t mod (cycleLength (nC k))) mem (schedule (nC k))
  by (simp add: Scheduler-L1)
show ?thesis
proof (cases i = k)
  assume i = k
  from sg9 and this show ?thesis by simp
next
  assume i ≠ k
from h7 and h4 and h1 and h2 and h3 and sg10 and a2 and h5 and
  this have sg11:
    nStore i t = nReturn k t
  by (simp add: fr-nStore-nReturn1)
from h6 and a2 have sg12:msg (Suc 0) (nReturn k) by auto
from a2 and h6 have sg13:length (nReturn k t) ≤ Suc 0
  by (simp add: msg-def)
from sg11 and sg13 show ?thesis by simp
qed
next
  assume ¬ (∃k<n. nSend k t ≠ [])
from h7 and this have sg14:recv t = [] by (simp add: Broadcast-nSend-empty1)

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

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

12.7 Refinement Properties

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

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

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

lemma FlexRayArch-nStore:
assumes h1: FlexRayArch n nReturn nC nStore nGet
    and h2: !i < n. msg (Suc 0) (nReturn i)
    and h3: DisjointSchedules n nC
    and h4: IdenticCycleLength n nC
    and h5: i < n
shows msg (Suc 0) (nStore i)
proof
  from assms obtain nSend recv where
    a1: Cable n nSend recv and
    a2: !i < n. FlexRayController (nReturn i) recv (nC i) (nStore i) (nSend i) (nGet i)
    by (simp add: FlexRayArch-def FlexRayArchitecture-def, auto)
  from h3 and h4 and a2 have sq1: inf-disj n nSend by (simp add: disjointFrame-L2)
  from a2 and h3 and h4 and sq1 and h5 and h2 and a1 show ?thesis
    by (rule msg-nStore)
qed
**13 Gateway: Types**

```plaintext
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**

```plaintext
theory Gateway
```
imports Gateway-types

begin

definition
  ServiceCenter ::
    ECall-Info istream ⇒ aType istream ⇒ bool
where
  ServiceCenter i a
  ≡
  ∀ (t::nat).
  a 0 = [] ∧ a (Suc t) = (if (i t) = [] then [] else [sc-ack])

definition
  Loss ::
    bool istream ⇒ aType istream ⇒ ECall-Info istream ⇒
    aType istream ⇒ ECall-Info istream ⇒ bool
where
  Loss lose a i2 a2 i
  ≡
  ∀ (t::nat).
  (if lose t = [False]
    then a2 t = a t ∧ i t = i2 t
    else a2 t = [] ∧ i t = [])

definition
  Delay ::
    aType istream ⇒ ECall-Info istream ⇒ nat ⇒
    aType istream ⇒ ECall-Info istream ⇒ bool
where
  Delay a2 i1 d a1 i2
  ≡
  ∀ (t::nat).
  (t < d −→ a1 t = [] ∧ i2 t = []) ∧
  (t ≥ d −→ (a1 t = a2 (t−d)) ∧ (i2 t = i1 (t−d)))

definition
  tiTable-SampleT ::
    reqType istream ⇒ aType istream ⇒
    stopType istream ⇒ bool istream ⇒
    (nat ⇒ GatewayStatus) ⇒ (nat ⇒ ECall-Info list) ⇒
    GatewayStatus istream ⇒ ECall-Info istream ⇒ vcType istream
    ⇒ (nat ⇒ GatewayStatus) ⇒ bool
where
  tiTable-SampleT req a1 stop lose st-in buffer-in
    ack i1 vc st-out
  ≡
  ∀ (t::nat)
  (r::reqType list) (x::aType list)
  (y::stopType list) (z::bool list).
(*1*)
(\(\text{st-in } t = \text{init-state} \land \text{req } t = [\text{init}]\))
\(\rightarrow\) \(\text{ack } t = [\text{call}] \land i1 t = [] \land \text{vc } t = []\)
\(\land\) \(\text{st-out } t = \text{call}\)

(*2*)
(\(\text{st-in } t = \text{init-state} \land \text{req } t \neq [\text{init}]\))
\(\rightarrow\) \(\text{ack } t = [\text{init-state}] \land i1 t = [] \land \text{vc } t = []\)
\(\land\) \(\text{st-out } t = \text{init-state}\)

(*3*)
(\(\text{st-in } t = \text{call} \lor (\text{st-in } t = \text{connection-ok} \land r \neq [\text{send}])\))
\(\land\) \(\text{req } t = r \land \text{lose } t = [\text{False}]\)
\(\rightarrow\) \(\text{ack } t = [\text{connection-ok}] \land i1 t = [] \land \text{vc } t = []\)
\(\land\) \(\text{st-out } t = \text{connection-ok}\)

(*4*)
(\(\text{st-in } t = \text{call} \lor \text{st-in } t = \text{connection-ok} \lor \text{st-in } t = \text{sending-data}\))
\(\land\) \(\text{lose } t = [\text{True}]\)
\(\rightarrow\) \(\text{ack } t = [\text{init-state}] \land i1 t = [] \land \text{vc } t = []\)
\(\land\) \(\text{st-out } t = \text{init-state}\)

(*5*)
(\(\text{st-in } t = \text{connection-ok} \land \text{req } t = [\text{send}]\))
\(\land\) \(\text{lose } t = [\text{False}]\)
\(\rightarrow\) \(\text{ack } t = [\text{send-data}] \land i1 t = \text{buffer-in } t \land \text{vc } t = []\)
\(\land\) \(\text{st-out } t = \text{send-data}\)

(*6*)
(\(\text{st-in } t = \text{send-data} \land a1 t = [] \land \text{lose } t = [\text{False}]\))
\(\rightarrow\) \(\text{ack } t = [\text{send-data}] \land i1 t = [] \land \text{vc } t = []\)
\(\land\) \(\text{st-out } t = \text{send-data}\)

(*7*)
(\(\text{st-in } t = \text{send-data} \land a1 t = [\text{sc-ack}]\))
\(\rightarrow\) \(\text{ack } t = [\text{voice-com}] \land i1 t = [] \land \text{vc } t = [\text{vc-com}]\)
\(\land\) \(\text{st-out } t = \text{voice-com}\)

(*8*)
(\(\text{st-in } t = \text{voice-com} \land \text{stop } t = [] \land \text{lose } t = [\text{False}]\))
\(\rightarrow\) \(\text{ack } t = [\text{voice-com}] \land i1 t = [] \land \text{vc } t = [\text{vc-com}]\)
\(\land\) \(\text{st-out } t = \text{voice-com}\)

(*9*)
(\(\text{st-in } t = \text{voice-com} \land \text{stop } t = [] \land \text{lose } t = [\text{True}]\))
\(\rightarrow\) \(\text{ack } t = [\text{voice-com}] \land i1 t = [] \land \text{vc } t = []\)
\(\land\) \(\text{st-out } t = \text{voice-com}\)

(*10*)
(\(\text{st-in } t = \text{voice-com} \land \text{stop } t = [\text{stop-vc}]\)

84
\[
\rightarrow \text{ack~} t = \text{[init-state]} \land i1~ t = [] \land vc~ t = [] \\
\land \text{st-out~} t = \text{init-state}
\]

definition
Sample-L ::
\[\begin{align*}
\text{reqType~} \text{istream} & \Rightarrow \text{ECall-Info~} \text{istream} \Rightarrow \text{aType~} \text{istream} \Rightarrow \\
\text{stopType~} \text{istream} & \Rightarrow \text{bool~} \text{istream} \Rightarrow \\
(\text{nat} \Rightarrow \text{GatewayStatus}) & \Rightarrow (\text{nat} \Rightarrow \text{ECall-Info~} \text{list}) \Rightarrow \\
\text{GatewayStatus~} \text{istream} & \Rightarrow \text{ECall-Info~} \text{istream} \Rightarrow \text{vcType~} \text{istream} \\
\Rightarrow (\text{nat} \Rightarrow \text{GatewayStatus}) & \Rightarrow (\text{nat} \Rightarrow \text{ECall-Info~} \text{list}) \\
\Rightarrow \text{bool}
\end{align*}\]

where
Sample-L \text{ req dt a1 stop lose st-in buffer-in}
ack i1 vc st-out buffer-out
\[
\equiv \\
(\forall (t::\text{nat}). \\
\text{buffer-out~} t = \\
(\text{if } dt~ t = [] \text{ then buffer-in~} t \text{ else } dt~ t) ) \\
\land \\
(t\text{tTable-SampleT req a1 stop lose st-in buffer-in} \\
\text{ack i1 vc st-out})
\]

definition
Sample ::
\[\begin{align*}
\text{reqType~} \text{istream} & \Rightarrow \text{ECall-Info~} \text{istream} \Rightarrow \text{aType~} \text{istream} \Rightarrow \\
\text{stopType~} \text{istream} & \Rightarrow \text{bool~} \text{istream} \Rightarrow \\
\text{GatewayStatus~} \text{istream} & \Rightarrow \text{ECall-Info~} \text{istream} \Rightarrow \text{vcType~} \text{istream} \\
\Rightarrow \text{bool}
\end{align*}\]

where
Sample \text{ req dt a1 stop lose ack i1 vc}
\[
\equiv \\
(\text{msg~} (1::\text{nat}) \text{ req}) \land \\
(\text{msg~} (1::\text{nat}) \text{ a1}) \land \\
(\text{msg~} (1::\text{nat}) \text{ stop})) \\
\rightarrow \\
(\exists \text{ st buffer}.
\text{Sample-L req dt a1 stop lose ack i1 vc st-out buffer-in})
\]

definition
Gateway ::
\[\begin{align*}
\text{reqType~} \text{istream} & \Rightarrow \text{ECall-Info~} \text{istream} \Rightarrow \text{aType~} \text{istream} \Rightarrow \\
\text{stopType~} \text{istream} & \Rightarrow \text{bool~} \text{istream} \Rightarrow \text{nat} \Rightarrow \\
\text{GatewayStatus~} \text{istream} & \Rightarrow \text{ECall-Info~} \text{istream} \Rightarrow \text{vcType~} \text{istream} \\
\Rightarrow \text{bool}
\end{align*}\]

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

definition
\[\text{GatewaySystem} ::\]
\[
\text{reqType} \ istream \Rightarrow \text{ECall-Info} \ istream \Rightarrow
\text{stopType} \ istream \Rightarrow \text{bool} \ istream \Rightarrow
\text{GatewayStatus} \ istream \Rightarrow \text{vcType} \ istream \Rightarrow \text{bool}
\]
\where
\[\text{GatewaySystem} \ req \ dt \ stop \ lose \ d \ ack \ vc\]
\[\equiv \exists \ a \ i.
(Sample \ req \ dt \ a \ stop \ lose \ d \ ack \ i \ vc) \land
(ServiceCenter \ i \ a)\]

definition
\[\text{GatewayReq} ::\]
\[
\text{reqType} \ istream \Rightarrow \text{ECall-Info} \ istream \Rightarrow \text{aType} \ istream \Rightarrow
\text{stopType} \ istream \Rightarrow \text{bool} \ istream \Rightarrow \text{nat} \Rightarrow
\text{GatewayStatus} \ istream \Rightarrow \text{ECall-Info} \ istream \Rightarrow \text{vcType} \ istream \Rightarrow \text{bool}
\]
\where
\[\text{GatewayReq} \ req \ dt \ a \ stop \ lose \ d \ ack \ i \ vc\]
\[\equiv (\forall (t::\text{nat}).
(req \ t = [\text{init-state}] \land \ \text{ack} \ t = [\text{init}] \land
\text{lose} \ (t+1) = [\text{False}] \land \text{lose} \ (t+2) = [\text{False}]
\rightarrow \ \text{ack} \ (t+2) = [\text{connection-ok}])
\land
(ack \ t = [\text{connection-ok}] \land \ \text{req} \ (Suc \ t) = [\text{send}] \land
(\forall (k::\text{nat}). \ k \leq (d+1) \rightarrow \ \text{lose} \ (t+k) = [\text{False}])
\rightarrow \ i \ ((Suc \ t) + d) = \text{inf-last-ti} \ dt \ t
\land \ \text{ack} \ (Suc \ t) = [\text{sending-data}])
\land
(\forall (k::\text{nat}). \ k \leq (d+1) \rightarrow \text{lose} \ (t+k) = [\text{False}])
\rightarrow \ \text{vc} \ ((Suc \ t) + d) = [\text{vc-com}])\)

definition
\[\text{GatewaySystemReq} ::\]
\[
\text{reqType} \ istream \Rightarrow \text{ECall-Info} \ istream \Rightarrow
\text{stopType} \ istream \Rightarrow \text{bool} \ istream \Rightarrow \text{nat} \Rightarrow
\]
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 sq1:length (a t) ≤ Suc 0 by (simp add: msg-def)
    from this and assms and a2 and a3 show ?thesis by auto
  qed
  qed

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

87
proof (cases a t)
  assume a1: a t = []
  from this and h2 show ?thesis by simp
next
  fix aa l
  assume a2: a t = aa # l
  from a2 and h1 have sg1: l = [] by (simp add: msg-nonempty1)
  from a2 and h1 and sg1 show ?thesis
  proof (cases aa)
    assume a3: aa = sc-ack
    from this and sg1 and h2 and a2 show ?thesis by simp
  qed
qed

lemma aType-lemma:
  assumes msg (Suc 0) a
  shows a t = [] ∨ a t = [sc-ack]
using assms
by (metis aType-nonempty)

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

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

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

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

lemma vcType-lemma:
  assumes msg (Suc 0) a
shows \( a \ t = [\] \lor a \ t = [\text{vc-com}] \)
using assms
by (metis vcType-empty)

15.2 Properties of the Delay component

lemma Delay-L1:
assumes h1: \( \forall t1 < t. \ i1 \ t1 = [\] \)
and h2: Delay y i1 d x i2
and h3: t2 < t + d
shows i2 t2 = [\]

proof (cases t2 < d)
assume a1: t2 < d
from h2 have sg1: t2 < d \( \rightarrow \) i2 t2 = []
by (simp add: Delay-def)
from sg1 and a1 show \( \vdots \)thesis by simp
next
assume a2: \( \neg \) t2 < d
from h2 have sg2: d \leq t2 \( \rightarrow \) i2 t2 = i1 (t2 - d)
by (simp add: Delay-def)
from a2 and sg2 have i2 t2 = i1 (t2 - d) by simp
from h1 and a2 and h3 and this show \( \vdots \)thesis by auto
qed

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

lemma Delay-L3:
assumes h1: \( \forall t1 \leq t. \ y \ t1 = [\] \)
and h2: Delay y i1 d x i2
and h3: t2 \leq t + d
shows x t2 = [\]

proof (cases t2 < d)
assume a1: t2 < d
from h2 have sg1: t2 < d \( \rightarrow \) x t2 = []
by (simp add: Delay-def)
from sg1 and a1 show \( \vdots \)thesis by simp
next
assume a2: \( \neg \) t2 < d
from h2 have sg2: d \leq t2 \( \rightarrow \) x t2 = y (t2 - d)
by (simp add: Delay-def)
from a2 and sg2 have sg3: x t2 = y (t2 - d) by simp
from h1 and a2 and h3 and sg3 show \( \vdots \)thesis by auto
qed

lemma Delay-L4:
assumes \( \forall t \leq t', y t' = [] \)
and Delay y i1 d x i2
shows \( \forall t' \leq t + d, x t' = [] \)
using assms by (clarify, rule Delay-L3, auto)

lemma Delay-lengthOut1:
assumes h1: \( \forall t. \) length (x t) \( \leq \) Suc 0
and h2: Delay x i1 d y i2
shows length (y t) \( \leq \) Suc 0
proof (cases t < d)
assume a1: t < d
from h2
have sg1: t < d \( \rightarrow \) y t = []
by (simp add: Delay-def)
from a1 and sg1
show ?thesis
by auto
next
assume a2: t \( \geq \) d
from h2
have sg2: t \( \geq \) 0 \( \rightarrow \) (y t = x (t−d))
by (simp add: Delay-def)
from a2 and sg2
and h1
show ?thesis
by auto
qed

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

15.3 Properties of the Loss component

lemma Loss-L1:
assumes \( \forall t < t'. i2 t' = [] \)
and Loss lose a i2 y i
and t2 < t
and ts lose
shows i t2 = []
using assms by (metis Loss-def)

lemma Loss-L2:
assumes \( \forall t < t'. i2 t' = [] \)
and Loss lose a i2 y i
and ts lose
shows \( \forall t < t', i2 t' = [] \)
using assms by (metis Loss-def)

lemma Loss-L3:
assumes \( \forall t < t'. a t2 = [] \)
and \( \text{Loss lose a i2 y i} \)
and \( t2 < t \)
and \( ts \text{ lose} \)
shows \( y \ t2 = [] \)
using \( \text{assms} \)
by (metis \( \text{Loss-def} \))

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

lemma \( \text{Loss-L5} \): 
assumes \( \forall t1 \leq t. \ a \ t1 = [] \)
and \( \text{Loss lose a i2 y i} \)
and \( t2 \leq t \)
and \( ts \text{ lose} \)
shows \( y \ t2 = [] \)
using \( \text{assms} \)
by (metis \( \text{Loss-def} \))

lemma \( \text{Loss-L5Suc} \): 
assumes \( \forall j \leq d. \ a \ (t + \text{Suc} j) = [] \)
and \( \text{Loss lose a i2 y i} \)
and \( \text{Suc j} \leq d \)
and \( ts\text{Lose:ts lose} \)
shows \( y \ (t + \text{Suc} j) = [] \)
using \( \text{assms} \)
proof (cases lose \( t + \text{Suc} j) = [\text{False}] \))
assume lose \( t + \text{Suc} j) = [\text{False}] \)
from \( \text{assms and this show \( \text{?thesis} \ by (simp add: Loss-def) \))}
next
assume lose \( t + \text{Suc} j) \neq [\text{False}] \)
from \( \text{this and ts\text{Lose have lose \( t + \text{Suc} j) = [\text{True}] \))}
by (simp add: ts-bool-True)
from \( \text{assms and this show \( \text{?thesis} \ by (simp add: Loss-def) \))}
qed

lemma \( \text{Loss-L6} \): 
assumes \( \forall t2 \leq t. \ a \ t2 = [] \)
and \( \text{Loss lose a i2 y i} \)
and \( ts \text{ lose} \)
shows \( \forall t2 \leq t. \ y \ t2 = [] \)
using \( \text{assms} \)
by (metis \( \text{Loss-L5} \))
lemma \textit{Loss-lengthOut1}:
\begin{itemize}
\item \textbf{assumes} $\forall t. \text{length} (a t) \leq \text{Suc 0}$
\item and $h2: \text{Loss lose } a \ i2 \ x \ i$
\end{itemize}
\textbf{shows} $\text{length} (x t) \leq \text{Suc 0}$
\textbf{proof} (cases \textit{lose} $t = [\text{False}]$)
\begin{itemize}
\item \textbf{assume} \textit{lose} $t = [\text{False}]$
\item from this and $h2$ have $sg1: x t = a t$ \textit{by (simp add: Loss-def)}
\item from $h1$ have $sg2: \text{length} (a t) \leq \text{Suc 0}$ \textit{by auto}
\item from $sg1$ and $sg2$ show \textit{thesis} \textit{by simp}
\end{itemize}
\textbf{next}
\begin{itemize}
\item \textbf{assume} \textit{lose} $t \neq [\text{False}]$
\item from this and $h2$ have $x t = []$ \textit{by (simp add: Loss-def)}
\item from this show \textit{thesis} \textit{by simp}
\end{itemize}
\textbf{qed}

lemma \textit{Loss-lengthOut2}:
\begin{itemize}
\item \textbf{assumes} $\forall t. \text{length} (a t) \leq \text{Suc 0}$
\item and $\text{Loss lose } a \ i2 \ x \ i$
\end{itemize}
\textbf{shows} $\forall t. \text{length} (x t) \leq \text{Suc 0}$
\textbf{using} \textit{assms}
\textbf{by} (simp add: \textit{Loss-lengthOut1})

lemma \textit{Loss-msg1}:
\begin{itemize}
\item \textbf{assumes} $\text{msg} (\text{Suc 0}) \ a$
\item and $\text{Loss lose } a \ i2 \ x \ i$
\end{itemize}
\textbf{shows} $\text{msg} (\text{Suc 0}) \ x$
\textbf{using} \textit{assms}
\textbf{by} (simp add: \textit{msg-def Loss-def Loss-lengthOut1})

15.4 Properties of the composition of Delay and Loss components

lemma \textit{Loss-Delay-length-y}:
\begin{itemize}
\item \textbf{assumes} $\forall t. \text{length} (a t) \leq \text{Suc 0}$
\item and $\text{Delay } x \ i1 \ d \ y \ i2$
\item and $\text{Loss lose } a \ i2 \ x \ i$
\end{itemize}
\textbf{shows} $\text{length} (y t) \leq \text{Suc 0}$
\textbf{using} \textit{assms}
\textbf{by} (metis \textit{Delay-msg1 Loss-msg1 msg-def})

lemma \textit{Loss-Delay-msg-a}:
\begin{itemize}
\item \textbf{assumes} $\text{msg} (\text{Suc 0}) \ a$
\item and $\text{Delay } x \ i1 \ d \ y \ i2$
\item and $\text{Loss lose } a \ i2 \ x \ i$
\end{itemize}
\textbf{shows} $\text{msg} (\text{Suc 0}) \ y$
\textbf{using} \textit{assms}
\textbf{by} (simp add: \textit{msg-def Loss-Delay-length-y})
15.5 Auxiliary Lemmas

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

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

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

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

lemma aux-lemma-lose-3a:
  assumes h1:∀ j≤2 * d + (4 + k). lose (t + j) = [False]
  and h2:ka ≤ Suc d
  shows lose (d + (t + (3 + k)) + ka) = [False]
proof –
  from h2 have sg1:(d + 3 + k + ka) ≤2 * d + (4 + k)
  by arith

93
from \( h1 \) and \( h2 \) and \( sg1 \) obtain \( j \) where \( a1\cdot j = (d + 3 + k + ka) \) and
\[ a2:\text{lose} (t + j) = [\text{False}] \]
by simp
from \( h2 \) and \( sg1 \) have \( \text{sg2}: (t + (d + 3 + k + ka)) = (d + (t + (3 + k)) + ka) \)
by arith
from \( h1 \) and \( h2 \) and \( a1 \) and \( a2 \) and \( sg2 \) show \( ?\text{thesis} \)
by simp
qed

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

**lemma** aux-arith1-Gateway7:
assumes \( t2 - t \leq (2::\text{nat}) \cdot d + (t + ((4::\text{nat}) + k)) \)
and \( t2 < t + (3::\text{nat}) + k + d \)
and \( \neg t2 - d < (0::\text{nat}) \)
shows \( t2 - d < t + (3::\text{nat}) + k \)
using assms by arith

**lemma** ts-lose-ack-st1ts:
assumes \( \text{ts lose} \)
and \( \text{lose} t = [\text{True}] \longrightarrow \text{ack} t = [x] \land \text{st-out} t = x \)
and \( \text{lose} t = [\text{False}] \longrightarrow \text{ack} t = [y] \land \text{st-out} t = y \)
shows \( \text{ack} t = [\text{st-out} t] \)
using assms
by (metis ts-bool-False)

**lemma** ts-lose-ack-st1:
assumes \( h1:\text{lose} t = [\text{True}] \lor \text{lose} t = [\text{False}] \)
and \( h2:\text{lose} t = [\text{True}] \longrightarrow \text{ack} t = [x] \land \text{st-out} t = x \)
and \( h3:\text{lose} t = [\text{False}] \longrightarrow \text{ack} t = [y] \land \text{st-out} t = y \)
shows \( \text{ack} t = [\text{st-out} t] \)
proof (cases \( \text{lose} t = [\text{False}] \))
assume \( \text{lose} t = [\text{False}] \)
from this and \( h3 \) show \( ?\text{thesis} \) by simp
next
assume \( a2:\text{lose} t \neq [\text{False}] \)
from this and \( h1 \) have \( \text{lose} t = [\text{True}] \) by (simp add: ts-bool-True)
from this and \( a2 \) and \( h2 \) show \( ?\text{thesis} \) by simp
qed

**lemma** ts-lose-ack-st2ts:
assumes \( \text{ts lose} \)
and \( \text{lose} t = [\text{True}] \longrightarrow \text{ack} t = [x] \land i1 t = [] \land vc t = [] \land \text{st-out} t = x \)
\[
\text{and } \text{lose } t = [\text{False}] \rightarrow \\
\text{ack } t = [y] \land i1 t = [] \land vc t = [] \land st-out t = y
\]
shows \( \text{ack } t = [\text{st-out } t] \)
using \text{assms}
by \((\text{metis ts-bool-True-False})\)

**lemma ts-lose-ack-st2:**
assumes \( h1: \text{lose } t = [\text{True}] \lor \text{lose } t = [\text{False}] \)
and \( h2: \text{lose } t = [\text{True}] \rightarrow \\
\text{ack } t = [x] \land i1 t = [] \land vc t = [] \land st-out t = x \)
and \( h3: \text{lose } t = [\text{False}] \rightarrow \\
\text{ack } t = [y] \land i1 t = [] \land vc t = [] \land st-out t = y \)
shows \( \text{ack } t = [\text{st-out } t] \)
proof \((\text{cases lose } t = [\text{False}])\)
assume \( \text{lose } t = [\text{False}] \)
from this and \( h3 \) show \( \text{thesis by simp} \)
next
assume \( a2: \text{lose } t \neq [\text{False}] \)
from this and \( h1 \) have \( \text{lose } t = [\text{True}] \) by \((\text{simp add: ts-bool-True})\)
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 \( h2: \text{lose } t = [\text{True}] \rightarrow \\
\text{ack } t = [x] \land i1 t = [] \land vc t = [] \land st-out t = x \)
and \( h3: \text{lose } t = [\text{False}] \rightarrow \\
\text{ack } t = [y] \land i1 t = [] \land vc t = [vc-com] \land st-out t = y \)
shows \( \text{ack } t = [\text{st-out } t] \)
proof \((\text{cases lose } t = [\text{False}])\)
assume \( \text{lose } t = [\text{False}] \)
from this and \( h3 \) show \( \text{thesis by simp} \)
next
assume \( a2: \text{lose } t \neq [\text{False}] \)
from this and \( h1 \) have \( \text{lose } t = [\text{True}] \) by \((\text{simp add: ts-bool-True})\)
from this and \( a2 \) and \( h2 \) show \( \text{thesis by simp} \)
qed

**lemma ts-lose-ack-st2send:**
assumes \( h1: \text{lose } t = [\text{True}] \lor \text{lose } t = [\text{False}] \)
and \( h2: \text{lose } t = [\text{True}] \rightarrow \\
\text{ack } t = [x] \land i1 t = [] \land vc t = [] \land st-out t = x \)
and \( h3: \text{lose } t = [\text{False}] \rightarrow \\
\text{ack } t = [y] \land i1 t = b t \land vc t = [] \land st-out t = y \)
shows \( \text{ack } t = [\text{st-out } t] \)
proof \((\text{cases lose } t = [\text{False}])\)
assume \( \text{lose } t = [\text{False}] \)
from this and \( h3 \) show \( \text{thesis by simp} \)
next
assume $a_2$ : lose $t \neq [\text{False}]$

from this and $h_1$ have lose $t = [\text{True}]$ by (simp add: ts-bool-True-False)

from this and $a_2$ and $h_2$ show ?thesis by simp

qed

lemma $tiTable$-ack-st-splitten:
assumes $h_1$ : ts lose
  and $h_2$ : msg (Suc 0) $a_1$
  and $h_3$ : msg (Suc 0) stop
  and $h_4$ : st-in $t$ = init-state $\land$ req $t$ = [init]$\rightarrow$
    ack $t$ = [call$] \land i_1 t = [] \land vc t = [] \land st-out $t$ = call
  and $h_5$ : st-in $t$ = init-state $\land$ req $t$ $\neq$ [init]$\rightarrow$
    ack $t$ = [init-state] $\land$ i_1 $t$ = [] $\land$ vc $t$ = [] $\land$ st-out $t$ = init-state
  and $h_6$ : (st-in $t$ = call $\lor$ st-in $t$ = connection-ok $\land$ req $t$ $\neq$ [send]$) \land$ lose $t$ = [False]$\rightarrow$
    ack $t$ = [connection-ok] $\land$ i_1 $t$ = [] $\land$ vc $t$ = [] $\land$ st-out $t$ = connection-ok
  and $h_7$ : (st-in $t$ = call $\lor$ st-in $t$ = connection-ok $\lor$ st-in $t$ = sending-data) $\land$
    lose $t$ = [True]$\rightarrow$
    ack $t$ = [init-state] $\land$ i_1 $t$ = [] $\land$ vc $t$ = [] $\land$ st-out $t$ = init-state
  and $h_8$ : st-in $t$ = connection-ok $\land$ req $t$ = [send]$\land$ lose $t$ = [False]$\rightarrow$
    ack $t$ = [sending-data] $\land$ i_1 $t$ = b $t$ $\land$ vc $t$ = [] $\land$ st-out $t$ = sending-data
  and $h_9$ : st-in $t$ = sending-data $\land$ a_1 $t$ = [] $\land$ lose $t$ = [False]$\rightarrow$
    ack $t$ = [sending-data] $\land$ i_1 $t$ = [] $\land$ vc $t$ = [] $\land$ st-out $t$ = sending-data
  and $h_{10}$ : st-in $t$ = sending-data $\land$ a_1 $t$ = [sc-ack]$\land$ lose $t$ = [False]$\rightarrow$
    ack $t$ = [voice-com] $\land$ i_1 $t$ = [] $\land$ vc $t$ = [vc-com] $\land$ st-out $t$ = voice-com
  and $h_{11}$ : st-in $t$ = voice-com $\land$ stop $t$ = [] $\land$ lose $t$ = [False]$\rightarrow$
    ack $t$ = [voice-com] $\land$ i_1 $t$ = [] $\land$ vc $t$ = [vc-com] $\land$ st-out $t$ = voice-com
  and $h_{12}$ : st-in $t$ = voice-com $\land$ stop $t$ = [] $\land$ lose $t$ = [True]$\rightarrow$
    ack $t$ = [voice-com] $\land$ i_1 $t$ = [] $\land$ vc $t$ = [] $\land$ st-out $t$ = voice-com
  and $h_{13}$ : st-in $t$ = voice-com $\land$ stop $t$ = [stop-vc]$\rightarrow$
    ack $t$ = [init-state] $\land$ i_1 $t$ = [] $\land$ vc $t$ = [] $\land$ st-out $t$ = init-state
shows ack $t$ = [st-out $t$]

proof
  from $h_1$ and $h_6$ and $h_7$ have $sg_1$ : lose $t = [\text{True}]$ $\lor$ lose $t = [\text{False}]$
  by (simp add: ts-bool-True-False)

show ?thesis

proof (cases st-in $t$
  assume $a_1$ : st-in $t$ = init-state
  from $a_1$ and $h_4$ and $h_5$ show ?thesis

proof (cases req $t$ = [init])
  assume $a_{11}$ : req $t$ = [init]
  from $a_{11}$ and $a_1$ and $h_4$ and $h_5$ show ?thesis by simp

next
  assume $a_{12}$ : req $t$ $\neq$ [init]
  from $a_{12}$ and $a_1$ and $h_4$ and $h_5$ show ?thesis by simp

qed

next
  assume $a_2$ : st-in $t$ = call
  from $a_2$ and $sg_1$ and $h_6$ and $h_7$ 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 h7 and h8 and sg1 show ?thesis
      apply simp
      by (rule ts-lose-ack-st2, assumption+)
next
assume a32: req t ≠ [send]
from this and a3 and h6 and h7 and h8 and sg1 show ?thesis
apply simp
by (rule ts-lose-ack-st2, assumption+)
qed

next
assume a4:st-in t = sending-data
from sg1 and a4 and h7 and h9 and h10 show ?thesis apply simp
proof (cases a1 t = [])
  assume a41: a1 t = []
  from this and a4 and sg1 and h7 and h9 and h10 show ?thesis
  apply simp
  by (rule ts-lose-ack-st2, assumption+)
next
assume a42: a1 t ≠ []
from this and h2 have a1 t = [sc-ack] by (simp add: aType-nonempty)
from this and a4 and a42 and sg1 and h7 and h9 and h10 show ?thesis
apply simp
by (rule ts-lose-ack-st2, assumption+)
qed

next
assume a5: st-in t = voice-com
from a5 and h11 and h12 and h13 show ?thesis
apply simp
proof (cases stop t = [])
  assume a51: stop t = []
  from this and a5 and h11 and h12 and h13 and sg1 show ?thesis
  apply simp
  by (rule ts-lose-ack-st2, assumption+)
next
assume a52: stop t ≠ []
from this and h3 have sg7: stop t = [stop-vc]
by (simp add: stopType-nonempty)
from this and a5 and a52 and h13 show ?thesis by simp
qed
qed
qed
lemma tiTable-ack-st:
assumes tiTable-SampleT req a1 stop lose st-in ack i1 vc st-out
and tsLose:ts lose
and a1Msg1:msg (Suc 0) a1
and stopMsg1:msg (Suc 0) stop
shows ack t = [st-out t]

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

qed

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

lemma tiTable-ack-connection-ok:
assumes tbl:tiTable-SampleT req x stop lose st-in b ack i1 vc st-out
and ackCon:ack t = [connection-ok]
and xMsg1:msg (Suc 0) x
and tsLose:ts lose
and stopMsg1:msg (Suc 0) stop
shows (st-in t = call ∨ st-in t = connection-ok ∧ req t = [send]) ∧
lose t = [False]

proof −
from tbl and tsLose have sg1:lose t = [True] ∨ lose t = [False]
by (simp add: ts-bool-True-False)
from tbl and xMsg1 have sg2:x t = [] ∨ x t = [sc-ack]
by (simp add: aType-lemma)
from tbl and stopMsg1 have sg3:stop t = [] ∨ stop t = [stop-vc]
by (simp add: stopType-lemma)
show ?thesis

proof (cases st-in t)
assuming a1:st-in t = init-state
show ?thesis
proof (cases req t = [init])
assuming a11:req t = [init]
from tbl and a1 and a11 and ackCon show ?thesis by (simp add: tiTable-SampleT-def)
next
assuming a12:req t ≠ [init]
from tbl and a1 and a12 and ackCon show ?thesis by (simp add: tiTable-SampleT-def)

qed

next
assuming a2:st-in t = call
show ?thesis
proof (cases lose t = [True])
assuming a21:lose t = [True]
from tbl and a2 and a21 and ackCon show ?thesis by (simp add: tiTable-SampleT-def)

next
assume a22:lose t \neq [True]
from this and tsLose have a22a:lose t = [False] by (simp add: ts-bool-False)
from tbl have
(st-in t = call \lor st-in t = connection-ok \land req t \neq [send]) \land
lose t = [False] \longrightarrow
ack t = [connection-ok] \land i1 t = [] \land vc t = [] \land st-out t = connection-ok
by (simp add: tiTable-SampleT-def)
from this and a2 and a22a and ackCon show ?thesis by simp
qed
next
assume a3:st-in t = connection-ok
show ?thesis
proof (cases lose t = [True])
assume a31:lose t = [True]
from tbl have
(st-in t = call \lor st-in t = connection-ok \lor st-in t = sending-data) \land
lose t = [True] \longrightarrow
ack t = [init-state] \land i1 t = [] \land vc t = [] \land st-out t = init-state
by (simp add: tiTable-SampleT-def)
from this and a3 and a31 and ackCon show ?thesis by simp
next
assume a32:lose t \neq [True]
from this and tsLose have a32a:lose t = [False] by (simp add: ts-bool-False)
show ?thesis
proof (cases req t = [send])
assume a321:req t = [send]
from tbl and a3 and a32a and a321 and ackCon show ?thesis
by (simp add: tiTable-SampleT-def)
next
dummy proof
qed
qed
next
assume a4:st-in t = sending-data
show ?thesis
proof (cases lose t = [True])
assume a41:lose t = [True]
from tbl and a4 and a41 and ackCon show ?thesis
by (simp add: tiTable-SampleT-def)
next
assume a42:lose t \neq [True]
from this and tsLose have a42a:lose t = [False] by (simp add: ts-bool-False)
show ?thesis
proof (cases x t = [sc-ack])
assume a421:x t = [sc-ack]
from tbl and a4 and a42a and a421 and ackCon show ?thesis
by (simp add: tiTable-SampleT-def)
next
  assume a422: x t ≠ [sc-ack]
  from this and xMsg1 have a422a: x t = [] by (simp add: aType-empty)
  from tbl and a4 and a42a and a422a and ackCon show ?thesis
  by (simp add: tiTable-SampleT-def)
qed
qed

qed

next
  assume a5: st-in t = voice-com
  show ?thesis
  proof (cases stop t = [stop-vc])
    assume a51: stop t = [stop-vc]
    from tbl and a5 and a51 and ackCon show ?thesis
    by (simp add: tiTable-SampleT-def)
  next
    assume a52: stop t ≠ [stop-vc]
    from this and stopMsg1 have a52a: stop t = [] by (simp add: stopType-empty)
    show ?thesis
    proof (cases lose t = [True])
      assume a521: lose t = [True]
      from tbl and a5 and a52a and a521 and ackCon show ?thesis
      by (simp add: tiTable-SampleT-def)
    next
      assume a522: lose t ≠ [True]
      from this and tsLose have a522a: lose t = [False] by (simp add: ts-bool-False)
      from tbl and a5 and a52a and a522a and ackCon show ?thesis
      by (simp add: tiTable-SampleT-def)
  qed
  qed
  qed

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

lemma tiTable-ack-call:
assumes \( tbl : tiTable-SampleT \) req \( x \) stop lose st-in b ack i1 vc st-out
and \( \text{ackCall} : \text{ack} \) t = [call]
and \( xMsg1 : \text{msg} (\text{Suc} \ 0) \) \( x \)
and \( \text{tsLose} : \text{ts} \) lose
and \( \text{stopMsg1} : \text{msg} (\text{Suc} \ 0) \) \( \text{stop} \)

shows \( \text{st-in} \ t = \text{init-state} \land \text{req} \ t = [\text{init}] \)

proof –
from \( tbl \) and \( \text{tsLose} \) have \( \text{sg1} : \text{lose} \ t = [\text{True}] \lor \text{lose} \ t = [\text{False}] \)
  by (simp add: ts-bool-True-False)
from \( tbl \) and \( xMsg1 \) have \( \text{sg2} : x \ t = [] \lor x \ t = [\text{sc-ack}] \)
  by (simp add: aType-lemma)
from \( tbl \) and \( \text{stopMsg1} \) have \( \text{sg3} : \text{stop} \ t = [] \lor \text{stop} \ t = [\text{stop-vc}] \)
  by (simp add: stopType-lemma)

show ?thesis

proof (cases \( \text{st-in} \ t \))
assume \( \text{a1} : \text{st-in} \ t = \text{init-state} \)

show ?thesis

proof (cases req \( t = [\text{init}] \))
assume \( \text{a11} : \text{req} \ t = [\text{init}] \)
from \( tbl \) and \( \text{a1} \) and \( \text{a11} \) and \( \text{ackCall} \) show ?thesis
  by (simp add: tiTable-SampleT-def)

next
assume \( \text{a12} : \text{req} \ t \neq [\text{init}] \)
from \( tbl \) and \( \text{a1} \) and \( \text{a12} \) and \( \text{ackCall} \) show ?thesis
  by (simp add: tiTable-SampleT-def)

qed

next
assume \( \text{a2} : \text{st-in} \ t = \text{call} \)
show ?thesis

proof (cases lose \( t = [\text{True}] \))
assume \( \text{a21} : \text{lose} \ t = [\text{True}] \)
from \( tbl \) and \( \text{a2} \) and \( \text{a21} \) and \( \text{ackCall} \) show ?thesis
  by (simp add: tiTable-SampleT-def)

next
assume \( \text{a22} : \text{lose} \ t \neq [\text{True}] \)
from \( \text{this} \) and \( \text{tsLose} \) have \( \text{a22a} : \text{lose} \ t = [\text{False}] \)
  by (simp add: ts-bool-False)
from \( tbl \) and \( \text{a2} \) and \( \text{a22a} \) and \( \text{ackCall} \) show ?thesis
  by (simp add: tiTable-SampleT-def)

qed

next
assume \( \text{a3} : \text{st-in} \ t = \text{connection-ok} \)
show ?thesis

proof (cases lose \( t = [\text{True}] \))
assume \( \text{a31} : \text{lose} \ t = [\text{True}] \)
from \( tbl \) and \( \text{a3} \) and \( \text{a31} \) and \( \text{ackCall} \) show ?thesis
  by (simp add: tiTable-SampleT-def)

next
assume \( \text{a32} : \text{lose} \ t \neq [\text{True}] \)
from this and tsLose have a32a:lose t = [False]
  by (simp add: ts-bool-False)
show ?thesis
proof (cases req t = [send])
  assume a321:req t = [send]
  from tbl and a3 and a32a and a321 and ackCall show ?thesis
    by (simp add: tiTable-SampleT-def)
next
  assume a322:req t ≠ [send]
  from tbl and a3 and a32a and a322 and ackCall show ?thesis
    by (simp add: tiTable-SampleT-def)
qed
qed
next
assume a4:st-in t = sending-data
show ?thesis
proof (cases lose t = [True])
  assume a41:lose t = [True]
  from tbl and a4 and a41 and ackCall show ?thesis
    by (simp add: tiTable-SampleT-def)
next
  assume a42:lose t ≠ [True]
  from this and tsLose have a42a:lose t = [False]
    by (simp add: ts-bool-False)
  show ?thesis
    proof (cases x t = [sc-ack])
      assume a421:x t = [sc-ack]
      from tbl and a4 and a42a and a421 and ackCall show ?thesis
        by (simp add: tiTable-SampleT-def)
    next
      assume a422:x t ≠ [sc-ack]
      from this and xMsg1 have a422a:x t = []
        by (simp add: aType-empty)
      show ?thesis
        proof (cases lose t = [True])
          assume a51:stop t = [stop-vc]
          from tbl and a5 and a51 and ackCall show ?thesis
            by (simp add: tiTable-SampleT-def)
        next
          assume a52:stop t ≠ [stop-vc]
          from this and stopMsg1 have a52a:stop t = []
            by (simp add: stopType-empty)
          show ?thesis
            proof (cases lose t = [True])
assume $a_{521} \cdot \text{lose } t = [\text{True}]
from \text{tbl and } a_{5} \text{ and } a_{52a} \text{ and } a_{521} \text{ and } \text{ackCall show } ?\text{thesis}$
by (simp add: \text{tiTable-SampleT-def})

next
assume $a_{522} \cdot \text{lose } t \neq [\text{True}]
from \text{this and tsLose have } a_{522a} \cdot \text{lose } t = [\text{False}] \text{ by (simp add: ts-bool-False)}
from \text{tbl and } a_{5} \text{ and } a_{52a} \text{ and } a_{522a} \text{ and } \text{ackCall show } ?\text{thesis}$
by (simp add: \text{tiTable-SampleT-def})
qed

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

lemma \text{tiTable-ack-init0:}
assumes \text{tbl:tiTable-SampleT req } a_{1} \text{ stop lose } \text{st-in } b \text{ ack } i{\text{l vc st-out}}
\text{and } ts \text{ lose}
\text{and } \text{msg } (\text{Suc } 0) \text{ a1}
\text{and } \text{msg } (\text{Suc } 0) \text{ stop}
\text{and } \text{ack } t = [\text{call}]
shows i1 t = []
proof
from \text{assms have } \text{st-in } t = \text{init-state } \land \text{req } t = [\text{init}]
by (simp add: \text{tiTable-ack-call})
from \text{this and tbl show } ?\text{thesis}
by (simp add: \text{tiTable-SampleT-def})
qed

lemma \text{tiTable-ack-init:}
assumes \text{tiTable-SampleT req } a_{1} \text{ stop lose}
\text{(fin-inf-append } [\text{init-state}] \text{ st)}
b \text{ ack } i{\text{l vc st}}
\text{and } \text{req0:req } 0 = []
shows ack 0 = [\text{init-state}]
proof
have \text{(fin-inf-append } [\text{init-state}] \text{ st)} (\text{0::nat}) = \text{init-state}
by (simp add: \text{fin-inf-append-def})
from \text{tbl and this and req0 show } ?\text{thesis by (simp add: tiTable-SampleT-def)}
qed

lemma \text{tiTable-ack-init:}
assumes \text{tiTable-SampleT req } a_{1} \text{ stop lose}
\text{(fin-inf-append } [\text{init-state}] \text{ st)}
b \text{ ack } i{\text{l vc st}}
\text{and } ts \text{ lose}
\text{and } \text{msg } (\text{Suc } 0) \text{ a1}
\text{and } \text{msg } (\text{Suc } 0) \text{ stop}
\text{and } \forall t{\text{l}} \leq t. \text{ req } t{\text{l}} = []
shows ack t = [\text{init-state}]
using \text{assms
proof (induction t)
case 0
  from this show ?case
    by (simp add: tiTable-ack-init0)
next
case (Suc t)
  from Suc have sg1: st t = hd (ack t)
    by (simp add: tiTable-ack-st-hd)
  from Suc and sg1 have sg2:
    (fin-inf-append [init-state] st) (Suc t) = init-state
    by (simp add: correct-fin-inf-append2)
  from Suc and sg1 and sg2 show ?case
    by (simp add: tiTable-SampleT-def)
qed

lemma tiTable-i1-3:
assumes tbl: tiTable-SampleT req x stop lose
  (fin-inf-append [init-state] st) ack t vc st
and tsLose:ts lose
and xMsg1:msg (Suc 0) x
and stopMsg1:msg (Suc 0) stop
and h5:∀ t1 ≤ t. req t1 = []
shows i1 t = []
proof
  from assms have sg1:ack t = [init-state]
    by (simp add: tiTable-ack-init)
  from assms have sg2:st t = hd (ack t)
    by (simp add: tiTable-ack-st-hd)
  from sg1 and sg2 have sg3:
    (fin-inf-append [init-state] st) (Suc t) = init-state
    by (simp add: correct-fin-inf-append2)
  from tbl and tsLose have sg4:lose t = [True] ∨ lose t = [False]
    by (simp add: ts-bool-True-False)
  from tbl and xMsg1 have sg5:x t = [] ∨ x t = [sc-ack]
    by (simp add: aType-lemma)
  from tbl and stopMsg1 have sg6:stop t = [] ∨ stop t = [stop-vc]
    by (simp add: stopType-lemma)
  show ?thesis
  proof (cases lose t = [True])
    assume a1:lose t = [True]
    from tbl and a2 and a21 show ?thesis
      by (simp add: tiTable-SampleT-def)
  next
    assume a2:lose t = [call]
    show ?thesis
    proof (cases lose t = [True])
      assume a21:lose t = [True]
      from tbl and a2 and a21 show ?thesis
        by (simp add: tiTable-SampleT-def)
next
  assume a22:lose t ≠ [True]
  from this and tsLose have a22a:lose t = [False]
   by (simp add: ts-bool-False)
  from tbl and a2 and a22a show thesis
   by (simp add: tiTable-SampleT-def)
qed
next
assume a3:fin-inf-append [init-state] st t = connection-ok
show thesis
proof (cases lose t = [True])
  assume a31:lose t = [True]
  from tbl and a3 and a31 show thesis
   by (simp add: tiTable-SampleT-def)
next
assume a32:lose t ≠ [True]
from this and tsLose have a32a:lose t = [False]
   by (simp add: ts-bool-False)
from h5 have a322:req t ≠ [send] by auto
from tbl and a3 and a32a and a322 show thesis
   by (simp add: tiTable-SampleT-def)
qed
next
assume a4:fin-inf-append [init-state] st t = sending-data
show thesis
proof (cases lose t = [True])
  assume a41:lose t = [True]
  from tbl and a4 and a41 show thesis by (simp add: tiTable-SampleT-def)
next
assume a42:lose t ≠ [True]
from this and tsLose have a42a:lose t = [False] by (simp add: ts-bool-False)
show thesis
proof (cases x t = [sc-ack])
  assume a421:x t = [sc-ack]
  from tbl and a4 and a42a and a421 and tsLose show thesis
   by (simp add: tiTable-SampleT-def)
next
assume a422: x t ≠ [sc-ack]
from this and xMsg have a422a:x t = [] by (simp add: aType-empty)
from tbl and a4 and a42a and a422a and tsLose show thesis
   by (simp add: tiTable-SampleT-def)
qed
qed
next
assume a5:fin-inf-append [init-state] st t = voice-com
show thesis
proof (cases stop t = [stop-vc])
  assume a51:stop t = [stop-vc]
from tbl and a5 and a51 and tsLose show ?thesis
  by (simp add: tiTable-SampleT-def)
next
  assume a52: stop t ≠ [stop-vc]
  from this and stopMsg1 have a52a: stop t = [] by (simp add: stopType-empty)
  show ?thesis
  proof (cases lose t = [True])
    assume a521: lose t = [True]
    from tbl and a5 and a52a and a521 and tsLose show ?thesis
    by (simp add: tiTable-SampleT-def)
  next
    assume a522: lose t ≠ [True]
    from this and tsLose have a522a: lose t = [False] by (simp add: ts-bool-False)
    from tbl and a5 and a52a and a522a and tsLose show ?thesis
    by (simp add: tiTable-SampleT-def)
  qed
qed

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

lemma tiTable-i1-4b:
assumes \( \text{tiTable-SampleT req x stop lose} \)

\[
(\text{fin-inf-append [init-state] st}) \ b \ack i1 \ vc \ st
\]

and \( ts \) lose
and \( \msg (\text{Suc 0}) \) \( x \)
and \( \msg (\text{Suc 0}) \) stop
and \( \forall \ t1 \leq t. \ \text{req} \ t1 = [] \)
and \( \text{req} (\text{Suc t}) = [\text{init}] \)
and \( \forall m < k + 3. \ \text{req} \ (t + m) \neq [\text{send}] \)
and \( \text{h7}\forall m \leq k. \ \ack (\text{Suc (Suc (t + m)))} = [\text{connection-ok}] \)
and \( \forall j \leq k + 3. \ \text{lose} \ (t + j) = [\text{False}] \)
and \( h9:t2 < (t + 3 + k) \)

shows \( i1 \ t2 = [] \)

proof \( (\text{cases } t2 \leq t) \)

assume \( t2 \leq t \)

from \( \text{assms and this show } \) ?thesis by \( (\text{simp add: tiTable-i1-3}) \)

next

assume \( a2:t2 \leq t \)

from \( \text{assms have } \text{sq1:ack } t = [\text{init-state}] \) by \( (\text{simp add: tiTable-ack-init}) \)

from \( \text{assms have } \text{sq2:st } t = \text{hd (ack } t \text{)} \) by \( (\text{simp add: tiTable-ack-st-hd}) \)

from \( \text{sq1 and sq2 have } \text{sq3:} \)

\( (\text{fin-inf-append [init-state] st}) \ (\text{Suc t}) = \text{init-state} \)

by \( (\text{simp add: correct-fin-inf-append2}) \)

from \( \text{assms and sq3 have } \text{sq4:st } (\text{Suc t}) = \text{call} \)

by \( (\text{simp add: tiTable-SampleT-def}) \)

show \( ?\text{thesis} \)

proof \( (\text{cases } t2 = \text{Suc } t) \)

assume \( a3:t2 = \text{Suc } t \)

from \( \text{assms and sq3 and a3 show } \) ?thesis

by \( (\text{simp add: tiTable-SampleT-def}) \)

next

assume \( a4:t2 \neq \text{Suc } t \)

from \( \text{assms and sq4 and a4 and a2 have } \text{sq7:st } (\text{Suc } (\text{Suc } t)) = \text{connection-ok} \)

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

from \( \text{assms have } \text{sq8:ack } (\text{Suc } (\text{Suc } t)) = [\text{st } (\text{Suc } (\text{Suc } t))] \)

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

show \( ?\text{thesis} \)

proof \( (\text{cases } t2 = \text{Suc } (\text{Suc } t)) \)

assume \( a5:t2 = \text{Suc } (\text{Suc } t) \)

from \( \text{h7 and h9 and a5 have } \text{sq9:ack } t2 = [\text{connection-ok}] \) by auto

from \( \text{assms and sq9 show } \) ?thesis by \( (\text{simp add: tiTable-i1-1}) \)

next

assume \( a6:t2 \neq \text{Suc } (\text{Suc } t) \)

from \( a6 \) and \( a4 \) and \( a2 \) have \( \text{sq10:Suc } (\text{Suc } t) < t2 \) by arith
from \( \text{h7 and h9 and sq10 have } \text{sq11:ack } t2 = [\text{connection-ok}] \)

by \( (\text{simp add: aux-ack-t2}) \)

from \( \text{assms and a6 and sq7 and sq8 and sq11 show } \) ?thesis

by \( (\text{simp add: tiTable-i1-1}) \)

qed

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

lemma \(\text{tiTable-ack-ok}\) : 
assumes \(h1: \forall j \leq d + 2. \text{ lose } (t + j) = [\text{False}]\)
and \(ts \text{ lose}\)
and \(\text{stopMsg1: msg (Suc 0) \text{ stop}}\)
and \(a1Msg1: msg (Suc 0) a1\)
and \(\text{reqNsend: req } (\text{Suc } t) \neq [\text{send}]\)
and \(\text{ackCon: ack } t = [\text{connection-ok}]\)
and \(\text{tbl: tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st) b ack i1 vc st}\) 
shows \(\text{ack } (\text{Suc } t) = [\text{connection-ok}]\)
proof -
from \(\text{tbl and tsLose and a1Msg1 and stopMsg1 have st t = hd } (\text{ack } t)\) 
by \((\text{simp add: tiTable-ack-st-hd})\)
from this and \(\text{ackCon have sg2: (fin-inf-append [init-state] st) } (\text{Suc } t) = \text{ connection-ok}\)
by \((\text{simp add: correct-fin-inf-append2})\)
have \(sg3a: Suc 0 \leq d + 2\) by \(\text{arith}\)
from \(h1\) and \(sg3a\) have \(sg3: \text{lose } (t + \text{Suc } 0) = [\text{False}]\) by \(\text{auto}\)
from \(sg2\) and \(sg3\) and \(\text{reqNsend and tbl show } ?\text{thesis}\)
by \((\text{simp add: tiTable-SampleT-def})\)
qed

lemma \(\text{Gateway-L7a}\) : 
assumes \(gw: \text{Gateway req dt a stop lose d ack i vc}\)
and \(a1Msg1: msg (Suc 0) a\)
and \(\text{stopMsg1: msg (Suc 0) stop}\)
and \(\text{reqMsg1: msg (Suc 0) req}\)
and \(ts \text{ lose}\)
and \(\text{loseFalse: } \forall j \leq d + 2. \text{ lose } (t + j) = [\text{False}]\)
and \(\text{nsend: req } (\text{Suc } t) \neq [\text{send}]\)
and \(\text{ackNCon: ack } (t) = [\text{connection-ok}]\)
shows \(\text{ack } (\text{Suc } t) = [\text{connection-ok}]\)
proof -
from gw and stopMsg1 and reqMsg1 and nsend obtain i1 i2 a1 a2 where
ah1:Sample req dt a1 stop lose ack i1 vc and
ah2:Delay a2 i1 d a1 i2 and
ah3:Loss a i2 a2 i
by (simp add: Gateway-def, auto)
from ah2 and ah3 and aMsg1 have sg1:msg (Suc 0) a1
by (simp add: Loss-Delay-msg-a)
from ah1 and sg1 and stopMsg1 and reqMsg1 obtain st buffer where
ah4:Sample-L req dt a1 stop lose (fin-inf-append [init-state] st)
       (fin-inf-append [] buffer)
       ack i1 vc st buffer
by (simp add: Sample-def, auto)
from ah4 have sg2:
tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st)
       (fin-inf-append [] buffer)
       ack i1 vc st
by (simp add: Sample-L-def)
from loseFalse and tsLose and stopMsg1 and sg1 and nsend and ackNCon and sg2 show ?thesis
by (simp add: tiTable-ack-ok)
qed

lemma Sample-L-buffer:
assumes
       Sample-L req dt a1 stop lose (fin-inf-append [init-state] st)
       (fin-inf-append [] buffer)
       ack i1 vc st buffer
shows buffer t = inf-last-ti dt t
proof –
from assms have
       ∀ t. buffer t =
       (if dt t = [] then fin-inf-append [] buffer t else dt t)
by (simp add: Sample-L-def)
from this show ?thesis
proof (induct t)
   case 0
   from this show ?case
   by (simp add: fin-inf-append-def)
next
   fix t
   case (Suc t)
   from this show ?case
   proof (cases dt t = [])
     assume dt t = []
     from this and Suc show ?thesis
     by (simp add: correct-fin-inf-append1)
   next
   assume dt t ≠ []
   from this and Suc show ?thesis
lemma \textit{tiTable-SampleT-i1-buffer}:
\begin{itemize}
  \item \textbf{assumes} \textit{ack t = [connection-ok]}
  \item \textit{reqSend:req (Suc t) = [send]}
  \item \textit{loseFalse:\forall k \leq Suc d. lose (t + k) = [False]}
  \item \textit{buf: buffer t = inf-last-ti dt t}
  \item \textit{tbl:tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st) (fin-inf-append [] buffer) ack i1 vc st}
  \item \textit{conOk:fin-inf-append [init-state] st (Suc t) = connection-ok}
\end{itemize}
\textbf{shows} \textit{i1 (Suc t) = inf-last-ti dt t}
\textbf{proof} –
\begin{itemize}
  \item \textit{have sg1:Suc 0 \leq Suc d by arith}
  \item \textit{from loseFalse and sg1 have sg2:lose (Suc t) = [False] by auto}
  \item \textit{from tbl have}
    \begin{itemize}
      \item \textit{fin-inf-append [init-state] st (Suc t) = connection-ok} \land
      \item \textit{req (Suc t) = [send]} \land
      \item \textit{lose (Suc t) = [False]} \rightarrow
      \item \textit{ack (Suc t) = [sending-data]} \land
      \item \textit{i1 (Suc t) = (fin-inf-append [] buffer) (Suc t)} \land
      \item \textit{vc (Suc t) = []} \land
      \item \textit{st (Suc t) = sending-data}
    \end{itemize}
    \item \textit{by (simp add: tiTable-SampleT-def)}
  \item \textit{from this and conOk and reqSend and sg2 have}
    \begin{itemize}
      \item \textit{i1 (Suc t) = (fin-inf-append [] buffer) (Suc t) by simp}
    \end{itemize}
  \item \textit{from this and buf show ?thesis by (simp add: correct-fin-inf-append1)}
\end{itemize}
qed

lemma \textit{Sample-L-i1-buffer}:
\begin{itemize}
  \item \textbf{assumes} \textit{msg (Suc 0) req}
  \item \textit{stopMsg1:msg (Suc 0) stop}
  \item \textit{a1Msg1:msg (Suc 0) a1}
  \item \textit{tsLose:ts lose}
  \item \textit{ackCon:ack t = [connection-ok]}
  \item \textit{reqSend:req (Suc t) = [send]}
  \item \textit{loseFalse:\forall k \leq Suc d. lose (t + k) = [False]}
  \item \textit{smpl:Sample-L reg dt a1 stop lose (fin-inf-append [init-state] st) (fin-inf-append [] buffer) ack i1 vc st buffer}
\end{itemize}
\textbf{shows} \textit{i1 (Suc t) = buffer t}
\textbf{proof} –
\begin{itemize}
  \item \textit{from smpl have sg1:buffer t = inf-last-ti dt t}
    \item \textit{by (simp add: Sample-L-buffer)}
  \item \textit{from smpl have sg2:}
    \begin{itemize}
      \item \textit{\forall t. buffer t = (if dt t = [] then fin-inf-append [] buffer t else dt t)}
    \end{itemize}
\end{itemize}

111
by (simp add: Sample-L-def)
from smpl have sg3:
  tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st)
  (fin-inf-append [[]] buffer) ack
  i1 vc st
by (simp add: Sample-L-def)
from sg3 and tsLose and a1Msg1 and stopMsg1 have sg4: st = hd (ack t)
by (simp add: tiTable-ack-st-hd)
from ackCon and sg4 have sg5:
  (fin-inf-append [init-state] st) (Suc t) = connection-ok
by (simp add: correct-fin-inf-append1)
from ackCon and reqSend and loseFalse and sg1 and
  sg3 and sg4 and sg5 have sg6:
  i1 (Suc t) = inf-last-ti dt t
by (simp add: tiTable-SampleT-i1-buffer)
from this and sg1 show ?thesis by simp
qed

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

lemma Sample-sending-data:
  assumes stopMsg1:msg (Suc 0) stop
  and tsLose:ts lose
  and reqMsg1:msg (Suc 0) req
  and a1Msg1:msg (Suc 0) a1
  and loseFalse: \( \forall j \leq 2 \times d \). lose (t + j) = [False]
  and ackSnd:ack t = [sending-data]
  and smpl:Sample req dt a1 stop lose ack i1 vc
  and add:x \leq d + d
  and h9: \( \forall t4 \leq t + d + d \). a1 t4 = []
  shows ack (t + x) = [sending-data]
using assms
proof -

112
from stopMsg1 and reqMsg1 and a1Msg1 and smpl obtain st buffer where
a1:
  Sample-L req dt a1 stop lose (fin-inf-append [init-state] st)
       (fin-inf-append [] buffer) ack
       if vc st buffer
       by (simp add: Sample-def, auto)
  from a1 have sg1:
    tiTable-SampleT req a1 stop lose (fin-inf-append [init-state] st)
       (fin-inf-append [] buffer)
       ack i1 vc st
       by (simp add: Sample-L-def)
  from a1 have sg2:
    \forall t. buffer t = (if dt t = [] then fin-inf-append [] buffer t else dt t)
       by (simp add: Sample-L-def)
  from stopMsg1 and tsLose and a1Msg1 and ackSnd and xdd and sg1 and sg2 show \textit{thesis}
  proof (induct x)
  case 0
  from this show \textit{case} by simp
next
  fix x
  case (Suc x)
  from this have sg3:st (t + x) = hd (ack (t + x))
       by (simp add: tiTable-ack-st-hd)
  from Suc have sg4:x \leq d + d by arith
  from Suc and sg3 and sg4 have sg5:
       (fin-inf-append [init-state] st) (Suc (t + x)) = sending-data
       by (simp add: fin-inf-append-def)
  from Suc have sg6:Suc (t + x) \leq 2 * d + t by simp
  from Suc have sg7:ack (t + x) = [sending-data] by simp
  from sg1 and loseFalse and h9 and sg7 and sg5 and sg6 have sg7:
       ack (Suc (t + x)) = [sending-data]
       by (simp add: tiTable-SampleT-sending-data)
  from this show \textit{case} by simp
qed
qed

15.6 Properties of the ServiceCenter component

lemma ServiceCenter-a-1:
  assumes ServiceCenter i a
  shows length (a t) \leq (Suc 0)
proof (cases t)
  case 0
  from this and assms show \textit{thesis} by (simp add: ServiceCenter-def)
next
  fix m assume t = Suc m
  from this and assms show \textit{thesis} by (simp add: ServiceCenter-def)
qed
lemma \textit{ServiceCenter-a-msg}:
\begin{itemize}
\item \textbf{assumes} ServiceCenter \( i \) \( a \)
\item \textbf{shows} \( \text{msg} \ (\text{Suc} \ 0) \) \( a \)
\end{itemize}
\textbf{using} \textit{assms}
\textbf{by} (simp add: \textit{msg-def ServiceCenter-a-1})

lemma \textit{ServiceCenter-L1}:
\begin{itemize}
\item \textbf{assumes} \( \forall \ t2 \prec x \). \( i \ t2 = \[] \)
\item \text{and} \ ServiceCenter \( i \) \( a \)
\item \text{and} \( t \leq x \)
\end{itemize}
\textbf{shows} \( a \ t = \[] \)
\textbf{using} \textit{assms}
\textbf{proof} (induct \( t \))
\begin{itemize}
\item \textbf{case} \( 0 \)
\item from this \textbf{show} \( \text{?case by} \ (\text{simp add: ServiceCenter-def}) \)
\end{itemize}
next
\begin{itemize}
\item \textbf{case} \( (\text{Suc} \ t) \)
\item from this \textbf{show} \( \text{?case by} \ (\text{simp add: ServiceCenter-def}) \)
\end{itemize}
\textbf{qed}

lemma \textit{ServiceCenter-L2}:
\begin{itemize}
\item \textbf{assumes} \( \forall \ t2 \prec x \). \( i \ t2 = \[] \)
\item \text{and} \ ServiceCenter \( i \) \( a \)
\end{itemize}
\textbf{shows} \( \forall \ t3 \leq x \). \( a \ t3 = \[] \)
\textbf{using} \textit{assms}
\textbf{by} (clarify, simp add: \textit{ServiceCenter-L1})

15.7 General properties of stream values

lemma \textit{streamValue1}:
\begin{itemize}
\item \textbf{assumes} \( h1:\forall j \leq D + (z::\text{nat}) \). \( \text{str} \ (t + j) = x \)
\item \text{and} \( h2: j \leq D \)
\end{itemize}
\textbf{shows} \( \text{str} \ (t + j + z) = x \)
\textbf{proof}
\begin{itemize}
\item from \( h2 \) \textbf{have} \( \text{sq1:} \ j + z \leq D + z \) \textbf{by} \ arith
\item have \( \text{sq2:} t + j + z = t + (j + z) \) \textbf{by} \ arith
\item from \( h1 \) \textit{and} \( \text{sq1} \) \textit{and} \( \text{sq2} \) \textbf{show} \( \text{?thesis by} \ (\text{simp (no-asn-simp)}) \)
\end{itemize}
\textbf{qed}

lemma \textit{streamValue2}:
\begin{itemize}
\item \textbf{assumes} \( \forall j \leq D + (z::\text{nat}) \). \( \text{str} \ (t + j) = x \)
\item \textbf{shows} \( \forall j \leq D \). \( \text{str} \ (t + j + z) = x \)
\end{itemize}
\textbf{using} \textit{assms} \textbf{by} (clarify, simp add: \textit{streamValue1})

lemma \textit{streamValue3}:
\begin{itemize}
\item \textbf{assumes} \( \forall j \leq D \). \( \text{str} \ (t + j + (\text{Suc} \ y)) = x \)
\item \text{and} \( j \leq D \)
\item \text{and} \( h3: \text{str} \ (t + y) = x \)
\end{itemize}
shows \( str(t + j + y) = x \)

using \( \text{assms} \)

proof (induct \( j \))
  case 0
  from \( h3 \) show \( ?\text{case by simp} \)
next
  case (Suc \( j \))
  from this show \( ?\text{case by auto} \)
qed

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

lemma \( \text{streamValue5} \):
  assumes \( \forall j \leq D. \ str(t + j + ((i::nat) + k)) = x \)
            and \( j \leq D \)
  shows \( \text{str}(t + i + k + j) = x \)
using \( \text{assms} \)
by (metis add.commute add.left-commute)

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

lemma \( \text{streamValue7} \):
  assumes \( h1: \forall j \leq d. \ str(t + i + k + d + \text{Suc } j) = x \)
            and \( h2: \text{str}(t + i + k + d) = x \)
            and \( h3: j \leq \text{Suc } d \)
  shows \( \text{str}(t + i + k + d + j) = x \)
proof –
  from \( h1 \) have \( sq1: \text{str}(t + i + k + d + \text{Suc } d) = x \)
     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 \( sq1 \) show \( ?\text{thesis by simp} \)
  next
  assume \( a2: j \neq \text{Suc } d \)
  from \( a2 \) and \( h3 \) have \( sq2: j \leq d \) by auto
  from \( \text{assms} \) and \( sq2 \) show \( ?\text{thesis} \)
  proof (cases \( j > 0 \))
    assume \( a3: 0 < j \)
    from \( a3 \) and \( h3 \) have \( sq3: j - (1::nat) \leq d \) by simp
  from \( a3 \) have \( sq4: \text{Suc}(j - (1::nat)) = j \) by arith
from $s_{g3}$ and $h1$ and $sg4$ have $sg5$: $\text{str}(t+i+k+d+j) = x$ by auto
from $sg5$ show $\text{thesis}$ by simp
next
  assume $a4': 0 < j$
  from $a4'$ have $sg6: j = 0$ by simp
  from $h2$ and $sg6$ show $\text{thesis}$ by simp
qed
qed
qed

lemma streamValue8:
assumes $\forall j \leq d. \text{str}(t+i+k+d+\text{Suc } j) = x$
and $\text{str}(t+i+k+d) = x$
shows $\forall j \leq \text{Suc } d. \text{str}(t+i+k+d+j) = x$
using assms streamValue7
by metis

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

lemma streamValue9:
assumes $h1: \forall j \leq 2 \cdot d. \text{str}(t + j + \text{Suc } (i + k)) = x$
and $h2: j \leq d$
shows $\text{str}(t + i + k + d + \text{Suc } j) = x$
proof
  from $h2$ have $(j + d) \leq 2 \cdot d$ by arith
  from $h1$ and this have $\text{str}(t + (j + d) + \text{Suc } (i + k)) = x$ by auto
  from this show $\text{thesis}$ by (simp add: arith-streamValue9aux)
qed

lemma streamValue10:
assumes $\forall j \leq 2 \cdot d. \text{str}(t + j + \text{Suc } (i + k)) = x$
shows $\forall j \leq \text{Suc } d. \text{str}(t + i + k + d + \text{Suc } j) = x$
using assms
apply clarify
by (rule streamValue9, auto)

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

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

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

lemma streamValue11:
assumes $h1: \forall j \leq 2 \cdot d + (4 + k). \text{lose } (t + j) = x$

and \( h2 \cdot j \leq \text{Suc} \ d \) shows lose \((t + 2 + k + j) = x \)

proof –
from \( h2 \) have \( sg1 : 2 + k + j \leq 2 \cdot d + (4 + k) \) by arith
have \( sg2 : \text{Suc} (t + k + j) = \text{Suc} (\text{Suc} (t + (k + j))) \) by arith
from \( sg1 \) and \( h1 \) have lose \((t + (2 + k + j)) = x \) by blast
from this and \( sg2 \) show \( \text{thesis} \) by (simp add: arith-sum2)
qed

lemma streamValue12:
assumes \( \forall j \leq 2 \cdot d + (4 + k) \cdot \text{lose} (t + j) = x \)
shows \( \forall j \leq \text{Suc} \ d \cdot \text{lose} (t + 2 + k + j) = x \)
using assms
apply clarify
by (rule streamValue11, auto)

lemma streamValue43:
assumes \( \forall j \leq 2 \cdot d + (4::\text{nat}) + k \cdot \text{lose} (t + j) = \text{[False]} \)
shows \( \forall j \leq 2 \cdot d \cdot \text{lose} ((t + (3::\text{nat}) + k) + j) = \text{[False]} \)
proof –
from assms have \( sg1 : \forall j \leq 2 \cdot d \cdot \text{lose} (t + j + (4 + k)) = \text{[False]} \)
by (simp add: streamValue2)
have \( sg2 : \text{Suc} (3 + k) = (4 + k) \) by arith
from \( sg1 \) and \( sg2 \) have \( sg3 : \forall j \leq 2 \cdot d \cdot \text{lose} (t + j + \text{Suc} (3 + k)) = \text{[False]} \)
by (simp (no-asn-simp))
from assms have \( sg4 : \text{lose} ((t + (3 + k)) = \text{[False]} \) by auto
from \( sg3 \) and \( sg4 \) have \( sg5 : \forall j \leq 2 \cdot d \cdot \text{lose} ((t + j + (3 + k)) = \text{[False]} \)
by (rule streamValue4)
from \( sg5 \) show \( \text{thesis} \) by (rule streamValue6)
qed

end

theory Gateway-proof
imports Gateway-proof-aux
begin

15.8 Properties of the Gateway

lemma Gateway-L1:
assumes \( h1 : \text{Gateway req dt a stop lose d ack i vc} \)
and \( h2 : \text{msg} \ (\text{Suc} \ 0) \ \text{req} \)
and \( h3 : \text{msg} \ (\text{Suc} \ 0) \ a \)
and \( h4 : \text{msg} \ (\text{Suc} \ 0) \ \text{stop} \)
and \( h5 : \text{ts lose} \)
and \( h6 : \text{ack} \ t = \text{[init-state]} \)
and \( h7 : \text{req} \ (\text{Suc} \ t) = \text{[init]} \)
and h8: 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
  i1 vc st
  by (simp add: Sample-def Sample-L-def, auto)
from a4 and h5 and sg1 and h4 have sg2: st
  by (simp add: tiTable-ack-st-hd)
from h6 and sg1 and h4 and h3 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 d ack i 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 a2 and h4 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 a4 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:

119
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 \((t + d) = \text{sending-data}\)
and \( h7 \): a (Suc \( t \)) = [sc-ack]
and \( h8 \): \( \forall k \leq \text{Suc} d. \) lose \((t + k) = \text{False}\)
shows \( \text{vc} (\text{Suc} (t + d)) = \text{vc-com} \)

proof –
from \( h1 \) obtain \( i1 \) \( i2 \) \( x \) \( y \)
where \( a1 \): Sample req dt \( x \) stop lose ack \( i1 \) vc
and \( a2 \): Delay \( y \) \( i1 \) \( d \) \( i2 \)
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
\((\text{fin-inf-append} \ [\text{init-state}] \ st) \ (\text{fin-inf-append} \ [[\ ]]] \ buffer) \) ack
\( i1 \) \( vc \) \( st \)
by (simp add: Sample-def Sample-L-def, auto)
from \( a4 \) and \( h5 \) and \( sg1 \) and \( h4 \) have \( sg2 \): \( t+d = \text{hd} (\text{ack} \ (t+d)) \)
by (simp add: correct-fin-inf-append1)

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 \((t + d) = \text{sending-data}\)
and \( h7 \): \( j \leq \text{Suc} d. \) a \((t+j) = []\)
and \( h8 \): \( \forall k \leq (d + d). \) lose \((t + k) = \text{False}\)
shows \( j \leq d \rightarrow \text{ack} \ (t+d+j) = \text{sending-data} \)

proof –
from \( h1 \) obtain \( i1 \) \( i2 \) \( x \) \( y \)
where a1: Sample req dt x stop lose ack i1 vc
  and a2: Delay y i1 d x i2
  and a3: Loss lose a i2 y i
by (simp only: Gateway-def, auto)
from a2 and a3 and h3 have sg1: msg (Suc 0) x
by (simp add: Loss-Delay-msg-a)
from a1 and h2 and h4 and sg1 obtain st buffer where a4:
tiTable-SampleT req x stop lose
  (fin-inf-append [init-state] st) (fin-inf-append [] buffer) ack i1 vc st
by (simp add: Sample-def Sample-L-def, auto)
from assms and a2 and a3 and sg1 and a4 show ?thesis
proof (induct j)
case 0 then show ?case by simp
next
case (Suc j)
then show ?case
proof (cases Suc j ≤ d)
  assume ¬ Suc j ≤ d then show ?thesis by simp
next
  assume a0: Suc j ≤ d
  then have d + Suc j ≤ d + d by arith
  then have sg2: Suc (d + j) ≤ d + d by arith
  from a4 and h2 and sg1 and h4 and h5 have sg4:
  st (t + d + j) = hd (ack (t + d + j))
  by (simp add: tiTable-ack-st-hd)
from Suc and a0 and sg4 have sg5:
  (fin-inf-append [init-state] st) (Suc (t + d + j)) = sending-data
by (simp add: correct-fin-inf-append1)
from h7 and a0 have sg6: ∀ j ≤ d. a (t + Suc j) = [] by auto
from sg6 and a3 and a0 and h5 have sg7: y (t + (Suc j)) = []
by (rule Loss-L5Suc)
from sg7 and a2 have sg8a: x (t + d + (Suc j)) = []
by (simp add: Delay-def)
then have sg8a: x (Suc (t + d + j)) = [] by simp
have sg9: Suc (t + d + j) = Suc (t + (d + j)) by arith
from a4 have sg10:
  fin-inf-append [init-state] st (Suc (t + d + j)) = sending-data ∧
  x (Suc (t + d + j)) = [] ∧
  lose (Suc (t + d + j)) = [False] →
  ack (Suc (t + d + j)) = [sending-data]
by (simp add: tiTable-SampleT-def)
from h8 and sg3 have sg11: lose (t + Suc (d + j)) = [False] by blast
have Suc (t + d + j) = t + Suc (d + j) by arith
from this and sg11 have lose (Suc (t + d + j)) = [False]
  by (simp (nu-asm-simp), simp)
from sg10 and sg5 and sg8a and this show ?thesis by simp
qed
qed

121
qed

lemma Gateway-L6-induction:
assumes h1: msg (Suc 0) req
  and h2: msg (Suc 0) x
  and h3: msg (Suc 0) stop
  and h4: ts lose
  and h5: \forall j \leq k. lose (t + j) = [False]
  and h6: \forall m \leq k. req (t + m) \neq [send]
  and h7: ack t = [connection-ok]
  and h8: Sample req dt x 1 stop lose ack i 1 vc
  and h9: Delay x 2 i 1 d x 1 i 2
  and h10: Loss lose x i 2 x 2 i
  and h11: m \leq k
shows ack (t + m) = [connection-ok]
using assms
proof (induct m)
case 0 then show ?case by simp
next
case (Suc m)
then have sg1: msg (Suc 0) x 1 by (simp add: Loss-Delay-msg-a)
from Suc and this obtain st buffer where
  a1: tiTable-SampleT req x 1 stop lose (fin-inf-append [init-state] st)
  (fin-inf-append [] buffer) ack i 1 vc st and
  a2: \forall t. buffer t = (if dt t = [] then fin-inf-append [] buffer t else dt t)
  by (simp add: Sample-def Sample-L-def, auto)
from a1 and sg1 and h3 and h4 have sg2: st (t + m) = hd (ack (t + m))
  by (simp add: tiTable-ack-st-hd)
from Suc have sg3: ack (t + m) = [connection-ok] by simp
from a1 and sg2 and sg3 have sg4:
  (fin-inf-append [init-state] st) (Suc (t + m)) = connection-ok
  by (simp add: fin-inf-append-def)
from Suc have sg5: Suc m \leq k by simp
from sg5 and h5 have sg6: lose (Suc (t + m)) = [False] by auto
from h6 and sg5 have sg7: req (Suc (t + m)) \neq [send] by auto
from a1 and sg3 and sg4 and sg5 and sg6 and sg7 show ?case
  by (simp add: tiTable-SampleT-def)
qed

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

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

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

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

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

have \( \text{Suc}\ 0 \leq k + d + 3 \) by arith

from \( h10 \) and this have lose \((t + \text{Suc}\ 0)\) = [False] by blast

then have \( \text{sg}1:\text{lose}\ \text{Suc}\ (t)\) = [False] by simp

have \( \text{Suc}\ (t) \leq k + d + 3 \) by arith

from \( h10 \) and this have lose \((t + \text{Suc}\ (t))\) = [False] by blast

then have \( \text{sg}2:\text{lose}\ \text{Suc}\ \text{Suc}\ (t)\) = [False] by simp

from \( h1 \) and \( h2 \) and \( h3 \) and \( h4 \) and \( h5 \) and \( h6 \) and \( h9 \) and \( \text{sg}1 \) and \( \text{sg}2 \)

have \( \text{sg}3:\text{ack}\ (t + 2) = \text{[connection-ok]}\)

by (simp add: Gateway-L1)

from \( h7 \) and this have \( \text{sg}4:\forall m \leq k.\ \text{req}\ ((t + 2) + m) \neq \text{[send]}\)

by (auto, simp add: aux-k3req)

from \( h10 \) have \( \text{sg}5:\forall j \leq k.\ \text{lose}\ ((t + 2) + j) = \text{[False]}\)

by (auto, simp add: aux3lose)

from \( h1 \) and \( \text{sg}4 \) and \( \text{sg}5 \) and \( \text{sg}3 \) and \( h5 \) and \( h4 \) and \( h3 \) and \( h2 \) have \( \text{sg}6:\forall m \leq k.\ \text{ack}\ ((t + 2) + m) = \text{[connection-ok]}\)

by (rule Gateway-L6a)

from \( \text{sg}6 \) have \( \text{sg}7:\text{ack}\ (t + 2 + k) = \text{[connection-ok]}\) by auto

from \( h1 \) obtain \( i1\ i2\ x\ y\ where\)

\[ a1:\text{Sample}\ \text{req}\ dt\ x\ \text{stop}\ \text{lose}\ \text{ack}\ i1\ \text{vc}\ \text{and}\]

\[ a2:\text{Delay}\ y\ i1\ d\ x\ i2\ \text{and}\]

\[ a3:\text{Loss}\ \text{lose}\ a\ i2\ y\ i\]

by (simp add: Gateway-def, auto)

from \( h3 \) and \( a2 \) and \( a3 \) have \( \text{sg}8:\text{msg}\ \text{Suc}\ 0\) \text{x}

by (simp add: Loss-Delay-msg-a)

from \( a1 \) and \( \text{sg}8 \) and \( h4 \) and \( h5 \) obtain \( \text{st}\ \text{buffer}\ where\)

\[ a4:\text{tiTable-SampleT}\ \text{req}\ x\ \text{stop}\ \text{lose}\ \text{ack}\ i1\ \text{vc}\ \text{st}\]

\[ \text{fin-inf-append}\ [\text{buffer}]\ \text{ack}\ i1\ \text{vc}\ \text{st}\ \text{and}\]

\[ a5:\forall t.\ \text{buffer}\ t = (\text{if}\ \text{dt}\ t = \text{[]}\ \text{then}\ \text{fin-inf-append}\ [\text{buffer}\ t\ \text{else}\ \text{dt}\ t})\]

by (simp add: Sample-def Sample-L-def, auto)

from \( a4 \) and \( h2 \) and \( \text{sg}8 \) and \( h4 \) and \( h11 \) and \( h6 \) and \( h7 \) and \( \text{sg}6 \) and \( h10 \)

have \( \text{sg}9:\forall t1 < (t + 3 + k).\ i1\ t1 = \text{[]}\)

by (simp add: tiTable-i1-4)

from \( \text{sg}9 \) and \( a2 \) have \( \text{sg}10: \forall t2 < (t + 3 + k + d).\ i2\ t2 = \text{[]}\)

by (rule Delay-L2)

from \( \text{sg}10 \) and \( a3 \) and \( h2 \) show \( \text{?thesis}\) by (rule Loss-L2)

qed

lemma Gateway-L8a:

assumes \( h1:\text{Gateway}\ \text{req}\ dt\ \text{a}\ \text{stop}\ \text{lose}\ d\ \text{ack}\ i1\ \text{vc}\)

and \( h2:\text{msg}\ \text{Suc}\ 0\) \text{req}

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

and \( h4:\text{msg}\ \text{Suc}\ 0\) \text{a}

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

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

and \( h7: \text{ack}\ t = \text{[sending-data]}\)

and \( h8: \forall t3 \leq t + d.\ a\ t3 = \text{[]}\)

and \( h9: x \leq d + d\)

shows \( \text{ack}\ (t + x) = \text{[sending-data]}\)
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: \( \forall t3 \leq t + d. \ y t3 = [] \) by (rule Loss-L6)
from sg1 and a2 have sg2: \( \forall t4 \leq t + d + d. \ x t4 = [] \) by (rule Delay-L4)
from h4 and a2 and a3 have sg3: msg (Suc 0) x by (simp add: Loss-Delay-msg-a)
from h3 and h5 and h2 and sg3 and h6 and h7 and a1 and h9 and sg2
show ?thesis
by (simp add: Sample-sending-data)
qed

lemma Gateway-L8:
assumes Gateway req dt a stop lose d ack i vc
shows GatewayReq req dt a stop lose d ack i vc
using assms
by (simp add: Gateway-L8)

15.9 Proof of the Refinement Relation for the Gateway Requirements

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

15.10 Lemmas about Gateway Requirements

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

proof –
from h9 have sg1:ack (Suc (Suc (t + k))) = \[connection-ok\] by auto
from h7 have sg2:
  \(\forall k \leq \operatorname{Suc} d. \, \operatorname{lose} (Suc (Suc (t + k + ka))) = \text{[False]}\)
  by (simp add: aux-lemma-lose-1)
from h1 and h2 and h3 and h4 and h6 and h10 and sg1 and sg2 have sg3:
ack (t + 2 + k) = \[connection-ok\] \land
  \(\text{req} (Suc (t + 2 + k)) = \text{[send]} \land (\forall k \leq \operatorname{Suc} d. \, \operatorname{lose} (t + k) = \text{[False]})) \rightarrow\)
ack (Suc (t + 2 + k)) = \[sending-data\]
  by (simp add: GatewayReq-def)
have \(t + 3 + k = Suc (Suc (Suc (t + k)))\) by arith
from sg3 and sg1 and h6 and h7 and this show \(?thesis\)
  by (simp add: eval-nat-numeral)
qed

lemma GatewayReq-L2:
assumes h1:msg (Suc 0) req
  and h2:msg (Suc 0) stop
  and h3:msg (Suc 0) a
  and h4:ts lose
  and h5:GatewayReq req dt a stop lose d i vc
  and h6:req (t + 3 + k) = \[send\]
  and h7:inf-last-ti dt t \(\neq \[]\)
  and h8:\(\forall j \leq 2 \ast d + (4 + k). \, \operatorname{lose} (t + j) = \text{[False]}\]
  and h9:\(\forall m \leq k. \, \operatorname{ack} (t + 2 + m) = \text{[connection-ok]}\)
shows \(t \ (t + 3 + k + d) \neq \[]\)
proof –
from h8 have sg1:\(\forall (x::nat). \, x \leq (d + 1) \rightarrow lose (t + x) = \text{[False]}\)
  by (simp add: aux-lemma-lose-2)
from h8 have sg2:\(\forall ka \leq \operatorname{Suc} d. \, \operatorname{lose} (Suc (Suc (t + k + ka))) = \text{[False]}\)
  by (simp add: aux-lemma-lose-1)
from h9 have sg3:\(\forall (t + 2 + k) = \text{[connection-ok]}\) by simp
from h1 and h2 and h3 and h4 and h5 and h6 and sg2 and sg3 have sg4:
ack (t + 2 + k) = \[connection-ok\] \land
  \(\text{req} (Suc (t + 2 + k)) = \text{[send]} \land (\forall k \leq \operatorname{Suc} d. \, \operatorname{lose} (t + k) = \text{[False]})) \rightarrow\)
i (Suc (t + 2 + k + d)) = \[\text{inf-last-ti} dt (t + 2 + k)\]
  by (simp add: GatewayReq-def, auto)
from h7 have sg5:inf-last-ti dt (t + 2 + k) \(\neq \[]\)
  by (simp add: inf-last-ti-nonempty-k)
have sg6:t + 3 + k = Suc (Suc (Suc (t + k))) by arith
have \(t + 2 + k = Suc (Suc (t + k))\) by arith
from sg1 and sg2 and sg3 and sg4 and sg5 and sg6 and this and h6 show \(?thesis\)
  by (simp add: eval-nat-numeral)
qed

15.11 Properties of the Gateway System

lemma GatewaySystem-L1aux:
**lemmas**

GatewaySystem-L1

**assumes**

msg (Suc 0) req
and msg (Suc 0) stop
and msg (Suc 0) a

and ts lose

and msg (Suc 0) req ∧ msg (Suc 0) a ∧ msg (Suc 0) stop ∧ ts lose ⟹

(∀ t. (ack t = [init-state] ∧
    req (Suc t) = [init] ∧ lose (Suc t) = [False] ∧
    lose (Suc (Suc t)) = [False] ⟹
    ack (Suc (Suc t)) = [connection-ok] ∧
    (ack t = [connection-ok] ∧ req (Suc t) = [send] ∧
    (∀ k≤Suc d. lose (t + k) = [False]) ⟹
    i (Suc (t + d)) = inf-last-ti dt t ∧ ack (Suc t) = [sending-data]) ∧
    (ack (t + d) = [sending-data] ∧ a (Suc t) = [sc-ack] ∧
    (∀ k≤Suc d. lose (t + k) = [False]) ⟹
    vc (Suc (t + d)) = [vc-com])

shows ack (t + 3 + k + d + d) = [sending-data] ∧
    a (Suc (t + 3 + k + d + d)) = [sc-ack] ∧
    (∀ ka≤Suc d. lose (t + 3 + k + d + ka) = [False]) ⟹
    vc (Suc (t + 3 + k + d + d)) = [vc-com]

using assms by blast

**lemma** GatewaySystem-L3aux:

**assumes** msg (Suc 0) req
and msg (Suc 0) stop
and msg (Suc 0) a

and ts lose

and msg (Suc 0) req ∧ msg (Suc 0) a ∧ msg (Suc 0) stop ∧ ts lose ⟹

(∀ t. (ack t = [init-state] ∧
    req (Suc t) = [init] ∧ lose (Suc t) = [False] ∧
    lose (Suc (Suc t)) = [False] ⟹
    ack (Suc (Suc t)) = [connection-ok] ∧
    (ack t = [connection-ok] ∧ req (Suc t) = [send] ∧
    (∀ k≤Suc d. lose (t + k) = [False]) ⟹
    i (Suc (t + d)) = inf-last-ti dt t ∧ ack (Suc t) = [sending-data]) ∧
    (ack (t + d) = [sending-data] ∧ a (Suc t) = [sc-ack] ∧
    (∀ k≤Suc d. lose (t + k) = [False]) ⟹
    vc (Suc (t + d)) = [vc-com])

shows ack (t + 2 + k) = [connection-ok] ∧
    req (Suc (t + 2 + k)) = [send] ∧
    (∀ j≤Suc d. lose (t + 2 + k + j) = [False]) ⟹
    i (Suc (t + 2 + k + d)) = inf-last-ti dt (t + 2 + k)

using assms by blast

**lemma** GatewaySystem-L1:

**assumes** h2:ServiceCenter i a
and h3:GatewayReq req dt a stop lose d ack i vc
and h4:msg (Suc 0) req
and h5:msg (Suc 0) stop
and h6:msg (Suc 0) a

127
and $h7$: $t \leq (4 + k)$.
and $h9$: $j \leq 2 \times d + (4 + k)$. 
and $h11$: $j \leq k$.
and $h14$: $\forall x \leq d + d$. 
shows $v c (2 \times d + (t + (4 + k))) = [v e - c o m]$.

proof –
from $h2$ have $\forall t$. 
a $(S u c t) = (i f i t = [] \ t h e n [] e l s e [s c - a c k])$
by $(s i m p a d d : S e r v i c e C e n t e r - d e f )$
then have $s q 1$: 
a $(S u c (t + 3 + k + d)) = (i f i (t + 3 + k + d) = [] \ t h e n [] e l s e [s c - a c k])$
by blast
from $s q 1$ and $h11$ have $s q 2$: $a (S u c (t + 3 + k + d)) = [s c - a c k] \ b y a u t o$
from $h14$ have $s q 3$: $a c k (t + 3 + k + 2 \times d) = [s e n d i n g - d a t a] \ b y s i m p$
from $h4$ and $h5$ and $h6$ and $h7$ and $h8$ have $s q 4$: 
$[s e n d i n g - d a t a] \wedge a (S u c (t + 3 + k + d)) = [s c - a c k] \wedge$
$(\forall k a \leq S u c d. \ l o s e (t + 3 + k + d + k a) = [F a l s e]) \l o n g e r$
$[v e (S u c (t + 3 + k + d + d)) = [v e - c o m]$,
apply $(s i m p o n l y : G a t e w a y R e q - d e f )$
by $(r u l e G a t e w a y S y s t e m - L t a u n x , a u t o )$
from $h9$ have $s q 5$: $\forall k a \leq S u c d. \ l o s e (d + (t + (3 + k)) + k a) = [F a l s e ]$
by $(s i m p \ a d d : a u x - l e m m a - l o s e - 3 )$
have $s q 5 a$: $d + (t + (3 + k)) = t + 3 + k + d$ by arith
from $s q 5$ and $s q 5 a$ have $s q 5 b$: $\forall k a \leq S u c d. \ l o s e (t + 3 + k + d + k a) = [F a l s e ]$
by auto
have $s q 6$: $(t + 3 + k + 2 \times d) = (2 \times d + (t + (3 + k)))$ by arith
have $s q 7$: $s u c (s u c (s u c (t + k + (d + d)))) = s u c (s u c (s u c (t + k + d + d))))$
by arith
have $s u c (s u c (s u c (t + k + d + d)))) =$
$S u c (S u c (S u c (S u c (d + d + (t + k)))))$ by arith
from $s q 4$ and $s q 3$ and $s q 2$ and $s q 5 b$ and $s q 6$ and $s q 7$ and this show $? t h e s i s$
by $(s i m p \ a d d : e v a l - n a t - n u m e r a l )$

qed

lemma $a u x - l o s e 1$: 
assumes $h1$: $\forall j \leq 2 \times + (4 + k)$. 
and $h2$: $j \leq k$
shows $l o s e (t + (2 \times n a t) + j) = [F a l s e ]$
proof –
from $h2$ have $(2 \times n a t) + j \leq (2 \times n a t) \times d + (4 + k)$ by arith
from $h1$ and this have $l o s e (t + (2 + j)) = [F a l s e ]$ by blast
then show $? t h e s i s$ by simp

qed

lemma $a u x - l o s e 2$: 
assumes $\forall j \leq 2 \times + (4 + k)$. 
and $3 + k + d \leq 2 \times d + (4 + k)$
shows lose \((t + (3::nat) + k + d) = [False]\)

proof –
  from assms have lose \((t + (3::nat) + k + d)) = [False] by blast then show ?thesis by (simp add: arith-sum1)
qed

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

lemma 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 \(h4: \text{msg} \ (\text{Suc} 0) \ \text{req}\)
  and \(h5: \text{msg} \ (\text{Suc} 0) \ \text{stop}\)
  and \(h6: \text{msg} \ (\text{Suc} 0) \ a\)
  and \(h7: \text{ts lose}\)
  and \(h8: \text{ack} \ t = [\text{init-state}]\)
  and \(h9: \text{req} \ (\text{Suc} \ t) = [\text{init}]\)
  and \(h10: \forall \text{t} \leq t. \ \text{req} \ \text{t} = []\)
  and \(h11: \forall m \leq k + 2. \ \text{req} \ (t + m) \neq [\text{send}]\)
  and \(h12: \text{req} \ (t + 3 + k) = [\text{send}]\)
  and \(h13: \text{inf-last-ti dt t} \neq []\)
  and \(h14: \forall j \leq 2 * d + (4 + k). \ \text{lose} \ (t + j) = [\text{False}]\)
shows \(vc \ (2 * d + (t + (4 + k))) = [\text{vc-com}]\)
proof –
  have \(\text{Suc} \ 0 \leq 2 * d + (4 + k)\) by arith
  from \(h14\) and this have lose \((t + \text{Suc} \ 0) = [\text{False}]\) by blast
  then have \(\text{sg1: lose} \ (\text{Suc} \ t) = [\text{False}]\) by simp
  have \(\text{Suc} \ (\text{Suc} \ 0) \leq 2 * d + (4 + k)\) by arith
  from \(h14\) and this have lose \((t + \text{Suc} \ (\text{Suc} \ 0)) = [\text{False}]\) by blast
  then have \(\text{sg2: lose} \ (\text{Suc} \ (\text{Suc} \ t)) = [\text{False}]\) by simp
  from \(h3\) and \(h4\) and \(h5\) and \(h6\) and \(h7\) and \(h8\) and \(h9\) and \(sg1\) and \(sg2\)
  have \(\text{sg3:}\)
    \(\text{ack} \ (t + 2) = [\text{connection-ok}]\)
    by (simp add: GatewayReq-def)
  from \(h14\) have \(\text{sg4: } \forall j \leq k. \ \text{lose} \ (t + 2 + j) = [\text{False}]\)
    by (clarify, rule aux4lose1, simp)
  from \(h11\) have \(\text{sg5: } \forall m \leq k. \ \text{req} \ (t + 2 + m) \neq [\text{send}]\)
    by (clarify, rule aux4req, auto)
  from \(h1\) and \(sg5\) and \(sg4\) and \(sg3\) and \(h4\) and \(h5\) and \(h6\) and \(h7\) have \(sg6:\)
∀ m ≤ k. ack (t + 2 + m) = [connection-ok]
  by (rule Gateway-L6)

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

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

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

from sg14 and h2 have sg15:∀ t3 ≤ (t + 3 + k + d). a t3 = []
  by (simp add: ServiceCenter-L2)

from h14 have sg18:∀ j ≤ 2 * d. lose ((t + 3 + k) + j) = [False]
  by (simp add: streamValue43)

from h14 have sg16a:∀ j ≤ 2 * d. lose (t + j + (4 + k)) = [False]
  by (simp add: streamValue2)

have sg16b:Suc (3 + k) = (4 + k) by arith

from sg16a and sg16b have sg16:∀ j ≤ 2 * d. lose (t + j + Suc (3 + k)) = [False]
  by (simp (no-asn-simp))

from h7 and h4 and h5 and h6 and h7 and sg18 and sg10 and sg15 have
  sg19:
  ∀ x ≤ d + d. ack (t + 3 + k + x) = [sending-data]
  by (simp add: Gateway-L8)

from sg19 have sg19a:ack (t + 3 + k + d + d) = [sending-data] by auto
from sg16 have sg20a:∀ j ≤ d. lose (t + 3 + k + d + Suc j) = [False]
  by (rule streamValue10)

have sg20b:3 + k + d ≤ 2 * d + (4 + k) by arith

from h14 and sg20b have sg20c:lose (t + 3 + k + d) = [False]
  by (rule aux2lose2)

from sg20a and sg20c have sg20:∀ j ≤ Suc d. lose (t + 3 + k + d + j) = [False]

  by (rule streamValue8)

from h4 and h5 and h6 and h7 and h8 have sg21:
  ack (t + 3 + k + d + d) = [sending-data] ∧
  a (Suc (t + 3 + k + d + d)) = [sc-ack] ∧
  (∀ j ≤ Suc d. lose (t + 3 + k + d + j) = [False]) →
  vc (Suc (t + 3 + k + d + d)) = [vc-com]

  apply (simp only: GatewayReq-def)
  by (rule GatewaySystem-L1aux, auto)

from h2 and sg11 have sg22:a (Suc (t + 3 + k + d)) = [sc-ack]
by (simp only: ServiceCenter-def, auto)
from sg21 and sg19a and sg22 and sg20 have sg23:

vc (Suc (t + 3 + k + d + d)) = [vc-com] by simp
have 2 * d + (t + (4 + k)) = (Suc (t + 3 + k + d + d)) by arith
from sg23 and this show thesis
by (simp (no-simp-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) \not= [] \lor dt (Suc (Suc t)) \not= []
and h9: ack t = [init-state]
and h10: req (Suc t) = [init]
and h11: \forall t1 \leq t. req t1 = []
and h12: \forall m \leq k + 2. req (t + m) \neq [send]
and h13: req (t + 3 + k) = [send]
and h14: \forall j \leq 2 * d + (4 + k). lose (t + j) = [False]

shows vc (2 * d + (t + (4 + k))) = [vc-com]

proof –

have Suc 0 \leq 2 * d + (4 + k) by arith
from h14 and this have lose (t + Suc 0) = [False] by blast
then have sg1: lose (Suc t) = [False] by simp
have Suc (Suc 0) \leq 2 * d + (4 + k) by arith
from h14 and this have lose (t + Suc (Suc 0)) = [False] by blast
then have sg2: lose (Suc (Suc t)) = [False] by simp
from h3 and h4 and h5 and h6 and h7 and h10 and h9 and sg1 and sg2
have sg3:

ack (t + 2) = [connection-ok]
by (simp add: GatewayReq-def)
from h14 have sg4: \forall j \leq k. lose (t + 2 + j) = [False]
by (clarify, rule aux4lose1, simp)
from h12 have sg5\forall m \leq k. req (t + 2 + m) \neq [send]
by (clarify, rule aux4req, auto)

from h1 and sg5 and sg4 and sg3 and h4 and h5 and h6 and h7 have sg6:
\forall m \leq 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 sq11a:
  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 sq12:Suc (t + 2 + k) = t + 3 + k by arith
from h13 and sq12 have sq12a: req (Suc (t + 2 + k)) = [send]
  by (simp add: eval-nat-numeral)
from h14 have sq13: ∀ j≤Suc d. lose ((t + 2 + k) + j) = [False]
  by (rule streamValue12)
from sq11a and sq6a and h13 and sq12a and sq13 have sq14:
  i (Suc (t + (2::nat) + k + d)) = inf-last-ti dt (t + 2 + k) by simp
from h8 have sq15: inf-last-ti dt (t + 2 + k) ≠ []
  by (rule inf-last-ti-Suc2)
from sq14 and sq15 have sq16: i (t + 3 + k + d) ≠ []
  by (simp add: arith-sum4)
from h14 have sq17: ∀ j≤k + d + 3. lose (t + j) = [False] by auto
from h12 have sq18: ∀ m < (k + 3), req (t + m) ≠ [send] by auto
from h1 and h4 and h5 and h6 and h7 and h10 and sq18 and h13 and h9
  and sq17 and h11
  have sq19: ∀ t2 < (t + 3 + k + d). i t2 = []
  by (simp add: Gateway-L7)
from h2 and sq19 have sq20: ∀ t3 ≤ (t + 3 + k + d). a t3 = []
  by (simp add: ServiceCenter-L2)
from h14 have sq21: ∀ j≤2 * d. lose (t + 3 + k + j) = [False]
  by (simp add: streamValue43)
from h1 and h4 and h5 and h6 and h7 and sq21 and sq10 and sq20 have
  ∀ x ≤ d + d. ack (t + 3 + k + x) = [sending-data]
  by (simp add: Gateway-L8)
from h2 and h3 and h4 and h5 and h6 and h7 and h14 and sq16 and this
show ?thesis
  by (simp add: GatewaySystem-L1)
qed

15.12 Proof of the Refinement for the Gateway System

lemma GatewaySystem-L0:
assumes GatewaySystem req dt stop lose d ack vc
shows GatewaySystemReq req dt stop lose d ack vc
proof –
from assms obtain x i where
  a1: Gateway req dt x stop lose d ack i vc and
  a2: ServiceCenter i x
  by (simp add: GatewaySystem-def, auto)
from a1 have sq1: GatewayReq req dt x stop lose d ack i vc
  by (simp add: Gateway-L0)
from a2 have sg2:msg (Suc 0) x
  by (simp add: ServiceCenter-a-msg)
from assms and a1 and a2 and sg1 and sg2 show ?thesis
  apply (simp add: GatewaySystemReq-def, auto)
  apply (simp add: GatewaySystem-L3)
  apply (simp add: GatewaySystem-L3)
  apply (simp add: GatewaySystem-L3)
  by (simp add: GatewaySystem-L2)
qed

end

References


