Theorem Prover Support for the Refinement of Stream Processing Functions

Robert Sandner and Olaf Müller*

Department of Computer Science, Munich University of Technology
80290 München, Germany
email: {mueller,sandnerr}@informatik.tu-muenchen.de

Abstract. In this paper, we show how to use the theorem prover Isabelle to provide tool support for FOCUS, a specification and verification framework for the stepwise development of distributed systems. FOCUS is embedded into Isabelle by modeling the basic notion of stream processing functions and by formalizing an appropriate set of assumption/ commitment refinement rules. Moreover, the refinement calculus is proven to be correct within this model. The model is based upon the logic HOLCF, an extension of higher order logic by the notions of domain theory. The well-known case study of a production cell is used to evaluate our proof support by mechanically verifying parts of a paper and pencil proof.

1 Introduction

Modeling distributed systems in a functional style by nondeterministic dataflow networks has a long tradition [Kok87, Bro87]. The system development methodology FOCUS [BDD+93, Bro93] follows this tradition and models distributed systems as networks of asynchronously communicating agents. The agents themselves are represented by a set of functions, where every function processes infinite streams of incoming messages and yields infinite streams of outgoing messages. The semantical foundation is provided by Scott's domain theory [Pau87]. Using for example the least fixed point theorem allows us to model feedbacks of message streams.

FOCUS also provides various refinement calculi. We concentrate on a particular calculus defined by a set of refinement rules in an Assumption/Commitment (A/C) style [SDW93].

The aim of this paper is to provide and evaluate mechanical proof support for FOCUS. Although quite a number of case studies have already been dealt with on paper using this design method (see, *e.g.*, [BFG+94]), there has not been any tool support for FOCUS until now. For our proof assistance we employ Isabelle [Pau94], an interactive theorem prover. Note that a model checking approach is not applicable here, as FOCUS components in general describe infinite state systems.

* Research supported by BMBF, *KorSys*.

Isabelle is generic in the sense that it provides an intuitionistic higher order metalogic wherein userdefined object logics may be embedded. Several object logics are already supported in the Isabelle distribution. For our purposes we choose the logic HOLCF [Reg95, Reg94], which provides the notions of domain theory as, *e.g.*, partial orders, continuity and least fixpoint induction. It is based on Isabelle/HOL, an object logic formalizing higher order logic.

We formalize the notions of streams, stream processing functions and the A/C refinement rules [SDW93] in HOLCF. The refinement calculus is definitionally embedded into HOLCF, i.e. its proof rules are not axiomatized, but mechanically proven with Isabelle.

The practicability of our formalization is evaluated by the well known case study of a production cell [Lin93]. We prove structural refinement in three hierarchy levels. The proof has already been done [FP93] in FOCUS, but the treatment was entirely mathematical, without computer support.

Our work is part of the project AUTOFOCUS [HSSS96], whose overall goal is a tool environment offering graphical description formalisms and appropriate analysis techniques which are embedded into the semantical framework of FOCUS. AUTOFOCUS will include graphical editors for hierarchical state transitions diagrams, network structure diagrams and message sequence charts. Analysis techniques will range from consistency checks over a simulation facility to formal verification. The work described in this paper fits into this toolset as a verification backend.

1.1 Outline of the Paper

The paper is organized as follows: Section 2 introduces the methods and tools used in our work. In Section 3 streams, stream processing functions and FOCUS components are formalized. Section 4 describes the embedding and verification of the refinement calculus. Finally, in Section 5 the case study of the production cell is presented.

2 Methods and Tools

In this section we give a survey of the formalisms and tools used in our approach.

2.1 FOCUS

FOCUS [BDD+93] provides a framework for the stepwise development of distributed systems. Starting from a requirement specification, a design specification is derived which is to be refined to an executable program in further steps. In this paper, we deal with the refinement of design specifications.

In the design phase, distributed systems are modeled in FOCUS as networks of asynchronously communicating agents. The agents themselves are modeled by continuous stream processing functions. A stream s is generated by the constructors ε (empty) and & (cons), elements are extracted by the usual head and

tail selectors $ft.s$ and $rt.s$. The operator $s|_k$ yields the prefix of the length k of s, $\#s$ the length of s, $alist©s$ filters all elements of $alist$ out of s, and $s \circ t$ concatenates s and t.

As mentioned above, FOCUS provides a calculus for the structured refinement of A/C-specifications. The calculus includes rules for sequential and parallel composition of agents (called SEQ and PAR) and for the introduction of feedback loops (FB), see Fig. 1.

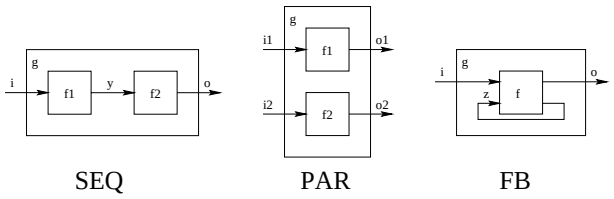

Fig. 1. Refinement Schemes for A/C-Specifications

Additional rules include a rule for specializing specifications by weakening the assumption and/or strengthening the commitment, and specialized feedback-rules. We write A/C-specifications as

$$f(i) = o \wedge assumption(i) \Rightarrow commitment(i, o) \tag{1}$$

where i and o are input and output streams and f is a stream processing function.

2.2 Isabelle/HOL and HOLCF

In our approach, we use the logic HOLCF [Reg95] both for formalizing specifications and for proving the refinement rules. HOLCF extends Isabelle's instantiation of HOL conservatively by the Logic of Computable Functions LCF [Pau87]. HOL formalizes Church's formulation of Higher Order Logic. To distinguish LCF types from HOL types, HOLCF introduces the type class `pcpo`, which is equipped with a complete partial order $\sqsubseteq$ and a least element $\bot$. `pcpo` is introduced as a subclass of `term`, the default class of HOL. HOLCF provides the type of monotonic, continuous functions between `pcpo`s. Elements of this type are called *operations*.

For operations a specific syntax is introduced for applications (`f`t`) and abstractions ($\Lambda$`x.t`). The fixpoint operator is denoted by `fix`. The syntax used for formulae is standard, except that there are two implications ($\longrightarrow$ and $\Longrightarrow$) and two equalities ($=$ and $\equiv$) which stand for object logic and metalogic respectively. Premises of theorems are enclosed in the brackets $[\![\ ]\!]$. In the following all formulae have been taken directly from Isabelle input and translated automatically into LaTeX, thanks to a version of Isabelle/HOL that allows the use of mathematical symbols like $\forall$ or $\exists$.

HOLCF comes with several standard domains. For the Cartesian product of domains the syntax <`a,b`> is introduced, with the selectors `cfst` and `csnd`. Further domains include strict products, strict sums and lifted types. There is also a datatype package supporting the introduction of user defined domains.

3 Formalizing FOCUS Specifications

In this section we present the formalization of stream processing functions and network specifications in HOLCF.

3.1 Formalizing Streams

FOCUS is based on continuous stream processing functions. The type of continuous functions is already provided by HOLCF. Therefore we only need to define a domain type of polymorphic streams. For the definition we use HOLCF's datatype package:

domain (α)`stream` = "&&" (`ft::`α) (**lazy** `rt::(`α`)stream`)

The definition is recursive: Streams are produced by the constructor &&, which appends the element `ft` of type α to the existing α stream `rt`. Empty streams are represented by the least element $\bot$. The operator && is lazy in its second argument: Otherwise the definition would yield an empty type.

Supplied with the above equation, the datatype package defines the new type `stream` with the constructors $\bot$ and `a&&s`, the counterparts of the FOCUS constructors ε and $a\&\&s$. It also provides definitions of the selectors `ft` and `rt` and derives a rich collection of theorems for the practical use of these definitions. In particular a rule for structural induction on streams is proven by the package. This rule is based on the functional `stream_take` which provides the functionality of the FOCUS operator $s|_k$.

In addition to the automatic definitions of the datatype package we introduce the operators $\#s$, $s \circ t$, $alist©s$ and the map functional `smap` on streams. These definitions include the introduction of the domain of lists and a datatype for infinite natural numbers. For the practical use of the operators about 120 theorems have been derived interactively in about 900 proof steps.

As an example, we discuss in the following the introduction of the length operator $\#s$. Since the length of a stream may be infinite, a datatype of possibly infinite natural numbers is required. We define this datatype by adding an infinity element to the HOL type `nat` for natural numbers:

datatype `inat = fin nat | infinity ("`∞`")`

The definition does not use the datatype package mentioned above, but a similar one for the logic HOL. For `inat` we redefine the relations < and <= and the successor function (`iSuc`). As we reused `nat`, we can directly use HOL's theories `Nat` and `Arith` for the subset of finite numbers. These theories provide an extensive formalization of arithmetic on natural numbers and therefore make our definition practically useful. We only need to derive theorems for the specific properties of the type `inat`. About 50 of the theorems mentioned above concerning the operators on streams deal with these properties. Equipped with the type `inat`, we can now define the operator #:

```
#s ≡ if stream_finite s
     then fin(μn. stream_take n`s = s)
     else ∞
```

Here the function `stream_finite` is used, which determines whether a stream is finite or not. In the finite case, we define the length of a finite stream as the least n ($\mu n.$) for which $s|_n$ yields s. Otherwise, `#s` yields ∞.

The definition shows a major benefit of HOLCF: Since every domain type is a HOL-type, the sublanguages HOL and LCF can be combined. We define the operator `#s` as a HOL function. Its domain is the `pcpo` type (α)stream and its range is the HOL type `inat`. This is not the only possible definition of `#s`. We may also introduce a domain type of infinite natural numbers and define `#s` as an *operation* of LCF. However, this would be a tedious task as we would have to develop a complete theory for natural numbers instead of making use of the existing theories `Nat` and `Arith`. However, the benefits of the above definition are not for free: the notion of continuity is not applicable to `#s` because of its range type. This complicates for example admissibility proofs considerably. (A predicate P is admissible, if it holds for the least upper bound of every chain satisfying P). For LCF terms admissibility often can be reduced to the continuity of the involved functions, which then can be proven automatically. However, it is possible to derive suitable counterparts of admissibility theorems in our setting. An example will be given in Section 5.1.

3.2 Formalizing Network Specifications

Our approach of formalizing specifications is based on the Agent Network Description Language ANDL [SS95]. ANDL provides graphical and textual specification schemes for FOCUS components which can be translated automatically into HOLCF notation. In the following we describe only the HOLCF syntax generated by the translation.

ANDL provides schemes for writing functional specifications both of *basic agents* and of *agent networks*. The description scheme for *basic agents* consists of just two implications (`Ass` and `Comm` are definition *schemes*)

```
basic_f f ≡ ∀i o. f`i = o ⟶ Ass i ⟶ Comm i o
```

which represents a direct encoding of (1) in HOLCF. The scheme for the description of *agent networks* is more interesting. We introduce these descriptions by an example, shown in Fig. 2. The graphical ANDL specification of this example has the following counterpart in HOLCF syntax:

```
network_table f ≡
  (∃ f1 f2. basic_control f1 ∧  basic_motor f2 ∧
            (∀ i a o. f`<i,a> = o ⟶
                      (∃ y z. f1`<i,a,z> = <o,y> ∧  f2`y = z ∧
                              minimal <o,y,z>)))
```

```
minimal <o,y,z> ≡
  (∀ o1 y1 z1. f1‘<i,a,z> = <o1,y1> ∧  f2‘y1 = z1 ⟶
               <o,y,z> ⊑ <o1,y1,z1>)
```

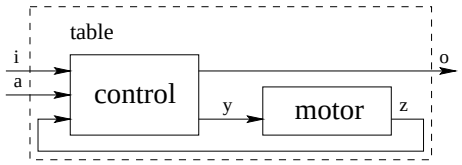

Fig. 2. Example of an agent network

The network specification `network_table` demands that there exist stream processing functions `f1` and `f2` which fulfill the specifications of the basic components, i.e. the basic specifications should be consistent. Furthermore, for all input and output streams `i,a` and `o` there must be a network configuration including the internal streams `y` and `z` which satisfies the network description. Finally, the internal and the output streams must be minimal. This implies according to Scott that the network computes a least fixed point. This constraint is necessary for the unique description of networks which contain feedback loops.

4 Formalizing and Verifying the Refinement Rules

As mentioned already in Section 2.1, we base our refinement notion upon the A/C refinement calculus of Stølen et al. described in [SDW93]. Refinement in our context means inclusion of the semantic models of stream processing functions. The definitions of [SDW93], however, have to be adapted to our setting first. Then a proof for each rule is given in the theorem prover Isabelle.

We formalized and verified two rules for sequential and one for parallel composition, a rule for specializing specifications and two rules for feedback loops. In the following we present only a rule for sequential composition and a rule for feedback loops.

4.1 A Rule for Sequential Composition

The rule SEQ is used for sequential composition of two components according to Fig. 1. We employ this simple rule to show the transformation of the format described in [SDW93] into ANDL/ HOLCF. Whereas in ANDL we describe networks uniformly by equations given by the network structure, Stølen et al. use explicit composition operators. They introduce the operators $\circ$ and $||$ for sequential and parallel composition and the operator μ for feedback loops. In [SDW93], the rule SEQ is written as follows:

$$\frac{\begin{array}{l} Ass\_g\ i \Rightarrow Ass\_f1\ i \\ Ass\_g\ i \wedge Comm\_f1\ (i,y) \Rightarrow Ass\_f2\ y \\ Ass\_g\ i \wedge Comm\_f1\ (i,y) \wedge Comm\_f2\ (y,o) \Rightarrow Comm\_g\ (i,o) \end{array}}{[Ass\_g, Comm\_g] \rightsquigarrow [Ass\_f1, Comm\_f1] \circ [Ass\_f2, Comm\_f2]} \quad \text{(SEQ)}$$

Here, A/C specifications are written as $[Ass\_f, Comm\_f]$. The term $spec1 \rightsquigarrow spec2$ means that *spec1* can be refined to *spec2*, which logically states that *spec2* implies *spec1*. In our approach, the rule SEQ has to be formalized in HOLCF as follows:

```
⟦ ∀i y. f1‘i = y ⟶ Ass_f1 i ⟶ Comm_f1 i y;
  ∀y o. f2‘y = o ⟶ Ass_f2 y ⟶ Comm_f2 y o;
  f1‘i = y; f2‘y = o;
  Ass_g i;

  Ass_g i ⟶ Ass_f1 i;
  Ass_g i ∧  Comm_f1 i y ⟶ Ass_f2 y;
  Ass_g i ∧  Comm_f1 i y ∧  Comm_f2 y o ⟶ Comm_g i o ⟧
⟹ Comm_g i o
```

The second part of the premises correspond exactly to the premises of SEQ. As we do not use an explicit operator for sequential composition, we added the network description to the premises. Furthermore, some implicit information is made visible: The basic specifications for *f1* and *f2*

$$f1(i) = o \wedge Ass\_f1(i) \Rightarrow Comm\_f1(i, o)$$

are also added to the premises, in order to bind *f1* to *Ass_f1* and *Comm_f1*. Finally, the specification $[Ass\_g, Comm\_g]$ is split into the premise *Ass_g* and the remaining proof goal *Comm_g*.

The proof of this rule can be done completely automatically in Isabelle.

4.2 A Rule for Feedback Loops

In this section we present a rule for feedback loops, called FB2, which allows to express liveness conditions in the assumption (see also Fig. 1). We start again with the version presented in [SDW93]:

$$\frac{\begin{array}{l} adm(Ass\_f) \\ Ass\_g\ i \Rightarrow Ass\_f(\hat{\imath}_0, \langle\rangle) \\ Ass\_g\ i \wedge Ass\_f(i, z) \wedge Comm\_f(i, z, o, z) \Rightarrow Comm\_g(i, o) \\ Ass\_g\ i \wedge Ass\_f(\hat{\imath}_j, x) \wedge Comm\_f(\hat{\imath}_j, x, o, z) \Rightarrow Ass\_f(\hat{\imath}_{j+1}, z) \end{array}}{[Ass\_g, Comm\_g] \rightsquigarrow \mu[Ass\_f, Comm\_f]} \quad \text{(FB2)}$$

The basic idea of the rule is the following: The assumption Ass_f holds initially and is preserved by every computation step. Since Ass_f is admissible, it also holds for the complete — possibly infinite — computation. Of course, the commitment predicate $Comm\_f$ must be strong enough to imply the commitment predicate for $Comm\_g$.

The steps of the computation are modeled by chains which approximate the input and output streams of f: The nth element of the chains consist of the consumed input and the produced output after n computation steps. To apply

the rule, the user has to supply a chain $\hat{\imath}$ which describes the consumption of the input stream i by the component f. In the formalization of the rule in HOLCF the chain $\hat{\imath}$ (`i_chain`) is required explicitly:

```
⟦ ∀i o. f‘i = o ⟶ Ass_f i ⟶ Comm_f i o;
  f‘<i, z> = <o,z>;
  ∀oz1. f‘<i, csnd‘oz1> = oz1 ⟶ <o,z> ⊑ oz1;
  Ass_g i;
  is_chain i_chain; (⨆n. i_chain n) = i;

  adm Ass_f;
  Ass_g i ⟶ Ass_f <i_chain 0,⊥>;
  Ass_g i ∧  Ass_f <i,z> ∧  Comm_f <i,z> <o,z> ⟶ Comm_g i o;
  ∀j x o z.
   Ass_g i ∧  Ass_f <i_chain j,x> ∧  Comm_f <i_chain j,x> <o,z>
   ⟶ Ass_f <i_chain (Suc j),z> ⟧
⟹ Comm_g i o
```

The proof of this rule uses a second chain (`oz_chain`) modeling the output produced during the computation. Its least upper bound is the pair $\langle o, z\rangle$. We show by induction that for all n that *Ass_f* holds for the nth computation step, i.e. for the nth element of both chains. The premise *adm(Ass_f)* ensures that *Ass_f* also holds for the least upper bounds of the chains, i.e. for the complete streams i and z. From the semantics of A/C-specifications and the third premise of rule FB2 it follows that *Comm_g* holds.

In [SDW93] the chain `oz_chain` modeling the produced output is only described by axioms. The main difficulty in the proof of the rule is to find a *definition* of this chain independent of the explicit construction of the user supplied chain `i_chain` and to prove the desired properties:

```
oz_chain 0 = <⊥,⊥>
oz_chain (Suc n) = f‘<i_chain n,csnd‘(oz_chain n)>
⨆n.oz_chain n = <o,z>
```

For this definition we need a sophisticated recursion principle that allows not only to refer to the nth element of the chain but to n explicitly, which is not available in HOLCF. Therefore we are forced to define the chain in pure HOL. Since theorems concerning least upper bounds of chains are not supported by the HOL library there is a lack of convenient theorems supporting the proof of the above properties, in particular of the last one. Therefore these proofs are rather tricky and require about 220 interactive proof steps, carried out in 11 lemmata. Actually, they demand the main part of the proof of the feedback rule which requires about 260 proof steps.

However the rule itself is relatively easy to apply, as will be shown in Section 5. The fact that the user must supply the construction of a chain may seem to indicate the opposite. However, the definition of a chain by the user makes the rule extremely flexible: Consider a component with more than one input

stream. In that case, various feedback recursions of the component are possible which can all be tackled by just one rule. Furthermore, in many cases feedback recursions are simple, which implies trivial chains. This will also be shown by an example in Section 5.

Note that for the refinement of specifications which do not contain liveness conditions there is a refinement rule available (FB) which does not require the construction of chains.

4.3 Syntactical Restrictions for Specifications

In this section we discuss some restrictions imposed on the formulation of the rules, which sometimes make transformations between specifications necessary. The problem is to formulate the rules independent on the number of input and output channels. Our solution is that we do not give a concrete type for a channel, but only determine the type *class* of the channel as the HOLCF default type class `pcpo`. Thus a channel of a rule can be instantiated for one stream only or for an arbitrary tuple of streams, encoded as nested pairs of streams. This is possible, as pairs of `pcpo`s also belong to the class `pcpo`. Note that thereby we generalize the refinement rules: They do not hold for (α)`stream` only, but also for every type of class `pcpo`, *e.g.* for timed streams.

However, some restrictions remain. First, in the feedback rule we have to divide the input channels into environment inputs and feedback inputs. They are coded as a pair `<i,z>`, where `i` and `z` are of class `pcpo` as described above. Therefore the syntax of a specification *spec* depends on the structure of the environment. Thus our approach is not completely modular: Different versions of semantically equivalent specifications may be necessary in different environments.

Second, if *spec* is further refined by the rule for parallel composition, there may be another pairing of input and feedback streams necessary. Therefore transformations between semantical equivalent specifications may be necessary. Although there is not a general refinement rule for such transformations, the corresponding refinement proofs are trivial, as the differences are only syntactical. Therefore, the proofs can be performed automatically.

Notice that all these restrictions are due to the type system of Isabelle/HOL. With dependent types, for example, a more flexible and modular solution would likely be possible.

5 Case Study: Production Cell

The example of a production cell has already been tackled using several formal methods, the task description was developed at FZI Karlsruhe [Lin93]. A first impression of the production cell is given by Fig. 3.

Our aim is to investigate the usability of the formalized refinement rules in practice. Emphasis is laid on the demonstration of the efficiency of our tool support. Therefore we do not develop specifications for all components of the system.

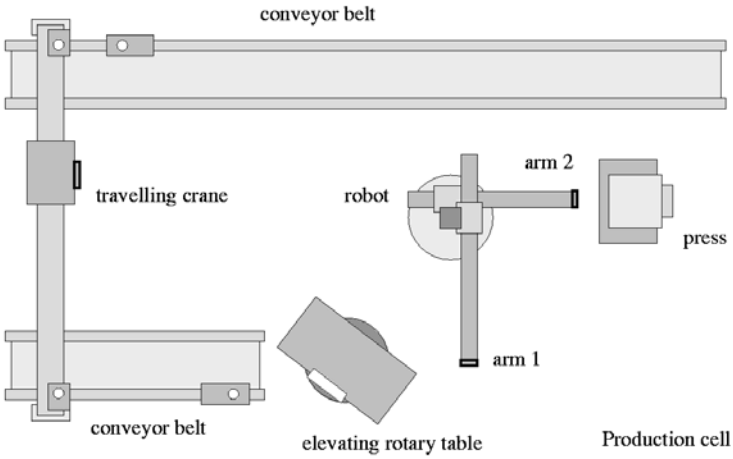

Fig. 3. The specified Production Cell

This has already been done in [FP93], where the whole production cell is developed in FOCUS and paper proofs of the refinement steps are given. Instead we focus on the development of one component of the production cell, reproducing the corresponding paper proofs of [FP93] in Isabelle. Starting with an abstract specification of the whole system, we carried out the complete development of the controller and motor of the elevating rotary table. The development process consists of three major steps (see Fig. 4):

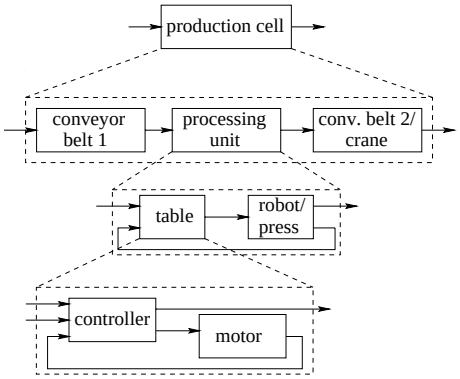

Fig. 4. Refinement Steps carried out in Isabelle

First, we divide the system into three units: *first conveyor belt*, *processing unit* and *second conveyor belt/crane*. The processing unit is refined to the subsystems *elevating table* and *robot/press* in the second step. As an example, we will illustrate this step in more detail in the following sections. Finally, the elevating table is itself refined to the components *motor* and *controller*.

5.1 Refinement of the Processing Unit: The Feedback Loop

The refinement process of the processing unit PU involves two steps: First, a feedback loop is introduced using the rule FB2. The resulting specification PU2 is divided into the components table and robot/press in a further refinement step.

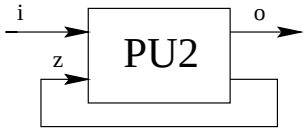

Fig. 5. Introducing a Feedback Loop for the Processing Unit

The Specification. The component specifications of PU and PU2 and the network specification of PU are shown below:

```
(* specification of PU *)
Ass_PU i ≡ True
Comm_PU i o ≡ smap‘process‘i = o
basic_PU f ≡
   (∀ i o. f‘i = o ⟶ Ass_PU i ⟶ Comm_PU i o)

(* specification of PU2 *)
Ass_PU2 <i,z> ≡ #z <= #i ∧  #i <= iSuc #z
Comm_PU2 <i,x> <o,z> ≡ smap‘process‘i = o ∧  #z = #o
basic_PU2 f ≡ (∀ i x o z.
    f‘<i,x> = <o,z> ⟶ Ass_PU2 <i,x> ⟶ Comm_PU2 <i,x> <o,z>)

(* network specification *)
network_PU f ≡ (∃ f1. basic_PU2 f1 ∧
                (∀ i o. f‘i = o ⟶
                        (∃ z. f1‘<i,z> = <o,z> ∧  minimal <o,z>)))
```

The component specification of PU is obviously a degenerate A/C specification: It imposes no constraints on the input stream and assures that the component processes every input, which is modeled by the function process. The network specification of PU is also simple: it consists of the component PU2 which in addition uses its second output stream as a second input. The specification of PU2 is more interesting: the assumption Ass_PU2 demands

$$\#z \leq \#i \leq \#z + 1$$

As the input stream z is produced by PU2, this is a constraint concerning the behavior of not only the environment of PU2, but also PU2 itself. It states that PU2 alternatingly consumes its input streams. This requirement is formulated with the next refinement step in mind: PU2 is refined to the sequential composition of the components table and robot/press. The table has to pass on every input to the robot, but before the next input can be processed an acknowledgement has to be received.

Proving the Correctness of the Refinement Step. The proof is straightforward: starting with the goal

$$\textit{network spec}\ (PU) \Rightarrow \textit{basic spec}\ (PU)$$

we first have to unfold the specifications and perform some simple transformations. This requires six trivial proof steps and can be done schematically.

The next major step in every refinement proof is to apply the suitable refinement rule. For using the rule FB2, we also have to supply the definition of a chain `i_chain` which models the consumption of the stream `i` by `PU2`. The construction of `i_chain` is quite simple:

$$\lambda \texttt{n. stream\_take n`i}$$

This means that after n computation steps, the prefix of length n of `i` has been consumed. Supplying the definition of `i_chain`, applying FB2 and proving the first four premises of FB2 requires another six interactive proof steps.

The remaining task is to prove that the premises of the mathematical representation of the rule discussed in Section 4.2 and the demanded properties of `i_chain` hold. Except proving the admissibility of `Ass_PU2`, this is relatively easy: it mainly requires term rewriting using some properties of FOCUS operators on streams. The proof of

$$adm(\#a \leq \#b)$$

is subtle because of the formalization of the operator # in HOL as discussed in Section 3.1. However, a more general version of this theorem could be proved which can be applied frequently throughout the case study. The proof of this theorem required 27 interactive proof steps. Using this theorem and the properties of the FOCUS's operators, proving the premises of FB2 requires just eight proof steps.

5.2 Further Refinement into Table and Robot/Press

The remaining task is now to divide the responsibilities of the elevating rotary table and the system robot/press (`RP`):

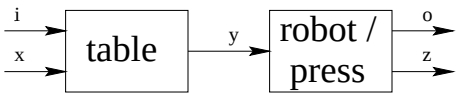

Fig. 6. Dividing the Processing Unit

The Specification. In HOLCF Fig. 6 looks as follows:

```
Ass_table <i,x> ≡ #x <= #i ∧  #i <= iSuc #x
Comm_table <i,x> o ≡ i = o
basic_table f ≡
  (∀ i x o. f`<i,x> = o ⟶ Ass_table <i,x> ⟶ Comm_table <i,x> o)
```

```
Ass_RP i ≡ True
Comm_RP i <o,z> ≡ smap‘process‘i = o ∧  #z = #o
basic_RP f ≡
  (∀ i o z. f‘i = <o,z> ⟶ Ass_RP i ⟶ Comm_RP i <o,z>)

network_PU2 f ≡
   (∃ f1 f2 . basic_table f1 ∧  basic_RP f2 ∧
      (∀ i x o z. f‘<i,x> = <o,z> ⟶
         (∃y. table‘<i,x> = y ∧  RP‘y = <o,z> ∧  minimal <y,o,z>)))
```

The specification of the component PU2 was already designed with respect to the specifications of table and RP. Therefore the specifications follow closely that of PU2. The table has to send every input to the robot provided the press has sent enough acknowledge messages. Hence the assumption Ass_table is identical with Ass_PU2 and the commitment assures the identity of the input and output stream. The assumption Ass_RP of the system robot/press is empty. The system processes each input element and generates the acknowledge messages for the table. Hence the commitment Comm_RP is identical with Comm_PU2.

Note that we refined this rather abstract specification in the next refinement step to a representation close to an implementation.

The Correctness Proof of the Refinement Step. The proof is nearly trivial and consists of three major steps analogous to the proof in the previous section. First some schematic transformations are performed. Second the refinement rule SEQ is applied. In the third step we have to prove the premises of SEQ. Since they consist of trivial implications only, they can be proven automatically.

6 Conclusion

In this paper we described the definitional embedding of an A/C refinement calculus for FOCUS in the logic HOLCF of the theorem prover Isabelle. As far as we know, this embedding is the first mechanical verification support applicable to FOCUS or similar approaches modeling distributed systems as nondeterministic dataflow networks.

The embedding has successfully been used to redo the structural development of a production cell component in a completely tool supported way. Our experience shows that the application of the formalized rules often follows a common scheme so that a high degree of automation seems to be possible.

In comparison to the paper proof we encountered a remarkable mismatch in the verification of the feedback rule. Whereas in [SDW93] a complicated chain was only *axiomatized* by demanding some properties, we had to *define* this chain and to prove these properties, which required more than 80% of the whole proof effort for the rule. In our opinion, this emphasizes the value of rigorous machine-checked proofs in contrast to (semiformal) paper proofs.

The refinement rules were formalized in a general way, so that they hold for various kinds of streams, i.e. also for timed streams or for finite streams only, although then the rules are not tailored for these specific streams.

The type system of Isabelle/HOL sometimes enforces us to apply syntactical transformations on component specifications. This requires identity proofs on the semantical level, which, however, can be automated. A richer type system, as *e.g.* dependent types, would allow a more modular handling of FOCUS components.

A major benefit of our work is the deeper experience in the use of HOLCF. On the one hand, we are convinced that HOLCF is exactly the right choice for a formalization of FOCUS. The notions of LCF are a necessity for the semantical foundation (coinduction would be the only alternative [LPM93]), and the expressivity of higher order logic allows for natural high-level specifications. On the other hand, HOLCF needs a lot of proof experience, in particular when combined with logical elements of pure HOL. HOL and its sublogic HOLCF have their own strengths and weaknesses, and using one of them at the wrong place often causes unexpectable trouble. For the definition of the length operator, for example, we could reuse the well established HOL theories for arithmetic on natural numbers. The price was to establish proof support for admissibility proofs for this operator in HOL, which in HOLCF in most cases are completely automated. Another example occurred in the proof of the feedback rule. A recursion functional not available in HOLCF forced us to define a chain in pure HOL. The missing proof support for arguing about limits of chains in HOL lengthened the proof considerably. What is needed is a clear interface between HOLCF and pure HOL which allows mutual reuse without loss of reasoning power. A first step into this direction has been developed in [MN97].

The project AUTOFOCUS [HSSS96] aims among other things at tool support for FOCUS based graphical description techniques, which in particular include the ANDL network specifications we used in our FOCUS formalization. This will provide a graphical interface for our HOLCF specifications, which is accessible for design engineers.

Acknowledgements. We wish to thank Franz Regensburger for intensive discussions. Helpful suggestions were also provided by Ketil Stølen and Bernhard Schätz. Last, but not least, we thank Jan Philipps for his case study, David Oheimb for his HOLCF expertise and Markus Wenzel for valuable comments and his excellent tea.

References

[BDD+93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems: An Introduction to Focus — Revised Version. Technical Report TUM-I9202-2, Technische Universität München, Fakultät für Informatik, 80290 München, Germany, 1993.

[BFG+94] M. Broy, M. Fuchs, T. F. Gritzner, B. Schätz, K. Spies, and K. Stølen. Summary of Case Studies in FOCUS – a Design Method for Distributed Systems. Technical Report TUM-I9423, Technische Universität München, Fakultät für Informatik, 80290 München, Germany, 1994.

[Bro87] M. Broy. Semantics of finite and infinite networks of concurrent communicating agents. *Distributed Computing*, 2:13–31, 1987.

[Bro93] M. Broy. Interaction Refinement – The Easy Way. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and System Sciences*. Springer, 1993.

[FP93] Max Fuchs and Jan Philipps. Formal development of a production cell in FOCUS – a case study. In Thomas Lindner and Claus Lewerentz, editors, *Formal Development of Reactive Systems – Case Study Production Cell*, number 891 in Lect Notes in Computer Science, chapter 11, pages 185 – 195. Springer Verlag, 1993.

[HSSS96] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus - a tool for distributed systems specification. In Joachim Parrow Bengt Jonsson, editor, *Proc. FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, pages 467–470. Springer-Verlag, 1996.

[Kok87] J. Kok. A fully abstract semantics for data flow nets. In *Computer Aided Verification*, volume 259 of *Lecture Notes in Computer Science*, pages 351–368. Springer-Verlag, 1987.

[Lin93] Thomas Lindner. Task deskription. In Thomas Lindner and Claus Lewerentz, editors, *Formal Development of Reactive Systems – Case Study Production Cell*, number 891 in Lect Notes in Computer Science, chapter 2, pages 7 – 19. Springer Verlag, 1993.

[LPM93] Francois Leclerc and Christine Paulin-Mohring. Programming with stream in coq, a case study: the sieve of eratosthenes. In H. Barendregt and T. Nipkow, editors, *Proc. Types for Proofs and Programs (TYPES'93)*, volume 806 of *Lecture Notes in Computer Science*, 1993.

[MN97] Olaf Müller and Tobias Nipkow. Traces of I/O-Automata in Isabelle/HOLCF. In *Proc. 7th Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[Pau87] Lawrence C. Paulson. *Logic and Computation*. Cambridge University Press, 1987.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Reg94] Franz Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. PhD thesis, Technische Universität München, 1994.

[Reg95] Franz Regensburger. HOLCF: Higher Order Logic of Computable Functions. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 293–307. Springer-Verlag, 1995.

[SDW93] Ketil Stølen, Frank Dederichs, and Rainer Weber. Assumption/commitment rules for networks of asynchronously communicating agents. TUM-I 9303, Technische Universität München, February 1993. SFB-Bericht Nr.342/2/93 A.

[SS95] B. Schätz and K. Spies. *Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik*. 1995. Technischer Bericht, TU München, Institut für Informatik, SFB-Bericht Nr. 342/16/95 A.

This article was processed using the LaTeX macro package with LLNCS style