The YAPI protocol for buffered data transfer

P. Lambooij

Computing Science Institute/

CSI-R9923 December

The YAPI protocol for buffered data transfer

A case study in object oriented specification and verification

Peter Lambooij

Philips Tass
P.O. Box 218, 5600 MD Eindhoven, The Netherlands
peter.lambooij@philips.com

abstract

A case study is presented of tool-assistent verification of a protocol used in industry to model Kahn processing networks. The object oriented specification language CCSL is used to describe the point to point communication between nodes in the network. CCSL is translated into the mathematical theories of coalgebras via a special compiler. These theories form the basis of a rigorous proof of correctness of data transfer and absence of deadlock in the protocol. Proofs are developed using the PVS system.
Keywords and Phrases: specification and verification, reasoning about programs, formal languages, object orientation.
AMS Subject Classification (1991): 68Q45, 68Q60.
CR Subject Classification (1991): D2.1, D2.4, F3.1

1 Introduction

1.1 Specification and verification

In the process of constructing reliable software it is essential to state the requirements of a program as precisely and completely as possible. Specification is basically the description of these requirements. A good specification allows for both verifying that the constructed program fulfills its requirements and checking whether all parties involved in the development process agree on the properties of the program to be build. The best way of avoiding the ambiguities inherent to natural languages (since their semantics rely heavily on context) is the use of a formal language. Another advantage of specifications in a formal language is the limited syntax allowing for a tool-assisted proof of the consistency and completeness of a specification. The work described here involves such a tool-assisted proof.

1.2 Tools and languages

A convenient tool for verification of specifications (and implementations) is the so-called LOOP tool ([JvdBH$^+$98], [HHJT98]). LOOP stands for Logic of Object Oriented Programming. LOOP is basically a compiler that translates certain object oriented specification languages into sets of logical theories. These theories can then be imported into a 'proof-assistant' to reason about specifications. By translating several class specifications with the tool, the generated logical theories can be used for checking their relative consistency.

The specification language that was used in this study is CCSL [Tew98], for Coalgebraic Class Specification Language, which is based on the theory of coalgebras[JR97]. CCSL provides a natural means of formal object oriented specification. All mathematical reasoning about the specifications presented in this study was done using the reasoning tool PVS [COR$^+$95]. PVS forms a combination of a mathematical specification language and a theorem prover. Thus, the LOOP Tool forms the bridge between CCSL and PVS.

1.3 Case study (the YAPI protocol)

In this report, a case study is described of the verification of the so-called YAPI protocol used in industry [KE99]. The protocol is part of an application programmer's interface which can be used to model signal processing applications as process networks. The communication between the processes is based on the Kahn model with blocking reads on (theoretically) unbounded fifo buffers [Kah74]. The purpose of YAPI is to provide a means to construct models of signal processing applications with a platform independent programmer's interface. The application programmer's interface is used in industry for performance analysis, hardware-software co-design and the mapping of software onto different platforms.

In this study, a crucial part of YAPI, the point to point transfer of data through a buffer is specified at different levels of abstraction. The relation between the specifications will be used to prove the correctness of the protocol under certain circumstances.

1.4 Overview

The rest of this report will be organised as follows: In section 2, the languages and tools that are used for this case study will be presented. First the proof-assistant and specification language of the Prototype Verification System will be discussed briefly. Then the Coalgebraic Class Specification Language will be presented that is used for the object oriented specifications. Finally the LOOP tool is discussed. In section 3 coalgebra, the theory behind proving refinement or implementation relations between specifications, is discussed. The proof techniques used rely on coalgebraic concepts like bisimilarity, refinements and invariants. These concepts are briefly discussed. Section 4 contains an informal decription of the YAPI protocol, including the background of the protocol. In section 5 a formal CCSL specification of YAPI is presented. Section 6 contains outlines of the proof of the correctness of the protocol and its implementation. Finally, the conclusions are presented in section 7.

2 Tools for Specification and Verification

2.1 PVS

All mathematical reasoning about the system at hand was done using the Prototype Verification System (PVS). PVS was developed at the Stanford Research Institute. Its purpose is to enable rapid development of mathematical models that can subsequently be formally verified. It consists of a specification language combined with a theorem prover. Both are integrated in an Emacs environment that allows for easy manipulation of files.

The specification language is based on typed higher-order logic. Models are written as (parameterised) theories that can contain axioms, definitions and lemmas/theorems. Many libraries with previously developed theories are available, including an extensive prelude file.

The theorem prover allows for both interactive and scripted proofs through a number of basic proof-commands. The system has many built-in inference rules that assist the user in finding a complete proof using a relative small number of steps. As an example of the capabilities of PVS a small theory and proof are presented in appendix 1 and 2. A complete description of PVS can be found in literature [OSR93a, OSR93b, SOR93, COR$^+$95].

2.2 CCSL

The specifications presented in this work were all written in CCSL. CCSL (Coalgebraic Class Specification Language) is an object oriented specification language ([Tew98]). It is based on the theory of coalgebras ([JR97] and [Jac99]). A coalgebraic specification of a class is characterised by an (hidden) state space and two types of operations. One kind reveals the visible aspects (attributes) of the objects in the class and the other kind changes the internal state (methods). Since CCSL is a specification language the behaviour of the object is not given in terms of algorithms but rather by assertions that specify the interdependency of the attributes and/or methods. Initial states are specified using constructors. The theory of coalgebras will be described in some more detail in section 3.

As a first example of the use of CCSL consider the system *PFC_abs* with two finite sequences (of type T) called respectively *source* and *dest*. It has an unspecified initial state *new*. The only operation that we define on *PFC_abs* is a transfer of a specified finite number items (of type T) from the front of *source* to the back of *dest*. A CCSL specification of such a system is shown in figure 2.1.

At the start of the file *nat* and *fin_seq* are declared as *BASETYPE*. This indicates that they are a pre-defined type (*nat*) or a user-defined type (*fin_seq*) that is known to PVS (see appendix 3). In the *CLASSSPEC* called *PFC_abs*, the attributes and methods are declared with keywords *ATTRIBUTE* and *METHOD* respectively. Only one assertion is declared to specify the transfer of items from *source* to *dest*. The type *Self* denotes the internal state space of *PFC_abs*. Assertions are given as predicates over *Self*, involving attributes and methods. They are written in the specification language of PVS and put between keywords *PVS* and *ENDPVS*. Hopefully, the meaning of *length*, *front* and *back* is clear in the context of this example. These terms are formally defined in appendix 3.

Contructor *new* is declared using keyword *CONSTRUCTOR*. In general, an assertion about the properties of new would be declared using keyword *CREATION*. However, since no assumptions were made about *new*, *CREATION* was not used. The style of “model-based” specification, with the use of abstraction functions (*Self* → *fin_seq*) will be maintained throughout the rest of this report.

```
TYPE nat: BASETYPE
TYPE fin_seq: BASETYPE

BEGIN PFC_abs[A: TYPE]: CLASSSPEC

IMPORTING "MyFinSeq[A]"

ATTRIBUTE
  source: Self -> fin_seq;
  dest:   Self -> fin_seq;

METHOD
  transfer: [Self, nat] -> Self;

ASSERTION
  transfer_ok:
  PVS
    FORALL (n: posnat):
      length(source(x)) >= n
      IMPLIES
      dest(transfer(x,n)) = append(dest(x), front(source(x),n))
      AND
      source(transfer(x,n)) = back(source(x),length(source(x))-n)
  ENDPVS

CONSTRUCTOR
  new: Self;

END PFC_abs
```

Figure 2.1: CCSL specification of class PFC_abs

2.3 LOOP

LOOP stands for Logic of Object Oriented Programming. The LOOP tool was developed as part of a joint project at the universities of Nijmegen and Dresden. Its aim is to facilitate computer-assisted reasoning about classes ([JvdBH+98], [HHJT98]) by acting as a front-end tool for theorem provers like PVS. It translates specifications of e.g. CCSL to theories in the target language as depicted in figure 2.2. It includes in the theories coalgebraic concepts like bisimilarity and invariants that are useful to prove refinement relations. The LOOP tool supports object oriented concepts like components, inheritance, late binding and overloading.

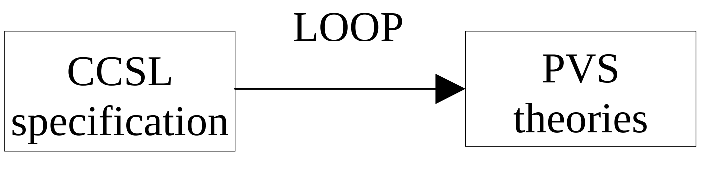

Figure 2.2: Schematic of the relation between CCSL, LOOP and PVS.

```
PFC_absModel[A : TYPE] : THEORY
BEGIN

  IMPORTING MyFinSeq[A]

  PFC_absState : TYPE = [# msource:  fin_seq, % labelled product of
                           mdest: fin_seq #]  % two finite sequences

  IMPORTING PFC_abs[PFC_absState , A]

  PFC_abs_c : [PFC_absState -> PFC_absIFace[PFC_absState , A]]=
    (LAMBDA (x: PFC_absState) :
      (# source   := msource(x), % source(x) = fin_seq
         dest     := mdest(x),   % dest(x) = fin_seq
         transfer :=
          (LAMBDA (p1: nat) :
            (# msource := back(msource(x),length(msource(x)) - p1),
               mdest   := append(mdest(x),front(msource(x),p1))
             #))
      #))
    ;

  assert_ok : LEMMA % :-)
    PFC_absAssert?(PFC_abs_c)

  PFC_abs_constr : PFC_absConstructors[PFC_absState , A] =
    (# msource := empty_seq,
       mdest   := empty_seq #);

  create_ok : LEMMA % :-)
    PFC_absCreate?(PFC_abs_c)(PFC_abs_constr)

END PFC_absModel
```

Figure 2.3: Model of CCSL class PFC_abs

In the case of the CCSL-specification of *PFC_abs* LOOP produces two files PFC_abs_basic.pvs and PFC_abs_user.pvs. The ‘basic’ file contains the theories that describe the behaviour of the system. One of its theories provides an interface to a coalgebra generated by the LOOP compiler. All assertions made in the CCSL specification are translated into predicates on a coalgebra with this interface. Other theories define the important notions of coalgebraic invariant and bisimilarity for the coalgebra at hand.

The ‘user’ file contains a theory with a framework for a model of the class specification. A model is an implementation of the specification with a concrete type as state space and concrete operations as attributes and methods. In figure 2.3 a model of *PFC_abs* is shown. Concrete type and operations are given for *Self*, *source*, *dest* and *transfer*. Lemmas *create_ok* and *assert_ok* state that the coalgebra defined by the model type and operations complies with the assertions written in the specification of *PFC_abs*.

3 Theory of Verification with CCSL / PVS

3.1 Coalgebras

Coalgebras are mathematical structures, which consist of a hidden state space (typically written as *Self*), together with a set of operations. These operations may either give information about *Self*, or they may be used to modify *Self*. The first kind of operations – sometimes called attribute – have a type of the form: [$Self \to A$] where A is a set. This operation gives information about an element of the state space *Self*. The second kind of operations – sometimes called method – may be of the form [$Self \times B \to Self$], changing the state, depending on a parameter from set B. The effect of a method can not be directly observed from the state, but only by the values of the attributes. Literature on (co)algebras and (co)induction can be found in [JR97] and [Jac99].

Coalgebras are very suitable for writing specifications for object oriented languages, since objects in object oriented languages also have a private state space, which should be accessed and modified through the public attributes and methods of the class. A class with its attributes and methods is comparable to a coalgebra. The most visible difference is the explicit reference in coalgebra to the state space, which is left implicit in common specifications of classes. An object of this class can be seen as an element of the state space of the coalgebra. In case of the example of *PFC_abs*, such a coalgebra contains a function of type [$Self \to fin\_seq$], which represents the attributes *source* and *dest*. The method *transfer* is represented by a function of the type [$Self \times nat \to Self$] changing the state of *PFC_abs*).

In the translated CCSL *PFC_abs* specification, such a coalgebra has the following form: *c:* [$Self \to PFC\_absIface$]. The type *PFC_absIface* is the interface of the coalgebra, giving access to its operations. In the translation, these operations are put into a labeled PVS record:

```
PFC_ABSIface: TYPE =
 [#   source: fin_seq,
      dest: fin_seq,
      transfer: [[nat] → Self]
 #]
```

Note that only the codomains of the operations are given. The types of the individual operations of *PFC_abs* are as follows:

source(c): [$Self \to fin\_seq$]
dest(c): [$Self \to fin\_seq$]
transfer(c): [$[Self, nat] \to Self$]

where *c:* [$Self \to PFC\_absIface$] is the coalgebra.

3.2 Coalgebraic invariants

In the context of coalgebras, an invariant is a unary predicate on the state space, which is closed under all operations of the coalgebra. If this predicate holds in a certain state, it will also hold in the states which are reached by executing any one of the operations of the coalgebra. If such a predicate holds in the initial states of the coalgebra, it will hold per definition for every reachable state.

Running the LOOP tool on a class generates a notion of invariance for the particular class (stating that the predicate is closed under every operation of the class). A predicate *invariant?* is generated, which takes a coalgebra *c* and a predicate *P* on the state space *Self* as arguments. The predicate *invariant?* holds if the predicate *P* is an invariant of the coalgebra. For the *PFC_abs* example in CCSL, the usual notion of invariance is expressed as follows:

invariant?(*c*) *:* [[*Self* $\rightarrow$ *bool*] $\rightarrow$ *bool*] =
  (*LAMBDA* (*P:* [*Self* $\rightarrow$ *bool*]) *:*
   FORALL (*x: Self*) *:* (*P*(*x*)) *IMPLIES* (*PFC_absPred*(*P*)(*c*(*x*))))

with *PFC_absPred* defined as:

PFC_absPred : [[*Self* $\rightarrow$ *bool*] $\rightarrow$ [*PFC_absIFace*[*Self* , *A*] $\rightarrow$ *bool*]] =
  (*LAMBDA* (*P:* [*Self* $\rightarrow$ *bool*]) *:*
   (*LAMBDA* (*rec: PFC_absIFace*[*Self* , *A*]) *:*
    ((*FORALL* (*p1: nat*) *: P*(*transfer*(*rec*)(*p1*))))))

Thus, in our example of *PFC_abs, invariant?*(*c*)(*P*) expresses that *P* is closed under operation *transfer*.

3.3 Coalgebraic refinements

Refinements can be used to prove that a ‘concrete’ class specification implements or refines an ‘abstract’ one. This is done by constructing a translation model of the abstract class, expressing the operations of the abstract class in terms of the operations of the concrete one.

The operations in the abstract model should also exist in the concrete model. This should be achieved by a translation of the concrete coalgebra into the abstract one. After making this translation, it is to be proved that the translated coalgebra satisfies the assertions of the abstract class specification.

In case of a non-trivial translation, the use of some predicate P on the state space of the concrete class is required. This predicate should satisfy the following three properties [Jac97]:

1: P is closed under the ‘abstract’ operations. This expresses that P is an invariant of the abstract class.

2: P holds for all initial states of the concrete class.

3: If a ‘concrete’ state statisfies P, the translation of that state satisfies the assertions of the abstract specification.

In other words, instead of proving the correctness for every state, we only have to prove the correctness for states where predicate P holds. This predicate may simplify the proof or even make it possible in the first place. In section 6 an example can be found of a coalgebraic invariant (called *start_pred*) that is used to prove a refinement relation between two coalgebras.

4 The Y-chart Application Programmer's Interface.

4.1 Parallel processing architectures

Typical signal processing systems are parallel architectures because they have to process enormous amounts of data at high rates, especially in the video domain. These throughput requirements significantly constrain the design space of signal processing systems. In order to explore the design space more efficiently the so-called YAPI model was developed at Philips Research [KE99].

YAPI is an acronym for Y-chart Application Programmer's Interface. The term Y-chart originates from the (independent) activities of application modelling and architecture modelling that are combined to create an overall performance model of a system. This process is depicted in the shape of a Y in figure 4.1.

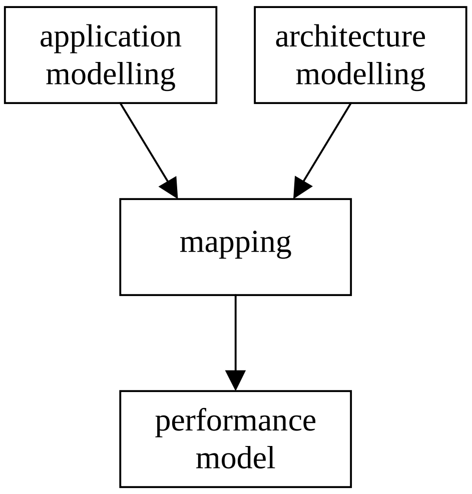

Figure 4.1: The so-called Y-chart

In YAPI the potential parallelism in the functional specification is explicitly described by modelling applications as networks of parallel processes. An example of a process network is shown in figure 4.2. The communication between the processes is based on the Kahn model with blocking reads on (theoretically) unbounded fifo buffers [Kah74]. Items are received in the same order as they were sent, hence the name fifo (first-in-first-out). In practice the fifo buffers are bounded such that writes can also suspend the execution of the writing process. Note that a buffer of size zero would correspond to the model of communicating sequential processes introduced by C.A.R. Hoare [Hoa78].

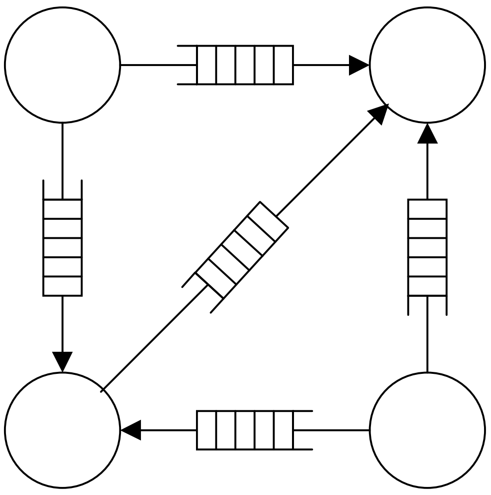
Figure 4.2: Schematic of a process network with four processes communicating through unbounded fifo buffers

4.2 Informal description of the YAPI protocol

In this study we have confined ourselves to one important aspect of YAPI: the point to point communication between two processes. Figure 4.3 shows a schematic of two processes called producer (P) and consumer (C) that are connected via a unidirectional buffered channel called the fifo (F). Data is transferred in elements (or items) of a certain arbitrary type. Data is received by the consumer in the same order as it is sent by the producer.

Both processes perform alternatingly internal computing steps and a communication command. The communication command of the producer is the *write(n)* command with *n* a natural number. The effect of the *write*(*n*) command is transfer of *n* items from producer to fifo on completion of the command. The fifo has a capacity of *N* items. The write command blocks if the fifo contains *N* items. Similar to the producer the consumer has the *read*(*n*) command. The effect of the *read*(*n*) command is transfer of n items from fifo to consumer. The read command blocks if the fifo is empty. The blocking of reading and writing avoids potential data loss during communication. It is the only form of synchronisation between producer and consumer.

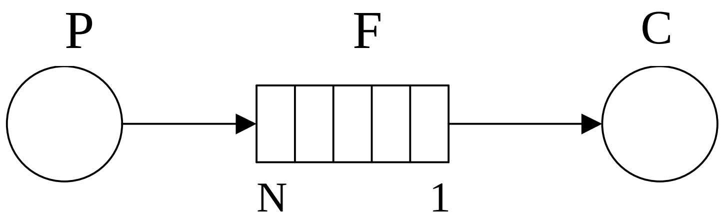

Figure 4.3: Schematic of a point to point communication between two processes

5 Specifications of the YAPI Protocol in CCSL

5.1 CCSL specification of YAPI

This section contains the decisions taken in writing down a specification for YAPI together with a description of the classes, attributes and methods that were used. The complete CCSL specification can be found in appendix 4.

Since producer, fifo and consumer are all parts of the process network it is natural to model the system as a class *ProcessNetwork* with a producer *prod*, a fifo *fifo* and a consumer *cons* as component classes. All assertions concerning more than one component are on the level of *ProcessNetwork.*

The transfer of data from producer to fifo and from fifo to consumer has to be modelled. We decided to do this by using an abstraction function *Self* $\to$ *fin_seq*. In this way specifying what happens to the data yields relatively simple assertions (with the help of a small number of auxilliary definitions, see PVS theory *MyFinSeq* in appendix 3). As an example, if one item is transferred from the head of a *fin_seq*, say f_1, to the end of another *fin_seq*, say f_2, then f_1 becomes *tail*(f_1) and f_2 becomes *add*(f_2,*head*(f_1)). Therefore, *prod, fifo* and *cons* all have a finite sequence *buff* as attribute.

The YAPI protocol does not specify the way in which producer and consumer share the fifo. We made the following decisions. Firstly, the communication between producer-fifo and fifo-consumer does never occur simultaneously (mutual exclusion). Secondly, once the producer (or consumer) has access to the fifo it will write (or read) items until either the required number is transferred, or until the fifo is full (or empty) after which the fifo is ‘released’. We will call such ‘bursts’ of transfer of items between fifo and producer or consumer write-cycles and read-cycles, respectively. Thirdly, when the fifo is idle the producer and consumer may try to access it simultaneously. In this case there are several possibilities to specify to whom access is granted like (i) always the consumer, (ii) always the producer or (iii) random choice between producer and consumer. For our purposes the differences between the possibilities are not important and we decided for (i), always granting access to the consumer.

Finally, *write*(*n*) and *read*(*n*) cannot be atomic steps if *n* is larger than the capacity of the fifo. As a consequence, we have to allow for some form of interleaving between *write*(*n*) and *read*(*n*). In addition, the producer and the consumer may issue communication commands with different numbers of items, say *write*(5) and *read*(7). If *write*(5) and *read*(7) are carried out as atomic steps then the state in which the 5 items written to the fifo are at the back of the *buff* of the consumer does not occur, since read(7) would transfer 7 items in one step from fifo to consumer. Therefore, interleaving is created by choosing as atomic step the transfer of one item from producer to fifo (or from fifo to consumer).

5.2 Component classes and attributes of ProcessNetwork

Figure 5.1 shows a diagram with all the classes used in the specification of *PFC_imp* of the YAPI protocol. Class *Sequence* was used as a basic wrapper around finite sequence *buff*. Auxilliary attributes *size* and *empty* in *Sequence* depend on *buff* as specified by assertions *size_sequence* and *empty_sequence*. They were introduced to obtain more readable expressions in assertions on the level of *ProcessNetwork.*

The description of *fifo* needs one other qualifier to indicate that it is *full*. *fifo* is an instantiation of class *BoundedSequence*, which inherits from class *Sequence* the attributes *buff*, *size* and *empty* and adds attribute *full*.

The producer and consumer can perform communication commands. In order to model the state of "in the process of carrying out a read or write command" a natural *nr_to_transmit* was used with *n* the number of items that is left for *prod* (or *cons*) to *write* (or *read*). Both *prod* and *cons* are members of class *TransmitSequence*. The difference in producer and consumer is only apparent in the assertions at the network level. *TransmitSequence* has a redundant method *set_TS(fin_seq, nat)* to obtain shorter specifications in class *ProcessNetwork*.

Abstract datatype *StatusValue* is introduced as a means to achieve mutual exclusion between producer and consumer. StatusValue has three has three elements *idle*, *writing* and *reading*. The LOOP tool automatically generates a corresponding PVS data type definition, containing precisely these three elements as constructors.

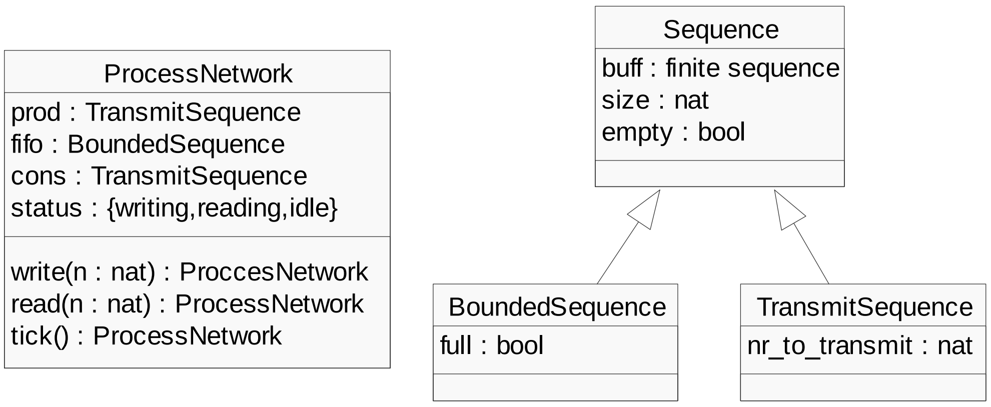

Figure 5.1 Classes of the CCSL specification of *PFC_imp*. Each class is represented by a rectangle with its name (top), attributes (middle) and methods (bottom). Methods to set attributes are omitted for clarity. The arrows denote inheritance. They point to the class from which the properties are inherited.

5.3 Methods and assertions of ProcessNetwork

Class ProcessNetwork has three methods *write*, *read* and *tick*. The methods *write* and *read* are used to model a communication command. In the model of YAPI processes perform alternating communication steps and computation steps. Therefore, the *read* method only has effect on *cons* when no "previous" *read* command is being carried out. In terms of the attributes this means if *nr_to_transmit* of *cons* equals 0 then the *nr_to_transmit* of *cons* after a *read(n)* command equals *n*. The *write(n)* command is similar to *read(n)* command. There is an additional constraint however: the size of *buff* should be sufficient for transfer of *n* items. The methods w*rite* and *read* are formally described in assertions *set_nr_to_write* and *set_nr_to_read* in the CCSL file in appendix 4.

The evolution of the system in time is modelled using method *tick*. Three assertions specify the effect of *tick*: *no_write_no_read*, *no_write_go_read* and *go_write_no_read*. They are all three written in the form: *pre-condition*(*x*) *IMPLIES post-condition*(*tick*(*x*)). The pre-conditions are all mutually exclusive and the disjunction of all three pre-conditions equals TRUE. Therefore, when method *tick* is aplied exactly one of the three assertions is always applicable. Furthermore, the post-condition of all three lemmas specify all attributes of *ProcessNetwork*. Hence, *no_write_no_read*, *no_write_go_read* and *go_write_no_read* fully specify the ‘temporal’ behavior of *ProcessNetwork*. As a final remark, transfer of items is done one by one as can be seen from the post-condition of *no_write_go_read* and *go_write_no_read*.

```
% ProcessNetwork (of PFC_imp.beh) is a refinement of PFC_abs
%
imp_ref_abs[Self, A: TYPE, N: posnat]: THEORY
BEGIN
  IMPORTING PFC_abs,
            ProcessNetwork[Self,A,N],
            ProcessNetwork_User[Self,A,N],
            MyDivMod

  c: VAR (ProcessNetworkAssert?)
  x: VAR Self

  refinement(c) : [Self -> PFC_absIFace[Self,A]] =
    (LAMBDA (x: Self):
      (# source := buff(prod(c)(x)), % source(x) : fin_seq
         dest   := buff(cons(c)(x)), % dest(x) : fin_seq
         transfer :=                 % transfer(x) : [nat -> Self]
           (LAMBDA (k: nat) :
             IF size(prod(c)(x)) >= k
             THEN LET T = div(k,N)*(2*N+2) + 2*mod(k,N)+2 IN
               iterate(tick(c),T)(read(c)(write(c)(x,k),k))
             ELSE x
           ENDIF
           )
       #)
     );

  start_pred(c): PRED[Self] =
    (LAMBDA (x: Self):
      nr_to_transmit(prod(c)(x)) = 0 AND
      size(fifo(c)(x)) = 0 AND
      nr_to_transmit(cons(c)(x)) = 0 AND
      status(c)(x) = idle
     )

  start_invariant: LEMMA % :-)
    invariant?(refinement(c))(start_pred(c))

  start_new: LEMMA % :-)
    FORALL (z: (ProcessNetworkCreate?(c))):
      start_pred(c)(new[Self,A](z))

  imp_refines_abs: LEMMA % :-)
    PFC_absAssert?[(start_pred(c)),A](refinement(c))

END imp_ref_abs
```

Figure 6.1 PVS theory imp_ref_abs

6 Verification of the YAPI protocol as a refinement

6.1 PFC_imp as a refinement of PFC_abs

The main question of the YAPI protocol as specified in *PFC_imp* is: will items eventually be transferred from *prod* to *cons*? In the most general case, with arbitrary numbers to read and write and the fifo buffer filled with an arbitrary number of items, the formal expression of “items eventually be transferred” becomes quite complicated. We decided to confine ourselves to a simpler situation: If *nr_to_transmit* of *prod* and *cons* equals 0, *fifo* is *empty* and *status* is *idle* will a subsequent *write*(*n*) and *read*(*n*) after a certain number of *ticks* lead to *n* items transferred from *buff* of *prod* to the back of the *buff* of *cons*?

An answer to this question is to view *PFC_imp* as a refinement of *PFC_abs*. The theory stating that *PFC_imp* is a refinement of *PFC_abs* is shown in figure 6.1. The theory follows the description of coalgebraic refinements given in section 3. The restriction to invariant *start_pred* is achieved through instantiation of *invariant?* with *start_pred*(*c*).

Function *refinement* is a mapping of coalgebra *c* of *ProcessNetwork* to the interface of a coalgebra of the abstract class. It is a record containing all attributes and methods of the abstract class expressed in terms of attributes and methods of the concrete class.

As invariant *P* predicate *start_pred* was defined. It states the restrictions on *prod*, *cons*, *fifo* and *status* given above. Lemma *start_invariant* expresses property 1 of coalgebraic refinements: *start_pred* is an invariant under the operations of *PFC_abs*. Lemma *start_new* states property 2: All of the initial states of the concrete class (*ProcessNetworkCreate?*(*c*)) satisfy *start_pred*. Finally, lemma *imp_refines_abs* reflects property 3: *refinement*(*c*) satisfies all assertions of *PFC_abs*.

6.2 Proof of refinement

The lemmas that were needed to proof the refinement relation between class *PFC_abs* and *ProcessNetwork* are given in appendix 5. All lemmas shown here were checked (interactively) using the PVS proof assistant. We will confine ourselves here to two crucial lemmas: *read_cycle* (which is representative for most of the others) and *transfer_all* (which is sufficient to prove property 3 of the refinement relation).

Lemma *read_cycle* is shown in figure 6.2. Function *eq_TS* (*eq_BS*) is short-hand for equality of all attributes of two objects of class *TransmitSequence* (*BoundSequence*). Informally, *read_cycle* states that once the network transports items from *fifo* to *cons*, it will continue to do so until either *fifo* is empty or the required nr of items is ‘read’. The main proof-step is induction on t. An initial case-distinction for t = 0 facilitates the induction proof. For proof of the induction step, lemma *no_write_go_read* was used.

```
read_cycle: LEMMA % :-)
  FORALL (x: Self):
   FORALL (t: nat):
     status(c)(x) = idle AND
     t <= size(fifo(c)(x)) AND
     t <= nr_to_transmit(cons(c)(x))
       IMPLIES
     eq_TS(prod(c)(ticks(c)(t)(x)), prod(c)(x)) AND
     eq_BS(fifo(c)(ticks(c)(t)(x)),
           set_BS(fifo(c)(x),back(buff(fifo(c)(x)),
                  size(fifo(c)(x))-t))) AND
     eq_TS(cons(c)(ticks(c)(t)(x)),
           set_TS(cons(c)(x),
                  append(buff(cons(c)(x)),front(buff(fifo(c)(x)),t)),
                  nr_to_transmit(cons(c)(x)) - t)) AND
     IF t = 0
     THEN status(c)(ticks(c)(t)(x)) = idle
     ELSE status(c)(ticks(c)(t)(x)) = reading
     ENDIF
```

Figure 6.2 Lemma *read_cycle*.

Lemma *transfer_all* is shown in figure 6.3. It states that after a *read*(*n*) and *write*(*n*) and a certain number of ticks, *n* items will have been transferred from *buff*(*prod*) to *buff*(*cons*). Combining previous lemmas of read and write cycles, *transfer_all* can be proven with relatively few steps. Method *transfer* is expressed in theory *imp_ref_abs* in concrete methods as iteration of a number of *tick*'s after a *read*(*n*) and *write*(*n*). Since the pre-condition for *transfer_all*: *nr_to_transmit* = *n* for both *prod* and *cons*, is fulfilled, *transfer_all* is sufficient to prove *transfer_ok* of *PFC_abs*.

```
transfer_all: LEMMA % :-)
  FORALL (x: Self):
    FORALL (k: nat):
      size(prod(c)(x)) >= k AND
      nr_to_transmit(prod(c)(x)) = k AND
      empty(fifo(c)(x)) AND
      nr_to_transmit(cons(c)(x)) = k AND
      status(c)(x) = idle
        IMPLIES
      LET T = ticks(c)(div(k,N)*(2*N+2) + 2*mod(k,N)+2)(x) IN
      eq_TS(prod(c)(T),
            set_TS(prod(c)(x),
                   back(buff(prod(c)(x)),
                   size(prod(c)(x)) - k),
                   0)) AND
      empty(fifo(c)(T)) AND
      eq_TS(cons(c)(T),
            set_TS(cons(c)(x),
                   append(buff(cons(c)(x)),front(buff(prod(c)(x)), k)),
                   0)) AND
      status(c)(T) = idle
```

Figure 6.3 Lemma *transfer_all*.

7 Conclusions

The aim of this study was to test the usefulness of CCSL specification in verification of the YAPI protocol, used in industry. An important property of the protocol, absence of deadlock and loss of data was mathematically proven.

In practice the study came down to writing specifications of classes and constructing refinements or implementations with a correctness proof. Since the concept of coalgebras plays an key-role in this approach, this concept was introduced. A coalgebra is a mathematical structure which consists of a hidden state space and a set of operations. Coalgebras are similar to classes in an object oriented environment. The set of operations of the coalgebra can be used to model the attributes and methods of a class.

The specification language CCSL was introduced and the LOOP tool was used to translate classes in CCSL into formal theories in the higher order logic of the reasoning environment: PVS. CCSL is a suitable language for specification. The bottleneck in the specification and proof of refinement relationships is the process of interactive proving. The proofs presented here have cost at least a month of full time interactive proving. In view of the labour-intensive nature of verification by mathematical proofs the size of problems that can be tackled is still limited.

In conclusion, the combination of CCSL / LOOP / PVS can form a useful set of tools to verify critical aspects of parts of software designs.

Acknowledgements

The author would like to thank Bart Jacobs for his overall support in this study, in particular for his help with the CCSL specifications and theory of coalgebras, Marieke Huisman for her valuable assistance in working with PVS, Erik Poll for a careful reading of the manuscript with useful feedback and Erwin de Cock for helpful discussions concerning the YAPI protocol.

References

[COR$^+$95] J. Crow, S. Owre, J. Rushby, N. Shankar and M. Srivas, *A tutorial introduction to PVS*, Computer Science Laboratory, April 1995

[HHTJ] U. Hensel, M. Huisman, B. Jacobs and H. Tews, *Reasoning about classes in object oriented languages: Logical models and tools*, in Ch. Hankin, editor, European Symposium on Programming, nr. 1381 in Lecture Notes in Computer Science, pages 105 – 121, Springer Verlag, Berlin 1998.

[Hoa78] C.A.R. Hoare, *Communicating Sequential Processes*, Communications of the ACM 21, 666 – 677, 1978

[Jac97] *Invariants, Bisimulations and the Correctness of Coalgebraic Refinements*, Lecture Notes in Computer Science, Vol. 1349, Springer, 1997, p. 276-291

[Jac99] B. Jacobs, *Exercises in Coalgebraic Specification*, proceedings of the Mathematics for Information Technology spring school, Oxford, April 2000.

[JR97] B. Jacobs and J. Rutten, *A tutorial on (co)algebras and (co)induction*, EATCS Bulletin 62, 1997.

[KE99] E.A. de Kock and G. Essink, *Y-chart Application Programmer's Interface*, Philips Naturkundig Laboratorium, March 1999, Technical Note 008/99

[Kah74] G. Kahn, *The Semantics of a Language for Parallel Programming*, Proc. of the IFIP Congress 74, North Holland Publishing Co., 1974

[Mey99] D. Meyer, *A case study in object oriented specification and verification: The MSMIE protocol*, master's thesis no. 452, Department of Computer Science, University of Nijmegen

[OSR93a] S. Owre, N. Shankar and J. Rushby, *The PVS Specification Language*, Computer Science Laboratory, February 1993.

[OSR93b] S. Owre, N. Shankar and J. Rushby, *User guide for the PVS Specification and Verification System*, Computer Science Laboratory, February 1993.

[SOR93] N. Shankar, S. Owre and J. Rushby, *The PVS Proof Checker: A Reference Manual*, Computer Science Laboratory, February 1993

[Tew98] H. Tews, *Coalgebraic Class specification Language* (*Reference manual*), Inst. Theor. Informatik, TU Dresden, D – 01062 Dresden, July 1998

Appendix 1. A sample theory in PVS

In this appendix a very brief, informal introduction is offered to provide a basic knowledge of the PVS specification language. A more complete description can be found in literature ([COR$^+$95] or [OSR93a]). In the explanation we make use of the example shown in figure A1.1. *MyFinSeq* is defined as a THEORY, i.e. a collection of definitions and lemmas/theorems. The theory has a type parameter T which means that all definitions and lemmas will be valid for arbitrary types of finite sequences, provided that the type is non-empty. PVS has several built-in types like *int*, *real*, *nat*, and *bool*. We assume the reader to be familiar with these types.

First, type *fin_seq* is defined as a "record" containing a natural *length* and a function *seq* from *below*[*length*] $\rightarrow$ *T* where *below*[*n*] is defined as [0 .. *n*). The sequences that are defined by *fin_seq* are finite but unbounded, since *length* can be any natural number.

Next, function *front* with signature: *fin_seq* $\times$ *nat* $\rightarrow$ *fin_seq* is defined by a case distinction *IF* .. *THEN* .. *ELSE* .. *ENDIF*. Informally, *front*(*fs*,*n*) yields a sequence with the first *n* elements of *fs* if *n* < *length*(*fs*) or else *fs*. This is expressed by a notation of substitution: (# length := .., seq := .. #), with the use of a LAMBDA function. Variables like *fs* and *n* are specification variables. They have only meaning in the context of the definition in which they are used.

Finally, a simple *LEMMA* with name *length_front* is specified. Lemmas are written as predicates (in higher order logic) over the definitions in the theory and built-in types of PVS. Function *min* is not a built-in function, but one of many functions and types that are defined in a prelude file that is loaded whenever the PVS tool is started. In appendix 2 a demonstration of the PVS proof of *length_front* is given.

```
MyFinSeq [T: TYPE+]: THEORY
 BEGIN
  fin_seq: TYPE = [# length: nat,
                     seq: [below[length] -> T]
                   #]

  front(fs: fin_seq, n: nat): fin_seq =
    IF n < length(fs)
    THEN (# length := n,
            seq := (LAMBDA (x: below[n]): seq(fs)(x))
         #)
    ELSE fs
    ENDIF

 length_front: LEMMA
    FORALL (fs: fin_seq, n: nat):
      length(front(fs,n))
      =
      min(length(fs),n)

END MyFinSeq
```

Figure A1.1 (A part of) PVS theory MyFinSeq

Appendix 2. A sample proof in PVS

This appendix contains a brief introduction to the use of the proof-assistant of PVS. A complete description can be found in literature ([COR$^+$95] or [OSR93a]). As an example the interactive proof of lemma *length_front* that was defined in appendix 1 is shown in figure A2.1. The dialog between machine and user consists of proof-obligations that are shown by PVS (*length_front:*) and commands that are typed in by the user at the prompt (*Rule?*). The format of the proof-obligations are so-called sequents. $A \wedge B \Rightarrow C \vee D$ is shown as:

```
(-1)   A
(-2)   B
|----------------
(1)    C
(2)    D
```

and tautology E is shown as:

```
|----------------
(1)    E
```

where A, B, C, D and E are predicates or formulas and the numbers are used to refer to a predicate or formula when issuing a proof-command, (in case of ambiguities).

The first command is (*skolem!*) which is used to remove the *FORALL* quantor. The bound variables *fs* and *n* are replaced by "free" variables *fs!1* and *n!1*. PVS keeps track of information like "*fs!1* is an arbitrary *fin_seq* and n*!1*is an arbitrary natural". Commands (*expand "front"*) and (*expand "min"*) request PVS to replace functions *front* and *min* by their definition from appendix 1 and the PVS prelude, respectively.

Most users would immediately recognise that the equality in the resultant sequent must hold, by mentally swapping *n!1* and *length*(*fs!1*). PVS cannot automatically match such complicated equalities. Therefore, to break down the *IF .. THEN* expression, the command (*case "n!1 < length*(*fs!1*)") introduces a case-distinction with two obligations. The first obligation is checked by the proof-command (*assert*). PVS automatically recognises that the equality can be simplified to *n!1 = n!1* and that this is true per definition. In the second proof obligation, the assumption that *n!1 < length*(*fs!1*) does not hold, is shown as a consequent, in the form of *n!1 < length*(*fs!1*), rather than as a presequent with a negation. Again, command (*assert*) is sufficient to let PVS recognise that the equality simplifies to *length*(*fs!1*) = *length*(*fs!1*) and PVS proof-assistant considers lemma *length_front* as proven.

```
length_front :

  |-------
{1}    FORALL (fs: fin_seq, n: nat):
              length(front(fs, n)) = min(length(fs), n)

Rule? (skolem!)

Skolemizing, this simplifies to:
length_front :

  |-------
{1}    length(front(fs!1, n!1)) = min(length(fs!1), n!1)
```

```
Rule? (expand "front" )

Expanding the definition of front, this simplifies to:
length_front :

  |-------
{1}    (IF n!1 < length(fs!1) THEN n!1 ELSE length(fs!1) ENDIF =
        min(length(fs!1), n!1))

Rule? (expand "min" )

Expanding the definition of min, this simplifies to:
length_front :

  |-------
{1}    (IF n!1 < length(fs!1) THEN n!1 ELSE length(fs!1) ENDIF =
        IF length(fs!1) > n!1 THEN n!1 ELSE length(fs!1) ENDIF)

Rule? (case "n!1 < length(fs!1)")

Case splitting on n!1 < length(fs!1), this yields  2 subgoals:
length_front.1 :

{-1}    n!1 < length(fs!1)
  |-------
[1]    (IF n!1 < length(fs!1) THEN n!1 ELSE length(fs!1) ENDIF
           = IF length(fs!1) > n!1 THEN n!1 ELSE length(fs!1)
        ENDIF)

Rule? (assert)

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of length_front.1.

length_front.2 :

  |-------
{1}    n!1 < length(fs!1)
[2]    (IF n!1 < length(fs!1) THEN n!1 ELSE length(fs!1) ENDIF
           = IF length(fs!1) > n!1 THEN n!1 ELSE length(fs!1)
        ENDIF)

Rule? (assert)

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of length_front.2.

Q.E.D.

Run time  = 0.35 secs.
Real time = 36.12 secs.
```

Figure A2.1 print-out of a interactive sessionusing the PVS proof-assistant

Appendix 3. PVS theory MyFinSeq

```
% PVS theory MyFinSeq written by Peter Lambooij

MyFinSeq [T: TYPE+]: THEORY
 BEGIN

  % type fin_seq is defined as a record
  %
  fin_seq: TYPE = [# length: nat,
                     seq: [below[length] -> T] #]

  % the empty sequence is defined with use of a dummy LAMBDA function
  %
  empty_seq: fin_seq =
    (# length := 0,
       seq := (LAMBDA (x: below[0]): epsilon! (t:T): true) #);

  % global declaration of local specification variables
  %
  fs, fs1, fs2: VAR fin_seq
  t: VAR T
  n: VAR nat

  % test whether a sequence is empty or not
  %
  empty?(fs): bool =
    (length(fs)=0)

  % append(fs1, fs2) yields the concatenation of fs1 and fs2
  %
  append(fs1, fs2): fin_seq =
    (# length := length(fs1) + length(fs2),
       seq := (LAMBDA (n:below[length(fs1)+length(fs2)]):
                 IF n < length(fs1)
                 THEN seq(fs1)(n)
                 ELSE seq(fs2)(n-length(fs1))
                 ENDIF) #);

  % add(fs,t) adds t to the end of fs
  %
  add(fs, t): fin_seq =
    (# length := length(fs)+1,
       seq := (LAMBDA (n:below[length(fs)+1]):
               IF n < length(fs)
               THEN seq(fs)(n)
               ELSE t
               ENDIF) #);

  % head(fs) yields the first element of fs
  %
  head(fs: {f: fin_seq | length(f)>0}): T =
    seq(fs)(0)

  % tail(fs) yields fs without the first element
  %
  tail(fs: {f: fin_seq | length(f)>0}): fin_seq =
    (# length := length(fs) - 1,
       seq := (LAMBDA (x: below[length(fs)-1]): seq(fs)(x + 1)) #)
  % front (fs) n yields: the first n elements of fs if n <  length(fs)
  %                    : fs                         if n >= length(fs)
  %
  front(fs, n): fin_seq =
    IF n < length(fs)
    THEN (# length := n, seq := (LAMBDA (x: below[n]): seq(fs)(x)) #)
    ELSE fs
    ENDIF

  % front (fs) n yields: the last n elements of fs if n <  length(fs)
  %                    : fs                        if n >= length(fs)
  %
  back(fs, n): fin_seq =
    IF n <= length(fs)
    THEN (# length := n,
            seq := (LAMBDA (x: below[n]): seq(fs)(x + length(fs)-n)) #)
    ELSE fs
    ENDIF

END MyFinSeq
```

Appendix 4. CCSL specification of YAPI

```
% -----------------------------------------------------
% PFC_imp specification in CCSL Producer-Fifo-Consumer
% written by Peter Lambooij 09/08/99
% added component_classes, inheritance and encapsulation
% added methods set to change attributes
% -----------------------------------------------------

TYPE bool: BASETYPE
TYPE nat: BASETYPE
TYPE posnat: BASETYPE
TYPE fin_seq: BASETYPE

BEGIN StatusValue: ADT
  CONSTRUCTOR
    idle: Carrier;
    writing: Carrier;
    reading: Carrier
END StatusValue


% -----------------------------------------------------
% class Sequence is an unbounded finite sequence buff
% with auxilliary attributes size and empty
% -----------------------------------------------------
BEGIN Sequence[A: TYPE]: CLASSSPEC

  IMPORTING "MyFinSeq[A]"

  ATTRIBUTE
    buff: Self -> fin_seq;
    size: Self -> nat;
    empty: Self -> bool;

  ASSERTION
    size_sequence:
    PVS
      size(x) = length(buff(x))
    ENDPVS

    empty_sequence:
    PVS
      empty(x) = (size(x) = 0)
    ENDPVS

  CONSTRUCTOR
    new_S: Self;

END Sequence


% -----------------------------------------------------
% class BoundSequence is a Sequence with upperbound N.
% -----------------------------------------------------
BEGIN BoundSequence[A: TYPE, N: posnat]: CLASSSPEC

  INHERIT FROM Sequence[A]
  ATTRIBUTE
    full: Self -> bool;

  METHOD
    set_BS: [Self, fin_seq] -> Self;

  ASSERTION
    full_fifo:
    PVS
      full(x) = (size(x) = N)
    ENDPVS

    set_bound_sequence:
    PVS
      FORALL (f: fin_seq):
        buff(set_BS(x,f)) = f
    ENDPVS

  CONSTRUCTOR
    new_BS: Self;

  CREATION
    new_bound_sequence:
    PVS
      buff(new_BS) = empty_seq
    ENDPVS

END BoundSequence


% -----------------------------------------------------
% class TransmitSequence is a Sequence
% with a natural nr_to_transmit
% -----------------------------------------------------
BEGIN TransmitSequence[A: TYPE]: CLASSSPEC

  INHERIT FROM Sequence[A]

  ATTRIBUTE
    nr_to_transmit: Self -> nat;

  METHOD
    set_TS: [Self,fin_seq, nat] -> Self;

  ASSERTION
    set_transmit_sequence:
    PVS
        FORALL (f: fin_seq, n: nat):
          buff(set_TS(x,f,n)) = f
          AND
          nr_to_transmit(set_TS(x,f,n)) = n
    ENDPVS

  CONSTRUCTOR
    new_TS: Self;

  CREATION
    new_transmit_sequence:
    PVS
      nr_to_transmit(new_TS) = 0
      AND
      buff(new_TS) = empty_seq
    ENDPVS

END TransmitSequence


% -------------------------------------------------------
% class ProcessNetwork consists of instantiations of
% TranferSequences prod en cons en BoundedSequence fifo
% and StatusValue fifo_status.
% Methods read, write and tick specify its behavior.
% -------------------------------------------------------
BEGIN ProcessNetwork[A: TYPE, N: posnat]: CLASSSPEC

  IMPORTING "MyFinSeq[A]"

  ATTRIBUTE
    prod: Self -> TransmitSequence[A]; % producer
    fifo: Self -> BoundSequence[A, N]; % fifo buffer
    cons: Self -> TransmitSequence[A]; % consumer
    status: Self -> StatusValue;

  METHOD
    read: [Self, nat] -> Self;
    write: [Self, nat] -> Self;
    tick: Self -> Self;

  ASSERTION
    set_nr_to_write:
      PVS
      FORALL (n: nat):
          IF size(prod(x)) >= n AND      % producer has enough items?
             nr_to_transmit(prod(x)) = 0 % previous write finished?
          THEN % set nr_to_transmit of prod to n
             prod(write(x,n)) = set_TS(prod(x),buff(prod(x)),n) AND
             fifo(write(x,n)) = fifo(x) AND
             cons(write(x,n)) = cons(x) AND
             status(write(x,n)) = status(x)
        ELSE % skip
             write(x,n) = x
        ENDIF
      ENDPVS

    set_nr_to_read:
    PVS
      FORALL (n: nat):
      IF   nr_to_transmit(cons(x)) = 0  % previous read finished?
        THEN % set nr_to_transmit of cons to n
           prod(read(x,n)) = prod(x) AND
           fifo(read(x,n)) = fifo(x) AND
           cons(read(x,n)) = set_TS(cons(x),buff(cons(x)),n) AND
           status(read(x,n)) = status(x)
      ELSE % skip
           read(x,n) = x
      ENDIF
    ENDPVS

    no_write_no_read:
    PVS
        ( % conditions to block writing
          empty(prod(x)) OR
          nr_to_transmit(prod(x)) = 0 OR
            status(x) = reading OR
          full(fifo(x))
          )
      AND ( % conditions to block reading
          nr_to_transmit(cons(x)) = 0 OR
          status(x) = writing OR
          empty(fifo(x))
        )
      IMPLIES
      % do not change prod, fifo and cons
      % and set status to idle
      prod(tick(x)) = prod(x) AND
      fifo(tick(x)) = fifo(x) AND
      cons(tick(x)) = cons(x) AND
      status(tick(x)) = idle
      ENDPVS

    no_write_go_read:
    % reading has priority over writing
    PVS
      NOT ( % conditions not to block reading
          nr_to_transmit(cons(x)) = 0 OR
          status(x) = writing OR
          empty(fifo(x))
        )
      IMPLIES
      % transfer one item from fifo to cons
      % and set status to sending
      prod(tick(x)) = prod(x) AND
      fifo(tick(x)) = set_BS(fifo(x),tail(buff(fifo(x)))) AND
      cons(tick(x)) =
      set_TS(cons(x),add(buff(cons(x)), head(buff(fifo(x)))),
nr_to_transmit(cons(x))-1) AND
      status(tick(x)) = reading
    ENDPVS

    go_write_no_read:
    PVS
      NOT ( % conditions not to block writing
          empty(prod(x)) OR
          nr_to_transmit(prod(x)) = 0 OR
            status(x) = reading OR
          full(fifo(x))
        )
      AND ( % conditions to block reading
          nr_to_transmit(cons(x)) = 0 OR
          status(x) = writing OR
          empty(fifo(x))
        )
      IMPLIES
      % transfer one item from prod to fifo
      % and set status fifo to receiving
      prod(tick(x)) = set_TS(prod(x),tail(buff(prod(x))),
nr_to_transmit(prod(x))-1) AND
      fifo(tick(x)) = set_BS(fifo(x),add(buff(fifo(x)),
head(buff(prod(x))))) AND
      cons(tick(x)) = cons(x) AND
      status(tick(x)) = writing
    ENDPVS
```

```
  CONSTRUCTOR
    new_PN: Self;

  CREATION
    new_ProcessNetwork:
    PVS
      prod(new_PN) = new_TS AND
      fifo(new_PN) = new_BS AND
      cons(new_PN) = new_TS AND
      status(new_PN) = idle
    ENDPVS

END ProcessNetwork
```

Appendix 5. Lemmas of YAPI

This appendix contains all lemmas that were used in proving the refinement relation between *PFC_abs* and *ProcessNetwork*. In figure A5.1 the dependencies between the main lemmas is shown.

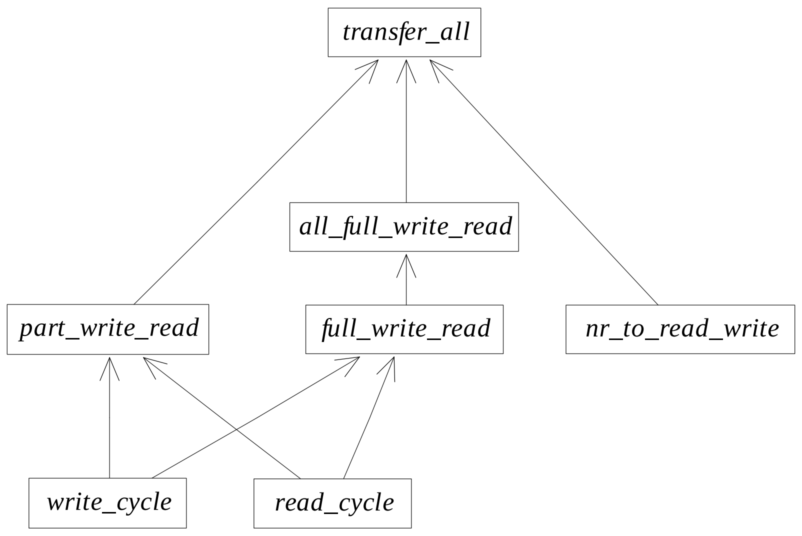

Figure A5.1 The main lemma's of specification *PFC_imp*. An arrow from lemma *a* to lemma *b* means that lemma *a* was used in the proof of lemma *b*.

```
ProcessNetwork_User[Self: TYPE, A: TYPE+, N: posnat]: THEORY
BEGIN

IMPORTING
  ProcessNetwork[Self, A, N],
  MyFinSeq[A],
  FinSeqLemmas[A],
  MyDivMod

c: VAR (ProcessNetworkAssert?[Self, A, N])
x: VAR Self

% -----------------------------------------------------------
% short-hand for multiple ticks
% -----------------------------------------------------------

ticks(c): [nat -> [Self -> Self]] =
  LAMBDA (n: nat):
    LAMBDA (x: Self):
      iterate(tick(c),n)(x)
```

```
one_tick: LEMMA % :-)
  FORALL (x: Self, t: nat):
    ticks(c)(t+1)(x) = tick(c)(ticks(c)(t)(x))

split_ticks: LEMMA % :-)
  FORALL (x: Self, t1: nat):
    FORALL (t2: nat):
      ticks(c)(t1+t2)(x) = ticks(c)(t2)(ticks(c)(t1)(x))

% ---------------------------------------------------------
% short-hand for pre-conditions of lemmas no_write_no_read,
% go_write_no_read and no_write_go_read
% ---------------------------------------------------------

go_write(c)(x): bool =
  NOT empty(prod(c)(x)) AND
  nr_to_transmit(prod(c)(x)) > 0 AND
  status(c)(x) /= reading AND
  NOT full(fifo(c)(x))

go_read(c)(x): bool =
  nr_to_transmit(cons(c)(x)) > 0 AND
  status(c)(x) /= writing AND
  NOT empty(fifo(c)(x))

complete_go_write_go_read: LEMMA % :-)
  Forall (x: Self):
    ( NOT go_write(c)(x) AND NOT go_read(c)(x) )
    OR
      go_read(c)(x)
    OR
    ( go_write(c)(x) AND NOT go_read(c)(x) )

% ---------------------------------------------------------
% bisimilarities to compare attributes of prod, fifo and cons
% ---------------------------------------------------------

eq_BS: [[LBoundSequence[A,N], LBoundSequence[A,N]] -> bool] =
  (LAMBDA (bs1, bs2: LBoundSequence[A,N]):
    buff(bs1) = buff(bs2)
  )

eq_TS: [[LTransmitSequence[A], LTransmitSequence[A]] -> bool] =
  (LAMBDA (bs1, bs2: LTransmitSequence[A]):
    buff(bs1) = buff(bs2)
    AND
    nr_to_transmit(bs1) = nr_to_transmit(bs2)
  )

bisim_eq_BS: LEMMA % :-)
  bisimulation?(loose_BoundedSequence,loose_BoundedSequence)(eq_BS)

bisim_eq_TS: LEMMA % :-)
  bisimulation?(loose_TransmitSequence,loose_TransmitSequence)(eq_TS)
```

```
% --------------------------------------------------------
% miscellaneous lemmas
% --------------------------------------------------------

plus_gt_0: LEMMA % :-)
  FORALL (a,b,c,d,e,f: nat):
    a >= 0 AND b >= 0 AND c >= 0 AND d >= 0 AND f >= 0
      IMPLIES
    a + b + c + d + e + f >= 0

times_gt_0: LEMMA % :-)
  FORALL (n: nat, m: posnat):
    n >= 0 IMPLIES n * m >= 0

equal_to_zero: LEMMA % :-)
  FORALL (n: nat):
    NOT n > 0  IMPLIES n = 0

transitivity: LEMMA % :-)
  FORALL(a,b,c: nat):
    a >= b AND b >=c IMPLIES a >= c

% --------------------------------------------------------
% lemmas to provide easy rewriting
% --------------------------------------------------------

size_fifo: LEMMA % :-)
  FORALL (x: Self):
    size(fifo(c)(x)) = length(buff(fifo(c)(x)))

empty_fifo: LEMMA % :-)
  FORALL (x: Self):
    empty(fifo(c)(x)) = (size(fifo(c)(x)) = 0)

full_fifo: LEMMA % :-)
  FORALL (x: Self):
    full(fifo(c)(x)) = (size(fifo(c)(x)) = N)

size_prod: LEMMA % :-)
  FORALL (x: Self):
    size(prod(c)(x)) = length(buff(prod(c)(x)))

empty_prod: LEMMA % :-)
  FORALL (x: Self):
    empty(prod(c)(x)) = (size(prod(c)(x)) = 0)

size_cons: LEMMA % :-)
  FORALL (x: Self):
    size(cons(c)(x)) = length(buff(cons(c)(x)))

empty_cons: LEMMA % :-)
  FORALL (x: Self):
    empty(cons(c)(x)) = (size(cons(c)(x)) = 0)

set_fifo_buff: LEMMA % :-)
  FORALL (x: Self, fs: fin_seq):
      buff(set_BS(fifo(c)(x),fs)) = fs

set_prod_buff: LEMMA % :-)
  FORALL (x: Self, fs: fin_seq, n: nat):
    buff(set_TS(prod(c)(x),fs,n)) = fs
```

```
set_prod_trans: LEMMA % :-)
  FORALL (x: Self, fs: fin_seq, n: nat):
    nr_to_transmit(set_TS(prod(c)(x),fs,n)) = n

set_cons_buff: LEMMA % :-)
  FORALL (x: Self, fs: fin_seq, n: nat):
    buff(set_TS(cons(c)(x),fs,n)) = fs

set_cons_trans: LEMMA % :-)
  FORALL (x: Self, fs: fin_seq, n: nat):
    nr_to_transmit(set_TS(cons(c)(x),fs,n)) = n


% ---------------------------------------------------------
% lemma about write(n) and read(n)
% ---------------------------------------------------------


nr_to_read_write: LEMMA % :-)
  FORALL (x: Self, n: nat):
    size(prod(c)(x)) >= n AND
    nr_to_transmit(prod(c)(x)) = 0 AND
    nr_to_transmit(cons(c)(x)) = 0
      IMPLIES
    eq_TS(prod(c)(read(c)(write(c)(x,n),n)),
          set_TS(prod(c)(x),buff(prod(c)(x)),n)) AND
    eq_BS(fifo(c)(read(c)(write(c)(x,n),n)),
          fifo(c)(x)) AND
    eq_TS(cons(c)(read(c)(write(c)(x,n),n)),
          set_TS(cons(c)(x),buff(cons(c)(x)),n)) AND
    status(c)(read(c)(write(c)(x,n),n)) = status(c)(x)

% ---------------------------------------------------------
% lemmas to express that prod (and cons) continue writing
% (reading) until they cannot anymore
% ---------------------------------------------------------

write_cycle: LEMMA % :-)
  FORALL (x: Self):
    FORALL (t: nat):
      t > 0 AND % added to avoid t = 0 case
      go_write(c)(x) AND
      status(c)(x) = idle AND
      empty(fifo(c)(x)) AND
      t <= size(prod(c)(x)) AND
      t <= N AND
      t <= nr_to_transmit(prod(c)(x))
        IMPLIES
      eq_TS(prod(c)(ticks(c)(t)(x)),
   set_TS(prod(c)(x), back(buff(prod(c)(x)),size(prod(c)(x))-t),
                     nr_to_transmit(prod(c)(x)) - t)) AND
      eq_BS(fifo(c)(ticks(c)(t)(x)),
   set_BS(fifo(c)(x), append(buff(fifo(c)(x)),front(buff(prod(c)(x)),t)))) AND
      eq_TS(cons(c)(ticks(c)(t)(x)), cons(c)(x)) AND
      IF t = 0
      THEN status(c)(ticks(c)(t)(x)) = idle
      ELSE status(c)(ticks(c)(t)(x)) = writing
      ENDIF
```

```
read_cycle: LEMMA % :-)
  FORALL (x: Self):
   FORALL (t: nat):
     status(c)(x) = idle AND
     t <= size(fifo(c)(x)) AND
     t <= nr_to_transmit(cons(c)(x))
       IMPLIES
     eq_TS(prod(c)(ticks(c)(t)(x)),
           prod(c)(x)) AND
     eq_BS(fifo(c)(ticks(c)(t)(x)),
           set_BS(fifo(c)(x),
                  back(buff(fifo(c)(x)), size(fifo(c)(x))-t))) AND
     eq_TS(cons(c)(ticks(c)(t)(x)),
           set_TS(cons(c)(x),
                  append(buff(cons(c)(x)),front(buff(fifo(c)(x)),t)),
                  nr_to_transmit(cons(c)(x)) - t)) AND
     IF t = 0
     THEN status(c)(ticks(c)(t)(x)) = idle
     ELSE status(c)(ticks(c)(t)(x)) = reading
     ENDIF

% ---------------------------------------------------------
% lemmas to couple a read-cycle to a write-cycle
% ---------------------------------------------------------

full_write_read: LEMMA % :-)
  FORALL (x: Self):
    FORALL (k: posnat):
      k >= N AND
      go_write(c)(x) AND
      status(c)(x) = idle AND
      empty(fifo(c)(x)) AND
      size(prod(c)(x)) >= k AND
      nr_to_transmit(prod(c)(x)) = k AND
      nr_to_transmit(cons(c)(x)) = k
        IMPLIES
      LET T = ticks(c)(2*N+2)(x) IN
      eq_TS(prod(c)(T), set_TS(prod(c)(x),
back(buff(prod(c)(x)),size(prod(c)(x)) - N), k - N)) AND
      empty(fifo(c)(T)) AND
      eq_TS(cons(c)(T),
set_TS(cons(c)(x),append(buff(cons(c)(x)),front(buff(prod(c)(x)),
N)), k - N)) AND
      status(c)(T) = idle

part_write_read: LEMMA % :-)
  FORALL (x: Self):
    FORALL (k: posnat):
      k > 0 AND
      k < N AND
      go_write(c)(x) AND
      empty(fifo(c)(x)) AND
      size(prod(c)(x)) >= k AND
      nr_to_transmit(prod(c)(x)) = k AND
      nr_to_transmit(cons(c)(x)) = k AND
      status(c)(x) = idle
        IMPLIES
      LET T = ticks(c)(2*k+2)(x) IN
      eq_TS(prod(c)(T), set_TS(prod(c)(x),
back(buff(prod(c)(x)),size(prod(c)(x)) - k), 0)) AND
      empty(fifo(c)(T)) AND
      eq_TS(cons(c)(T),
set_TS(cons(c)(x),append(buff(cons(c)(x)),front(buff(prod(c)(x)),
k)), 0)) AND
      status(c)(T) = idle

all_full_write_read_pred(c)(x: Self, k,m: posnat): bool =
    LET T = ticks(c)(m*(2*N+2))(x) IN
    eq_TS(prod(c)(T), set_TS(prod(c)(x),
back(buff(prod(c)(x)),size(prod(c)(x)) - m*N), k - m*N)) AND
    empty(fifo(c)(T)) AND
    eq_TS(cons(c)(T),
set_TS(cons(c)(x),append(buff(cons(c)(x)),front(buff(prod(c)(x)),
m*N)), k - m*N)) AND
    status(c)(T) = idle

% ---------------------------------------------------------
% lemma to describe repeated full write-read cycles
% ---------------------------------------------------------

all_full_write_read: LEMMA % :-)
  FORALL (x: Self):
    FORALL (k: posnat):
      FORALL (m: posnat):
        m * N <= k AND
        k >= N AND
        go_write(c)(x) AND
        empty(fifo(c)(x)) AND
        size(prod(c)(x)) >= k AND
        nr_to_transmit(prod(c)(x)) = k AND
        nr_to_transmit(cons(c)(x)) = k AND
        status(c)(x) = idle
          IMPLIES
        all_full_write_read_pred(c)(x,k,m)

% ---------------------------------------------------------
% lemma to describe transfer of all items from prod to cons
% ---------------------------------------------------------

transfer_all: LEMMA % :-)
  FORALL (x: Self):
    FORALL (k: nat):
      empty(fifo(c)(x)) AND
      size(prod(c)(x)) >= k AND
      nr_to_transmit(prod(c)(x)) = k AND
      nr_to_transmit(cons(c)(x)) = k AND
      status(c)(x) = idle
        IMPLIES
      LET T = ticks(c)(div(k,N)*(2*N+2) + 2*mod(k,N)+2)(x) IN
      eq_TS(prod(c)(T),
            set_TS(prod(c)(x),
                   back(buff(prod(c)(x)),size(prod(c)(x)) - k),
                   0)) AND
      empty(fifo(c)(T)) AND
      eq_TS(cons(c)(T),
            set_TS(cons(c)(x),
                   append(buff(cons(c)(x)),front(buff(prod(c)(x)), k)),
                   0)) AND
      status(c)(T) = idle

END ProcessNetwork_User
```