ALICE
An Advanced Logic for Interactive Component Engineering

Borislav Gajanovic and Bernhard Rumpe

Software Systems Engineering Institute
Carl-Friedrich-Gauß Faculty for Mathematics and Computer Science
Braunschweig University of Technology, Braunschweig, Germany
http://www.sse-tubs.de

Abstract. This paper presents an overview of the verification framework ALICE in its current version 0.7. It is based on the generic theorem prover Isabelle [Pau03a]. Within ALICE a software or hardware component is specified as a state-full black-box with directed communication channels. Components send and receive asynchronous messages via these channels. The behavior of a component is generally described as a relation on the observations in form of streams of messages flowing over its input and output channels. Untimed and timed as well as state-based, recursive, relational, equational, assumption/guarantee, and functional styles of specification are supported. Hence, ALICE is well suited for the formalization and verification of distributed systems modeled with this stream-processing paradigm.

1 Introduction

1.1 Motivation

As software-based systems take ever more and more responsibility in this world, correctness and validity of a software-based system is increasingly important. As the complexity of such systems is also steadily increasing, it becomes ever more complicated to ensure correctness. This especially concerns the area of distributed systems like bus systems in transportation vehicles, operating systems, telecommunication networks or business systems on the Internet. Expenses for verification are an order of magnitude higher than the expenses of the software testing up to now. This, on the one hand, will not change easily in the short run but it will also become evident that crucial parts of software need a different handling than less critical ones. So verification will go along with testing in the future. Full verification, however, will at least be used for critical protocols and components. To reduce verification expenses, a lot has been achieved in the area of theorem provers, like Isabelle [Pau03a, Pau03b, NPW02], in the last years. Based on these foundational works and on the increasing demand for powerful domain specific theories for such theorem provers, we have decided to realize ALICE as a stream-processing-oriented, formal framework for distributed, asynchronously communicating systems.

ALICE is a still growing framework within Isabelle for the verification of logically or physically distributed, interactive systems, where the concept of communication or message exchange plays a central role. An interactive system (see also [BS01] for a characterization) consists of a number of components with precisely defined interfaces. An interactive component interacts with its environment via asynchronous message sending and receiving over directed and typed communication channels. Each channel incorporates an implicit, unbounded buffer that decouples the sending and arrival of messages, and thus describing asynchronous communication. In timed channels, we can control how long these messages remain in this implicit buffer. Fig. 1 illustrates the graphical notation for the syntactical interface of a simple interactive component with one input and one output channel.

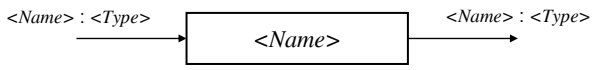

Fig. 1. Illustration of an interactive component as a black-box

In ALICE message flow over channels is modeled by possibly infinite sequences of messages called streams. Such a stream represents the observation of what happens on a channel over time. Since infinite sequences are also included, the liveness and fairness properties of systems can also be dealt with. ALICE provides type constructors *astream* for building (untimed) streams and *tastream* for timed streams over arbitrary types.

As an advanced verification framework, ALICE will offer precisely formalized and comfortably usable concepts based on an underlying logic language called HOL [NPW02] as available in Isabelle. Using a well explored and rather expressive logic language allows us to build on an elaborated set of verification techniques for ALICE.

ALICE will provide support for a number of techniques to specify a component. A specification can be a relation between input and output streams, a stream-processing function, a mapping of input to output, or a set of stream-processing functions allowing to describe non-determinism and underspecification. All variants can be timed or untimed. Further support will be given to map between these styles, allowing to choose appropriate specification techniques for each problem and integrating those later.

Although ALICE does already provide some of these features in its current version, this workshop paper also reports on work still to be done (for the previous version see [GR06]). In a future version ALICE will provide the following:

– A verification framework based on Isabelle supporting development methods for real time, distributed, asynchronously communicating and object oriented systems, respectively. This supports e.g. the development methodologies of [Rum96] and FOCUS [BS01].

– A formal semantics framework for various languages based on stream-processing, e.g. UML's composite structure diagrams that will be formalized based on streams [BCR06, BCR07a, BCR07b].
– A sophisticated verification tool for distributed, interactive systems or, at least, their communication protocols based on stream-processing (see [Ste97] for a survey of stream-processing).

In the following we give a compact overview of Isabelle's HOL and HOLCF that acts as a reminder for experts of the field. An introduction can be found in [NPW02, Reg94] before we start describing features of ALICE in Section 2 and demonstrating the use of ALICE in Section 3 on the Alternating Bit Protocol. Section 4 concludes the paper with a discussion.

1.2 HOL

Isabelle is a generic theorem prover, hence, it can be instantiated with object logics and appropriate proof tactics. Isabelle/HOL [NPW02], in short HOL, is such an elaborated higher order logic, dealing amongst others with sets, relations, total functions, natural numbers, and induction.

HOL provides a term syntax close to mathematical syntax and constructs from functional languages. It also provides basic types like *bool* or *nat*. For building sets over arbitrary types, HOL provides the type constructor *set*. Function types are built with the infix type constructor $\Rightarrow$ for total functions. To build more complex types, the mentioned, and a number of additional basic types and type constructors are provided.

HOL inherits the type system of Isabelle's metalogic including its automatic type inference for variables. There are two kinds of implication: the implication on the level of object logic, in this case HOL, symbolized by $\longrightarrow$, and the symbol $\Longrightarrow$ for Isabelle's inference. Analogously, there is an object logics symbol for the equality, in this case $=$, and the metalogics symbol $\equiv$ for the definitional equality.

In Isabelle assumptions of an inference rule are enclosed in $[\![\ ]\!]$ and separated by ;. The metalogics universal quantifier is symbolized by $\bigwedge$.

1.3 HOLCF

Isabelle/HOLCF [Reg94, MNvOS99], shortly HOLCF, is a conservative extension of HOL with domain theoretical concepts, such as chains, continuity, admissibility, fixpoint recursion and induction, as well as some basic types and datatypes e.g. for lazy lists.

HOLCF extends HOL with the central type class *pcpo* for "pointed complete partial orders". Any type that is a member of this type class features a special relation symbol $\sqsubseteq$ for a partial order on its elements, the least element symbolized by $\bot$, and the existence of the least upper bound for any chain of its elements with respect to $\sqsubseteq$.

This extension is carried out in layers of theories, beginning with the definition of type class *po* for partial orders. *po* is extended to type class *cpo*, where the existence of the least upper bound for any chain, symbolized by $\bigsqcup i.\ Y\ i$, is introduced. Here, *Y* is a chain of growing elements and *i* the index over natural numbers. Based on these theories, monotonicity and continuity for HOL functions on *cpo* types is formalized.

Type class *pcpo* finally introduces the existence of the least element in its members. We call the members of this class HOLCF types. Subsequently, HOLCF provides the new infix type constructor $\rightarrow$ for the construction of continuous functions on HOLCF types. Analogously to the HOLCF types, we call these functions HOLCF functions or operations. These functions, by definition, exhibit the advantages of continuous functions, such as composability, implementability etc. A lambda-abstraction, denoted by Λ (not to confuse with HOL's λ) and a corresponding function application, using the symbol $\cdot$ (opposite to HOL's white space) is provided accordingly.

Subsequently, the fixpoint theory *Fix* mainly implements a continuous fixpoint operator, symbolized by *fix*, and the fixpoint induction principle. Hence, with $\rightarrow$, *fix*, and HOLCF datatypes a complete HOLCF syntax for defining and reasoning about HOLCF functions and types is provided, which is separate from HOL's function space. As an advantage, by construction, HOLCF function abstraction and application remains in the HOLCF world.

1.4 Related Work

A good outline on different approaches to formalize possibly infinite sequences in theorem provers like Isabelle or PVS, as well as a detailed comparison can be found in [DGM97, Mül98]. In contrast to a HOLCF formalization given in [Mül98], where finite, partial, and infinite sequences are defined to model traces of I/O-Automata, our streams have been developed using only partial sequences and their infinite completions, which are more appropriate for modeling interactive systems as these are generally non-terminating. A pure HOL approach based on coinduction and corecursion is described in [Pau97].

Another approach is the formal specification language ANDL introduced in [SS95]. ANDL is a formalization of a subset of FOCUS with an untimed syntax and a fixed and an untimed semantics. Currently, ANDL does not provide an appropriate verification infrastructure or extended sophisticated definition principles, but it is HOLCF oriented. In [SM97] ANDL is used as interface for an A/C refinement calculus for FOCUS in HOLCF. In [Hin98] ANDL is extended to deal with time.

A recent work in this area is [Spi06], where a pure HOL approach to formalize timed FOCUS streams is used. By this approach (see also [DGM97, Mül98]), an infinite stream is represented by a higher-order function from natural numbers

to a set of messages. Furthermore a time-driven approach, as it will briefly be mentioned in Section 2.4, has been chosen there.

Apart from our idea of building such a logical framework, the realization of ALICE is based on a rudimentary formalization of FOCUS streams in HOLCF, developed by D. von Oheimb, F. Regensburger, and B. Gajanovic (the session HOLCF/FOCUS in Isabelle's release Isabelle2005), a concise depiction of HOLCF in [MNvOS99], as well as on the conclusions from [DGM97, SM97]. It is elaborately explained in [GR06]. Additionally, it is worth mentioning that, in the current version HOL's construct `typedef` has been used to define `astream`.

2 ALICE

The newly defined logic ALICE includes the following parts:

- HOL - the full HOL definitions.
- HOL/HOLCF - all theories from HOLCF, like *Pcpo*, *Cont*, etc. that are used on the "interface" between HOL and HOLCF (as discussed in Section 1.3).
- HOLCF - using HOLCF application/abstraction (LCF sublanguage) only.
- ALICE - basic type constructors `astream` and `tastream`, as well as recursion, pattern-matching, automata, etc.
- ALICE - lemmas provided by ALICE theories (they are generally partitioned in timed and untimed properties).

Please note that, for the development of ALICE, we use a combination of HOL and HOLCF syntax, but the user of ALICE does not need to. This is due to the fact that we internally use HOLCF to build up necessary types, operators, and proving techniques, but will encapsulate these as much as possible.

2.1 Basic Features of ALICE

To understand ALICE in more detail, we first summarize its basic features. ALICE provides:

- polymorphic type constructors `astream` and `tastream` for timed and untimed streams over arbitrary HOL types,
- sophisticated definition principles for streams and functions over streams, such as pattern-matching, recursion, and state-based definition techniques,
- incorporated domain theory (concepts of approximation and recursion),
- various proof principles for streams,
- incorporated automata constructs for state-based modeling, also supporting underspecification or non-determinism,
- extensive theories for handling timed streams, functions and properties,
- a powerful simplifier (while developing ALICE, a proper set of simplification rules has been defined carefully in such a way as to be used by ALICE automatically), and

– an extensive library of functions on streams and theorems, as well as commonly needed types (just like in any other programming language, a good infrastructure makes a language user friendly).

The following sections provide brief insights in the above listed features. For a deeper understanding we refer to [GR06].

2.2 Specifying Streams

ALICE provides a basic type constructor called *astream* for specifying untimed streams. For any Isabelle type *t*, the type *t astream* is member of the HOLCF type class *pcpo* as described in Section 1.3. The following exhaustion rule describes the basic structure of untimed streams as well as the fundamental operators for their construction:

$\bigwedge s.\ s = \varepsilon \vee (\exists h\ rs.\ s = \langle h \rangle^\frown rs)$

A stream *s* is either empty, symbolized by ε, or there is a first message *h* and a remaining stream *rs* so that pre-pending *h* to *rs* yields the stream *s*. The operator <.> builds single element streams and $.^\frown.$ defines the concatenation on streams. It is associative and continuous in its second argument and has the empty stream (ε) as a neutral element. If the first argument of concatenation is infinite, the second is irrelevant and the first is also the result of the concatenation. This effectively means that the messages of the second stream then never appear in the observation at all.

According to the above rules, ALICE also offers selection functions, named *aft* for the head and *art* for the rest of a stream, respectively. Function *atake* allows us to select the first *n* symbols from a stream. Function *adrop* acts as a counterpart of *atake* as it drops the first *n* messages from the beginning of a stream *s*. The operator *anth* yields for a number *n* and a stream *s*, the *n*-th message. Beyond that, ALICE provides many other auxiliary functions, e.g. # for the length of a stream, yielding ∞ for infinite streams, *aflatten* for the flattening of streams of streams, *aipower* for the infinite repetition, *afilter* for message filtering. In Section 2.5 we give a tabular review of operators that are available in the current version of ALICE.

Since streams are HOLCF datatypes, they carry a partial order (see also Section 1.3), which is described by the following lemma

$s1 \sqsubseteq s2 \Longrightarrow \exists t.\ s1^\frown t = s2$

The above rule characterizes the prefix ordering on streams. It is induced by a flat order on the messages, disregarding any internal structure of the messages themselves. Based on these operators, a larger number of lemmas is provided to deal with stream specifications, like case analysis, unfolding rules, composition rules, associativity, injectivity, and idempotency. Some foundational lemmas are given in Tab. 1.

Table 1. Some foundational lemmas on stream concatenation

$\varepsilon \frown s = s \frown \varepsilon = s$
$(s \frown t) \frown u = s \frown (t \frown u)$
$\#\varepsilon = 0$
$\#\langle m \rangle = 1$
$\#(s \frown t) = \#s + \#t$
$\#s = \infty \Longrightarrow s \frown t = s$

2.3 Timed Streams

Built on the untimed case, ALICE provides another type constructor called *tastream* for specifying timed streams. Structurally, both are rather similar. Again, for any Isabelle type *t*, the type *t tastream* is a member of *pcpo*. The following exhaustion rule describes the basic structure of timed streams. It shows that timed streams may still be empty, contain a message or a tick as their first element:

$\bigwedge ts.\ ts = \varepsilon \vee (\exists z.\ ts = \langle\surd\rangle \frown z) \vee (\exists m\ z.\ ts = \langle Msg\ m\rangle \frown z)$

In addition to ordinary messages, we use a special message $\surd$, called the tick, to model time progress. Each $\surd$ stands for the end of a time frame. To differentiate between the tick and ordinary messages, we use the constructor *Msg* as shown above. This operator is introduced by type constructor *addTick* that extends any type with the tick.

Please note that any timed stream of type *t tastream* is also an ordinary stream of type *(t addTick) astream*. Therefore, all machinery for *astream* types is available.

In addition, ALICE provides a timed take function. *ttake n·s* yields at most *n* time frames from the beginning of a timed stream *s*.

To allow inductive definitions, *tastream* streams may be empty. However, for specifications we restrict ourselves to observations over infinite time, which means that we will only use the subset of timed streams with infinitely many ticks. Therefore, additional machinery is necessary to deal with those. For example, the predicate *timeComplete* is provided to check whether a stream contains infinitely many time frames.

For an integration of both stream classes, operator *timeAbs* maps a timed stream into an untimed one, just keeping the messages, but removing any time information.

2.4 Stream Based Proof Principles

Having the necessary types and type classes as well as auxiliary functions and lemmas at hand, we can introduce proof principles for streams now. At first, we handle the untimed case, as the timed case can be built on that.

Proof Principles for Untimed Streams. A rather fundamental proof principle for untimed streams is the so called take-lemma for streams that gives us an inductive technique for proving equality

```
(∀n. atake n·x = atake n·y) ⟹ x = y
```

Two streams are equal if all finite prefixes of the same length of the streams are equal. More sophisticated proof principles, like pointwise comparison of two streams using the operator *anth* or the below given induction principles are built on the take-lemma. The following is an induction principle for proving a property *P* over finite (indicated by the constructor *Fin*) streams

```
⟦#x = Fin n; P ε; ⋀a s. P s ⟹ P (<a>⌢s)⟧ ⟹ P x
```

As said, when necessary, we base our proof principles directly on HOLCF but try to avoid their extensive exposure. Here is a principle that uses admissibility from HOLCF (*adm*) for predicates to span validity to infinite streams (see [Reg94])

```
⟦adm P; P ε; ⋀a s. P s ⟹ P (<a>⌢s)⟧ ⟹ P x
```

The above induction principles have also been extended to the general use of concatenation, where not only single element streams, but arbitrary streams can be concatenated.

The concept of approximation (provided by HOLCF) and induction on natural numbers can also be used to prove properties involving continuous functions over streams as discussed in Section 2.5.

Proof Principles for Timed Streams. Since timed streams can also be seen as normal untimed streams, the above given proof principles can also be used to prove properties of timed streams.

Please note that we have taken a *message driven* approach to inductively define timed streams. Messages are added individually to extend a stream. This also leads to event driven specification techniques. In the contrary, it would have been possible to model timed streams inductively as a stream *(t list) astream*, where each list denotes the finite list of messages of type *t* occurring in one time frame. This definition would lead to time-driven specification principles. It is up to further investigation to understand and integrate both approaches. As a first step in this direction, ALICE provides a timed-take-lemma for timed streams arguing that streams are equal if they are within first n time frames for each n, as given in the following.

```
(∀n. ttake n·x = ttake n·y) ⟹ x = y
```

Analogously, the following proof principle is based on time frame comparison

```
(∀n. tframe n·x = tframe n·y) ⟹ x = y
```

ALICE provides more sophisticated proof principles for timed streams, but also for special cases of timed streams, such as time-synchronous streams, containing exactly one message per time unit, and the already mentioned time complete streams, containing infinitely many time frames.

2.5 Recursive Functions on Streams

Specifying streams allows us to define observations on communication channels. However, ALICE focusses on specification of components communicating over those channels. The behavior of a component is generally modeled as function over streams and is often defined recursively or even state-based.

A recursively defined function *f* processes a prefix of its input stream *s* by producing a piece of the output stream and continues to process the remaining part of *s* recursively. All functions defined in this specification style are per construction correct behaviors for distributed components. This makes such a specification style rather helpful. Functions of this kind are defined in their simplest form as illustrated in the following (using the function *out* to process the message *x* appropriately)

```
f (<x>⌒s) = (out x)⌒(f s)
```

By construction, these functions are monotonic and continuous (lub-preserving, see below) wrt. their inputs, which allows us to define a number of proof principles on functions.

Table 2. Basic operators in ALICE

Operator	Signature
`<.>`	`'a ⇒ 'a astream`
`aft`	`'a astream ⇒ 'a`
`art`	`'a astream → 'a astream`
`atake`	`nat ⇒ 'a astream → 'a astream`
`adrop`	`nat ⇒ 'a astream → 'a astream`
`anth`	`nat ⇒ 'a astream ⇒ 'a`
`#.`	`'a astream → inat`
`.⌒.`	`'a astream ⇒ 'a astream → 'a astream`
`aipower`	`'a astream ⇒ 'a astream`
`apro1`	`('a * 'b) astream → 'a astream`
`apro2`	`('a * 'b) astream → 'b astream`
`amap`	`('a ⇒ 'b) ⇒ 'a astream → 'b astream`
`azip`	`'a astream → 'b astream → ('a * 'b) astream`
`afilter`	`'a set ⇒ 'a astream → 'a astream`
`atakew`	`('a ⇒ bool) ⇒ 'a astream → 'a astream`
`adropw`	`('a ⇒ bool) ⇒ 'a astream → 'a astream`
`aremstutter`	`'a astream → 'a astream`
`aflatten`	`'a astream astream → 'a astream`
`ascanl`	`nat ⇒ ('a ⇒ 'b ⇒ 'a) ⇒ 'a ⇒ 'b astream → 'a astream`
`aiterate`	`('a ⇒ 'a) ⇒ 'a ⇒ 'a astream`

A number of predefined auxiliary operators assist in specifying components. Due to expressiveness, we also allow to use operators that are not monotonic or continuous in some arguments, such as ⌒ in its first argument or *aipower*. In ALICE, it is also possible to define more such functions using pattern-matching and recursion. The above notions can also be found in standard literature on semantics like [Win93]. In the following we concentrate on continuous functions.

Continuous Functions - The Approximation Principle. As briefly discussed, continuous functions capture the notion of computability in interactive systems and therefore play a prominent role in stream-processing specification techniques. The behavior of a continuous function for an infinite input can be predicted by the behavior for the finite parts of the input. Thus, its behavior can be approximated. As it has been shown amongst others in [Win93], composition of continuous functions results in continuous functions. Therefore, based on a number of basic functions and equipped with appropriate definition techniques, it becomes easy to specify further functions. ALICE provides amongst others

- pattern-matching and recursion (like in functional languages),
- state-based definitions (using I/O*-automata [Rum96], see Section 2.6),
- fixpoint recursion (using HOLCF), and
- continuous function-chain construction (using HOL's *primrec* and approximation, see [GR06])

Currently, we do have at least the operators on streams depicted in Tab. 2 and Tab. 3 available. For the sake of brevity, we do not explain those further, but refer to [GR06] as well as Section 2.2 and 2.3 and furthermore assume that readers will recognize the functionality through name and signature.

Table 3. Basic operators for timed specifications

Operator	Signature
timeComplete	*'a tastream ⇒ bool*
timeSync	*'a tastream ⇒ bool*
injectTicks	*nat astream → 'a astream → 'a tastream*
timeAbs	*'a tastream → 'a astream*
ttake	*nat ⇒ 'a tastream → 'a tastream*
tframe	*nat ⇒ 'a tastream → 'a astream*
stretchTimeFrame	*nat ⇒ 'a tastream → 'a tastream*
getTime	*nat ⇒ 'a tastream ⇒ nat*

2.6 State-Based Definition Techniques

There is quite a number of variants of state machines available that allow for a state-based description. We use I/O*-automata that do have transitions with one occurring message (event) as input and a sequence of messages (events) as output (hence I/O*). They have been defined in [Rum96] together with a formal semantics based on streams and a number of refinement techniques. In contrast to I/O automata [LT89], they couple incoming event and reaction and need no intermediate states.

As they are perfectly suited for a state-based description of component behavior, we provide assistance for the definition of an I/O*-automaton *A* in ALICE by modeling the abstract syntax as a 5-tuple in form of

```
A = (stateSet A, inCharSet A, outCharSet A, delta A, initSet A)
```

Automata of this structure can be defined using the type constructor *ioa*. I/O*-automata consist of types for its states, input and output messages. *delta* denotes the transition relation of an automaton. It consists of tuples of source state, input message, destination state and a sequence of output messages. The 5th element *initSet* describes start states and possible initial output (that is not a reaction to any incoming message).

As an illustration, we define[1] an I/O*-automaton representing a component dealing with auctions in the American style, where bidders spontaneously and repeatedly spend money and after a certain (previously unknown) timeout the last spender gets the auctioned artifact. The auction component is initialized with an arbitrary but a non-zero timeout. It counts down using the ticks and stores the last bidder as he will be the winner.

```
amiauction :: "((nat * Bid * IAP), Bid addTick, BidUclosed addTick) ioa"
amiauction_def:
  "amiauction ≡
    (UNIV, UNIV, UNIV,
     {t.∃k b m x.

         (* handle time and accept the last bid
              as soon as the time limit is reached *)
         t = ((k+1,b,x), √, (k,b,x), <√>) ∧ k > 0 ∨
         t = ((0,b,x), √, (0,b,x), <√>⌢<Msg closed>) ∨
         t = ((1,b,I), √, (0,b,I), <√>⌢<Msg closed>) ∨
         t = ((1,b,A), √, (0,b,A), <√>⌢<Msg (accept b)>⌢<Msg closed>) ∨

         (* store the new bid m if necessary *)
         t = ((k+1,b,x), Msg m, (k+1,m,A), ε) ∨ t = ((0,b,x), Msg m, (0,b,x), ε)},

     {((ϵ s. fst s > 0 ∧ snd (snd s) = I), ε)})"
```

The above automaton is well-defined, deterministic and complete. By applying the operator *ioafp*, we map this automaton into a function that is continuous by construction. The recursive definition of a stream-processing function is now embedded in the *ioafp* operator, leaving a non-recursive but explicit definition of the actual behavior in an event based style.

In fact, a number of proof principles are established on these state machines that do not need inductive proof anymore, but just need to compare transitions and states. More precisely, the behaviors can then be compared by establishing a (bi-)simulation relation between the automata.

A non-deterministic I/O*-automaton is defined in an analogous form and not mapped to a single but a set of stream-processing functions. This is especially suitable to deal with underspecification.

As said, ALICE is still in development. Although we have initial results on this kind of specification style, we will further elaborate ALICE to comfortably deal with I/O*-automata of this kind in the future.

[1] Due to lack of space, we skip HOL's keyword *constdefs* in front of a definition but symbolize it by indentation. We also do not introduce the necessary type declarations, which is actually straightforward for the specifications used here.

3 Alternating Bit Protocol - An Example

Based on the theory introduced so far, we show the usefulness of ALICE by developing a small, yet not trivial and well known example.

The Alternating Bit Protocol (ABP) is a raw transmission protocol for data over an unreliable medium. Goal of the ABP is to transmit data over a medium that looses some messages, but does not create, modify, rearrange or replicate them. The key idea is that the sender adds an identifier to each message that is being sent back as acknowledgement by the receiver. If the acknowledgement does not arrive, the sender sends the same message again. When only one single message is in transmission, the identifier can boil down to a single bit with alternating value – hence the name of the protocol.

The ABP specification involves a number of typical issues, such as underspecification, unbounded non-determinism and fairness. Fig. 2 illustrates the overall structure of the ABP. A detailed explanation of a similar specification can be found in [BS01].

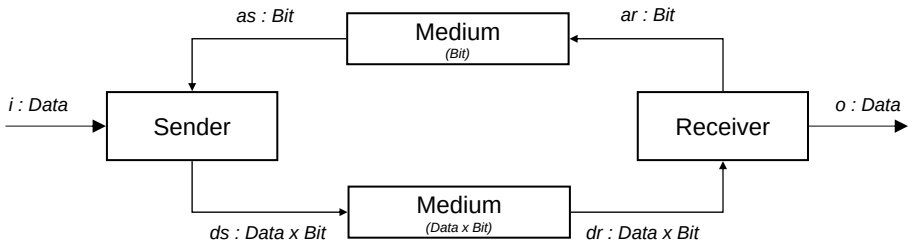

Fig. 2. The architecture of the Alternating Bit Protocol (ABP)

3.1 The ABP Medium

Please note that the medium is modeled after the existing, real world, while sender and receiver need to be specified and later implemented in such a way that they can safely deal with the given medium. So, we first specify the behavior of the medium as described above.

```
Med :: "'t astream ⇒ 't astream ⇒ bool"
Med_def:
  "Med x y ≡
     ∃p. #(afilter {True}·p) = ∞ ∧
         y = apro1·(afilter {a. ∃b. a = (b, True)}·(azip·x·p))"
```

Through the use of an internal oracle stream p, we can describe that a medium does eventually transmit a message if we retry long enough. The fairness, as described below, is deduced from the above specification as follows.

$$[\![\#x = \infty;\ Med\ x\ y]\!] \Longrightarrow \#y = \infty$$

The lemma is proven easily using the following auxiliary lemma, since the lengths of the first and the second pointwise projection (*apro1* and *apro2* respectively) of a stream consisting of ordered pairs are equal.

```
∀x. #x = ∞ ⟶ apro2·(afilter {a. ∃b. a = (b, z)}·(azip·x·p)) = afilter {z}·p
```

The above auxiliary lemma is again proven by induction on the free stream variable *p* using an appropriate proof principle from Section 2.4.

3.2 The Sender

Now, relative to a given medium, we have to define a sender and a receiver that establish the desired behavior: safe transmission of messages. The sender receives data from outside and transmits them together with the alternating bit. We give a specification in a functional style:

```
Snd :: "Data astream ⇒ Bit astream ⇒ (Data * Bit) astream ⇒ bool"
Snd_def:
  "Snd i as ds ≡
     let
        fas = aremstutter·as;
        fb = apro2·(aremstutter·ds);
        fds = apro1·(aremstutter·ds)
     in
        fds ⊑ i ∧
        fas ⊑ fb ∧
        aremstutter·fb = fb ∧
        #fds = imin #i (iSuc (#fas)) ∧
        (#fas < #i ⟶ #ds = ∞)"
```

We explicitly define the channel observations for the sender in Fig. 2. The conjuncts in the *in* part of the definition constrain the sender in the order of their appearance, using the abbreviations from the *let* part, as follows

1. Abstracting from consecutive repetitions of a message via *aremstutter*, we see that the sender is sending the input messages in the order they arrive.
2. The sender also knows which acknowledgement bit it is waiting for, nevertheless, it is underspecified which acknowledgment bit is sent initially.
3. Each new element from the data input channel is assigned a bit different from the bit previously assigned.
4. When an acknowledgment is received, the next data element will eventually be transmitted, given that there are more data elements to transmit.
5. If a data element is never acknowledged then the sender never stops transmitting this data element.

3.3 The Receiver

The receiver sends each acknowledgment bit back to the sender via the acknowledgment medium and the received data messages to the data output channel removing consecutive repetitions, respectively.

```
Rcv :: "(Data * Bit) astream ⇒ Bit astream ⇒ Data astream ⇒ bool"
Rcv_def: "Rcv dr ar o ≡ ar = apro2·dr ∧ o = apro1·(aremstutter·dr)"
```

3.4 The Composed System

The overall system is composed as defined by the architecture in Fig. 2. This composition is straightforwardly to formulate in ALICE:

```
ABP :: "Data astream ⇒ Data astream ⇒ bool"
ABP_def:
  "ABP i o ≡ ∃as ds dr ar. Snd i as ds ∧ Med ds dr ∧ Rcv dr ar o ∧ Med ar as"
```

This formalization of the ABP uses a relational approach similar to the specification in [BS01]. However, formalizations as sets of functions or in a state-based manner are possible as well. Using a more elaborate version of ALICE, we will be able to define a state-based version of sender and receiver (similar to [GGR06]), which is on the one hand more oriented towards implementation and on the other hand might be more useful for inductive proof on the behaviors. Most important however, we will be able to prove that this relational and the state-based specifications will coincide.

For this case study, we remain in the relational style and specify the expected property of the overall system (without actually presenting the proof):

```
ABP i o ⟹ o = i
```

Please note that, at this stage of the development of ABP, there are neither realizability nor sophisticated timing constraints considered in the above formalization. Due to relational semantics, additional refinement steps are then needed to reduce the underspecification towards an implementation oriented or timing-aware style, since there are infinite streams fulfilling the specification that are not valid protocol histories. These, however, would not occur, when using sets of stream-processing functions as they impose continuity on the overall behavior.

4 Discussion

In this paper we have introduced ALICE, an advanced logic for formal specification and verification of communication in distributed systems. ALICE is embedded in the higher order logic HOL, which itself is formalized using the Isabelle generic theorem prover.

Our approach is based on using HOLCF to deal with partiality, infinity, recursion, and continuity. We provide techniques to use ALICE directly from HOL, thus preventing the user to actually deal with HOLCF specialities.

ALICE is currently under development. So not all concepts and theories presented here are already completely mature. Further investigations will also deal with the question of expressiveness, applicability and interoperability. Beyond the ABP, we already have some experience with other formalizations that show that the overhead of formalizing a specification in ALICE as apposed to a mere paper definition is not too bad. However, it also shows where to improve comfort.

References

[BCR06] M. Broy, M. V. Cengarle, and B. Rumpe. Semantics of UML. Towards a System Model for UML. The Structural Data Model. Technical Report TUM-I0612, Munich University of Technology, 2006.

[BCR07a] M. Broy, M. V. Cengarle, and B. Rumpe. Semantics of UML, Towards a System Model for UML, Part 2: The Control Model. Technical Report TUM-I0710, Munich University of Technology, 2007.

[BCR07b] M. Broy, M. V. Cengarle, and B. Rumpe. Semantics of UML, Towards a System Model for UML, Part 3: The State Machine Model. Technical Report TUM-I0711, Munich University of Technology, 2007.

[BS01] M. Broy and K. Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement.* Springer Verlag Heidelberg, 2001.

[DGM97] M. Devillers, D. Griffioen, and O. Müller. Possibly Infinite Sequences in Theorem Provers: A Comparative Study. In E. L. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, pages 89–104, Murray Hill, New Jersey, 1997. LNCS 1275, Springer Verlag.

[GGR06] B. Gajanovic, H. Grönniger, and B. Rumpe. Model Driven Testing of Time Sensitive Distributed Systems. In J-P. Babau, J. Champeau, and S. Gerard, editors, *Model Driven Engineering for Distributed Real-Time Embedded Systems: From MDD Concepts to Experiments and Illustrations*, pages 131–148. ISTE Ltd, 2006.

[GR06] B. Gajanovic and B. Rumpe. Isabelle/HOL-Umsetzung strombasierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen. Technical Report Informatik-Bericht 2006-03, Braunschweig University of Technology, 2006.

[Hin98] U. Hinkel. *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL.* Dissertation, Munich University of Technology, 1998.

[LT89] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[MNvOS99] O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.

[Mül98] O. Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory.* Dissertation, Munich University of Technology, 1998.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* LNCS 2283, Springer Verlag, 2002.

[Pau97] L. C. Paulson. Mechanizing Coinduction and Corecursion in Higher-Order Logic. *Journal of Logic and Computation*, 7(2):175–204, 1997.

[Pau03a] L. C. Paulson. *Introduction to Isabelle.* Computer Laboratory, University of Cambridge, 2003.

[Pau03b] L. C. Paulson. *The Isabelle Reference Manual. With Contributions by Tobias Nipkow and Markus Wenzel.* Computer Laboratory, University of Cambridge, 2003.

[Reg94] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF.* Dissertation, Munich University of Technology, 1994.

[Rum96] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Herbert Utz Verlag Wissenschaft, 1996.

[SM97] R. Sandner and O. Müller. Theorem Prover Support for the Refinement of Stream Processing Functions. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97).* Springer, 1997.

[Spi06] M. Spichkova. FlexRay: Verification of the FOCUS Specification in Isabelle/HOL. A Case Study. Technical Report TUM-I0602, Munich University of Technology, 2006.

[SS95] B. Schätz and K. Spies. Formale Syntax zur logischen Kernsprache der FOCUS-Entwicklungsmethodik. Technical Report TUM-I9529, Munich University of Technology, 1995.

[Ste97] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7):491–541, 1997.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing. The MIT Press, Cambridge, Massachusetts, 1993.