A Formal Proof of Correctness of a Distributed Presentation Software System

Ievgen Ivanov, Taras Panchenko[1]

Taras Shevchenko National University of Kyiv,
64/13, Volodymyrska st., Kyiv, 01601, Ukraine,
[1] `tp@infosoft.ua`,
WWW home page: `https://www.facebook.com/tpanchenko`

Abstract. In this paper we present a formal proof of total correctness for Infosoft e-Detailing 1.0 distributed presentation software using Isabelle proof assistant. We model execution of a distributed software as a transition system with a global state that is composed of states of the system's components and show that under a certain progress assumption, after a presenter switches the current slide to a given target slide, the executions of this transition system reaches a state which all viewers (clients) can see the target slide.

Keywords: software correctness, shared memory concurrency, interleaving concurrency, safety property, liveness property, total correctness, formal methods.
Key Terms: Software System, Environment, Characteristic, Methodology, Experience.

1 Introduction

Infosoft e-Detailing (`www.e-detailing.pro` [2,3]) is a commercially distributed presentation software which operates over a computer network and allows a presenter ("manager") to display slides to several viewers ("clients"). This software is distributed among several computers/laptops/mobile devices - the manager's device and client's devices. The manager gives a presentation consisting of a sequence of slides to the clients. The content of the slides does not change during the presentation, but the order in which they are displayed and the duration of the presentation of each particular slide are freely controlled by the manager in real time. In general, the operation of the system can be considered as a sequence of operation cycles, each of which consists of switching a slide on the manager's device and subsequent switching the view on all client devices. The number of such cycles is unlimited, and the cycles continue till the end of the presentation.

The goal of the software is to ensure the that the clients see the slide currently selected by the manager. Obviously, there are delays between the time the manager switches a new slide and the time the viewers see the new slide, mostly caused by network propagation, but the system is designed to minimize such delays. In particular, the following techniques are used for this purpose: pre-sharing slides content among the clients, sending only the current slide index to the clients instead of sending the slide content, implementing a execution-time-optimized reaction to the slide updating notification.

In the previous works [2,3] we described the algorithm implemented by this system in the imperative compositional language (ICPL) notation [4,5] and considered the problem of proving partial [2] as well as total [6] correctness of this algorithm in the following sense: if at the start of a system operation cycle the manager switches to a new slide s, then when the programs on the client's devices reach the end of their operation cycle, the clients will see the slide s.

In this paper we will consider this situation and formalize and prove a total correctness property with more rigorous approach (automated instead of manual proof), which guarantees that if at the start of a system operation cycle the manager switches to a new slide s, then the program on the clients' devices eventually reach the end of the operation cycle and when this happens the clients see the slide s. We will prove this property under an assumption that at each point in time during a system's operation cycle, each client device program is either in the final state of the operation cycle (it already displays the slide currently chosen by the manager and performs no actions until the next operation cycle, i.e. slide change by the manager), or is still working in the sense that either at the current or at some future time moment it will make an execution step in accordance with its algorithm. This assumption excludes a situation when some client program stops working (is unable to perform an execution step) in the course of a system's operation cycle. The execution steps of client programs consist of network data exchange steps (which consist of sending/receiving fixed-length messages over a computer network) and internal computation steps (which take place on the device). Thus a client program's inability to make an execution step may be caused by the reasons like: a client exits the software on his/her device during the presentation, shuts off the device, disconnects the device from the network (so no network data exchange steps can be completed), etc. When such issues are excluded, our proof shows that the system eventually reaches a state in which all the clients display the slide chosen the manager, so the operation cycle may be called "successful". Thus this proof excludes the possibility of non-terminating system operation cycles, which could arise e.g. as a result of falling of client programs into infinite loops in the course of manager slide change processing.

2 Overview of Infosoft e-Detailing

The Infosoft e-Detailing software operates on a hardware system which consists of the central server, the manager's device, and one or more client devices (such as computers, mobile devices, tables). All client devices and the manager's device should be able to communicate with the central server over a computer network. The components of the system are illustrated in Fig. 1.

Fig. 1. Infosoft e-Detailing Interactive Presentation System

Communications include HTTP(S)-requests with AJAX technology over the internet on client and manager devices for data transfer and the server software, which "synchronizes" the current slide between manager and clients in the way depicted on Fig. 1 and described in [3,2] and in the next section in more details. This architecture was designed to minimize continuous connection number and time elapsed in waiting mode on the server side and for more flexibility in client connections and supported devices range:

- possible temporary disconnections
- some network instability is acceptable
- no need to fix the count of clients
- most of devices supports AJAX and HTTP(S) as transferring technology

Despite of huge variety of existing presentation software on the market, this system has a combination of unique characteristics or function set, which cannot be found in any other system:

- like/dislike function for every slide with registration and post-analysis statistics available
- support for enhanced requirements for secure material storage
- voting and testing during the presentation
- lecturer notes and per-slide time-log statistics in peer-to-peer mode
- rich slides content (video-, audio-materials)

All these requirements are not met by any well-known solution (we mean partial analogues from Google, Apple, Microsoft, Webinar.*, Adobe, a large amount of other world-wide and Ukrainian-market products) and this was one of the reasons to develop and real need to have the new software presenting system mentioned above.

3 The Model of System's Behavior

In order to be able to formalize and prove the system correctness condition which described above, we need to give a mathematical model of its behavior. We will use state transition system as a model. The states in this transition system are global states which include execution phases of all software components of the system and the data stored in the memories of these components. Transitions denote quite basic, but not elementary execution steps of software components, including memory reading/storage (program variable assignments) and exchanges of fixed length messages over a network (e.g. reading server-stored slide index from a client device).

The global state of the system's model includes:

- the index of the current slide stored on the central server (denoted as S);

- the index of the slide $slideM$ that is currently displayed by the manager's device;
- the index of the slide the is currently displayed on the i-th client device which we will denote as $slideC_i$.
- the state of the local variables of the program of the i-th client device.

The behavior of the software system (including its server, manager, and client parts) during one system operation cycle can be described in a simplified form using imperative compositional language (IPCL) notation [4] as follows:

Manager $\equiv$
  $[M1]S := slideM[M2]$

Client $\equiv$
  $[C1]newSlide := S;$
  while$[C2](slideC = newSlide)$**do**
    $[C3]newSlide := S;$
  end while
  $[C4]slideC := newSlide[C5]$

Note that here the identifiers in square brackets ([]) mean the labels which denote the code execution phases. E.g. the $[M1]S := slideM[M2]$ means that the manager's program starts execution in a phase denoted as M1 (initial phase) and sends the value $slideM$ the server where it is stored in S, and when this operation completes, the server updates the value S, and acknowledges success of this operation, the manager's program transitions to an execution phase denotes as $M2$.

The behaviour of the distributed presentation software system with n clients can be described in ICPL as follows:

$$\textbf{Software} = Manager \mid\mid Client^n.$$

Here the n-th power means execution of n instances of a program in an interleaving manner [5,7,4].

4 The Formalization and Proof of Total Correctness in Isabelle

To formalize and prove the total correctness condition we will use Isabelle proof assistant [8] which is a generic interactive proof assistant [9] based on a small logical core. The core provides a meta-logic based on a weak form of type theory. This meta-logic is used encode stronger ("object") logics which can be used for formalizing and proving mathematical statements and properties of programs: first-order logic (FOL), higher-order logic (HOL), Zermelo-Fraenkel set theory (ZFC), etc.

The proof assistant is interactive and requires user guidance. However, certain steps in the proof process can be performed automatically (using automated theorem provers). Thus, in general, proof process may be called semi-automatic. In order to formalize a particular fact, a user can introduce definitions of particular data types and predicates and functions on them which describe a certain application domain in the language of the chosen object logic, and then state and prove the lemmas and theorems about the introduced predicates and functions. The correctness of such proofs is checked automatically by Isabelle.

In this paper we use the Isabelle's HOL object logic to formalize the behaviour of the presentation software system and state and prove its total correctness condition.

Below we give the text of our Isabelle formalization with comments which describe the purpose of various introduced elements. For more information about the syntax and semantics Isabelle theories please refer to [8].

theory *eDetailing* **imports** *Main*
begin

— Firstly, let us define data types of the components of the state of the software system (manager's program state, client's program state, server state (*GlobalState*) and the data type of the state of the whole system (State) which includes them all. We also define auxiliary data types *ManagerLabel*, *ClientLabel* which range over labels (code execution phases) in the ICPL code given above and *GlobalData*, *ManagerData*, and *ClientData* as polymorphic record types which range over memory states of the server, manager, and client programs. These records consist of components which represent assignments of values to variables which are referenced in the ICPL code (e.g. *S* is the server variable which stores slide index, *newSlide*, *slideC* are client variables which store slide indices, and *slideC* is the current slide on the client's device, *slideM* is a manager variable which stores the index of the manager's current slide).

— '*a* is a type parameter which denotes the type of slide index (e.g. natural number)
— '*c::finite* is a type parameter which denotes the type of client index (e.g. a finite type which has the same number of elements as the number of client devices)

datatype *ManagerLabel* = *M1* | *M2*

datatype *ClientLabel* = *C1* | *C2* | *C3* | *C4* | *C5*

record *'a GlobalData* =
  S :: *'a*

record *'a ManagerData* =
  slideM :: *'a*

record *'a ClientData* =
  newSlide :: *'a*
  slideC :: *'a*

datatype (*'a*, *'c::finite*) *State* = *State ManagerLabel*

$'c$::*finite* $\Rightarrow$ *ClientLabel*
$'a$ *GlobalData*
$'a$ *ManagerData*
$'c$::*finite* $\Rightarrow$ $'a$ *ClientData*

— Now let us define the transition relation on states of the system. We define it as a predicate *Tr* on pairs of states (s_1, s_2) which is true, if the system can transition from s_1 to s_2 in accordance with the semantics of the ICPL code given above. This transition predicate is represented as a disjunction of 6 simpler transition predicates which denote possible transitions by the manager's and client's programs. Please see [2,3] for the details on the meaning of $Tr1$-$Tr6$ and on how the transition predicate can be obtained from ICPL code.

fun *Tr1* :: ($'a$, $'c$::*finite*) *State* $\Rightarrow$ ($'a$, $'c$::*finite*) *State* $\Rightarrow$ *bool*
where
Tr1 (*State M-1 CS1 GD1 MD1 CD1*) (*State M-2 CS2 GD2 MD2 CD2*) = ((*M-1* = *M1*) & (*M-2* = *M2*) & (*CS1* = *CS2*) & (*S GD2* = *slideM MD2*) & (*MD1* = *MD2*) & (*CD1* = *CD2*))

fun *Tr2j* :: $'c$::*finite* $\Rightarrow$ ($'a$, $'c$::*finite*) *State* $\Rightarrow$ ($'a$, $'c$::*finite*) *State* $\Rightarrow$ *bool*
where
Tr2j j (*State M-1 CS1 GD1 MD1 CD1*) (*State M-2 CS2 GD2 MD2 CD2*) =
((*M-1* = *M-2*) & (*GD1* = *GD2*) & (*MD1* = *MD2*) &
((*CS1 j* = *C1* & *CS2 j* = *C2*
& ($\forall$ *i* . (*i* $\neq$ *j* $\longrightarrow$ *CS1 i* = *CS2 i* & *CD1 i* = *CD2 i*))
& *slideC* (*CD1 j*) = *slideC* (*CD2 j*)
& *newSlide* (*CD2 j*) = *S GD2*)))

fun *Tr2* :: ($'a$, $'c$::*finite*) *State* $\Rightarrow$ ($'a$, $'c$::*finite*) *State* $\Rightarrow$ *bool*
where
Tr2 s1 s2 = ($\exists$ *j* . *Tr2j j s1 s2*)

fun *Tr3j* :: $'c$::*finite* $\Rightarrow$ ($'a$, $'c$::*finite*) *State* $\Rightarrow$ ($'a$, $'c$::*finite*) *State* $\Rightarrow$ *bool*
where
Tr3j j (*State M-1 CS1 GD1 MD1 CD1*) (*State M-2 CS2 GD2 MD2 CD2*) =
((*M-1* = *M-2*) & (*GD1* = *GD2*) & (*MD1* = *MD2*) & (*CD1* = *CD2*)
&
(*CS1 j* = *C2* & *CS2 j* = *C4*
& ($\forall$ *i* . (*i* $\neq$ *j* $\longrightarrow$ *CS1 i* = *CS2 i*))
& *slideC* (*CD1 j*) $\neq$ *newSlide* (*CD1 j*)))

fun *Tr3* :: ($'a$, $'c$::*finite*) *State* $\Rightarrow$ ($'a$, $'c$::*finite*) *State* $\Rightarrow$ *bool*
where
Tr3 s1 s2 = ($\exists$ *j* . *Tr3j j s1 s2*)

fun *Tr4j* :: $'c$::*finite* $\Rightarrow$ ($'a$, $'c$::*finite*) *State* $\Rightarrow$ ($'a$, $'c$::*finite*) *State* $\Rightarrow$ *bool*
where
Tr4j j (*State M-1 CS1 GD1 MD1 CD1*) (*State M-2 CS2 GD2 MD2 CD2*) =
((*M-1* = *M-2*) & (*GD1* = *GD2*) & (*MD1* = *MD2*) & (*CD1* = *CD2*)
&

```
                    ( CS1 j = C2 & CS2 j = C3
                          & (∀ i . (i ≠ j ⟶ CS1 i = CS2 i))
                          & slideC (CD1 j) = newSlide (CD1 j)))
```

```
fun Tr4 :: ('a, 'c::finite) State ⇒ ('a, 'c::finite) State ⇒ bool
where
  Tr4 s1 s2 = (∃ j . Tr4j j s1 s2)

fun Tr5j :: 'c::finite ⇒ ('a, 'c::finite) State ⇒ ('a, 'c::finite) State ⇒ bool
where
  Tr5j j (State M-1 CS1 GD1 MD1 CD1) (State M-2 CS2 GD2 MD2 CD2) =
                   ((M-1 = M-2) & (GD1 = GD2) & (MD1 = MD2) &
                    ( (CS1 j = C3 & CS2 j = C2
                          & (∀ i . (i ≠ j ⟶ CS1 i = CS2 i & CD1 i = CD2 i))
                          & slideC (CD1 j) = slideC (CD2 j)
                          & newSlide (CD2 j) = S GD2) ) )

fun Tr5 :: ('a, 'c::finite) State ⇒ ('a, 'c::finite) State ⇒ bool
where
  Tr5 s1 s2 = (∃ j . Tr5j j s1 s2)

fun Tr6j :: 'c::finite ⇒ ('a, 'c::finite) State ⇒ ('a, 'c::finite) State ⇒ bool
where
  Tr6j j (State M-1 CS1 GD1 MD1 CD1) (State M-2 CS2 GD2 MD2 CD2) =
                   ((M-1 = M-2) & (GD1 = GD2) & (MD1 = MD2) &
                    ( (CS1 j = C4 & CS2 j = C5
                          & (∀ i . (i ≠ j ⟶ CS1 i = CS2 i & CD1 i = CD2 i))
                          & newSlide (CD1 j) = newSlide (CD2 j)
                          & slideC (CD2 j) = newSlide (CD1 j)) ) )

fun Tr6 :: ('a, 'c::finite) State ⇒ ('a, 'c::finite) State ⇒ bool
where
  Tr6 s1 s2 = (∃ j . Tr6j j s1 s2)

definition ClientTr :: 'c ⇒ ('a, 'c::finite) State ⇒ ('a, 'c) State ⇒ bool
where
  ClientTr i s1 s2 = (Tr2j i s1 s2 ∨ Tr3j i s1 s2 ∨ Tr4j i s1 s2 ∨ Tr5j i s1 s2 ∨ Tr6j
i s1 s2)

definition Tr :: ('a, 'c::finite) State ⇒ ('a, 'c::finite) State ⇒ bool
where
  Tr s1 s2 = (Tr1 s1 s2 ∨ Tr2 s1 s2 ∨ Tr3 s1 s2 ∨ Tr4 s1 s2 ∨ Tr5 s1 s2 ∨ Tr6 s1
s2)
```

— Let us introduce several auxiliary predicates,

— The predicate "*StartState s*" means that s is a state in which the programs of the manager's device and all client devices are in their initial execution phases.

primrec *StartState* :: ('*a*, '*c*::*finite*) *State* ⇒ *bool*
where
StartState (*State M CS* - - -) = (*M* = *M1* & (∀ *i* . (*CS i* = *C1*)))

— The predicate "*PreCond x s*" ("precondition") means that *s* is a state in which the programs of all client devices are synchronized with the server, but the program of the manager's device is not, and the manager's device displays the slide number *x*.
primrec *PreCond* :: '*a* ⇒ ('*a*, '*c*::*finite*) *State* ⇒ *bool*
where
PreCond x (*State* - - *GD MD CD*) = ((∀ *i* . (*slideC* (*CD i*) = *S GD*)) & *slideM MD* ≠ *S GD* & *slideM MD* = *x*)

— The predicate '*'PostCond x s*" ("postcondition") means that *s* is a state in which the programs of the manager's device and all client devices are synchronized with the server and display the same slide *x*.
primrec *PostCond* :: '*a* ⇒ ('*a*, '*c*::*finite*) *State* ⇒ *bool*
where
PostCond x (*State* - - *GD MD CD*) = ((∀ *i* . (*slideC* (*CD i*) = *x*)) & *slideM MD* = *x* & *S GD* = *x*)

— The predicate '*'ManagerStop s*" means that *s* is a state in which the execution phase of the program of the manager's device is *M2*, i.e. this program has finished execution after switching to a new slide.
primrec *ManagerStop* :: ('*a*, '*c*::*finite*) *State* ⇒ *bool*
where
ManagerStop (*State M* - - *MD* -) = (*M* = *M2*)

— The predicate '*'ClientStop i s*" means that *s* is a state in which the execution phase of the program of the *i*-th client device is *C5*, i.e. this program has finished execution after switching to a new slide.
primrec *ClientStop* :: '*c*::*finite* ⇒ ('*a*, '*c*::*finite*) *State* ⇒ *bool*
where
ClientStop i (*State* - *CL* - - -) = (*CL i* = *C5*)

— The predicate '*'SwitchingDone x s*" ("*s* is a state after switching to the slide *x*") defined below means that s is a state in which the manager program reaches the label *M2* and which is reachable from from some starting state *s0* in which all clients are synchronized with the server (i.e. display the slide the number of which is stored in the variable *S* on the server) and the manager is not synchronized with the server (i.e. the number of the slide displayed by the manager differs from the slide index stored in the variable *S* on the server) and display the slide x by following transitions of the transition system.

definition *SwitchingDone* :: '*a* ⇒ ('*a*, '*c*::*finite*) *State* ⇒ *bool*
where
SwitchingDone x s = (∃ *s0* . *StartState s0* ∧ *PreCond x s0* ∧ *Tr*^** *s0 s* ∧ *ManagerStop s*)

— The predicate "*Run x sn*" ("*sn* is a run after switching to *x*") defined below means that *sn* is a finite or infinite sequence of states, each consecutive pair of which is related by a transition in the transition system, which starts in a state that satisfies "*SwitchingDone x s*", i.e. *sn* models an execution of the transition system which takes place after the manager program completes execution.

definition $Run :: 'a \Rightarrow (nat \Rightarrow ('a, 'c::finite)\ State\ option) \Rightarrow bool$
where
$$Run\ x\ sn = ((sn\ 0) \neq None \wedge SwitchingDone\ x\ (the\ (sn\ 0)) \wedge (\forall\ n\ .\ sn\ (Suc\ n) \neq None \longrightarrow sn\ n \neq None \wedge Tr\ (the\ (sn\ n))\ (the\ (sn\ (Suc\ n)))))$$

— If *sn* is a run, *i* is a client device index, *k* is an index in the domain of *sn* (i.e. a discrete time moment), the predicate "*Live sn i k*" ("*i*-th client device program is live at time *k* in *sn*") means that at some index $k' \geq k$ (i.e. in the future relative to *k*) sn contains a transition caused by the execution of the *i*-th client device program.

definition $Live :: (nat \Rightarrow ('a, 'c::finite)\ State\ option) \Rightarrow 'c \Rightarrow nat \Rightarrow bool$
where
$Live\ sn\ i\ k = (\exists\ k'\ .\ k' \geq k \wedge (sn\ (Suc\ k')) \neq None \wedge ClientTr\ i\ (the\ (sn\ k'))\ (the\ (sn\ (Suc\ k'))))$

— The predicate "*Liverun x sn*" ("*sn* is a live run after switching to *x*") defined below means that *sn* is a run after switching to *x* in which for each *k* and each client device *i*, the *i*-th client device program is either live at *k* in *sn*, or its execution phase is *C5* (i.e. it terminated).

definition $Liverun :: 'a \Rightarrow (nat \Rightarrow ('a, 'c::finite)\ State\ option) \Rightarrow bool$
where
$Liverun\ x\ sn = (Run\ x\ sn \wedge (\forall\ i\ k\ .\ (sn\ k) \neq None \longrightarrow (Live\ sn\ i\ k \vee ClientStop\ i\ (the\ (sn\ k)))))$

— If sn is a run, the predicate "*Successful x sn*" ("in the run *sn*, eventually, all devices become synchronized with the server and display the slide *x*") defined below means that *sn* contains a state ("*the (sn k)*"), after which each state in *sn* ("*the (sn k')*" for $k' >= k$) is such a state ("*PostCond*") that the manager's device and all client devices are synchronized with the server and display the slide *x*.

definition $Successful :: 'a \Rightarrow (nat \Rightarrow ('a, 'c::finite)\ State\ option) \Rightarrow bool$
where
$Successful\ x\ sn = (\exists\ k\ .\ (sn\ k) \neq None \wedge (\forall\ k'\ .\ k' \geq k \wedge (sn\ k') \neq None \longrightarrow PostCond\ x\ (the\ (sn\ k'))))$

— The e-Detailing presentation software correctness condition which we prove in this paper is formalized as "*Liverun x sn* implies *Successful x sn*" (for any *x*, *sn*), i.e. if *sn* is a live run after switching to *x*, then in *sn*, eventually, all devices become synchronized with the server and display the slide *x*. We prove this implication in the main theorem given below.

— In order to prove the main result let us introduce a sequence of auxiliary definitions and lemmas. We introduce predicates $I1 - I5$ which denote invariants [1] of the system (statements about the states of the system preserved by the transition relation). Their conjunction is denoted as Inv. It is later used to prove properties of runs of the system.

```
primrec I1 :: ('a, 'c::finite) State ⇒ bool
where
  I1 (State M - GD MD -) = ((M = M2) ⟶ (S GD = slideM MD))

primrec I2 :: ('a, 'c::finite) State ⇒ bool
where
  I2 (State M CS GD MD CD) = (M = M1 ⟶ ((S GD ≠ slideM MD
                            & (∀ i . (slideC (CD i) = S GD &
                                 ((CS i) = C1 ∨ (CS i) = C2 ∨ (CS i) = C3) )))))
primrec I3 :: ('a, 'c::finite) State ⇒ bool
where
  I3 (State - CS GD - CD) = (∀ i . (CS i = C4 ⟶ (
                               slideC (CD i) ≠ S GD & newSlide (CD i) = S GD )))

primrec I4 :: ('a, 'c::finite) State ⇒ bool
where
  I4 (State - CS GD - CD) = (∀ i . (CS i = C5 ⟶ slideC (CD i) = S GD))

primrec I5 :: ('a, 'c::finite) State ⇒ bool
where
  I5 (State M CS GD - CD) = (∀ i . (CS i = C1 ∨ slideC (CD i) = newSlide (CD
i) ∨ (M = M2 & newSlide (CD i) = S GD)))

primrec I7j :: 'c::finite ⇒ ('a, 'c::finite) State ⇒ bool
where
  I7j i (State - CS GD - CD) = ( (CS i = C3 ⟶ newSlide (CD i) = (slideC (CD
i))))

fun I7 :: ('a, 'c::finite) State ⇒ bool
where
  I7 s = (∀ i . I7j i s)

definition Inv :: ('a, 'c::finite) State ⇒ bool
where
  Inv s = (I1 s & I2 s & I3 s & I4 s & I5 s)

primrec slideMstate :: ('a, 'c::finite) State ⇒ 'a ⇒ bool
where
  slideMstate (State M - - MD -) x = (slideM MD = x)

primrec ManagerComplete :: ('a, 'c::finite) State ⇒ 'a ⇒ bool
where
  ManagerComplete (State M - - MD -) x = (M = M2 ∧ slideM MD = x)

definition PostInv :: 'a ⇒ ('a, 'c::finite) State ⇒ bool
```

where
  $PostInv\ x\ s = (ManagerComplete\ s\ x \wedge Inv\ s \wedge I7\ s)$

primrec $ClientComplete :: {}'c{::}finite \Rightarrow ('a,\ 'c{::}finite)\ State \Rightarrow bool$
where
  $ClientComplete\ i\ (State\ \text{-}\ \text{-}\ GD\ \text{-}\ CD) = (newSlide\ (CD\ i) = S\ GD \wedge slideC\ (CD\ i) = S\ GD)$

definition $clientcompl :: (nat \Rightarrow ('a,\ 'c{::}finite)\ State\ option) \Rightarrow {}'c \Rightarrow nat \Rightarrow bool$
where
  $clientcompl\ sn\ i\ k = ((sn\ k) \neq None \wedge ClientComplete\ i\ (the\ (sn\ k)))$

— Let us introduce a natural number-valued function $f\ s\ i$, where s is a state (in the transition system) and i is a client. For each fixed i, $\lambda s.(f\ s\ i)$ plays the role of a termination measure for the i-th client program, namely, its value is 0, if the client's program reached state corresponding to the end of the system's operation cycle (the i-th client device, the manager device, and the server are synchronized), and is greater than 0, if it is still working. Moreover, the value of $\lambda s.(f\ s\ i)$ decreases in course of operation i-th client program becoming smaller, when this program is closer to the state in which $f\ s\ i = 0$. This function is used to prove the termination of the client's program reaction to manager's slide change notification

fun $f :: ('a,\ 'c{::}finite)\ State \Rightarrow {}'c{::}finite \Rightarrow nat$
where
$f\ (State\ \text{-}\ CS\ GD\ \text{-}\ CD)\ i =$
 $(if\ (newSlide\ (CD\ i) = slideC\ (CD\ i) \wedge (newSlide\ (CD\ i) = S\ GD))\ then\ 0$
 $else\ (if$
   $(newSlide\ (CD\ i) = slideC\ (CD\ i) \wedge (newSlide\ (CD\ i) \neq S\ GD))\ then\ (if\ CS\ i = C1\ then\ 3\ else\ (if\ CS\ i = C2\ then\ 4\ else\ (if\ CS\ i = C5\ then\ 2\ else\ 3)))$
 $else\ (if$
   $(newSlide\ (CD\ i) \neq slideC\ (CD\ i) \wedge (newSlide\ (CD\ i) = S\ GD))\ then\ (if\ CS\ i = C1\ then\ 3\ else\ (if\ CS\ i = C2\ then\ 2\ else\ 1))$
 $else\ 5)))$

definition $fsum :: ('a,\ 'c{::}finite)\ State \Rightarrow nat$
where
  $fsum\ s = setsum\ (f\ s)\ \{x\ .\ True\}$

definition $ff :: {}'c{::}finite \Rightarrow ('a,\ 'c{::}finite)\ State \Rightarrow ('a,\ 'c{::}finite)\ State \Rightarrow bool$
where
  $ff\ i\ s1\ s2 = (f\ s2\ i < f\ s1\ i \vee (f\ s1\ i = 0 \wedge f\ s2\ i = 0))$

definition $decrj :: {}'c \Rightarrow ('a,\ 'c{::}finite)\ State \Rightarrow ('a,\ 'c{::}finite)\ State \Rightarrow bool$
where
  $decrj\ i\ s1\ s2 = (ff\ i\ s1\ s2 \wedge (\forall\ j\ .\ j = i \vee f\ s1\ j = f\ s2\ j))$

definition $decr :: ('a,\ 'c{::}finite)\ State \Rightarrow ('a,\ 'c{::}finite)\ State \Rightarrow bool$
where
  $decr\ s1\ s2 = (\exists\ i\ .\ decrj\ i\ s1\ s2)$

```
definition fsumval :: (nat ⇒ ('a, 'c::finite) State option) ⇒ nat ⇒ bool
where
  fsumval sn v = (∃ k . (sn k) ≠ None ∧ v = fsum (the (sn k)))

definition isminfsum :: (nat ⇒ ('a, 'c::finite) State option) ⇒ nat ⇒ bool
where
  isminfsum sn m = (fsumval sn m ∧ (∀ v . fsumval sn v ⟶ m ≤ v))


lemma start: StartState s ⟹ PreCond x s ⟹ Inv s
  by (cases s, auto simp add: Inv-def)

lemma trans1: Tr1 s1 s2 ⟹ Inv s1 ⟹ Inv s2
  apply(cases s1, cases s2)
  apply(simp add: Inv-def)
  by (metis ClientLabel.distinct)

lemma trans21234: Tr2 s1 s2 ⟹ Inv s1 ⟹ I1 s2 & I2 s2 & I3 s2 & I4 s2
  apply(cases s1, cases s2)
  apply(simp add: Inv-def)
  by (metis ClientLabel.distinct)

lemma trans25: Tr2 s1 s2 ⟹ Inv s1 ⟹ I5 s2
  apply(cases s1, cases s2)
  apply(auto simp add: Inv-def)
  apply(metis ManagerLabel.exhaust)
  apply(metis)
  apply(metis)
  apply(metis)
  apply(metis ManagerLabel.exhaust)
  by metis

lemma trans2: Tr2 s1 s2 ⟹ Inv s1 ⟹ Inv s2
  using Inv-def trans21234 trans25 by blast

lemma trans3: Tr3 s1 s2 ⟹ Inv s1 ⟹ Inv s2
  apply(cases s1, cases s2)
  apply(simp add: Inv-def)
  by (metis ClientLabel.distinct)

lemma trans4: Tr4 s1 s2 ⟹ Inv s1 ⟹ Inv s2
  apply(cases s1, cases s2)
  apply(simp add: Inv-def)
  by (metis ClientLabel.distinct)

lemma trans51234: Tr5 s1 s2 ⟹ Inv s1 ⟹ I1 s2 & I2 s2 & I3 s2 & I4 s2
  apply(cases s1, cases s2)
  apply(simp add: Inv-def)
  by (metis ClientLabel.distinct)
```

lemma *trans55*: *Tr5 s1 s2* $\Longrightarrow$ *Inv s1* $\Longrightarrow$ *I5 s2*
apply(*cases s1*, *cases s2*)
apply(*auto simp add*: *Inv-def*)
apply(*metis ManagerLabel.exhaust*)
apply(*metis*)
apply(*metis*)
apply(*metis*)
apply(*metis ManagerLabel.exhaust*)
by *metis*

lemma *trans5*: *Tr5 s1 s2* $\Longrightarrow$ *Inv s1* $\Longrightarrow$ *Inv s2*
using *Inv-def trans51234 trans55* **by** *blast*

lemma *trans6*: *Tr6 s1 s2* $\Longrightarrow$ *Inv s1* $\Longrightarrow$ *Inv s2*
apply(*cases s1*, *cases s2*)
apply(*simp add*: *Inv-def*)
by (*metis ClientLabel.distinct*)

lemma *trans7j*: *Tr1 s1 s2* $\vee$ *Tr2j i s1 s2* $\vee$ *Tr3j i s1 s2* $\vee$ *Tr4j i s1 s2* $\vee$ *Tr5j i s1 s2* $\vee$ *Tr6j i s1 s2* $\Longrightarrow$ *I7j i s1* $\Longrightarrow$ *I7j i s2*
apply(*cases s1*, *cases s2*)
by *auto*

lemma *trans7*: *Tr s1 s2* $\Longrightarrow$ *I7 s1* $\Longrightarrow$ *I7 s2*
apply(*simp only*: *Tr-def*)
apply(*cases s1*, *cases s2*)
apply(*auto simp add*: *trans7j*)
apply(*metis ClientLabel.distinct*(*9*))
apply(*metis ClientLabel.distinct*(*15*))
apply(*metis*)
apply(*metis ClientLabel.distinct*(*9*))
by *metis*

lemma *trans-inv*: *Tr s1 s2* $\Longrightarrow$ *Inv s1* $\Longrightarrow$ *Inv s2*
using *Tr-def trans1 trans2 trans3 trans4 trans5 trans6* **by** *blast*

lemma *lem-mgrc-mon*: *Tr s1 s2* $\Longrightarrow$ *ManagerComplete s1 x* $\Longrightarrow$ *ManagerComplete s2 x*
apply(*cases s1*)
apply(*cases s2*)
by (*auto simp add*: *Tr-def*)

lemma *postinv-mon*: *Tr s1 s2* $\Longrightarrow$ *PostInv x s1* $\Longrightarrow$ *PostInv x s2*
by (*metis PostInv-def lem-mgrc-mon trans7 trans-inv*)

lemma *postinv-rmon*: *Tr*ˆ++ *s1 s2* $\Longrightarrow$ *PostInv x s1* $\Longrightarrow$ *PostInv x s2*
apply(*erule tranclp-induct*)
by (*auto simp add*: *postinv-mon*)

lemma *mcpinv*: *StartState s0* $\Longrightarrow$ *PreCond x s0* $\Longrightarrow$ *Tr*ˆ∗∗ *s0 s1* $\Longrightarrow$ (*ManagerComplete s1 x* $\longrightarrow$ *PostInv x s1*)
apply(*erule rtranclp-induct*)
apply(*cases s0*)
apply(*simp*)
apply(*auto simp only*: *PostInv-def*)
apply(*metis* (*no-types*, *hide-lams*) *rtranclp-induct start trans-inv*)
apply(*subgoal-tac I7 s0*)
apply(*metis* (*mono-tags*, *hide-lams*) *rtranclp-induct trans7*)
apply(*cases s0*)
apply(*simp*)
apply(*simp add*: *trans-inv*)
using *trans7* **apply** *blast*
done

lemma *slmd-mon*: *slideMstate s0 x* $\Longrightarrow$ *Tr s0 s1* $\Longrightarrow$ *slideMstate s1 x*
apply(*auto simp add*: *Tr-def*)
apply (*metis State.exhaust Tr1.simps slideMstate.simps*)
apply (*metis State.exhaust Tr2j.simps slideMstate.simps*)
apply (*metis State.exhaust Tr3j.simps slideMstate.simps*)
apply (*metis State.exhaust Tr4j.simps slideMstate.simps*)
apply (*metis State.exhaust Tr5j.simps slideMstate.simps*)
by (*metis State.exhaust Tr6j.simps slideMstate.simps*)

lemma *slmdinv*: *PreCond x s0* $\Longrightarrow$ *Tr*ˆ∗∗ *s0 s1* $\Longrightarrow$ *slideMstate s1 x*
apply(*erule rtranclp-induct*)
apply(*metis PreCond.simps State.exhaust slideMstate.simps*)
apply(*simp add*: *slmd-mon*)
done

lemma *mstopcompl*: *PreCond x s0* $\Longrightarrow$ *Tr*ˆ∗∗ *s0 s1* $\Longrightarrow$ (*ManagerStop s1* $\longrightarrow$ *ManagerComplete s1 x*)
apply(*erule rtranclp-induct*)
apply(*cases s0*, *cases s1*)
apply(*simp*)
apply(*auto*)
apply(*metis* (*full-types*) *ManagerComplete.simps ManagerStop.simps State.exhaust slideMstate.simps slmd-mon slmdinv*)
apply(*simp add*: *lem-mgrc-mon*)
done

lemma *sdpinv*: *SwitchingDone x s* $\Longrightarrow$ *PostInv x s*
by (*metis SwitchingDone-def mcpinv mstopcompl*)

lemma *lt1*: *ManagerComplete s1 x* $\Longrightarrow$ *Tr1 s1 s2* $\Longrightarrow$ *decr s1 s2*
apply(*simp only*: *ff-def decr-def*)
apply(*cases s1*)
apply(*cases s2*)
by *auto*

lemma *lt2*: *Tr2j i s1 s2* $\Longrightarrow$ *ff i s1 s2*
apply(*simp only*: *ff-def*)
apply(*cases s1*)
apply(*cases s2*)
by *auto*

lemma *lt2a*: *Tr2j i s1 s2* $\Longrightarrow$ $j \neq i$ $\Longrightarrow$ *f s1 j* = *f s2 j*
by (*cases s1*, *cases s2*, *simp*)

lemma *lt3*: *Inv s2* $\Longrightarrow$ *Tr3j i s1 s2* $\Longrightarrow$ *ff i s1 s2*
apply(*simp only*: *ff-def*)
apply(*cases s1*)
apply(*cases s2*)
by (*auto simp add*: *Inv-def*)

lemma *lt3a*: *Tr3j i s1 s2* $\Longrightarrow$ $j \neq i$ $\Longrightarrow$ *f s1 j* = *f s2 j*
by (*cases s1*, *cases s2*, *simp*)

lemma *lt4*: *Tr4j i s1 s2* $\Longrightarrow$ *ff i s1 s2*
apply(*simp only*: *ff-def*)
apply(*cases s1*)
apply(*cases s2*)
by *auto*

lemma *lt4a*: *Tr4j i s1 s2* $\Longrightarrow$ $j \neq i$ $\Longrightarrow$ *f s1 j* = *f s2 j*
by (*cases s1*, *cases s2*, *simp*)

lemma *lt5*: *I7 s1* $\Longrightarrow$ *Tr5j i s1 s2* $\Longrightarrow$ *ff i s1 s2*
apply(*simp only*: *ff-def*)
apply(*cases s1*)
apply(*cases s2*)
by *auto*

lemma *lt5a*: *Tr5j i s1 s2* $\Longrightarrow$ $j \neq i$ $\Longrightarrow$ *f s1 j* = *f s2 j*
by (*cases s1*, *cases s2*, *simp*)

lemma *lt6*: *I3 s1* $\Longrightarrow$ *Tr6j i s1 s2* $\Longrightarrow$ *ff i s1 s2*
apply(*simp only*: *ff-def*)
apply(*cases s1*)
apply(*cases s2*)
by *auto*

lemma *lt6a*: *Tr6j i s1 s2* $\Longrightarrow$ $j{\neq}i$ $\Longrightarrow$ *f s1 j* = *f s2 j*
by (*cases s1*, *cases s2*, *simp*)

lemma *trjdecr*: *PostInv x s1* $\Longrightarrow$ *ClientTr i s1 s2* $\Longrightarrow$ *decrj i s1 s2*
apply(*auto simp add*: *PostInv-def ClientTr-def*)
apply(*metis lt2 lt2a decrj-def*)
apply(*metis Tr3.simps lt3 lt3a trans3 decrj-def*)
apply(*metis lt4 lt4a decrj-def*)

apply(*metis I7.elims*(*3*) *lt5 lt5a decrj-def*)
apply(*metis Inv-def lt6 lt6a decrj-def*)
done

lemma *postinv-tr*: *PostInv x s1* $\Longrightarrow$ *Tr s1 s2* $\Longrightarrow$ *decr s1 s2*
apply(*auto simp add*: *PostInv-def Tr-def*)
apply(*metis lt1*)
apply(*metis lt2 lt2a decr-def decrj-def*)
apply(*metis Tr3.simps lt3 lt3a trans3 decr-def decrj-def*)
apply(*metis lt4 lt4a decr-def decrj-def*)
apply(*metis I7.elims*(*3*) *lt5 lt5a decr-def decrj-def*)
apply(*metis Inv-def lt6 lt6a decr-def decrj-def*)
done

lemma *ftrmon*: *PostInv x s1* $\Longrightarrow$ *Tr s1 s2* $\Longrightarrow$ *f s2 i* $\leq$ *f s1 i*
apply(*subgoal-tac decr s1 s2*)
apply (*metis decr-def decrj-def ff-def nat-le-linear not-less*)
by (*simp only*: *postinv-tr*)

lemma *fttrmon*: *PostInv x s1* $\Longrightarrow$ *Tr*^++ *s1 s2* $\Longrightarrow$ ($\forall$ *i* . *f s2 i* $\leq$ *f s1 i*)
apply(*erule tranclp-induct*)
apply(*subgoal-tac decr s1 y*)
apply(*simp add*: *ftrmon*)
apply(*simp only*: *postinv-tr*)
apply(*metis ftrmon order.trans postinv-rmon*)
done

lemma *fsttrmon*: *PostInv x s1* $\Longrightarrow$ *Tr*^** *s1 s2* $\Longrightarrow$ *fsum s2* $\leq$ *fsum s1*
apply(*subgoal-tac PostInv x s1* $\Longrightarrow$ *Tr*^++ *s1 s2* $\Longrightarrow$ *fsum s2* $\leq$ *fsum s1*)
apply(*metis Nitpick.rtranclp-unfold eq-iff*)
by (*simp add*: *fttrmon fsum-def setsum-mono*)

lemma *ccf0*: *ClientComplete i s* $\Longrightarrow$ *f s i* = *0*
apply(*cases s*)
by *auto*

lemma *fnzcc*: *f s i* $\neq$ *0* $\vee$ *ClientComplete i s*
apply(*cases s*)
by *auto*

lemma *cceq*: *ClientComplete i s* = (*f s i* = *0*)
  using *ccf0 fnzcc* **by** *blast*

lemma *fsumsub*: *f s* (*i*::(*'c*::*finite*)) $\leq$ *fsum s*
apply(*simp only*: *fsum-def*)
apply(*subgoal-tac setsum* (λ *j* . (*if j* = *i then* (*f s i*) *else 0*)) {*x* . *True*} = *f s i*)
apply(*smt eq-iff not-less not-less0 setsum-mono*)
apply(*smt finite mem-Collect-eq setsum.cong setsum.delta'*)
done

lemma *fs0cc*: *fsum s* = *0* $\Longrightarrow$ *ClientComplete i s*
apply(*subgoal-tac f s i* = *0*)
apply(*simp only*: *cceq*)
by (*metis fsumsub le-0-eq*)

lemma *ccfs0*: $\forall$ *i* . *ClientComplete i s* $\Longrightarrow$ *fsum s* = *0*
by (*simp add*: *ccf0 fsum-def*)

lemma *fseq*: *fsum s* = *0* $\longleftrightarrow$ ($\forall$ *i* . *ClientComplete i s*)
by (*auto simp add*: *fs0cc ccfs0*)

lemma *fs0pc*: *ManagerComplete s x* $\Longrightarrow$ *Inv s* $\Longrightarrow$ *fsum s* = *0* $\Longrightarrow$ *PostCond x s*
apply(*subgoal-tac* $\forall$ *i* . *ClientComplete i s*)
apply(*cases s*)
apply(*auto simp add*: *Inv-def PostCond-def*)
apply(*simp only*: *fs0cc*)
done

lemma *decfsum*: *decrj i* (*s1*::('*a*,'*c*::*finite*) *State*) *s2* $\Longrightarrow$ (*f s1 i* $\neq$ *0*) $\Longrightarrow$ *fsum s2* < *fsum s1*
apply(*subgoal-tac f s2 i* < *f s1 i*)
apply(*auto simp add*: *decrj-def*)
apply(*subgoal-tac* $\forall$ *i*. *f s2 i* $\leq$ *f s1 i* $\wedge$ *finite* {*c*::'*c* . *True*})
apply(*simp only*: *fsum-def*)
using *setsum-strict-mono-ex1* **apply** *blast*
using *le-eq-less-or-eq* **apply** *fastforce*
apply (*simp add*: *ff-def*)
done

lemma *seqtr*: *Run x sn* $\Longrightarrow$ (*sn k*) $\neq$ *None* $\Longrightarrow$ (*sn k'*) $\neq$ *None* $\Longrightarrow$ *k* $\leq$ *k'* $\Longrightarrow$ *Tr*^** (*the* (*sn k*)) (*the* (*sn k'*))
apply(*induct k'*)
apply(*simp*)
by (*metis le-Suc-eq rtranclp.simps Run-def*)

lemma *seqpi*: *Run x sn* $\Longrightarrow$ (*sn k*) $\neq$ *None* $\Longrightarrow$ *PostInv x* (*the* (*sn k*))
apply(*induct k*)
apply(*metis sdpinv Run-def*)
by (*metis postinv-mon Run-def*)

lemma *seqfsmon*: *Run x sn* $\Longrightarrow$
(*sn k*) $\neq$ *None* $\Longrightarrow$ (*sn k'*) $\neq$ *None* $\Longrightarrow$ *k*$\leq$*k'* $\Longrightarrow$ *fsum* (*the* (*sn k'*)) $\leq$ *fsum* (*the* (*sn k*))
by (*metis seqpi fsttrmon seqtr*)

lemma *rundom*: *Run x sn* $\Longrightarrow$ (*sn k'*) $\neq$ *None* $\Longrightarrow$ $\forall$ *k* . *k*$\leq$*k'* $\longrightarrow$ (*sn k*) $\neq$ *None*
apply(*induct k'*)
apply *simp*
by (*metis le-Suc-eq Run-def*)

lemma *runfsdecr*: *Run x sn* $\Longrightarrow$
Live sn i k $\Longrightarrow \exists\ k' \geq k$. $(sn\ k') \neq None \wedge (fsum(the\ (sn\ k')) < fsum\ (the\ (sn\ k)) \vee ClientComplete\ i\ (the\ (sn\ k')))$
apply(*auto simp only*: *Live-def*)
apply(*subgoal-tac* $(decrj\ i\ (the\ (sn\ k'))\ (the\ (sn\ (Suc\ k'))) \wedge f\ (the\ (sn\ k'))\ i \neq 0) \vee ClientComplete\ i\ (the\ (sn\ k'))$)
apply(*subgoal-tac* $(sn\ (Suc\ k')) \neq None \wedge (fsum\ (the\ (sn\ (Suc\ k'))) < fsum\ (the\ (sn\ k)) \vee ClientComplete\ i\ (the\ (sn\ k')))$)
apply (*meson le-Suc-eq Run-def*)
apply(*subgoal-tac* $fsum\ (the\ (sn\ (Suc\ k'))) < fsum\ (the\ (sn\ k')) \vee ClientComplete\ i\ (the\ (sn\ k'))$)
apply(*subgoal-tac* $fsum\ (the\ (sn\ k')) \leq fsum\ (the\ (sn\ k))$)
using *less-le-trans* **apply** *blast*
apply(*subgoal-tac* $(sn\ k) \neq None \wedge (sn\ k') \neq None$)
apply(*simp only*: *seqfsmon*)
apply(*simp add*: *seqfsmon*)
apply(*metis option.distinct(1) Run-def rundom*)
using *decfsum* **apply** *auto*[*1*]
apply(*metis cceq option.distinct(1) Run-def seqpi trjdecr*)
done

lemma *cscc*: *PostInv x s* $\Longrightarrow$ *ClientStop i s* $\Longrightarrow$ *ClientComplete i s*
apply(*cases s*)
using *Inv-def PostInv-def* **apply** *force*
done

lemma *lfsdecr*: *Liverun x sn* $\Longrightarrow (sn\ k) \neq None \Longrightarrow$
$\exists\ k' \geq k$. $(sn\ k') \neq None \wedge (fsum(the\ (sn\ k')) < fsum\ (the\ (sn\ k)) \vee ClientComplete\ i\ (the\ (sn\ k')))$
apply(*subgoal-tac* $(Live\ sn\ i\ k \vee ClientStop\ i\ (the\ (sn\ k)))$)
apply(*metis cscc eq-imp-le Liverun-def runfsdecr seqpi*)
apply(*simp add*: *Liverun-def*)
done

lemma *minfslv*: *Liverun x sn* $\Longrightarrow$
$(sn\ k) \neq None \Longrightarrow ismininfsum\ sn\ m \Longrightarrow$
$m < fsum\ (the\ (sn\ k)) \vee (\exists\ k'\ .\ clientcompl\ sn\ i\ k')$
apply(*subgoal-tac* $(\exists\ k' \geq k\ .\ (sn\ k') \neq None \wedge\ fsum(the\ (sn\ k')) < fsum\ (the\ (sn\ k))) \vee (\exists\ k' \geq k\ .\ (sn\ k') \neq None \wedge ClientComplete\ i\ (the\ (sn\ k')))$)
apply(*simp only*: *clientcompl-def*)
apply(*metis eq-iff fsumval-def isminfsum-def not-le*)
apply(*metis lfsdecr*)
done

lemma *clcompl*:
assumes *a0*: *Liverun x sn*
shows $\exists\ k'$. *clientcompl sn i k'*
proof −
have *fsumval sn* $(fsum\ (the\ (sn\ 0)))$ **using** *assms fsumval-def Liverun-def Run-def* **by** *metis*

then have $\exists$ *m . isminfsum sn m* **by** (*metis* (*full-types*) *ex-least-nat-le isminfsum-def le0 not-le*)
then obtain *k m* **where** (*sn k*) $\neq$ *None* $\wedge$ *fsum*(*the* (*sn k*)) = *m* $\wedge$ *isminfsum sn m* **by** (*metis fsumval-def isminfsum-def*)
then show *?thesis* **using** *minfslv a0* **by** (*metis not-less-iff-gr-or-eq*)
qed

lemma *ccik*: *Liverun x sn* $\Longrightarrow$
clientcompl sn i0 k $\Longrightarrow$ *clientcompl sn i k'* $\Longrightarrow$ *k'* $\leq$ *k* $\Longrightarrow$ *clientcompl sn i k*
apply(*simp only*: *clientcompl-def*)
apply(*subgoal-tac PostInv x* (*the* (*sn k'*)) $\wedge$ *Tr*^** (*the* (*sn k'*)) (*the* (*sn k*)))
apply(*smt cceq fttrmon le-0-eq rtranclp-induct rtranclp-into-tranclp1*)
by (*metis Liverun-def seqpi seqtr*)

lemma *clcompla*:
assumes *a0*: *Liverun x sn*
shows $\exists$ *k* . $\forall$ *i . clientcompl sn i k*
proof −
let *?A* = λ *i* . (*SOME k'* . *clientcompl sn i k'*)
let *?k* = *Max*(*range*(*?A*))
have *?k* $\in$ *range*(*?A*) **by** *simp*
then obtain *i0* **where** *?k* = *?A i0* **by** *blast*
then have *kdef*: *clientcompl sn i0 ?k* **by** (*metis assms clcompl someI-ex*)
have *ccA*: $\forall$ *i . clientcompl sn i* (*?A i*) **by** (*metis assms clcompl someI-ex*)
have $\forall$ *i . clientcompl sn i ?k*
proof
fix *i*
let *?k'* = *?A i*
have *?k'* $\leq$ *?k* $\wedge$ *clientcompl sn i ?k'* **by** (*simp add*: *ccA*)
then show *clientcompl sn i ?k* **using** *ccik kdef a0* **by** *metis*
qed
then show *?thesis* **by** *blast*
qed

lemma *corr0*:
assumes *a0*: *Liverun x sn*
assumes *a1*: *isminfsum sn m*
assumes *a2*: *m* $\neq$ *0*
shows *False*
proof −
obtain *k* **where** $\forall$ *i . clientcompl sn i k* **using** *clcompla a0* **by** *metis*
then have (*sn k*) $\neq$ *None* $\wedge$ *fsum* (*the* (*sn k*)) = *0* **by** (*simp add*: *clientcompl-def fseq*)
then have *isminfsum sn 0* **by** (*metis fsumval-def isminfsum-def le0*)
then have *m* $\leq$ *0* **using** *a1* **using** *isminfsum-def* **by** *blast*
then show *?thesis* **using** *a2* **by** *simp*
qed

lemma *corr1*: *Liverun x sn* $\Longrightarrow$ $\exists$ *k* . (*sn k*)$\neq$*None* $\wedge$ *fsum*(*the*(*sn k*)) = *0*
apply(*subgoal-tac isminfsum sn 0*)

apply(*simp add*: *fsumval-def isminfsum-def*)
apply(*subgoal-tac* $\exists$ *m* . *isminfsum sn m*)
using *corr0* **apply** *fastforce*
apply(*subgoal-tac fsumval sn* (*fsum* (*the* (*sn 0*))))
apply(*metis* (*full-types*) *ex-least-nat-le isminfsum-def le0 not-le*)
apply(*metis fsumval-def Liverun-def Run-def*)
done

lemma *corr2*: *Run x sn* $\Longrightarrow$
(*sn k*)$\neq$*None* $\wedge$ *fsum*(*the*(*sn k*)) = *0* $\Longrightarrow$ $k' \geq k$ $\Longrightarrow$ (*sn* k') $\neq$ *None* $\Longrightarrow$ *fsum* (*the* (*sn* k')) = *0*
apply(*subgoal-tac PostInv x* (*the* (*sn k*)) $\wedge$ *Tr*^** (*the* (*sn k*)) (*the* (*sn* k')))
using *fsttrmon* **apply** *fastforce*
by (*auto simp add*: *seqpi seqtr*)

— Now let us formulate and prove the main result – the e-Detailing presentation software correctness condition, which, informally, means that if *sn* is a live run after switching to *x*, then in *sn*, eventually, all devices become synchronized with the server and display the slide *x*.

theorem *Main-result*: *Liverun x sn* $\Longrightarrow$ *Successful x sn*
apply(*subgoal-tac* $\forall$ *k* . *Run x sn* $\wedge$ (*sn k*) $\neq$ *None* $\wedge$ *fsum* (*the* (*sn k*)) = *0* $\longrightarrow$ *PostCond x* (*the* (*sn k*)))
apply(*metis corr1 corr2 Liverun-def Successful-def*)
apply(*metis PostInv-def fs0pc seqpi*)
done

end

5 Conclusions

We have presented proof of total correctness of the Infosoft e-Detailing 1.0 software system using Isabelle proof assistant. This gives us confidence that the software provides a correct implementation of the slide presentation algorithm and demonstrates a method of application of interactive theorem proving to software verification problems which are not limited to safety property checking.

Other examples ([10,11] etc.) could also be checked and re-proved with Isabelle proof assistant help as extended samples of this rigorous approach to software engineering.

References

1. Hoare, C.A.R. An Axiomatic Basis for Computer Programming. Communications of the ACM. Vol. 12, no. 10. 576–583 (1969)
2. Polishchuk, N., Kartavov, M. and Panchenko, T. Safety Property Proof using Correctness Proof Methodology in IPCL. Proceedings of the 5th International Scientific Conference "Theoretical and Applied Aspects of Cybernetics". Kyiv: Bukrek. 37–44 (2015)

3. Kartavov, M., Panchenko, T. and Polishchuk, N. Properties Proof Method in IPCL Application To Real-World System Correctness Proof. International Journal "Information Models and Analyses". Sofia, Bulgaria, ITHEA. Vol. 4, No. 2. 142–155 (2015)
4. Panchenko, T. The Methodology for Program Properties Proof in Compositional Languages IPCL [in Ukrainian]. Proceedings of the International Conference "Theoretical and Applied Aspects of Program Systems Development" (TAAPSD'2004). Kyiv. 62–67 (2004)
5. Panchenko, T. The Method for Program Properties Proof in Compositional Nominative Languages IPCL [in Ukrainian]. Problems of Programming. No. 1. 3–16 (2008)
6. Kartavov, M., Panchenko, T. and Polishchuk, N. Infosoft e-Detailing System Total Correctness Proof in IPCL [in Ukrainian]. Bulletin of Taras Shevchenko National University of Kyiv. Series: Physical and Mathematical Sciences. No. 3. 80–83 (2015)
7. Panchenko, T. Compositional Methods for Software Systems Specification and Verification (PhD Thesis) [in Ukrainian]. Kyiv. 177 p. (2006)
8. Nipkow, T., Paulson, L. C., Wenzel, M. Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer. 226 p. (2003)
9. Wiedijk F. The Seventeen Provers of the World. Foreword by Dana S. Scott. F. Wiedijk (editor), Lecture Notes in Artificial Intelligence, Vol. 3600, Springer-Verlag Berlin Heidelberg (2006)
10. Panchenko, T. Application of the Method for Concurrent Programs Properties Proof to Real-World Industrial Software Systems. Proceedings of the International Conference on ICT in Education, Research, and Industrial Applications (ICTERI'2016). 119–128 (2016)
11. Ostapovska, Yu., Panchenko, T., Polishchuk, N. and Kartavov, M. Correctness Property Proof for the Banking System for Money Transfer Payments [in Ukrainian]. Problems of Programming. No. 2-3. 119–132 (2016)