FlexRay: Verification of the FOCUS Specification in Isabelle/HOL. A Case Study.

Maria Spichkova

TUM-I0602
Februar 06

FlexRay: Verification of the Focus Specification in Isabelle/HOL. A Case Study[*]

Maria Spichkova

February 20, 2006

Abstract

This paper represents a translation of the Focus [2] specifications of the FlexRay communication protocol [4] the Higher Order Logic specification in Isabelle/HOL [5] and the corresponding formal Isabelle/HOL proofs for the translated specifications, which shows that the specified FlexRay architecture fulfills the specified FlexRay requirements.

[*]This work was partially funded by the German Federal Ministry of Education and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the author.

Contents

1 Introduction

This paper represents the verification of the FlexRay protocol. The formalization of the FlexRay protocol as the Focus [2] specifications (the requirements specification and the architecture specification) is represented in [4]. The Focus specifications [4] of FlexRay are equal modulo syntax to presented here Isabelle/HOL predicates that represent they semantics.

In practice a stepwise development of systems is used – the requirements specification is refined into a concrete implementation stepwise, via a number of intermediate specifications (see [1]). The paper contains the translation of the Focus specifications in Isabelle/HOL [5] and the proof that the architecture specification fulfill the requirements specification, i.e. is its behavioral refinement.

1.1 Focus

In Focus any specification characterizes the relation between the *communication histories* for the external *input* and *output channels*. The formal meaning of a specification is exactly this external *input/output relation*. The Focus specifications can be structured into a number of formulas each characterizing a different kind of property, the most prominent classes of them are *safety* and *liveness properties*.

The central concept in Focus are *streams*, that represent communication histories of *directed channels*.

In Focus streams are represented as functions mapping natural numbers to messages, where a message for the case of timed stream can be either a data message or *time tick*.

We discuss here only a small subset of that are used In the Focus specification of FlexRay [4] are used the following Focus operators together with standard logical operators:

- An empty stream is represented by $\langle\rangle$.
- $\mathsf{dom}.s$ yields the list $[1...\#s]$, where $\#s$ denotes the length of the stream s.
- $\mathsf{rng}.s$ converts the stream s into a set of its elements : $\{s.j \mid j \in \mathsf{dom}.s\}$.

For detailed description of Focus see [2].

1.2 Isabelle

Isabelle [5] is a specification and verification system implemented in the functional programming language ML. Isabelle/HOL is the specialization of Isabelle for Higher Order Logic. To specify a system with Isabelle means creating *theories*. A theory is a named collection of types, functions (constants), and theorems (lemmas). Similar to the module concept from Focus, we can understand a theory in Isabelle as a module.

The base types in Isabelle/HOL are `bool`, the type of truth values and `nat`, the type of natural numbers. The base type constructors are `list`, the type of lists, and `set`, the type of sets. Function types are denoted by $\Rightarrow$. The operator $\Rightarrow$ is right-associative. The type variables are denoted by `'a`, `'b` etc.

Terms in Isabelle/HOL are formed as in functional programming by applying functions to arguments. Terms may also contain λ-abstractions.

For detailed description of Isabelle/HOL see [5] and [6].

We represent the Focus streams used in the FlexRay specification [4] in Isabelle/HOL as follows:

- Finite untimed streams of type `'a` are represented by the list type: `list 'a`.
- Infinite timed streams of type `'a` are represented by the type `istream 'a` that is a functional type `(nat => list 'a)`.

1.3 FlexRay

FlexRay is a communication protocol for safety critical real time automotive applications that has been developed by the FlexRay Consortium [3]. It is a static time division multiplexing network protocol and supports fault-tolerant clock synchronization via a global time base.

The static message transmission mode 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.

The formal specification of FlexRay in FOCUS is given in [4].

2 FlexRay Specification in Isabelle/HOL

In this section we discuss the FOCUS specifications of FlexRay that are schematically translated into Isabelle/HOL using the representation of FOCUS streams given above.

2.1 Auxiliary Definitions

The type `Message` corresponds here to the message type *Message* that consists of a slot identifier `slot` and the payload `data`. It represents the type *Message* from the FOCUS specifications [4]:

$$\mathsf{type}\ Message = msg(slot : Slot, data : Data)$$

The type of payload is defined as a finite list of type *FT_CNI_Entity* that consists of a message ID of type $\mathbb{N}$ and data of type *DataType*:

$$\mathsf{type}\ MessageId = \mathbb{N}$$
$$\mathsf{type}\ FT\_CNI\_Entity = entity(message\_id \in MessageId, ftcdata \in DataType)$$

Because the type *DataType* is not specified in FOCUS exactly (to have a polymorphic type)[1], we need to specify in Isabelle/HOL the type of data also as a polymorphic type:

```
record 'a FT_CNI_Entity =
   message_id :: nat
   ftcdata    :: 'a

record 'a Message =
   slot :: nat
   data :: "('a FT_CNI_Entity) list"
```

Remark: We also can underspecify the type *DataType*, e.g. as the type of natural numbers:

[1]This also implies that the types *FT_CNI_Entity* and *Message* are polymorph.

```
record FT_CNI_Entity =
  message_id :: nat
  ftcdata    :: nat
```

If the type *DataType* is underspecified in such a way, the types *FT_CNI_Entity* and *Message* are not polymorph any more. In this case we will use in Isabelle/HOL specification `FT_CNI_Entity`, `Message` and `nMessage` instead of `'a FT_CNI_Entity`, `'a Message` and `'a nMessage`.

The proofs for the underspecified specification is the same as for the polymorph specification.

The type `nat` can be easily changed to any other type – it has also no influence on any definition or proof. □

The type `Config` represents the bus configuration and contains the scheduling table `schedule` of a node and the length of the communication round `cycleLength`. This type is defined in the FOCUS specification [4] as follows:

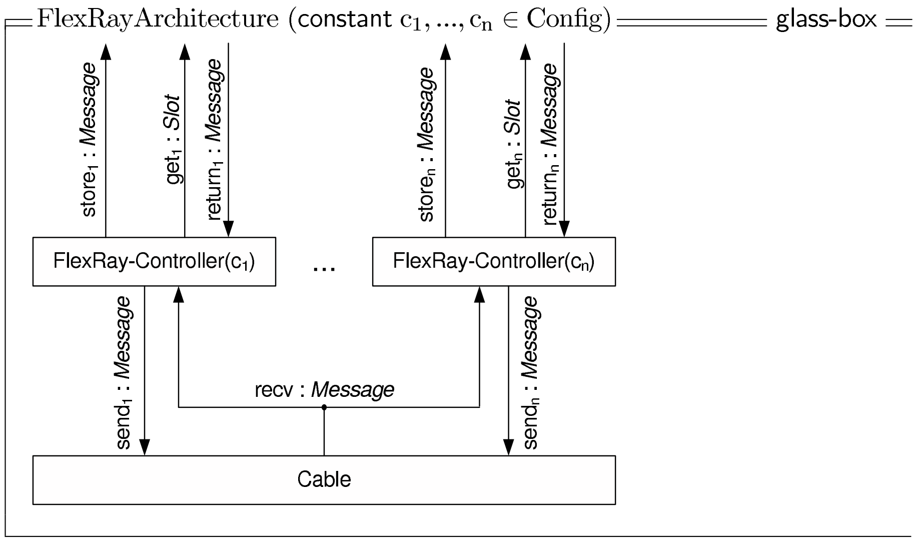
type $Config = conf(schedule : Slot^*, cycleLength : \mathbb{N})$

The corresponding representation in Isabelle/HOL is given below:

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

If a number of channels (streams) of the same type must be represented, the concept of *sheaf of channels* can be used. A sheaf of channels in FOCUS can be understood as an indexed set of channels. We say that a sheaf of channels $x_1, \dots, x_n$ is *correct*, if all the channels $x_1, \dots, x_n$ are of the same type and the number n is greater then zero. In the FOCUS specification *FlexRayArchitecture* [4] (see below) we have the following sheafs of channels:

- $store_1, ..., store_n$
- $get_1, ..., get_n$
- $return_1, ..., return_n$
- $send_1, ..., send_n$



The types `nMessage` and `nSlot` are used to represent sheafs of channels of corresponding types. In the similar way we define the type `nConfig` for the list of parameter constants $c_1, \ldots, c_n$ of the type *Config*.

```
types 'a nMessage = "nat ⇒ ('a Message) istream"

types nSlot = "nat ⇒ nat istream"

types nConfig = "nat ⇒ Config"
```

A sheaf will be represented as a single variable of corresponding type, e.g. the sheaf $send_1, \ldots, send_n$ will be represented as a variable `nSend` of type `'a nMessage`.

To argue in Isabelle/HOL about channels (streams) from a sheaf, e.g. to say that the predicate p is true for any stream of the sheaf $send_1, \ldots, send_n$

$$\forall\, i \in [1..n] :\; p(s_i)$$

the following notation can be used[2]:

```
∀ i < n. p (nSend i)
```

The relation $<$ must be used, because the elements in Isabelle/HOL are counted from 0, in contrast to Focus, where the count goes from 1.

The predicate `disjunctMessage` corresponds to the Focus operator disjunct for the sheaf of channels. It is true, if all streams are disjunct, i.e. in every time unit only one of the streams has any messages to transfer.

```
constdefs
  disjunctMessage :: "nat ⇒ 'a nMessage ⇒ bool"
  "disjunctMessage n nSend
  ≡
  ∀ t k. k < n  ∧ (nSend k t) ≠ [] ⟶
         (∀ j. j < n ∧ j ≠ k ⟶ (nSend j t) = [])"
```

The predicate `CorrectSheaf` is true for nonempty sheaf of channels (the number of channels is greather then zero).

```
constdefs
  CorrectSheaf :: "nat ⇒ bool"
  "CorrectSheaf n ≡ 0 < n"
```

The predicate `maxmsg` is equal modulo syntax to the Focus operator $\mathsf{maxmsg}_n(s)$ [4] that holds for a timed stream s, if this stream contains at every time unit at most n messages.

```
constdefs
  maxmsg :: "nat ⇒ 'a istream ⇒ bool"
 "maxmsg n s ≡  ∀ t. length (s t) ≤ n"
```

The Focus operator $\mathsf{ti}(s, n)$ [4] yields the list of messages that are in the timed stream s between the ticks $n-1$ and n (at the nth time unit). According our representation of the timed Focus streams this operator corresponds in Isabelle/HOL simply to `s n`.

[2]Note that for the cases of many sheafs as well as for the cases of additional restrictions we can use the logical rule $a \rightarrow b \rightarrow c \;\equiv\; a \wedge b \rightarrow c$.

2.2 FlexRay

The requirements specification *FlexRay* [4] of FlexRay in FOCUS is an assumption/guarantee one and is given below:

```
FlexRay (constant $c_1, ..., c_n \in$ Config)                                timed
in    $return_1, ..., return_n$ : $Message$
out   $store_1, ..., store_n$ : $Message$;  $get_1, ..., get_n$ : $Slot$
-----------------------------------------------------------------------------
 asm  $\forall\, i \in [1..n]$ : $\mathsf{maxmsg}_1(return_i)$
      $DisjointSchedules(c_1, ..., c_n)$
      $IdenticCycleLength(c_1, ..., c_n)$
 gar  $MessageTransmission(return_1, ..., return_n, store_1, ..., store_n, get_1, ..., get_n,$
          $c_1 ..., c_n)$
      $\forall\, i \in [1..n]$ : $\mathsf{maxmsg}_1(get_i) \wedge \mathsf{maxmsg}_1(store_i)$
```

The predicate `FlexRay` represents the semantics of the FOCUS specification *FlexRay*:

constdefs
```
  FlexRay ::
    "nat ⇒ 'a nMessage ⇒ nConfig ⇒ 'a nMessage ⇒ nSlot ⇒ bool"
 "FlexRay n nReturn nC nStore nGet
  ≡
   (CorrectSheaf n  ∧
   (∀ i < n. maxmsg 1 (nReturn i)) ∧
   (DisjointSchedules n nC) ∧ (IdenticCycleLength n nC)
    ⟶
   ((MessageTransmission n nReturn nStore nGet nC)  ∧
   (∀ i < n. maxmsg 1 (nGet i) ∧ maxmsg 1 (nStore i))))"
```

The predicates *DisjointSchedules*, *IdenticCycleLength*, *MessageTransmission* from the FOCUS specifications [4] are equal modulo syntax to predicates the same name that we specify in Isabelle/HOL.

The predicate `DisjointSchedules` is true for sheaf of channels of type `nConfig`, if all bus configurations have disjoint scheduling tables:

```
DisjointSchedules
 $c_1, ..., c_n \in Config$
-----------------------------------------------------------------------------
 $\forall\, i, j \in [1..n], j \neq i$ :
    $\forall\, x \in \mathsf{rng}.schedule(c_i),\ y \in \mathsf{rng}.schedule(c_j)$ :
       $x \neq y$
```

The corresponding representation in Isabelle/HOL (`mem` denotes here the Isabelle/HOL operator "member of the list"):

constdefs
```
  DisjointSchedules :: "nat ⇒ nConfig ⇒ bool"
 "DisjointSchedules n nC
 ≡
 ∀ i j. i < n ∧ j < n ∧ i ≠ j ⟶
 (∀  x y. (x mem (schedule (nC i))) ∧ (y mem (schedule (nC j)))  ⟶ x ≠ y)"
```

The predicate `IdenticCycleLength` is true for sheaf of channels of type `nConfig`, if all bus configurations have the equal length of the communication round:

IdenticCycleLength

$c_1, ..., c_n \in \mathit{Config}$

$\forall\, i, j \in [1..n]:$

$\quad \mathit{cycleLength}(c_i) = \mathit{cycleLength}(c_j)$

The corresponding representation in Isabelle/HOL:

constdefs
```
  IdenticCycleLength :: "nat ⇒ nConfig ⇒ bool"
 "IdenticCycleLength n nC
 ≡
 ∀ i j. i < n ∧ j < n ⟶
  cycleLength (nC i) = cycleLength (nC j)"
```

The predicate `MessageTransmission` [4] defines the correct message transmission:

MessageTransmission

$\mathit{store}_1, ..., \mathit{store}_n, \mathit{return}_1, ..., \mathit{return}_n \in \mathit{Message}^{\underline{\omega}}$
$\mathit{get}_1, ..., \mathit{get}_n \in \mathit{Slot}^{\underline{\omega}}$
$c_1, ..., c_n \in \mathit{Config}$

$\forall\, t \in \mathbb{N}, k \in [1..n]:$

$\quad s \in \mathit{schedule}(c_k) : s = t \bmod \mathit{cycleLength}(c_k) \rightarrow$

$\qquad \mathsf{ti}(\mathit{get}_k, t) = \langle s \rangle \ \wedge$

$\qquad \forall j \in [1..n], j \neq k : \mathsf{ti}(\mathit{store}_j, t) = \mathsf{ti}(\mathit{return}_k, t)$

The corresponding representation in Isabelle/HOL:

constdefs
```
  MessageTransmission ::
    "nat ⇒ 'a nMessage ⇒ 'a nMessage ⇒ nConfig ⇒ nSlot ⇒ bool"
 "MessageTransmission n nReturn nStore nGet nC
 ≡
  CorrectSheaf n ∧
  ( ∀ t. ∀ k < n.
    ((t mod (cycleLength (nC k))) mem (schedule (nC k)))
    ⟶
    (nGet k t) = [t mod (cycleLength (nC k))]
     ∧  (∀ j. j < n ∧ j ≠ k ⟶ (nStore j t) =  (nReturn k t)) )"
```

The predicate `FlexRayArch` represents the semantics of the FOCUS specification *FlexRayArch* [4] that is an assumption/guarantee one. The assumption part of the specification *FlexRayArch* is the same as of the specification *FlexRay*. The guarantee part is represented by the specification *FlexRayArchitecture* (see above) that is a composite one and consists of the component *Cable* and n components *FlexRay_Controller* (for n nodes).

The specification *FlexRayArch* is refinement of of the specification *FlexRay* – this will be shown in the Section 3.

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

in $return_1, ..., return_n : Message$

out $store_1, ..., store_n : Message;\ get_1, ..., get_n : Slot$

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

$DisjointSchedules(c_1, ..., c_n)$

$IdenticCycleLength(c_1, ..., c_n)$

gar $FlexRayArchitecture$ (constant $c_1, ..., c_n \in Config$)

$(return_1, ..., return_n, store_1, ..., store_n, get_1, ..., get_n)$

The corresponding representation in Isabelle/HOL:

constdefs
```
   FlexRayArch ::
     "nat ⇒ 'a nMessage ⇒ nConfig ⇒ 'a nMessage ⇒ nSlot ⇒ bool"
  "FlexRayArch n nReturn nC nStore nGet
   ≡
   (CorrectSheaf n  ∧
   (DisjointSchedules n nC) ∧ (IdenticCycleLength n nC) ∧
   (∀ i < n. maxmsg 1 (nReturn i))
   ⟶
   (FlexRayArchitecture n nReturn nC nStore nGet))"
```

where

constdefs
```
   FlexRayArchitecture ::
     "nat ⇒ 'a nMessage ⇒ nConfig ⇒ 'a nMessage ⇒ nSlot ⇒ bool"
  "FlexRayArchitecture n nReturn nC nStore nGet
   ≡
   ∃ nSend recv.
    CorrectSheaf n ∧ (Cable n nSend recv) ∧
    (∀ i < n. FlexRay_Controller (nReturn i) recv (nC i)
             (nStore i) (nGet i) (nSend i))"
```

2.3 Cable

The predicate `Cable` represents the semantics of the FOCUS Assumption/Gurantee-specification *Cable*:

Cable ═══════════════════════════════ timed

in $send_1, ..., send_n : Message$

out $recv : Message$

asm disjunct($send_1, ..., send_n$)

gar Broadcast($send_1, ..., send_n, recv$)

constdefs
```
  Cable :: "nat ⇒ nMessage ⇒ Message istream ⇒ bool"
 "Cable n nSend recv
  ≡
  CorrectSheaf n ∧ (disjunctMessage n nSend)
  ⟶
  (Broadcast n nSend recv)"
```

The predicate `Broadcast` represents Isabelle/HOL the semantics of the corresponding predicate defined in Focus:

Broadcast

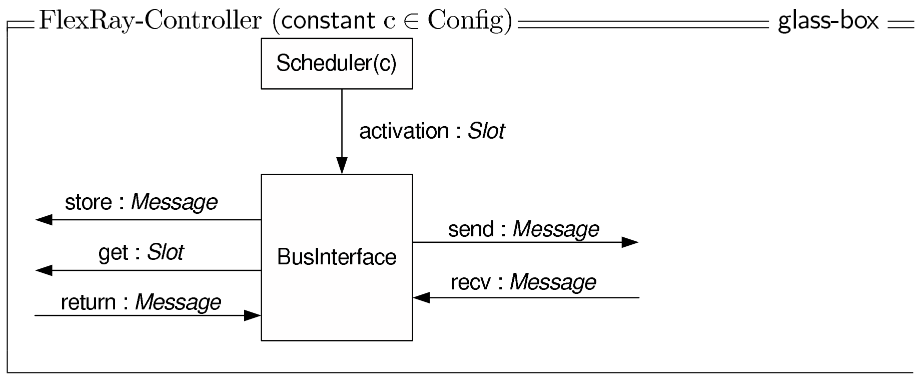
$send_1, ..., send_n, recv \in Message^{\underline{\omega}}$

```
∀ t ∈ ℕ :
   if  ∃ k ∈ [1...n] : ti(send_k, t) ≠ ⟨⟩
      then  ti(recv, t) = ti(send_k, t)
      else  ti(recv, t) = ⟨⟩
   fi
```

constdefs
```
  Broadcast ::
    "nat ⇒ nMessage ⇒ Message istream ⇒ bool"
 "Broadcast n nSend recv
  ≡
  CorrectSheaf n ∧
 (∀ t.
   ( if ∃ k < n. ((nSend k) t) ≠ []
     then (recv t) = ((nSend (SOME k. k < n ∧ ((nSend k) t) ≠ [])) t)
     else (recv t) = []) )"
```

2.4 FlexRay-Controller

The predicate `FlexRay_Controller` represents the semantics of the Focus specification of the component FlexRay-Controller that is a composite one and consists of the components *Scheduler* and *BusInterface*.



constdefs
```
  FlexRay_Controller ::
    "Message istream ⇒ Message istream ⇒ Config ⇒
     Message istream ⇒ nat istream ⇒ Message istream ⇒ bool"
 "FlexRay_Controller return recv c store get send
  ≡
  ∃ activation.
  (Scheduler c activation) ∧
  (BusInterface activation recv return get send store)"
```

2.4.1 Scheduler

The predicate `Scheduler` represents the semantics of the FOCUS specification of the corresponding component.

Scheduler (constant c ∈ Config) — timed

out $activation : Slot$

univ $slot \in \mathbb{N}$

```
∀ $t \in \mathbb{N}$ :
  $slot = t \bmod cycleLength(c)$
  if $slot \in schedule(c)$
    then $\mathrm{ti}(activation, t) = \langle slot \rangle$
    else $\mathrm{ti}(activation, t) = \langle\rangle$
  fi
```

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

2.4.2 BusInterface

The predicate `BusInterface` represents the semantics of the Focus specification of the component *BusInterface* that represents the receive and the send of messages.

```
BusInterface                                                    timed
in   activation : Slot; recv, return : Message
out  get : Slot; send, store : Message
----------------------------------------------------------------------
Receive(recv, store, activation)

Send(activation, get, return, send)
```

constdefs

```
  BusInterface ::
    "nat istream ⇒ Message istream ⇒ Message istream ⇒
     nat istream ⇒ Message istream ⇒ Message istream ⇒ bool"
 "BusInterface activation recv return get send store
 ≡
  (Receive recv store activation) ∧
  (Send return send get activation)"
```

The predicates `Send` and `Receive` define the corresponding Focus relations on the streams.

```
Receive
recv, store ∈ Message^ω; activation ∈ Slot^ω
----------------
∀ t ∈ ℕ :
  if ti(activation, t) = ⟨⟩
    then ti(store, t) = ti(recv, t)
    else ti(store, t) = ⟨⟩
  fi
```

constdefs

```
  Receive ::
    "Message istream ⇒ Message istream ⇒ nat istream ⇒ bool"
 "Receive recv store activation
 ≡
 ∀ t.
  ( if  (activation t) = []
    then (store t) = (recv t)
    else (store t) = [])"
```

Send

$return, send \in Message^{\underline{\omega}};\ get, activation \in Slot^{\underline{\omega}}$

```
∀ $t \in \mathbb{N}$ :
  if $\mathsf{ti}(activation, t) = \langle\rangle$
    then $\mathsf{ti}(get, t) = \langle\rangle \wedge \mathsf{ti}(send, t) = \langle\rangle$
    else $\mathsf{ti}(get, t) = \mathsf{ti}(activation, t) \wedge \mathsf{ti}(send, t) = \mathsf{ti}(return, t)$
  fi
```

constdefs
```
  Send ::
   "Message istream ⇒ Message istream ⇒ nat istream ⇒ nat istream ⇒ bool"
 "Send return send get activation
  ≡
  ∀ t.
  ( if  (activation t) = []
    then (get t) = [] ∧ (send t) = []
    else (get t) = (activation t) ∧ (send t) = (return t)
   )"
```

3 Proofs in Isabelle/HOL

A Specification S_2 is called a *behavioral refinement* (written $S_1 \leadsto S_2$) of a specification S_1 if they have the same syntactic interface and any I/O history of S_2 is also an I/O history of S_1. Formally, we need to show that any I/O history of S_2 is an I/O history of S_1, but S_1 may have additional I/O histories:

$$[\![S_2]\!] \Rightarrow [\![S_1]\!]$$

In Isabelle it means to prove that the formula that corresponds to $[\![S_2]\!]$ implies the formula that corresponds to $[\![S_1]\!]$.

3.1 Proof of the Refinement

The lemma `main_fr_refinement` says that the specification *FlexRayArch* is refinement of the specification *FlexRay*: the predicate `FlexRayArch` that represents the semantics of the architecture specification *FlexRayArch* implies the `FlexRay` that represents the semantics of the requirements specification *FlexRay* .

To prove this lemma we used the definitions of the predicates `FlexRay`, `FlexRayArch`, `FlexRayArchitecture` and `CorrectSheaf`, Isabelle/HOL reasoning methods [5] `slarify`, `clarsimp` and `auto` that works automatically and the rule `conjI` that splits the subgoal of the form $P \wedge Q$ into two new subgoal P and Q. To prove the resulting subgoals we used auxiliary lemmas `fr_refinement_MessageTransmission` (see Section 3.2) and `fr_refinement_maxmsg` (see Section 3.3).

lemma `main_fr_refinement:`
```
"⋀ n nReturn nC nStore nGet.
  FlexRayArch n nReturn nC nStore nGet  ⟹ FlexRay n nReturn nC nStore nGet"
  apply (simp add: FlexRayArch_def FlexRay_def)
  apply (simp add: FlexRayArchitecture_def)
  apply (simp add: CorrectSheaf_def)
  apply clarify
  apply (rule conjI)
  apply clarsimp
  apply (simp add: fr_refinement_MessageTransmission)
  apply clarsimp
  apply (erule fr_refinement_maxmsg)
  apply auto
  done
```

3.2 Lemma fr_refinement_MessageTransmission

This lemma says: for any natural number *n*, for any stream *recv*, for all parameters *nC* (corresponds to $c_1, \ldots, c_n$) and for all sheafs of channels

- *nReturn* (corresponds to the channles $return_1, \ldots, return_n$),
- *nStore* (corresponds to the channles $store_1, \ldots, store_n$),
- *nGet* (corresponds to the channles $get_1, \ldots, get_n$) and
- *nSend* (corresponds to the channles $send_1, \ldots, send_n$)

the following holds: If holds that

- the number *n* is greater then zero,
- the predicate *Cable* is true for the corresponding streams,
- the predicate *FlexRay_Controller* is true for the corresponding streams on every node *i* of the *n* nodes,
- the parameters $c_1, \ldots, c_n$ has disjoint scheduling tables and the equal cycle lengths: *DisjointSchedules n nC* and *IdenticCycleLength n nC*,
- the streams $return_1, \ldots, return_n$ contain at every time unit at most 1 message,

then the predicate *MessageTransmission* for the corresponding streams must be true.

To prove the lemma we have used the definitions of the predicates *MessageTransmission*, *CorrectSheaf*, *FlexRay_Controller*, *Send*, *Scheduler*, an auxiliary lemma *fr_nStore_nReturn* (see Section 3.4) as well as the standard Isabelle/HOL rules and methods like the method *clarify*.

```
lemma fr_refinement_MessageTransmission:
"⋀n nReturn nC nStore nGet nSend recv.
 ⟦ Cable n nSend recv;
   ∀i<n. FlexRay_Controller (nReturn i) recv
         (nC i) (nStore i) (nGet i) (nSend i);
   0 < n;
   DisjointSchedules n nC; IdenticCycleLength n nC;
   ∀i<n. maxmsg (Suc 0) (nReturn i)⟧
⟹ MessageTransmission n nStore nReturn nGet nC"
  apply (simp add: MessageTransmission_def)
  apply (simp add: CorrectSheaf_def)
  apply clarify
  apply (rule conjI)
  apply (erule_tac x="k" in allE)
  apply (simp add: FlexRay_Controller_def)
  apply (simp add: BusInterface_def)
apply clarify
apply (simp add: Send_def)
apply (simp add:  Scheduler_def)
apply (erule_tac x="t" in allE)
apply (simp add: Let_def)
apply (simp add: fr_nStore_nReturn)
done
```

3.3 Lemma fr_refinement_maxmsg

This lemma says: for any natural numbers *n* and *i*, for any stream *recv*, for all parameters *nC* and for all sheafs of channels *nReturn*, *nStore*, *nGet* and *nSend* the following holds: If holds that

- the number *n* is greater then zero and the number *i* is less then *n*,
- the predicate *Cable* is true for the corresponding streams,
- the streams $return_1, \dots, return_n$ contain at every time unit at most 1 message,
- the parameters $c_1, \dots, c_n$ has disjoint scheduling tables and the equal cycle lengths: *DisjointSchedules n nC* and *IdenticCycleLength n nC*,
- the predicate *FlexRay_Controller* is true for the corresponding streams on every node *i* of the *n* nodes,

then the streams get_i and $store_i$ must contain at every time unit at most 1 message: *maxmsg (Suc 0) (nGet i)* and *maxmsg (Suc 0) (nSend i)*.

To prove the lemma we prove first that the streams of the sheaf *nSend* are disjunct (an additional subgoal *"disjunctMessage n nSend"*) using an auxiliary lemma *disjunctMessage_lemma* (see Section 3.5), and after that we have used auxiliary lemmas *fr_refinement_maxmsg_nget* (see Section 3.10) and *fr_refinement_maxmsg_nstore* (see Section 3.12) as well as the automatical Isabelle/HOL method *auto*.

```
lemma fr_refinement_maxmsg:
"⋀n nReturn nC nStore nGet nSend recv i.
 ⟦ Cable n nSend recv;
   i < n;  0 < n;
   ∀i<n. maxmsg (Suc 0) (nReturn i);
   DisjointSchedules n nC; IdenticCycleLength n nC;
   ∀i<n. FlexRay_Controller (nReturn i) recv
         (nC i) (nStore i) (nGet i) (nSend i)⟧
 ⟹ maxmsg (Suc 0) (nGet i) ∧ maxmsg (Suc 0) (nStore i)"
  apply (subgoal_tac "disjunctMessage n nSend")
    prefer 2
    apply (simp add: disjunctMessage_lemma)
  apply (rule conjI)
  apply (simp add: fr_refinement_maxmsg_nGet)
  apply (erule fr_refinement_maxmsg_nStore)
  apply auto
  done
```

3.4 Lemma fr_nStore_nReturn

The lemma *fr_nStore_nReturn* says: for any time unit *t*, for any natural numbers *n* and *k*, for any stream *recv*, for all parameters *nC* and for all sheafs of channels *nReturn*, *nStore*, *nGet* and *nSend* the following holds: If holds that

- the number *n* is greater then zero and the number *k* is less then *n*,

- the predicate `Cable` is true for the corresponding streams,
- the parameters $c_1, \dots, c_n$ has disjoint scheduling tables and the equal cycle lengths: `DisjointSchedules n nC` and `IdenticCycleLength n nC`,
- the predicate `FlexRay_Controller` is true for the corresponding streams on every node `i` of the `n` nodes,
- the slot `t mod cycleLength (nC k)` occurs in the the scheduling table of the `k`th node

then the list of messages at every time unit `t` in every `j`th stream ($j \neq k$) from the sheaf `nStore` must be equal to list of messages at the time unit `t` in the `k`th stream from the sheaf `nReturn`.

To prove the lemma we have used the standard Isabelle/HOL methods `clarify` and `simp`. First we prove that the streams of the sheaf `nSend` (excepting the `k`th stream) have an empty list of messages at the time unit `t` (an additional subgoal `"∀ j. j < n ∧ j ≠ k" ⟶ (nSend j) t = [])` using an auxiliary lemma `fr_nC_Send` (see Section 3.9).

Then we use the definitions of the predicates `Cable` and `CorrectSheaf` and prove that the streams of the sheaf `nSend` are disjunct (an additional subgoal `"disjunctMessage n nSend"`) using an auxiliary lemma `disjunctMessage_lemma` (see Section 3.5), and we make a copy of an assumption about the `FlexRay_Controller` predicate to instantiate its quantifier with the variables `k` and `j`.

After that we use the definitions of the predicates `Broadcast` and `CorrectSheaf` (the second predicate `CorrectSheaf` comes from the definition of `Broadcast`), instantiate the quantifier from the definition of `Broadcast` with `t`, use the definitions of the predicates `FlexRay_Controller` and `Scheduler`, and instantiate the quantifiers from the definition `Scheduler` also with `t`.

Then we prove that the slot `t mod cycleLength (nC j)` cannot occur in the scheduling table of the `j`th node because it occurs in the the scheduling table of the `k`th node (because the tables are disjunct) using an auxiliary lemma `correct_DisjointSchedules1` (see Section 3.7).

After that we use the definitions of the predicates `BusInterface`, `Send` and `Receive` together with standard Isabelle/HOL rules and methods.

```
lemma fr_nStore_nReturn:
"⋀n nReturn nC nStore nGet nSend recv t k.
 ⟦Cable n nSend recv;
  ∀ i<n. FlexRay_Controller (nReturn i) recv
        (nC i) (nStore i) (nGet i) (nSend i);
  0 < n; k < n;
  DisjointSchedules n nC;
  IdenticCycleLength n nC;
  t mod cycleLength (nC k) mem schedule (nC k)⟧
  ⟹
   ∀j. j < n ∧ j ≠ k ⟶ nStore j t = nReturn k t"
  apply clarify
  apply (subgoal_tac
    "∀j. j < n ∧ j ≠ k ⟶ (nSend j) t = []")
    prefer 2
    apply (simp add: fr_nC_Send)
  apply (simp add: Cable_def)
  apply (simp add: CorrectSheaf_def)
  apply (subgoal_tac "disjunctMessage n nSend")
    prefer 2
    apply (simp add: disjunctMessage_lemma)
apply simp
apply (subgoal_tac
 "∀i<n. FlexRay_Controller (nReturn i) recv
       (nC i) (nStore i) (nGet i) (nSend i)")
  prefer 2
  apply simp
apply (erule_tac x="j" in allE)
apply (rotate_tac 8)
apply (erule_tac x="k" in allE)
apply (simp add: Broadcast_def)
apply (simp add: CorrectSheaf_def)
apply (erule_tac x="t" in allE)
apply (simp add: FlexRay_Controller_def)
apply clarify
apply (simp add:  Scheduler_def)
apply (rotate_tac 8)
apply (erule_tac x="t" in allE)
apply (erule_tac x="t" in allE)
apply (simp add: Let_def)
apply (subgoal_tac
  "¬ (t mod cycleLength (nC j) mem schedule (nC j))")
  prefer 2
  apply (erule correct_DisjointSchedules1)
  apply assumption+
  apply simp
apply (simp add: BusInterface_def)
apply clarify
apply (simp add: Send_def)
apply (rotate_tac 9)
apply (erule_tac x="t" in allE)
apply (erule_tac x="t" in allE)
apply (simp split add: split_if_asm)
apply (subgoal_tac
  "(SOME i. i < n ∧  nSend i t ≠ []) = ka")
  prefer 2
  apply (simp add: correct_disjunctMessage)
apply simp
apply (erule_tac
  V="(SOME i. i < n ∧ nSend i t ≠ []) = ka"
  in thin_rl)
apply (simp only: disjunctMessage_def)
apply (erule_tac x="t" in allE)
apply (rotate_tac 8)
apply (erule_tac x="ka" in allE)
apply simp
apply (erule_tac x="ka" in allE)
apply simp
apply (erule_tac V="ka = k" in thin_rl)
apply (simp add: Receive_def)+
done
```

3.5 Lemma disjunctMessage_lemma

The lemma *disjunctMessage_lemma* says: for any natural number *n*, for any stream *rcv*, for all parameters *nC* and for all sheafs of channels *nReturn*, *nStore*, *nGet* and *nSend* the following holds: If holds that

- the number *n* is greater then zero,

- the parameters $c_1, \ldots, c_n$ has disjoint scheduling tables and the equal cycle lengths: *DisjointSchedules n nC* and *IdenticCycleLength n nC*,
- the predicate *FlexRay_Controller* is true for the corresponding streams on every node *i* of the *n* nodes,

then all the streams from the sheaf *nSend* are disjunct.

```
lemma disjunctMessage_lemma:
"⋀ n nReturn rcv nC nStore nGet nSend.
⟦ DisjointSchedules n nC; 0 < n;
  IdenticCycleLength n nC;
  ∀ i < n. FlexRay_Controller (nReturn i) rcv
          (nC i) (nStore i) (nGet i) (nSend i) ⟧
⟹ disjunctMessage n nSend"
  apply (subgoal_tac
    "∀ i<n. FlexRay_Controller (nReturn i) rcv
           (nC i) (nStore i) (nGet i) (nSend i)")
    prefer 2
    apply simp
  apply (simp only: disjunctMessage_def)
  apply (rule allI)+
  apply (erule_tac x="k" in allE)
  apply clarify
  apply (erule_tac x="j" in allE)
  apply (simp add: FlexRay_Controller_def)
  apply clarify
  apply (simp add: DisjointSchedules_def)
  apply (erule_tac x="k" in allE)
  apply (erule_tac x="j" in allE)
  apply (simp add:  BusInterface_def)
  apply clarify
  apply (simp add: Send_def)
  apply (rotate_tac 8)
  apply (erule_tac x="t" in allE)
  apply (erule_tac x="t" in allE)
  apply (simp split add: split_if_asm)
  apply auto
  apply (simp add: Scheduler_def)
  apply (erule_tac x="t" in allE)
  apply (erule_tac x="t" in allE)
  apply (simp add: Let_def)
  apply (simp split add: split_if_asm)
  apply (erule_tac x="t mod cycleLength (nC k)" in allE)
  apply (erule_tac x="t mod cycleLength (nC j)" in allE)
  apply simp
  apply (simp only: IdenticCycleLength_def)
  apply (erule_tac x="k" in allE)
  apply (erule_tac x="j" in allE)
  apply simp
  done
```

3.6 Lemma correct_disjunctMessage

The lemma *disjunctMessage_lemma* says: for any time unit *t*, for any natural numbers *n* and *k*, for any stream *recv* and for all sheafs of channels *nSend* the following holds: If holds that all the streams from the sheaf *nSend* are disjunct and the *k*th stream from the sheaf *nSend* is nonempty at the time unit *t*, then the Isabelle/HOL operator *SOME*[3] returns *k* for the description *SOME i. i < n ∧ nSend i t ≠ []* (there exists exactly one stream from the sheaf *nSend* that is nonempty at the time unit).

```
lemma correct_disjunctMessage:
"⋀ n nSend recv t k.
 ⟦ disjunctMessage n nSend;  nSend k t ≠ []; k < n ⟧
  ⟹ (SOME i. i < n ∧  nSend i t ≠ []) = k"
  apply (simp add: disjunctMessage_def)
  apply (erule_tac x="t" in allE)
  apply (erule_tac x="k" in allE)
  apply auto
  done
```

3.7 Lemma correct_DisjointSchedules1

The lemma *correct_DisjointSchedules1* says: for any time unit *t*, for any natural numbers *n*, *k* and *j*, for all parameters *nC* the following holds: If holds that

- the numbers *k* and *j* are unequal and less then *n*,
- the parameters $c_1, \dots, c_n$ has disjoint scheduling tables and the equal cycle lengths: *DisjointSchedules n nC* and *IdenticCycleLength n nC*,
- the slot *t mod cycleLength (nC k)* occurs in the the scheduling table of the *k*th node

the slot *t mod cycleLength (nC j)* cannot occur in the scheduling table of the *j*th node.

```
lemma correct_DisjointSchedules1:
"⋀ n nC k t j.
 ⟦ DisjointSchedules n nC; IdenticCycleLength n nC;
   (t mod cycleLength (nC k)) mem schedule (nC k);
   k < n; j < n; k ≠ j ⟧
  ⟹
 ¬ (t mod cycleLength (nC j) mem schedule (nC j))"
  apply (simp add: DisjointSchedules_def)
  apply (erule_tac x="k" in allE)
  apply (erule_tac x="j" in allE)
  apply clarify
  apply (simp only: IdenticCycleLength_def)
  apply (erule_tac x="k" in allE)
  apply (erule_tac x="j" in allE)
  apply auto
  done
```

3.8 Lemma fr_Send

The lemma *fr_Send* says: for any time unit *t*, for any natural numbers *n*, *k* and *i*, for any stream *recv*, for all parameters *nC* and for all sheafs of channels *nReturn*, *nStore*, *nGet* and *nSend* the following holds: If holds that

- the number *n* is greater then zero and the numbers *k* and *i* are less then *n*,
- the parameters $c_1, \dots, c_n$ has disjoint scheduling tables and the equal cycle lengths: *DisjointSchedules n nC* and *IdenticCycleLength n nC*,

[3] HOL provides an indefinite description formalizes the word "some", as in "some member of" (see [5] for details).

- the predicate *FlexRay_Controller* is true for the corresponding streams on the node *i*,
- the slot *t mod cycleLength (nC i)* does't occur in the scheduling table of the *i*th node,

then the *i*th stream from the sheaf *nSend* has no messages at the time unit *t*.

```
lemma fr_Send:
"⋀n nReturn nC nStore nGet nSend recv t k i.
 ⟦ i < n;  0 < n;  k < n;
  FlexRay_Controller (nReturn i) recv (nC i)
    (nStore i) (nGet i) (nSend i);
  DisjointSchedules n nC;
  IdenticCycleLength n nC;
  ¬ (t mod cycleLength (nC i) mem schedule (nC i))⟧
  ⟹
   (nSend i) t = []"
  apply (simp add: FlexRay_Controller_def)
  apply clarify
  apply (simp add:  Scheduler_def)
  apply (erule_tac x="t" in allE)
  apply (simp add: Let_def)
  apply (simp add: BusInterface_def)
  apply clarify
  apply (simp add: Send_def)
  done
```

3.9 Lemma fr_nC_Send

The lemma *fr_nC_Send* says: for any time unit *t*, for any natural numbers *n* and *k*, for any stream *recv*, for all parameters *nC* and for all sheafs of channels *nReturn*, *nStore*, *nGet* and *nSend* the following holds: If holds that

- the number *n* is greater then zero and the number *k* is less then *n*,
- the parameters $c_1, \dots, c_n$ has disjoint scheduling tables and the equal cycle lengths: *DisjointSchedules n nC* and *IdenticCycleLength n nC*,
- the predicate *FlexRay_Controller* is true for the corresponding streams on every node *i* of the *n* nodes,
- the slot *t mod cycleLength (nC k)* occurs in the scheduling table of the *k*th node,

then every stream from the sheaf *nSend* except the *k*th stream has no messages at the time unit *t*.

```
lemma fr_nC_Send:
"⋀n nReturn nC nStore nGet nSend recv t k.
 ⟦ ∀i<n. FlexRay_Controller (nReturn i) recv
            (nC i) (nStore i) (nGet i) (nSend i);
  0 < n; k < n;
  DisjointSchedules n nC;
  IdenticCycleLength n nC;
  t mod cycleLength (nC k) mem schedule (nC k)⟧
  ⟹
   ∀j.  j < n ∧ j ≠ k ⟶ (nSend j) t = []"
  apply clarify
apply (subgoal_tac
  "¬ (t mod cycleLength (nC j) mem schedule (nC j))")
  prefer 2
  apply (erule correct_DisjointSchedules1)
  apply assumption+
  apply simp
apply (erule_tac x="j" in allE)
apply (simp add: fr_Send)
done
```

3.10 Lemma fr_refinement_maxmsg_nGet

The lemma *fr_refinement_maxmsg_nGet* says: for any natural numbers *n* and *i*, for any streams *recv* and *activation*, for all parameters *nC* and for all sheafs of channels *nReturn*, *nStore*, *nGet* and *nSend* the following holds: If the number *i* is less then *n* and the predicate *FlexRay_Controller* is true for the corresponding streams on every node *i* of the *n* nodes, then every stream from the sheaf *nGet* contains at every time unit at most one message.

```
lemma fr_refinement_maxmsg_nGet:
"⋀n nReturn nC nStore nGet nSend recv i activation.
  ⟦  i < n;
     ∀i<n. FlexRay_Controller (nReturn i) recv
           (nC i) (nStore i) (nGet i) (nSend i)⟧
 ⟹ maxmsg (Suc 0) (nGet i)"
  apply (simp add: FlexRay_Controller_def)
  apply (erule_tac x="i" in allE)
  apply clarify
  apply (simp add: BusInterface_def)
  apply (simp add: maxmsg_def)
  apply clarify
  apply (simp add: Send_def)
  apply (simp add: Scheduler_def)
  apply (erule_tac x="t" in allE)+
  apply (simp add: Let_def)
  apply (simp split add: split_if_asm)
  done
```

3.11 Lemma fr_refinement_maxmsg_nSend

The lemma *fr_refinement_maxmsg_nSend* says: for any natural numbers *n* and *i*, for any streams *recv* and *activation*, for all parameters *nC* and for all sheafs of channels *nReturn*, *nStore*, *nGet* and *nSend* the following holds: If every stream from the sheaf *nReturn* contains at every time unit at most one message and the predicate *BusInterface* is true for the corresponding streams, then every stream from the sheaf *nSend* contains at every time unit at most one message.

```
lemma fr_refinement_maxmsg_nSend:
"⋀n nReturn nC nStore nGet nSend recv i activation.
 ⟦ maxmsg (Suc 0) (nReturn i);
   BusInterface activation recv (nReturn i)
     (nGet i) (nSend i) (nStore i)⟧
⟹ maxmsg (Suc 0) (nSend i)"
  apply (simp add: maxmsg_def)
  apply (simp add: BusInterface_def)
  apply clarify
  apply (simp add: Send_def)
apply (erule_tac x="t" in allE)+
apply (simp split add: split_if_asm)
done
```

3.12 Lemma fr_refinement_maxmsg_nStore

The lemma *fr_refinement_maxmsg_nStore* says: for any natural numbers *n* and *i*, for any stream *recv* and for all parameters *nC* and for all sheafs of channels *nReturn*, *nStore*, *nGet* and *nSend* the following holds:

- the number *n* is greater then zero and the number *i* is less then *n*,
- the parameters $c_1, \dots, c_n$ has disjoint scheduling tables and the equal cycle lengths: *DisjointSchedules n nC* and *IdenticCycleLength n nC*,
- all the stream from the sheaf *nSend* are disjunct,
- the predicate *Cable* is true for the corresponding streams,
- the predicate *FlexRay_Controller* is true for the corresponding streams on every node *i* of the *n* nodes,
- every stream from the sheaf *nReturn* contains at every time unit at most one message,

then every stream from the sheaf *nStore* contains at every time unit at most one message.

```
lemma fr_refinement_maxmsg_nStore:
"⋀n nReturn nC nStore nGet nSend recv i.
 ⟦ DisjointSchedules n nC;  IdenticCycleLength n nC;
   disjunctMessage n nSend; i < n;  0 < n;
   ∀ i<n. maxmsg (Suc 0) (nReturn i);
   Cable n nSend recv;
   ∀i<n. FlexRay_Controller (nReturn i) recv
          (nC i) (nStore i) (nGet i) (nSend i)⟧
⟹ maxmsg (Suc 0) (nStore i)"
  apply (simp (no_asm) add: maxmsg_def)
  apply clarify
  apply (simp add: Cable_def)
  apply (simp add: CorrectSheaf_def Broadcast_def)
  apply (rotate_tac 5)
  apply (erule_tac x="t" in allE)
  apply (simp split add: split_if_asm)
  apply (subgoal_tac
    "(SOME i. i < n ∧  nSend i t ≠ []) = k")
    prefer 2
    apply (simp add: correct_disjunctMessage)
  apply simp
  apply (erule_tac
     V="(SOME i. i < n ∧ nSend i t ≠ []) = k"
     in thin_rl)
  apply (simp only: disjunctMessage_def)
  apply (rotate_tac 1)
  apply (erule_tac x="t" in allE)
  apply (rotate_tac -1)
  apply (erule_tac x="k" in allE)
  apply simp
  apply (rotate_tac 5)
apply (subgoal_tac
 "∀ i<n. FlexRay_Controller (nReturn i) recv
       (nC i) (nStore i) (nGet i) (nSend i)")
  prefer 2
  apply simp
apply (erule_tac x="i" in allE)
apply (rotate_tac -2)
apply (erule_tac x="k" in allE)
apply clarify
apply (simp add: FlexRay_Controller_def)
apply (rotate_tac 5)
apply (erule_tac x="k" in allE)
apply clarify
apply (subgoal_tac " maxmsg (Suc 0) (nSend k)")
  prefer 2
  apply (erule fr_refinement_maxmsg_nSend)
  apply assumption
apply (simp add: BusInterface_def)
apply clarify
apply (simp add: Receive_def)
apply (rotate_tac -2)
apply (erule_tac x="t" in allE)
apply (rotate_tac -3)
apply (erule_tac x="t" in allE)
apply (simp split add: split_if_asm)
apply (simp add: maxmsg_def)+
apply (erule_tac x="i" in allE)
apply (simp add: FlexRay_Controller_def)
apply clarify
apply (simp add: BusInterface_def)
apply clarify
apply (simp add: Receive_def)
apply (rotate_tac -2)
apply (erule_tac x="t" in allE)
apply (simp split add: split_if_asm)
done
```

4 Conclusions

This paper represents the formal verification of the FlexRay protocol specification [4]. It contains the translation of the FlexRay Focus specifications in in Isabelle/HOL. The case study has shown that the translation from a timed Focus specification to the specification in Isabelle/HOL can be done in the schematical way. The corresponding proof in Isabelle/HOL has shown that the Focus specification of the FlexRay architecture is a refinement of the FlexRay requirements specification and the proof that the architecture specification fulfill the requirements specification, i.e. is its behavioral refinement.

References

[1] Broy, M. Compositional refinement of interactive systems. *J. ACM 44*, 6 (1997), 850–891.

[2] Broy, M., and Stølen, K. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer, 2001.

[3] FlexRay Consortium. `http://www.flexray.com`.

[4] Kühnel, C., and Spichkova, M. FlexRay und FTCom: Formale Spezifikation in FOCUS. Tech. Rep. TUM-I0601, Technische Univerität München, 2006.

[5] Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.

[6] Wenzel, M. *The Isabelle/Isar Reference Manual.* TU München, 2004.