University of Oslo
Department of Informatics

A PVS Proof Environment for OUN

Einar B. Johnsen
and Olaf Owe

Research report 295
ISBN 82-7368-245-5

June 2001

A PVS Proof Environment for OUN

Einar Broch Johnsen and Olaf Owe

Department of informatics
University of Oslo, Norway
email: {einarj, olaf}@ifi.uio.no

Abstract

OUN is an object-oriented design language with support for an aspect-wise specification style, constructed for the formal specification of open distributed systems. In this paper, we propose a proof environment for OUN using the PVS theorem prover. It is shown how properties of the design language as well as properties of OUN designs can be formalized and given proofs within the proof environment.

1 Introduction

The Oslo University Notation (OUN) is developed as part of the ADAPT-FT research project and is designed for the specification of open, distributed systems. A goal for the ADAPT-FT project is to investigate the development process for such systems by means of formal methods, integrating the Unified Modeling Language (UML [2]) and OUN in a common platform [20], [21], [22]. In this setting, OUN is used as a means to formalize graphical specifications in UML. Object-orientation is advocated as a natural framework for the development of open distributed systems [8] and OUN is object-oriented, specifying objects by means of communication traces. Furthermore, OUN is aspect-oriented, thus offering specification by separation of concerns, with multiple viewpoints to each object. In this paper, we develop a proof environment for OUN specifications in the Prototype Verification System (PVS) and show how machine-assisted proofs can be obtained for properties of OUN specifications.

OUN is a high-level object-oriented design language developed to support the development of open distributed systems. The language is designed in a restricted way so that reasoning is manageable, particularly, reasoning control is based on static typing and proofs, and generation of verification conditions is based on static analysis of pieces of programs or specification texts [14]. At the highest level of abstraction, an OUN design consists of interfaces and contracts. An object may support a number of interfaces and this number may change dynamically. The relationship between several given objects can be described by means of contracts. The semantics of the language is based on communication traces. Requirement specifications of interfaces consist of assumptions about the behavior of the environment and invariants about the behavior of self, by means of first-order predicates on traces. In OUN, objects are always considered through their given roles, i.e. by means of the interfaces they offer.

We consider objects that run in parallel and communicate by method calls. The state of an object at a given execution point is given by its observed behavior in the form of a trace, i.e. a finite sequence of observable communication events, reflecting calls to the methods of the object by other objects and calls to methods of other objects by the current one. At any given time in the life of an object, there has only been a finite number of such method calls, so the trace of communication events reflecting the activity of the object up to this point in time is also finite.

The set of traces that reflect possible runs of the object, give us an external, or observational description of the object. For any trace in such a set, all prefixes of the trace represent possible traces of the object at points in time prior to the one of the considered trace. Consequently, trace sets are prefix-closed. Internal activity is not captured directly in this model of observable communication between objects, but such internal activity may manifest itself at the level of communication traces as non-deterministic choice, i.e. a trace may have several different extensions in the trace set (due to internal activity).

A specification of an object consists of an alphabet of communication events and a prefix-closed set of traces over this alphabet. When an object is specified at any level of abstraction, some details concerning its behavior are ignored, in order to focus the specification on a relevant aspect of the behavior of the object. In our case, such details may be a subset of the methods offered by the object or a subset of its communication with other objects in its environment. Hence, the alphabet of the specification of an object is a subset of the communication events of that object. We may derive a specification of an object from an OUN interface declaration supported by that object. By generalization, specifications can describe the behavior of several objects. In particular, the composition of two specifications is a new specification.

There may be several specifications of the same object, describing the communication traces built over different subsets of the alphabet of that object. These partial specifications correspond to different roles of the object. We propose a refinement relation that allows alphabet expansion, i.e. the addition of new events through refinement. Thus, various (partial) specifications of an object may have a common refinement, although their alphabets differ. This refinement relation is introduced in the OUN specification language [5], [15]. In this paper, the relation is expanded to allow the introduction of new objects in a specification through refinement.

The Prototype Verification System (PVS) is a specification and verification environment that provides mechanized support for formal verification in a specification language based on higher-order logic. The verification system is implemented in Common Lisp and available from the SRI International Computer Science Laboratory. The language has a rich type system including the notion of predicate subtypes and dependant types. These features make the language very expressive, but type checking becomes undecidable. There are type constructors for functions, tuples, records and abstract data types. PVS specifications are organized into theories and modularity and reuse are supported by means of parameterized theories. The language was designed to describe computer systems, but concentrates on abstract descriptions. PVS has a powerful verification tool which uses decision procedures for simplifying and discharging proofs and provides many proof techniques such as induction, rewrite, backward proofs, forward proofs and proof by cases for interactive user intervention.

Technicalities of PVS are not treated in this paper, except when these are crucial to our choice of representation. A tutorial presentation of PVS is given in [3]. For a thorough description, see the reference manuals: [17], [19], and [18]. Other trace models have been encoded in PVS, see for example [6] for an encoding of Communicating Sequential Processes (CSP) [7].

In this paper, a PVS proof environment for partial object specifications based on finite communication traces is proposed. The proof environment is then extended with constructs particular to the OUN specification language, and we show by examples how OUN specifications can be translated in a fairly straightforward manner into PVS theories and how properties of these specifications are given machine-assisted proofs, using the PVS decision procedures. Thus, the proof environment gives us support for formal reasoning about OUN specifications.

The paper is structured as follows. We present a theory of finite sequences (Section 2) and build a theory for partial specifications (Section 3). Then we introduce the OUN syntax (Section 4) and show how it can be represented in the theory for partial specifications (Section 5). Examples of specifications and properties we can prove about them are given (Section 7). The complete PVS theories are included in appendixes. They may also be downloaded from the net [10].

2 Finite Sequences in PVS

In this section, we develop a theory for finite sequences in PVS. The sequences are parameterized over some type T, which will be the type of communication events needed in later sections. Finite sequences over T are defined as an abstract data-type in PVS.

```
seq[T: TYPE]: DATATYPE
 BEGIN
  empty: empty?
  addr(lr: seq, rt: T): nonempty?
 END seq
```

Note that this definition gives us selector-functions `lr` and `rt` as well as identifiers `empty?` and `nonempty?` and the constructors `empty` and `addr`. Thus, we can state and prove properties such as

```
AddrEta: LEMMA FORALL (t: (nonempty?)): addr(lr(t), rt(t)) = t
```

This and some other similar properties of abstract datatypes are in fact automatically constructed by the PVS theorem prover [16]. For our datatype, these are kept in a separate theory `seq_adt`. We proceed by defining further functions on the finite sequences by induction over the constructors of the abstract datatype (i.e. terminating generator inductions [4]). These functions are assembled in a theory `sequence`, parameterized over the element type T, and using the abstract datatype `Seq[T]`. The complete theory is included in appendix A.1. The singleton sequence of an element x of type T is defined as

```
  singleton_seq(x: T): seq[T] = addr(empty, x)
```

A length function on sequences is defined by induction as follows:

```
length(s: seq[T]): RECURSIVE nat =
  CASES s OF empty: 0,
             addr(x, y): length(x) + 1
    ENDCASES
   MEASURE reduce_nat(0, (LAMBDA (n: nat, x: T): n + 1))
```

Notice the definition by cases construct and the measure, which generates TCC's (type correctness conditions, see [18]) to guarantee that the definition is well-founded. The length of a sequence will be used as a measure in further definitions. Define concatenation as

```
conc(s, t: seq[T]): RECURSIVE seq[T] =
  CASES t OF empty: s,
             addr(t1, x): addr(conc(s, t1), x)
    ENDCASES
   MEASURE length(t)
```

With finite sequences, it makes sense to extend a sequence at either end. The operation `addr` was introduced in the definition of the abstract datatype to add elemtent to a sequence; we now define its complementary operation `addl`:

```
addl(x: T, s: seq[T]): RECURSIVE seq[T] =
  CASES s OF empty: addr(empty, x),
             addr(s1, y): addr(addl(x, s1), y)
    ENDCASES
   MEASURE length(s)
```

The new constructor is accompanied by new selector functions `lt` and `rr`. These are defined by

```
lt(s: (nonempty?)): RECURSIVE T =
  IF empty?(lr(s)) THEN rt(s) ELSE lt(lr(s)) ENDIF
   MEASURE length(s)

rr(s: (nonempty?)): RECURSIVE seq[T] =
  IF empty?(lr(s)) THEN empty ELSE addr(rr(lr(s)), rt(s)) ENDIF
   MEASURE length(s)
```

In the sequel, we will often need filtering operations on sequences over type T in order to restrict the sequences to a subtype of T. This is either done by including only the elements of a given set or by removing elements of the given set.

```
restrict(s: seq[T], a: set[T]): RECURSIVE seq[T] =
  CASES s
    OF empty: empty,
       addr(t, x):
         IF member(x, a) THEN addr(restrict(t, a), x)
                         ELSE restrict(t, a)
         ENDIF
    ENDCASES
   MEASURE length(s)
```

```
hide(s: seq[T], a: set[T]): RECURSIVE seq[T] =
  CASES s
    OF empty: empty,
       addr(t, x):
         IF member(x, a) THEN hide(t, a)
                         ELSE addr(hide(t, a), x)
         ENDIF
    ENDCASES
   MEASURE length(s)
```

Finally, we introduce a prefix ordering on sequences and the notion of a subsequence:

```
ord?(s, t: seq[T]): bool = EXISTS (u: seq[T]): conc(s, u) = t

sub?(s, t: seq[T]): bool =
  EXISTS (u, v: seq[T]): conc(u, conc(s, v)) = t
```

Using these definitions, we will state and prove some elementary properties about the defined operations on finite sequences. Due to an extensionality axiom generated by PVS for the abstract datatype, we may prove an equality lemma:

```
EQ: LEMMA
  FORALL (s, t: seq[T]):
    s = t IFF
     (s = empty AND t = empty) OR
      ((NOT (s = empty) AND NOT (t = empty)) AND
        ((rt(s) = rt(t)) AND (lr(s) = lr(t))))
```

Furthermore, other properties of equality and distribution for the defined functions can be proved, for instance

```
EqualLengthLemma: LEMMA
  FORALL (s, t: seq[T]): s = t IMPLIES length(s) = length(t)

ConcAssocLemma: LEMMA
  FORALL (s, t, u: seq[T]):
    conc(s, conc(t, u)) = conc(conc(s, t), u)

ConcAddlAddrLemma: LEMMA
  FORALL (s, t: seq[T], x: T):
    conc(s, addl(x, t)) = conc(addr(s, x), t)
```

These and similar properties are assembled in a theory `seq_lemmas`, which is included in appendix A.2.

Of particular interest to our exposition, and especially to the development of Section 3, are properties of the restriction and hiding operators on finite sequences.

```
RestrictHideDist: LEMMA
  FORALL (t: seq[T], s1, s2: set[T]):
    restrict(hide(t, s1), s2) = hide(restrict(t, s2), s1)
```

```
RestrictHideDifference: LEMMA
  FORALL (t: seq[T], s1, s2: set[T]):
    restrict(hide(t, s1), s2) = restrict(t, difference(s2, s1))

SubsetRestrict: LEMMA
  FORALL (t: seq[T], s1, s2: set[T]):
    subset?(s1, s2)
      => restrict(restrict(t, s2), s1) = restrict(t, s1)

SubsetRestrict2: LEMMA
  FORALL (t: seq[T], s1, s2: set[T]):
    (subset?(s1, s2) AND restrict(t, s1) = t)
       => restrict(t, s2) = t

RestrictHideInterchange: LEMMA
  FORALL (t: seq[T], s1, s2: set[T]):
    empty?(intersection(s1, s2))
      AND restrict(t, union(s1, s2)) = t
        => restrict(t, s1) = hide(t, s2)
```

Notice for lemmas `SubsetRestrict2` and `RestrictHideInterchange` that the sequences are actually over an (implicit) subtype of `T`, i.e. they are over `s1` and `union(s1, s2)`, respectively. We have adapted this implicit subtyping style to avoid TCC's regarding the commutativity of the union of two subtypes of T in the next section. These and related lemmas concerning restriction and hiding are assembled in a theory `restriction`, see appendix A.3.

3 Encoding Partial Specifications

Specifications in OUN consist of behavioral constraints on communication patterns between objects. As aspects of objects are specified, there may be several specifications that describe the same object (identifiers). In this section, we propose PVS constructs for the basic notions of a specification, refinement, and composition of specifications in OUN. Then we present sufficient restrictions to obtain a compositional refinement property for OUN specifications, following earlier work [9]. This property is here given a proof in the PVS proof environment. In order to do this, we introduce a PVS type `Event` for communication events with explicit object identifiers. As the theory of sequences of the previous section is parameterized, we will import sequences of type `Event` and define the OUN operations of composition and refinement on specifications that include sets of such sequences.

A communication event in our setting is a structured event reflecting a remote method call from one object to another. In PVS, we define objects and methods as types.

```
Object: TYPE

Method: TYPE
```

Events are defined as structured triples[1] that consist of two object identifiers

[1] For full expressivity in OUN, we extend these triples in Section 5.1.

and a method name. For structured triples, we use the PVS record type.

```
Event: TYPE = [# caller: Object, to: Object, name: Method #]
```

Note that the fields of the triple are given the names `caller`, `to`, and `name`, respectively, which can be used to access the different fields of the triple. (While the order of tuples is important and access is by projection functions, the order of record types in PVS is unimportant and the accessor functions have explicit names.) If `e` is an element of `Event`, then `caller(e)`, `to(e)` and `name(e)` access the sender, receiver, and method name of `e`, respectively.

3.1 General Specifications

The specifications we consider will describe safety properties, which means that it suffices to consider prefix-closed sets of sequences [1]. Prefix-closure is defined in PVS by the following predicate:

```
prefixclosed?: [set[seq[Event]] -> bool] =
    (LAMBDA (s: set[seq[Event]]):
       FORALL (t: seq[Event]):
         member(empty, s) AND
          ((nonempty?(t) AND member(t, s)) => member(lr(t), s)))
```

A safety specification is a triple which consists of a set of objects, a set of events, and a prefix-closed set of traces over these events. In PVS, consider a record type `Triple`, with fields `alpha:set[Event]`, `traces:(prefixclosed?)`, and `obj:set[Objects]`. By the PVS notation `(prefixclosed?)`, we define the subtype of `set[seq[Event]]` which satisfies the `prefixclosed?` predicate. This is a standard way of constructing subtypes in PVS, where types are modeled as sets. Now, a specification is a subtype of `Triple` such that some objects are specified and the *internal activity* of these objects is not observable. The `Specification` subtype is defined as follows:

```
Specification: TYPE =
  {x: Triple |
     nonempty?(obj(x)) AND
      (FORALL (m: Event):
         member(m, alpha(x)) =>
          (member(caller(m), obj(x)) XOR member(to(m), obj(x))))}
```

Remark that unlike the field `traces` of `Triple`, we do not restrict `alpha` to the subset of Events `e` such that `caller(e)`$\neq$`to(e)`. Rather, we have a more general notion of internal activity for specifications that consist of any number of objects, by using the exclusive or construct `XOR` of PVS on membership in the object set `obj(x)` of the triple `x`. A specification Γ is said to be *singleton* if `obj`(Γ) is a singleton set.

3.2 Refinement of Specifications

OUN offers a notion of refinement which is particularly suitable for aspect-oriented specifications, as the relation allows multiple inheritance of specifications through projection on alphabets. Thus, we can join several aspects of an

object into a common viewpoint through refinement. A discussion of refinement for aspect-oriented specifications using this formalism can be found in [9]. The refinement relation for aspect-oriented specifications is formalized as follows in PVS:

```
refines?(S1, S2: Specification): bool =
    subset?(alpha(S2), alpha(S1)) AND
     subset?(obj(S2), obj(S1)) AND
      (FORALL (t: seq[Event]):
         member(t, traces(S1)) =>
          member(restrict(t, alpha(S2)), traces(S2)))
```

3.3 Composition of Specifications

We will understand by composition an encapsulation of the specified objects, such that the internal activity of a composition is not reflected in the communication sequences. The internal events of a set of objects are given by a function `int`, defined as

```
int(Obj_set: set[Object]): set[Event] =
    {x: Event | member(to(x), Obj_set)
                   AND member(caller(x), Obj_set)}
```

Composition of specifications is then defined by hiding internal communication and by projection on the traces of the components.

```
comp(S1, S2: Specification): Specification =
    LET alph =
          {x: Event |
             member(x,
                    difference(union(alpha(S1), alpha(S2)),
                               int(union(obj(S1), obj(S2)))))}
      IN
      (# alpha := alph,
         traces
           := {t: seq[Event] |
                 restrict(t, alph) = t AND
                  (EXISTS (t1: seq[Event]):
                     restrict(t1, union(alpha(S1), alpha(S2))) = t1 AND
                      t = hide(t1, int(union(obj(S1), obj(S2)))) AND
                       member(restrict(t1, alpha(S1)), traces(S1)) AND
                        member(restrict(t1, alpha(S2)), traces(S2)))},
         obj := union(obj(S1), obj(S2)) #)
```

Due to the aspect-oriented specification style of OUN, a communication event may be part of several specifications, either as internal or in the alphabet of the specifications. This may give rise to certain difficulties that are avoided by restricting composition to *composable* objects. Define a predicate `composable?` on specifications:

```
composable?(s1, s2: Specification): bool =
    empty?(intersection(alpha(s1), int(obj(s2)))) AND
     empty?(intersection(alpha(s2), int(obj(s1))))
```

We can now state and prove commutativity and associativity for *composable* specifications in PVS:

```
CompCommute: LEMMA
  FORALL (x, y: Specification): comp(x, y) = comp(y, x)

CompAssoc: LEMMA
  FORALL (x, y, z: Specification):
    (composable?(comp(x, y), z) AND composable?(x, comp(y, z)))
      => comp(comp(x, y), z) = comp(x, comp(y, z))
```

In a refinement step, OUN allows the introduction of new object identifiers in a specification. When we refine a specification that is part of a composition, the new objects may interfere with the behavior of the other specification. In order to avoid such interference, a *properness* criterion is introduced [9]. In PVS, this is defined as a predicate:

```
proper?(s1, s2, s3: Specification): bool =
    empty?(intersection({x: Event |
                           (member(to(x),obj(s1))
                             OR member(caller(x), obj(s1)))
                            AND NOT (member(to(x), obj(s2))) AND
                             NOT (member(caller(x), obj(s2)))},
                        alpha(s3)))
```

In PVS, we state and prove the following compositional refinement property:

```
SpecificationCompRef: LEMMA
  FORALL (x, y, z: Specification):
    (refines?(x, y) AND proper?(x, y, z) AND composable?(x, z))
      => refines?(comp(x, z), comp(y, z))
```

4 OUN Syntax

In this section, we briefly present the syntax of the OUN specification language, partly following [13]. For a motivation of design choices and a further discussion of the language, the reader is referred to [13], [14], and [15]. The specification language allows us to specify interfaces and contracts. Classes are not part of the specification language, but are part of the OUN design language [13]. The language is strongly typed, and objects are typed by interfaces that correspond to different roles. An object may have several roles and different objects may have the same role, so interfaces are declared independently of their objects and objects that are typed by the same interfaces can be used in the same places. We say that an object offers an interface to its environment, or that an objects supports an interface. An object supporting an interface F may be used anywhere an object supporting a *super-interface* of F can be used. This substitutability requires a rigorous definition of interface inheritance. We will refer to the collection of interfaces (offered by objects) and contracts that specify a computing system as a design.

4.1 Asynchronous Communication

In distributed systems, we find it natural to think of communication between objects as asynchronous. In the traces of the communication model, each remote method call is represented by two distinct events, which we refer to as the initiation and the completion of the call, respectively. Constraints on the sequences of communication events can be used to capture synchronous communication, as discussed later.

Events contain information about input and output values as well as about the kind[2] of event we are dealing with, either an initiation or a completion event, in addition to the identities of the sending and receiving objects and the name of the method. In the syntax, an operation is declared (in an interface) as

$$\textbf{opr}\ m(\textbf{in}\ p_1{:}T_1, \ldots, p_i{:}T_i; \textbf{out}\ p_{i+1}{:}T_{i+1}, \ldots, p_j{:}T_j).$$

Say that an object o_1 offers this method to its environment, and that the method is invoked by another object o_2. This remote call is first reflected in the traces by an initiation event, represented as

$$o_2 \rightarrow o_1.m(p_1, \ldots, p_i).$$

If the call is answered by o_1, i.e. o_1 computes an answer to the method invocation and this answer is transmitted to o_2, this transmission is reflected by a completion event in the traces, denoted

$$o_2 \leftarrow o_1.m(p_1, \ldots, p_i; p_j, \ldots, p_n).$$

The semicolon separates input and output values. For methods without explicit output values, there is simply no semicolon nor values after it. Considering communication events that reflect a remote method call, we will refer to o_2 as the caller and to o_1 as the receiver of both events $o_2 \rightarrow o_1.m(\ldots)$ and $o_2 \leftarrow o_1.m(\ldots)$. Denote by $\textbf{caller}(o)$ the set of events corresponding to remote calls that are called by an object o and by $\textbf{receiver}(o)$ the corresponding events received by o. As the communication events model remote calls, we do not observe internal activity in the objects directly and hence:

$$\textbf{caller}(o) \cap \textbf{receiver}(o) = \emptyset.$$

Global traces are communication histories for an entire system or sub-system, whereas local traces describe the histories restricted to a subset of the communication events. Asynchronicity for the communication means that the calling object need not wait for the completion of a call. This is directly reflected in the (local) traces by the fact that other communication events may occur between the initiation and completion events of a particular call. Synchronous communication can be captured in the communication model by disallowing other events between the initiation and completion events reflecting a particular method call. In OUN, synchronous communication is often represented in the traces by an event

$$o_2 \leftrightarrow o_1.m(p_1, \ldots, p_i; p_j, \ldots, p_n),$$

[2] In other papers, we consider different kinds of completion events, see [12] and [11].

which is shorthand for the subsequence

$$\langle o_2 \rightarrow o_1.m(p_1, \ldots, p_i),\ o_2 \leftarrow o_1.m(p_1, \ldots, p_i; p_j, \ldots, p_n) \rangle.$$

This is not, however, the same as handshake communication, where the answer to a call must be ready before the call is made, which is the case in for example CSP [7]. In OUN, although a call appears as synchronous from the point of view of one of the objects involved, it may be asynchronous for the other. For full synchrony, the call must be perceived as such by both parts.

4.2 Interfaces

An interface contains syntactic definitions of operations and semantic requirement specifications. The semantic requirement has the form of an assumption-guarantee specification. The guarantee is an invariant and the assumption is a requirement on the behavior of the objects in the communication environment. As customary in the assumption-guarantee paradigm, the invariant is guaranteed to hold only when the assumption is respected by the environment. In the interfaces, auxiliary functions can be introduced for specification purposes.

As objects are typed by interfaces, only the semantic behavior of objects is considered in the specification language and the specifications do not depend on class definitions. The semantic requirements of an interface relies on the available communication history (the local trace) of an object offering the interface up to the present point in time, i.e. predicates on the finite traces. An interface declaration has the following general form:

interface F [⟨type parameters⟩] (⟨context parameters⟩)
  inherits $F_1, F_2, \ldots, F_m$
begin
  with G
    opr $m_1(\ldots)$
      ⋮
    opr $m_n(\ldots)$
  asm <formula on local trace restricted to one calling object>
  inv <formula on local trace>
where <auxiliary function definitions>
end

where $F, F_1, \ldots, F_m$, and G are interfaces. We will now explain the syntax in terms of alphabets and trace sets. Inheritance is momentarily ignored, we return to issues concerning inheritance at the end of this section. Remark only that the **inherits** clause as well as the **with** clause indicate an ordering of the interface declarations.

The type parameters (between square brackets) can be data types or interfaces. The context parameters (between ordinary brackets) are values (typed by data types) and objects parameters (typed by interfaces). These context parameters describe the minimal environment that any object supporting the interface must know about at the point of creation. Type parameterized interfaces are understood as interface templates and as such do not describe aspects of objects. We will now consider interfaces where both type and context parameters have been given actual values; these "fixed" interfaces describe viewpoints to objects.

Sometimes, we may wish that only objects of a particular kind are allowed to invoke the methods we specify in an interface. For this purpose, the **with** clause is used to restrict the communication environment of an object (considered through the interface) to objects offering a particular interface. Thus, the behavior of the current interface can involve calling methods of the calling object, as an interface of this object is known. In the case where no such knowledge is required and access should be open to all objects in the environment, the **with** clause may be omitted, or we can use the special keyword **any**, which denotes any interface, i.e. an "empty" interface without declared operations and with assumption and invariant predicates **true**. The interface declared in the **with** clause is referred to as a cointerface.

In OUN, we denote by $o{:}\,F$ an object o which offers an interface F (as above, but for given parameters). For $o{:}\,F$, we can derive an alphabet $\alpha(o{:}\,F)$ of communication events from the declaration of the interface F. To each method, we associate initiation and a completion events, varying over possible input and output values and possible calling objects. If $o'{:}\,F'$ is included as a value for a context parameter to F, we include the relevant part of its alphabet in $\alpha(o{:}\,F)$, i.e. $\alpha(o'{:}\,F') \cap \mathbf{caller}(o) \subseteq \alpha(o{:}\,F)$. Furthermore, the alphabets of objects included as parameters to F are included in $\alpha(o{:}\,F)$. Also, if F has a cointerface G, we include the events in the alphabet $\alpha(o'{:}\,G)$ that may be called by o, i.e. $[\bigcup_{o'\in \mathtt{Object}} \alpha(o'{:}\,G) \cap \mathbf{caller}(o)] \subseteq \alpha(o{:}\,F)$. Likewise, if G' is the type of an object parameter transmitted as input to one of the methods declared in F, events of G' that may be called by o are included in the alphabet $\alpha(o{:}\,F)$. As the language is strongly typed, all actual calls are guaranteed to be type-correct, so this need not be captured in the alphabet.

For convenience, we have defined the alphabet of $o{:}\,F$ as maximal, including all possible such objects. In reality, the communication environment of an object evolves over time, due to information about calling objects and object identifiers transmitted as method parameters. To capture this evolution is part of the task of specifying the acceptable communication traces. Alternatively, we could define the alphabet of $o{:}\,F$ as a function over the trace up to present time, to catch the idea of transmitted object identifiers more precisely. However, this would force us to use more complicated reasoning for the specifications, in particular for refinement and composition issues.

The assumption is a predicate on traces and object identifiers, declared following the **asm** keyword. As the assumption is a requirement on the environment, it is to be pointwise, i.e. it is expected to hold for the local traces restricted to a particular object in the environment at a time. Hence, in the declaration of an interface, the assumption predicate varies over traces as well as both calling and current objects. (When an object identifier offers the interface, it is understood as the current object.) Also, since assumptions are the responsibility of the environment, these are only expected to hold at points in the traces that end with input to the object o considered.

Inputs to an object o are either events $o'{\rightarrow}o.m(\ldots)$ or events $o{\leftarrow}o'.m(\ldots)$, reflecting initiations of calls to methods of o or answers to calls by o to methods of objects in the environment. Let $\mathbf{in}_o(h)$ denote a trace h where $rt(h)$ is hidden (recursively) while it is not an input to o, defined as

$\mathbf{in}_o(\varepsilon) = \varepsilon$
$\mathbf{in}_o(h \vdash o'{\rightarrow}o.m(\ldots)) = h \vdash o'{\rightarrow}o.m(\ldots)$

$\mathbf{in}_o(h \vdash o{\leftarrow}o'.m(\ldots)) = h \vdash o{\leftarrow}o'.m(\ldots)$
$\mathbf{in}_o(h \vdash \mathbf{others}) = \mathbf{in}_o(h)$.

Outputs are either events $o'{\leftarrow}o.m(\ldots)$ or events $o{\rightarrow}o'.m(\ldots)$, reflecting completions of calls to methods of o or initiations of calls by o to methods of objects in the environment. Denote by $\mathbf{out}_o(h)$ the corresponding predicate for output traces.

Given an assumption predicate in an interface declaration, we need to be explicit about its arguments. If A is such a predicate, declared in F, we define

$$A^{in}(h, o) = \forall x \neq o : A(\mathbf{in}_o(h/x), o, x) \quad \text{for } h \in \mathrm{Seq}[\alpha(o{:}\,F)],$$

and a similar predicate $A^{out}(h, o)$, hiding inputs to o by the function $\mathbf{out}_o(h)$. When we specify an assumption A in an interface supported by an object o, the quantification over calling objects is implicit: we actually expect $A^{in}(h, o)$ to hold. (See Section 7 for examples of OUN interface declarations.)

The invariant is a predicate on traces and object identifiers, declared following the **inv** keyword. Define $I^{out}(h)$ in a similar manner for the invariant predicate I, except that we do not quantify over calling objects x in the environment:

$$I^{out}(h, o) = I(\mathbf{out}_o(h/\alpha(o{:}\,F))) \quad \text{for } h \in \mathrm{Seq}[\alpha(o{:}\,F)].$$

The invariant of a specification is guaranteed to hold for the object offering the interface, so it is a predicate on the entire (local) trace. The (local) trace of $o{:}\,F$ is the global trace of the system designed, restricted to $\alpha(o{:}\,F)$. Hence, by an invariant I declared in an interface, we expect $I^{out}(h, o)$ to hold. If the assumption predicate is omitted in the declaration of an interface, it is understood as **true**.

If auxiliary functions or predicates are needed for specification purposes, these may be defined following the **where** keyword.

In OUN, traces are used explicitly in interface declarations to determine the behavior at some point in time. We therefore regard the behavior of objects supporting the interface as a set $\mathcal{T}$ of possible (finite) traces. For every trace h in such a set $\mathcal{T}$, all prefixes of h represent prior points in the life of the object. Hence, the set $\mathcal{T}$ must be prefix-closed, i.e. for all sequences h and h', $h \vdash\!\dashv h' \in \mathcal{T} \Rightarrow h \in \mathcal{T}$. Prefix-closed sets of finite traces represent safety specifications in the sense of Alpern and Schneider [1].

The trace set $\mathcal{T}(o{:}\,F)$ of $o{:}\,F$ is defined as the prefix-closure of the set of finite sequences over $\alpha(o{:}\,F)$ defined by

$$\{h : \mathrm{Seq}[\alpha(o{:}\,F)] \mid A^{in}(h, o) \Rightarrow (I^{out}(h, o) \wedge A^{out}(h, o))\}.$$

We do not want the output from the object to violate the assumption of future extensions to a trace, so $A^{out}(h, o)$ is included as part of the invariant.

Now, consider interfaces with inheritance. When an interface F inherits another interface F_i and o offers F to the environment, the alphabet of $o{:}\,F_i$ is included in $\alpha(o{:}\,F)$. For the traces, inclusion is by projection, so $h/\alpha(o{:}\,F_i) \in \mathcal{T}(o{:}\,F_i)$ for all $h \in \mathcal{T}(o{:}\,F)$. The trace set $\mathcal{T}(o{:}\,F)$of $o{:}\,F$, where F inherits interfaces $F_1, \ldots, F_m$ and where A and I are the assumption and invariant predicates of F, is then given by the prefix-closure of

$$\left\{ h : \mathrm{Seq}[\alpha(o{:}\,F)] \;\middle|\; \begin{array}{l} h/\alpha(o{:}\,F_1) \in \mathcal{T}(o{:}\,F_1) \wedge \ldots \wedge h/\alpha(o{:}\,F_m) \in \mathcal{T}(o{:}\,F_m) \\ \wedge\, A^{in}(h, o) \Rightarrow (I^{out}(h, o) \wedge A^{out}(h, o)) \end{array} \right\}$$

4.3 Contracts

Whereas interfaces describe roles of single objects, OUN offers the possibility to define contracts in order to describe interaction between several objects, as a kind of glass-box specification (in contrast to the composition of these objects, where internal activity is hidden). A contract has the form

contract C
begin
  with $o_1{:}\,F_1, \ldots, o_n{:}\,F_n$
  inv <formula on traces with appropriate restriction>
where <auxiliary function definitions>
end

This contract describes the interaction between the objects $o_1, \ldots, o_n$, typed by interfaces $F_1, \ldots, F_n$, respectively. Let $\alpha = \alpha(o_1{:}\,F_1) \cup \ldots \cup \alpha(o_n{:}\,F_n)$. The communication events $\alpha(C)$ considered by a contract can be derived from the alphabets of the objects considered, typed by their respective interfaces, as follows:

$$\alpha(C) = \{o{\rightarrow}o'.m(\ldots), o{\leftarrow}o'.m(\ldots) \in \alpha \mid o, o' \in \{o_1, \ldots, o_n\}\}.$$

Hence, the events considered by the contract are those regarding intercommunication between the objects considered by that contract, that are visible in the respective alphabets.

Let C be a contract between $o_1{:}\,F_1, \ldots, o_n{:}\,F_n$ with a predicate I as invariant. The contract should be respected by all (global) traces of the system, i.e. for any trace $h : \mathrm{Seq}[\alpha(o_1{:}\,F_1) \cup \ldots \cup \alpha(o_n{:}\,F_n)]$, h is a possible trace of the system if and only if $I(h/\alpha(C))$ holds. Auxiliary functions are as before.

5 Embedding OUN Specifications in PVS

In this section, we extend the PVS proof environment of Section 3 with constructs for OUN designs. For this purpose, we commence by a richer theory for communication events, reflecting the remote method calls of OUN. Then the OUN notion of a trace is formalized, and OUN specifications are introduced as a subtype of `Specification`. A framework for representing interface declarations in PVS is introduced and we discuss the representation of objects supporting interfaces, of interfaces, and of contracts in PVS.

5.1 Communication Events for OUN

The `Event` type of Section 3 needs to be modified to include the additional information of OUN events. In this section, we will briefly present such a modification. The complete PVS declarations for OUN communication events are included in Appendix C. Objects in OUN are typed by interface, so we need a type for interface names in PVS:

```
Interface: TYPE
```

Introduce a type for the different kinds of events that we consider in the communication model, i.e. initiation and completion, by enumeration:

```
EvtKind: TYPE = {i,c}
```

We use a uniform type for the possible data values that are transmitted as either input or output values to the communication events reflecting method calls in the model. For convenience, we here limit the possible data values to the natural numbers and to objects typed by interface, i.e. to the set

$$\{x \in \texttt{nat}\} \cup \{y{:}\, z \mid y \in \texttt{Object} \wedge z \in \texttt{Interface}\}.$$

In PVS, the union of two types is typically created by wrapping them into a new type, using different constructors and identifiers for the two subtypes. This technique is used several times in the paper. Define `Data` as an abstract datatype as follows:

```
Data: DATATYPE
 BEGIN
  numb(x: nat): numb?
  ref(x: Object, y: Interface): ref?
 END Data
```

Thus, `Data` is the disjoint union of the natural numbers and the objects, typed by interface, encoded in a common datatype. We cannot construct the (disjoint) union of two types directly in PVS [16].

The modified type `Event` is a subtype of the PVS record type

```
[# caller: Object, to: Object, name: Method, kind: EvtKind,
   input: list[Data], output: list[Data] #]
```

such that the list of output data is empty for initiation events and an event does not have the same caller and receiver. All the results of Section 3 still hold when we use this modified `Event` type.

5.2 A Theory of Traces

In OUN, a remote method call is reflected in the traces by an initiation and a completion event. Therefore, there is an underlying assumption that every completion in a trace is preceded by a corresponding initiation. To any event `m`, we can construct the initiation event `init(m)` that corresponds to `m`. (If `kind(m)=i`, then `init(m)=m`.) Define

```
init(m: Event): Event =
    (# name := name(m),
       to := to(m),
       caller := caller(m),
       kind := i,
       input := input(m),
       output := null #)
```

Say that a set of events is *balanced* if there is an initiation event in the set that corresponds to every completion event in the set. The balanced sets of events satisfy the following predicate:

```
balanced?(s: set[Event]): bool =
    (FORALL (m: Event):
       kind(m) = c AND member(m, s) => member(init(m), s))
```

A sequence over a set of events S is *causal* if, for every balanced subset of S, initiation events occur more often than completions in all prefixes of the sequence. This requirement is formalized in the predicate

```
causal?(tr: seq[Event]): bool =
    FORALL (s: set[Event]):
      balanced?(s)
          => dom(restrict(tr, s), {x: Event | kind(x) = i})
```

where the domination predicate `dom(t,s)` states that events from the set `s` occur more often than not as one moves from left to right in the sequence `t`, see Appendix A.1 for details. Note that for every completion event e occurring in a trace, causality implies that the initiation corresponding to that termination has already occurred, as the set $\{e, \texttt{init}(e)\}$ is balanced.

Define a type `Trace` to represent OUN traces as the *causal* subtype of `Seq[Event]`:

```
Trace: TYPE = (causal?)
```

The empty sequence of events satisfies the causality predicate, so it inhabits this type. The $\mathbf{in}_o(h)$ and $\mathbf{out}_o(h)$ functions are represented in PVS be the two functions `in_prefix` and `out_prefix` on `Trace`, where the former is defined as

```
in_prefix(tr: Trace, O: Object): RECURSIVE Trace =
  CASES tr
    OF empty: empty,
       addr(s, x):
         IF (kind(x) = i AND to(x) = O) OR
             (kind(x) = c AND caller(x) = O)
           THEN tr
         ELSE in_prefix(s, O)
         ENDIF
    ENDCASES
   MEASURE length(tr)
```

The output prefix function `out_prefix`has a similar definition.

5.3 Objects typed by Interfaces

An object o that offers an interface F to its environment has a defined alphabet $\alpha(o{:}\,F)$ and a defined set of traces $\mathcal{T}(o{:}\,F)$. We can therefore interpret $o{:}\,F$ as a singleton specification in the sense of Section 3. Whereas the alphabet may be translated into a PVS set of events `S` in a fairly straightforward manner (provided that all inherited interfaces and their cointerfaces are already defined), the translation of the trace set depends upon our ability to represent the assumption and invariant predicates in PVS. In this section, we show how OUN specifications can be translated into PVS, given a representation of the assumption and invariant predicates. The complete PVS representation can be found in Appendix E. Remark that the alphabet of an OUN specification is balanced, which is helpful for the consideration of the traces. An OUN specification is defined as the subtype of `Specification` with balanced alphabets:

```
OUNSpecification: TYPE = {s: Specification | balanced?(alpha(s))}
```

We now define the types for assumption and invariant predicates as

```
  AsmPred: TYPE = [Trace, Object, Object -> bool]

  InvPred: TYPE = [Trace, Object -> bool]
```

restricted to predicates that are true for the empty trace (cf. complete theory in appendix). By construction, we assume that an alphabet includes the alphabets of all inherited interfaces (as supported by the same object). The traces in the trace set must be included in the trace set of every inherited interface, with the appropriate alphabet restriction. This requirement is defined as a predicate on traces:

```
inheritanceReq?(h: Trace, Interfaces: set[OUNSpecification]): bool =
      FORALL (s: OUNSpecification):
        member(s, Interfaces)
          => member(restrict(h, alpha(s)), traces(s))
```

From this requirement and the assumption and invariant predicates, the following predicate is constructed on traces:

```
tracepred?(h: Trace, Asm: AsmPred, Inv: InvPred, O: Object,
            Inherited: set[OUNSpecification]): bool =
    ((FORALL (x: Object): NOT (x = O) => Asm(in_prefix(h, O), O, x))
        => (Inv(out_prefix(h, O), O) AND
              (FORALL (x: Object):
                 NOT (x = O) => Asm(out_prefix(h, O), O, x))))
      AND inheritanceReq(h, Inherited)
```

The trace set of $o{:}F$ can then be represented as the set of traces h such that all prefixes of h satisfy the predicate `tracepred` above:

```
AsmInvTraceSet(a: (balanced?), Asm: AsmPred, Inv: InvPred,
                O: Object, Inherited: set[OUNSpecification]):
  set[Trace] =
      {h: Trace |
         restrict(h, a) = h AND
          (FORALL (prefix: Trace):
             (ord?(prefix, h)
                => tracepred(prefix, Asm, Inv, O, Inherited)))}
```

These trace sets are prefix-closed. In PVS, we state and prove the following lemma:

```
PrefixclosureLemma: LEMMA
    FORALL (a: (balanced?), Asm: AsmPred, Inv: InvPred, O: Object,
            Inherited: set[OUNSpecification]):
      prefixclosed?((AsmInvTraceSet(a, Asm, Inv, O, Inherited)))
```

Hence, the prefix closure of trace sets for actual specifications will not become proof obligations in PVS.

5.4 Reasoning about Interfaces

So far, we have only discussed how we can represent in PVS an interface declaration that is a viewpoint to a particular object. An interface F that has not been associated with such an object, can be understood as the union of all (possible) objects offering F. Thus, an interface F is represented by the specification

$$\langle Object, \bigcup_{o:Object} \alpha(o{:}\,F), \bigcup_{o:Object} \mathcal{T}(o{:}\,F)\rangle.$$

With an appropriate translation, this is a `Specification` in PVS. Therefore, by choosing this representation, we can reason in PVS about interface refinement and composition directly, even before it is decided which objects shall actually support a particular interface. Remark that all formal parameters to interface declarations can be dealt with in the same manner as the supporting object is treated here.

5.5 Contracts

Contracts restrict the interaction between a given set of objects. If a contract with alphabet `alpha` and invariant predicate `I` is included in a design, we will interpret the contract (for specification purposes) as a restriction on the global traces of the system designed. In PVS, contracts are used to define subtypes of `Trace`. Thus, let

```
SystemTrace: TYPE = {h:Trace | I(restrict(h,alpha)}
```

Additional contracts are added as additional conjuncts. To verify that a contract is respected by a design, it is sufficient to show that for every h of type `Trace` (i.e. not restricted to `SystemTrace`) that is in the trace set of every interface , h validates the contract in the sense that `I(restrict(h,alpha)` holds.

6 Regular expressions

A convenient predicate for safety specifications is the predicate that defines prefixes of regular expressions, because the trace set defined by such a predicate is always prefix-closed. Thus, using these predicates for assumptions and invariants in OUN give an intuitive, graphical specifcation style. A theory for such predicates is now formalized in PVS and will be used in the examples of Section 7 in order to express assumption and invariant predicates of OUN interfaces. Regular expressions can be defined as an abstract datatype in PVS, parameterized over some type T:

```
reg[T: TYPE]: DATATYPE
 BEGIN
  rempty: rempty?
  single(elt: T): single?
  AND(fst: reg, snd: reg): and?
  OR(lft: reg, rgt: reg): or?
  star(body: reg): star?
 END reg
```

We want to define constructively, i.e. by recursion using `addl`, a predicate `prs?(t,exp)` which is true if `t` is a prefix of a trace described by the regular expression `exp`. In order to do this, we first wrap the regular expressions in another datatype, to encode error situations that can occur in the recursion. Define

```
ereg[T: TYPE]: DATATYPE
 BEGIN
  IMPORTING reg[T]

  okreg(rexp: reg): okreg?
  emptyreg: emptyreg?
  nomatchreg: nomatchreg?
 END ereg
```

The prefix of regular expression function `prs?` can now be defined as follows:

```
  prs?(s: seq, e: reg): RECURSIVE bool =
    CASES s
      OF empty: TRUE,
         addr(q, y):
           LET z = lt(s), q = rr(s), f = pop(z, e) IN
             IF emptyreg?(f) OR nomatchreg?(f)
               THEN FALSE
             ELSE prs?(q, rexp(f))
             ENDIF
      ENDCASES
     MEASURE length(s)
```

where `pop(z,e)` returns an element of type `ereg`, and, in the case of success, a new regular expression where the element matching `z` (from the left) has been removed. See Appendix F for the complete PVS encoding. The prefix-closure of the `prs?` predicate can now be proved:

```
  PrefixLemma2: LEMMA
    FORALL (s,t:seq, r:reg): (ord?(s,t) and prs?(t,r)) => prs?(s,r)
```

7 Examples

To illustrate how PVS specifications can be derived from OUN designs, we consider the declaration of OUN interfaces *Reader*, *Writer*, and *ReaderWriter*, describing different viewpoints to an object controlling read and write access to some shared data, following [5]. For the *Reader* viewpoint, we want to allow concurrent read operations from the objects of the environment, whereas by the *Writer* viewpoint, access is restricted so that only one object in the environment may perform write operations at the time. Finally the interface *ReaderWriter* allows read operations when a calling object is granted write access.

7.1 A Reader Interface in PVS

When we consider an object controlling read and write access to some shared data, the *Reader* interface describes a viewpoint to the object which says that

concurrent read access should be granted to all objects in the environment. In OUN, the *Reader* interface can be declared as

interface *Reader*[T : Data-type]
begin
  with any
    opr *Read*(**out** $d : T$)
  asm true
  inv true
end

For the example, we will consider an object `object` supporting the interface. In PVS, we declare

```
Read: Method

object: Object
```

The alphabet of *Reader*, supported by `object`, is then defined as

```
AlphaReader: set[Event] =
    {e: Event | to(e) = object
                  AND name(e) = Read AND input(e) = null}
```

The assumption and invariant predicates always return **true**, so define

```
AsmReader: AsmPred = (LAMBDA ((tr: Trace), (o1, o2: Object)): TRUE)

InvReader: InvPred = (LAMBDA ((tr: Trace), (O: Object)): TRUE)
```

The trace set of `object`, when considered through the viewpoint offered the *Reader* interface, is of course the set of all *local* traces, i.e. the traces over the alphabet `AlphaReader`:

```
TracesReader: set[Trace] =
    restrict({h: seq[Event] | restrict(h, AlphaReader) = h})
```

However, in order to follow the standard scheme for defining trace sets that we outlined in Section 5.3, we prefer to define the trace set we consider as follows:

```
TracesReader2: set[Trace] =
    AsmInvTraceSet(AlphaReader, AsmReader,
                   InvReader, object, emptyset)
```

Define `ObjectReader` as a `Specification` in PVS following the scheme:

```
ObjectReader: OUNSpecification =
    (# alpha := AlphaReader,
       traces := (TracesReader2),
       obj := singleton(object) #)
```

Finally, we can state and prove that the intuitive and schematical formulations of the trace set are in fact equal:

```
ReaderLemma: LEMMA TracesReader = TracesReader2
```

The complete PVS theories for this example can be found in Appendix G.1.

7.2 A Writer Interface in PVS

In order to specify access control, a calling object must first call a method *Open_write*. It is first upon completion of this call that the calling object can perform write operations. To return access control, the calling object invokes the *Close_write* method. Using this convention, the *Writer* interface limits write access to one object at the time. In OUN, it can be declared as

interface $Writer[T$: Data-type]
begin
  with any
    opr *Open_write*()
    opr $Write$(**in** $d : T$)
    opr *Close_write*()
  asm h prs $(\leftrightarrow.open\_write \leftrightarrow.write^* \leftrightarrow.close\_write)^*$
  inv $h/ \leftarrow$ prs $(\leftarrow.open\_write \leftarrow.write^* \leftarrow.close\_write)^*$
end

The sequence of events required for the *Writer* interface to guarantee that write access is held by one object at a time, is formulated as an assumption on the environment. The assumption predicate therefore has three implicit parameters: the history as well as the calling and receiving objects. (In this example, the receiving object is `object`.) Likewise, the invariant predicate has two implicit parameters: the history and the receiving object.

In PVS, declare three elements `Open_write`, `Write`, and `Close_write` of type `Method`. Now, the alphabet `AlphaWriter` of `object`, offering the *Writer* interface, can be defined. Remark that it is declared to be of type `(balanced?)`.

The assumption and invariant predicates of *Writer* consider only the name and kind of the communication events, ignoring the object identities of the caller and receiver. This abstraction on events can be represented directly in PVS. Define a type `AbstractEvent` that only includes these fields, import `reg_exp[AbstractEvent]` into the current theory, and define a function

```
liftseq(t: seq[Event]): RECURSIVE seq[AbstractEvent]
```

that will lift a trace to a sequence of `AbstractEvent`. The assumption predicate can then be defined as

```
AsmWriter: AsmPred =
    (LAMBDA ((tr: Trace), (o1, o2: Object)):
       prs?(liftseq(tr),
            AND(AND(AND(AND(AND(single((# name := Open_write,
                                        kind := i #)),
                                single((# name := Open_write,
                                        kind := c #))),
                            single((# name := Write, kind := i #))),
                        single((# name := Write, kind := c #))),
                    single((# name := Close_write, kind := i #))),
                single((# name := Close_write, kind := c #)))))
```

The invariant can be defined in a similar manner, for traces restricted to completion events only. Finally, the trace set `TracesWriter` can be defined following the scheme, and we get:

```
ObjectWriter: OUNSpecification =
    (# alpha := AlphaWriter,
       traces := (TracesWriter),
       obj := singleton(object) #)
```

The complete encoding of this example in PVS can be found in Appendix G.2.

7.3 A ReaderWriter Interface in PVS

The *ReaderWriter* interface allows an object to perform read operations when it has been granted access to perform write operations. Remark that the interface inherits *Writer* as well as *Reader*. The interface can be declared in OUN as

interface *ReaderWriter*[T : Data-type]
  inherits *Writer*, *Reader*
begin
  with any
    opr *Open_read*()
    opr *Close_read*()
  asm h prs $(\leftrightarrow.open\_write \leftrightarrow.write^* \leftrightarrow.close\_write$
        $\mid \leftrightarrow.open\_read \leftrightarrow.read^* \leftrightarrow.close\_read)^*$
  inv $\sharp(h/ \leftarrow.open\_read) - \sharp(h/ \leftarrow.close\_read) = 0$
      $\vee\ \sharp(h/ \leftarrow.open\_write) - \sharp(h/ \leftarrow.close\_write) = 0$
end

In this example, we must take into account the inheritance-clause, and see how it modifies the schematic translation of the previous examples. First, the theory for the *Writer* interface is imported and we declare `Open_read`, `Read` and `Close_read` to be of type `Method`. Then, construct a set of events `AlphaReader` using these method names. The **with** clause states that any object may make calls to this interface, so we do not get any additional events form the **with** clause. The alphabet of *ReaderWriter* becomes

```
AlphaRW: (balanced?) = union(AlphaReader, AlphaWriter)
```

The assumption predicate `AsmRW` resembles the predicates of `Writer`. The invariant predicate `InvRW` is defined as

```
InvRW: InvPred =
    (LAMBDA ((tr: Trace), (O: Object)):
       (length(restrict(tr, {e: Event | name(e) = Open_read})) -
         length(restrict(tr, {e: Event | name(e) = Close_read}))
         = 0)
        OR
        (length(restrict(tr, {e: Event | name(e) = Open_write})) -
         length(restrict(tr, {e: Event | name(e) = Close_write}))
         = 0))
```

The trace set `TracesRW` can then be constructed from predicates `AsmRW` and `InvRW`, but for this interface, we need to include the inherited specification:

```
TracesRW: set[Trace] =
    AsmInvTraceSet(AlphaRW, AsmRW, InvRW,
                   object, singleton(ObjectWriter))
```

Finally, we get an OUN specification for the `object` offering the *ReaderWriter* interface, which can be represented as `ObjectRW`. Finally, we state and prove in PVS that

```
RWrefinesW: LEMMA refines?(ObjectRW, ObjectWriter)
```

The complete PVS encoding of this example can be found in Appendix G.3.

8 Conclusion

In order to be of any practical use, a formal specification language needs the support of a proof environment. Without the assistance of the theorem proving facilities of such an environment, it is impossible to perform formal reasoning about specifications of some complexity. In this paper, we present a PVS proof environment for a formalism for partial object specifications. This formalism supports specification by separation of concerns, multiple inheritance, and refinement. In PVS, the formalism is shown to have the compositional refinement property. We extend the proof environment with constructs to facilitate its use as a formal reasoning environment for OUN specifications, i.e. for objects supporting interfaces in OUN. In the paper, we also discuss how the formalism for partial object specifications can be used to represent OUN notions such as interfaces and contracts. In the latter case, both specification and verification concerns are considered.

In OUN, prefixes of regular expressions are often employed to specify assumption and invariant predicates. These predicates give an intuitive, graphical style of safety specifications because the trace sets defined by such predicates are prefix closed. A theory for prefixes of regular expressions is constructed in PVS and added to the formalism in order to facilitate the translation of such specifications into the proof environment, and the prefix closure of these predicates are proved. Finally, we show by examples how some OUN interfaces can be translated into PVS, and also how refinement claims are represented and proved.

References

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, 1999.

[3] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, Apr. 1995. Available, with specification files, at `http://www.csl.sri.com/wift-tutorial.html`.

[4] O.-J. Dahl. *Verifiable Programming*. International Series in Computer Science. Prentice-Hall, New York, N.Y., 1992.

[5] O.-J. Dahl and O. Owe. Formal methods and the RM-ODP. Technical Report 261, Department of Informatics, University of Oslo, 1998.

[6] B. Dutertre and S. Schneider. Embedding CSP in PVS. an application to authentication protocols. In E. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *LNCS*, pages 121–136, Murray Hill, NJ, 1997. Springer.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[8] ITU Recommendation X.901-904 ISO/IEC 10746 1-4. *Open Distributed Processing - Reference Model*. ISO/IEC, July 1995.

[9] E. B. Johnsen. Composition and refinement for partial object specifications. Submitted for publication, 2001.

[10] E. B. Johnsen. PVS theories for OUN specifications, 2001. Available at `http://www.ifi.uio.no/~einarj/PVSOUN.tar.bz2`.

[11] E. B. Johnsen, O. Owe, E. Munthe-Kaas, and J. Vain. Incremental design of dependable systems in OUN. Technical Report 293, Department of informatics, University of Oslo, 2000.

[12] E. B. Johnsen, O. Owe, E. Munthe-Kaas, and J. Vain. Incremental fault-tolerant design in an object-oriented setting. submitted, 2001.

[13] O. Owe and I. Ryl. A notation for combining formal reasoning, object orientation and openness. Technical Report 278, Department of informatics, University of Oslo, 1999.

[14] O. Owe and I. Ryl. On combining object orientation, openness and reliability. In *NIK '99 — Proceedings of Norsk Informatikkonferanse 1999*. Tapir, 1999.

[15] O. Owe and I. Ryl. OUN: a formalism for open, object oriented, distributed systems. Technical Report 270, Department of informatics, University of Oslo, 1999.

[16] S. Owre and N. Shankar. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993. Extensively revised June 1997.

[17] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1998.

[18] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1998.

[19] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1998.

[20] I. Traoré. The UML specification of the Integrator. Technical Report 275, Department of Informatics, University of Oslo, 1999.

[21] I. Traore, D. B. Aredo, and K. Stølen. Formal development of open distributed systems: Towards an integrated framework. In *Proc. Workshop on Object-Oriented Specification Techniques for Distributed Systems and Behaviours (OOSDS'99)*, September 1999.

[22] I. Traoré and K. Stølen. Towards the definition of a platform supporting the formal development of open distributed systems. Technical Report 271, Department of Informatics, University of Oslo, 1999.

A Finite Sequences in PVS

A.1 Sequence Definitions

Sequences over a type T are defined as an abstract datatype in PVS.

```
seq[T: TYPE]: DATATYPE
 BEGIN
  empty: empty?
  addr(lr: seq, rt: T): nonempty?
 END seq
```

Note that `empty` and `addr` are constructors for the datatype, `empty?` and `nonempty?` are identifiers, giving rise to subtypes, and `lr` and `rt` are selectors for the abstract datatype.

In the theory `sequence[T]`, further functions and predicates on sequences are introduced, using the constructors and selectors of the datatype. We define a length function, a concatenation operator `conc`, an addleft operator `addl` (cf. `addr` above), an ordering predicate `ord` on prefixes, a subsequence operator, projection functions `restrict` and `hide`, selectors `lt` and `rr` for the addleft operator (cf, the selectors of `addr` in the datatype). Finally, we define a predicate "ends with", abbreviated `ew`, and the domination predicate of Dahl [4], which states that elements of a set a occur more often than not as we move from left to right along a sequence s.

```
sequence[T: TYPE]: THEORY
 BEGIN

  IMPORTING seq[T]

  singleton_seq(x: T): seq[T] = addr(empty, x)

  length(s: seq[T]): RECURSIVE nat =
    CASES s OF empty: 0, addr(x, y): length(x) + 1 ENDCASES
     MEASURE reduce_nat(0, (LAMBDA (n: nat, x: T): n + 1))

  conc(s, t: seq[T]): RECURSIVE seq[T] =
    CASES t OF empty: s, addr(t1, x): addr(conc(s, t1), x) ENDCASES
     MEASURE length(t)
```

```
addl(x: T, s: seq[T]): RECURSIVE seq[T] =
  CASES s OF empty: addr(empty, x),
             addr(s1, y): addr(addl(x, s1), y)
    ENDCASES
   MEASURE length(s)

ord?(s, t: seq[T]): bool =
  EXISTS (u: seq[T]): conc(s, u) = t

sub?(s, t: seq[T]): bool =
  EXISTS (u, v: seq[T]): conc(u, conc(s, v)) = t

restrict(s: seq[T], a: set[T]): RECURSIVE seq[T] =
  CASES s
    OF empty: empty,
       addr(t, x):
         IF member(x, a)
           THEN addr(restrict(t, a), x)
         ELSE restrict(t, a)
         ENDIF
    ENDCASES
   MEASURE length(s)

hide(s: seq[T], a: set[T]): RECURSIVE seq[T] =
  CASES s
    OF empty: empty,
       addr(t, x):
         IF member(x, a) THEN hide(t, a)
                         ELSE addr(hide(t, a), x)
         ENDIF
    ENDCASES
   MEASURE length(s)

lt(s: (nonempty?)): RECURSIVE T =
  IF empty?(lr(s)) THEN rt(s) ELSE lt(lr(s)) ENDIF
   MEASURE length(s)

rr(s: (nonempty?)): RECURSIVE seq[T] =
  IF empty?(lr(s)) THEN empty
                   ELSE addr(rr(lr(s)), rt(s))
  ENDIF
   MEASURE length(s)

ew?(s: (nonempty?[T]), x: T): bool = rt(s) = x
```

```
  dom(s: seq[T], a: setof[T]): RECURSIVE bool =
    CASES s
      OF empty: TRUE,
         addr(s1, x):
           dom(s1, a) AND
            ((length(restrict(s1, a)) = length(hide(s1, a)))
              IMPLIES member(x, a))
      ENDCASES
     MEASURE length(s)
 END sequence
```

A.2 Properties of Sequences

We state and prove equality and distribution properties for the functions we have introduced. Note that the equality lemma `EQ` follows from the definition of the datatype.

```
seq_lemmas[T: TYPE]: THEORY
 BEGIN

  IMPORTING sequence[T]

  EQ: LEMMA
    FORALL (s, t: seq[T]):
      s = t IFF
       (s = empty AND t = empty) OR
        ((NOT (s = empty) AND NOT (t = empty)) AND
          ((rt(s) = rt(t)) AND (lr(s) = lr(t))))

  EqualityLemma1: LEMMA FORALL (s: (nonempty?)): NOT (s = empty)

  EqualLengthLemma: LEMMA
    FORALL (s, t: seq[T]): s = t IMPLIES length(s) = length(t)

  EqualityLemma2: LEMMA
    FORALL (s: seq[T], x: T): NOT (s = addr(s, x))

  EqualityLemma3: LEMMA
    FORALL (s, t: seq[T]):
      s = t IFF
       (s = empty AND t = empty) OR
        ((NOT (s = empty) AND NOT (t = empty)) AND
          lt(s) = lt(t) AND rr(s) = rr(t))

  ConcEqualityLemma1: LEMMA
    FORALL (s, t, u: seq[T]): conc(s, u) = conc(t, u) IFF s = t

  RrLengthLemma: LEMMA
    FORALL (s: (nonempty?[T])): length(rr(s)) = length(lr(s))
```

```
AddlLemma1: LEMMA
  FORALL (s: seq[T], x: T):
    EXISTS (t: seq[T], y: T): addr(s, x) = addl(y, t)

AddlLemma2: LEMMA
  FORALL (s: seq[T], x: T):
    EXISTS (t: seq[T], y: T): addl(x, s) = addr(t, y)

AddlLemma3: LEMMA
  FORALL (s: seq[T], x: T): lt(addl(x, s)) = x AND rr(addl(x, s)) = s

AddlLemma4: LEMMA FORALL (s: seq[T], x: T): nonempty?(addl(x, s))

ConcEmptyLemma: LEMMA FORALL (s: seq[T]): conc(empty, s) = s

ConcAssocLemma: LEMMA
  FORALL (s, t, u: seq[T]): conc(s, conc(t, u)) = conc(conc(s, t), u)

ConcAddlAddrLemma: LEMMA
  FORALL (s, t: seq[T], x: T): conc(s, addl(x, t)) = conc(addr(s, x), t)

ConcSingLemma: LEMMA
  FORALL (s: seq[T], x: T): addr(s, x) = conc(s, singleton_seq(x))

LrConcLemma: LEMMA
  FORALL (s: seq[T], t: (nonempty?)): lr(conc(s, t)) = conc(s, lr(t))

AddrLrRtLemma: LEMMA FORALL (s: (nonempty?)): s = addr(lr(s), rt(s))

EmptyConcLemma: LEMMA
  FORALL (s, t: seq[T]):
    conc(s, t) = empty IMPLIES (s = empty AND t = empty)

LtLemma: LEMMA FORALL (x: T, s: seq[T]): lt(conc(singleton_seq(x), s)) = x

RrLemma: LEMMA FORALL (x: T, s: seq[T]): rr(conc(singleton_seq(x), s)) = s

LtRrConcLemma: LEMMA
  FORALL (s: seq[T], x: T):
    addr(s, x) = conc(singleton_seq(lt(addr(s, x))), rr(addr(s, x)))

LrConcDistr: LEMMA
  FORALL (s, t: (nonempty?)): lr(conc(s, t)) = conc(s, lr(t))

LtConcDistr: LEMMA
  FORALL (s: (nonempty?), t: seq[T]): lt(conc(s, t)) = lt(s)

RrConcDistr: LEMMA
  FORALL (s: (nonempty?), t: seq[T]): rr(conc(s, t)) = conc(rr(s), t)
```

```
 LtAddrDistr: LEMMA FORALL (s: (nonempty?), x: T): lt(addr(s, x)) = lt(s)

 RrAddrDistr: LEMMA
   FORALL (s: (nonempty?), x: T): rr(addr(s, x)) = addr(rr(s), x)

 AddlConcDistLemma: LEMMA
   FORALL (s, t: seq[T], x: T): conc(addl(x, s), t) = addl(x, conc(s, t))

 OrdSingletonLeft: LEMMA
   FORALL (x: T, s, t: seq[T]):
     ord?(conc(singleton_seq(x), s), t) IFF
      NOT (t = empty) AND x = lt(t) AND ord?(s, rr(t))

 OrdTrans: LEMMA
   FORALL (s, t, u: seq[T]):
     ord?(s, u) AND ord?(t, u) IMPLIES ord?(s, t) OR ord?(t, s)

 SubConcLemma: LEMMA
   FORALL (x: T, s, t: seq[T]):
     sub?(conc(singleton_seq(x), s), t) IFF
      NOT (t = empty) AND
       ((lt(t) = x AND ord?(s, rr(t))) OR
         sub?(conc(singleton_seq(x), s), rr(t)))

 OrdLemma: LEMMA
   FORALL (s, t: seq[T]):
     (nonempty?(t) AND ord?(s, t)) IMPLIES (s = t OR ord?(s, lr(t)))

 OrdEmptyLemma: LEMMA FORALL (t: seq[T]): ord?(t, empty) => t = empty

 ConcLengthLemma: LEMMA
   FORALL (t1, t2: seq[T]): length(t1) <= length(conc(t1, t2))

 OrdLengthLemma: LEMMA
   FORALL (t1, t2: seq[T]): ord?(t1, t2) => length(t1) <= length(t2)

 OrdAddrLemma: LEMMA
   FORALL (t1, t2: seq[T], m: T): ord?(t1, t2) => ord?(t1, addr(t2, m))

 ConcEqualityLemma2: LEMMA
   FORALL (s, t, u: seq[T]): conc(s, t) = conc(s, u) IFF t = u

 ConcEqualityLemma3: LEMMA
   FORALL (s, t, u, v: seq[T]):
     conc(s, t) = conc(u, v) IFF
      ((s = u AND t = v) OR
        (EXISTS (a: (nonempty?[T])): conc(s, a) = u AND conc(a, v) = t) OR
         (EXISTS (a: (nonempty?[T])): conc(u, a) = s AND conc(a, t) = v))
END seq_lemmas
```

A.3 Properties of Projection

The properties of the projection functions `restrict` and `hide` are of particular interest to Section 3. They are presented in a separate PVS theory.

```
restriction[T: TYPE]: THEORY
 BEGIN

  IMPORTING seq_lemmas[T]

  RestrictIntersect: LEMMA
    FORALL (s: seq[T], a, b: set[T]):
      restrict(restrict(s, a), b) = restrict(s, intersection(a, b))

  RestrictLength: LEMMA
    (FORALL (t: seq[T], s: set[T]): length(t) >= length(restrict(t, s)))

  RestrictHideDist: LEMMA
    FORALL (t: seq[T], s1, s2: set[T]):
      restrict(hide(t, s1), s2) = hide(restrict(t, s2), s1)

  RestrictHideDifference: LEMMA
    FORALL (t: seq[T], s1, s2: set[T]):
      restrict(hide(t, s1), s2) = restrict(t, difference(s2, s1))

  SubsetRestrict: LEMMA
    FORALL (t: seq[T], s1, s2: set[T]):
      subset?(s1, s2) => restrict(restrict(t, s2), s1) = restrict(t, s1)

  NotAddrRestrictLemma: LEMMA
    FORALL (t: seq[T], m: T, s: set[T]): NOT (restrict(t, s) = addr(t, m))

  SubsetRestrict2: LEMMA
    FORALL (t: seq[T], s1, s2: set[T]):
      (subset?(s1, s2) AND restrict(t, s1) = t) => restrict(t, s2) = t

  RestrictHideNonIntersect: LEMMA
    FORALL (t: seq[T], s1, s2: set[T]):
      empty?(intersection(s1, s2)) =>
       restrict(t, s2) = restrict(hide(t, s1), s2)

  RestrictHideInterchange: LEMMA
    FORALL (t: seq[T], s1, s2: set[T]):
      empty?(intersection(s1, s2)) AND restrict(t, union(s1, s2)) = t =>
       restrict(t, s1) = hide(t, s2)
```

```
  RestrictLemma1: LEMMA
    FORALL (t1, t2: seq[T], s1, s2, s3: set[T]):
      (intersection(s1, s2) = s3 AND
        restrict(t1, s1) = t1 AND
         restrict(t2, s2) = t2 AND restrict(t1, s3) = restrict(t2, s3)
        =>
        (EXISTS (t3: seq[T]):
           restrict(t3, union(s1, s2)) = t3 AND
            restrict(t3, s1) = t1 AND restrict(t3, s2) = t2))

  RestrictLemma2: LEMMA
    FORALL (t1, t2: seq[T], s1, s2, s3: set[T]):
      (subset?(intersection(s1, s2), s3) AND
        restrict(t1, s1) = t1 AND
         restrict(t2, s2) = t2 AND restrict(t1, s3) = restrict(t2, s3)
        =>
        (EXISTS (t3: seq[T]):
           restrict(t3, union(s1, s2)) = t3 AND
            restrict(t3, s1) = t1 AND restrict(t3, s2) = t2))

  RestrictReplaceLemma: LEMMA
    FORALL (t: seq[T], s1, s2, s3: set[T]):
      intersection(s1, s2) = intersection(s1, s3) =>
       restrict(restrict(t, s2), s1) = restrict(restrict(t, s3), s1)

  RestrictIdemLemma: LEMMA
    FORALL (t: seq[T], s: set[T]):
      restrict(restrict(t, s), s) = restrict(t, s)
 END restriction
```

B A Theory for Compositional Refinement

In this PVS theory, we formalize specifications as triples that consist of a set of events with distinct caller and receiver objects, a prefix-closed set of traces and a non-empty set of objects. For specifications, we define composition by hiding of the internal events `int` in both the alphabet and the traces. We define the OUN refinement relation. Commutativity and associativity for the composition rule as well as compositionality of the refinement relation for proper and composable specifications are proved. The theory relies on a theory `Universe` that we have omitted here, but it need only contain the `Object`, `Method`, and `Event` types at the beginning of Section 3. However, it will also work for the OUN specific universe of Appendix C.

```
compref: THEORY
 BEGIN

  IMPORTING Universe, restriction[Event]
```

```
prefixclosed?: [set[seq[Event]] -> bool] =
    (LAMBDA (s: set[seq[Event]]):
       FORALL (t: seq[Event]):
         member(empty, s) AND
          ((nonempty?(t) AND member(t, s)) => member(lr(t), s)))

Triple: TYPE =
[# alpha: set[Event], traces: (prefixclosed?), obj: set[Object] #]

Specification: TYPE =
{x: Triple |
         nonempty?(obj(x)) AND
          (FORALL (m: Event):
             member(m, alpha(x)) =>
              (member(caller(m), obj(x)) XOR member(to(m), obj(x))))}

SenderNotReceiver: LEMMA
  FORALL (m: Event, s: Specification):
    (member(to(m), obj(s)) AND member(caller(m), obj(s))) =>
     NOT (member(m, alpha(s)))

Singleton_Spec: TYPE = {x: Specification | singleton?(obj(x))}

refines?(S1, S2: Specification): bool =
    subset?(alpha(S2), alpha(S1)) AND
     subset?(obj(S2), obj(S1)) AND
      (FORALL (t: seq[Event]):
         member(t, traces(S1)) =>
          member(restrict(t, alpha(S2)), traces(S2)))

int(Obj_set: set[Object]): set[Event] =
    {x: Event | member(to(x), Obj_set) AND member(caller(x), Obj_set)}

int(Obj_set1, Obj_set2: set[Object]): set[Event] =
    {x: Event |
       (member(to(x), Obj_set1) AND member(caller(x), Obj_set2)) OR
        (member(to(x), Obj_set2) AND member(caller(x), Obj_set1))}

proper?(s1, s2, s3: Specification): bool =
    empty?(intersection({x: Event |
                           (member(to(x), obj(s1)) OR
                             member(caller(x), obj(s1)))
                            AND
                            NOT (member(to(x), obj(s2))) AND
                             NOT (member(caller(x), obj(s2)))},
                       alpha(s3)))
```

```
 comp(S1, S2: Specification): Specification =
    LET alph =
          {x: Event |
             member(x,
                    difference(union(alpha(S1), alpha(S2)),
                               int(union(obj(S1), obj(S2)))))}
      IN
      (# alpha := alph,
         traces
           := {t: seq[Event] |
                 restrict(t, alph) = t AND
                  (EXISTS (t1: seq[Event]):
                     restrict(t1, union(alpha(S1), alpha(S2))) = t1 AND
                      t = hide(t1, int(union(obj(S1), obj(S2)))) AND
                       member(restrict(t1, alpha(S1)), traces(S1)) AND
                        member(restrict(t1, alpha(S2)), traces(S2)))},
        obj := union(obj(S1), obj(S2)) #)

composable?(s1, s2: Specification): bool =
    empty?(intersection(alpha(s1), int(obj(s2)))) AND
     empty?(intersection(alpha(s2), int(obj(s1))))

AlphabetLemma: LEMMA
  FORALL (s1, s2: Specification):
    subset?(alpha(comp(s1, s2)), union(alpha(s1), alpha(s2))) AND
     subset?(union(alpha(s1), alpha(s2)),
             union(alpha(comp(s1, s2)), int(obj(comp(s1, s2)))))

SingletonSetsLemma: LEMMA
  FORALL (x, y: (singleton?[Object])):
    nonempty?(y) AND subset?(y, x) => x = y

SingletonInternalLemma: LEMMA
  FORALL (x, y, z: Singleton_Spec):
    refines?(x, y) =>
     int(union(obj(x), obj(z))) = int(union(obj(y), obj(z)))

SingletonCompRef: LEMMA
  FORALL (x, y, z: Singleton_Spec):
    refines?(x, y) => refines?(comp(x, z), comp(y, z))

SingletonCompCommute: LEMMA
  FORALL (x, y: Singleton_Spec): comp(x, y) = comp(y, x)

CompCommute: LEMMA FORALL (x, y: Specification): comp(x, y) = comp(y, x)

AlphaCompCommute: LEMMA
  FORALL (x, y: Specification): alpha(comp(x, y)) = alpha(comp(y, x))
```

```
AlphaCompAssoc: LEMMA
  FORALL (x, y, z: Specification):
    alpha(comp(comp(x, y), z)) = alpha(comp(x, comp(y, z)))

setlemma1: LEMMA FORALL (s: set[Event]): ({x: Event | s(x)}) = s

CompAlphaLemma: LEMMA
  FORALL (x, y, z: Specification):
    difference(union(alpha(x), alpha(comp(y, z))),
               int(union(obj(x), union(obj(y), obj(z)))))
     = alpha(comp(x, comp(y, z)))

CompTraceLemma1: LEMMA
  FORALL (x, y, z: Specification):
    composable?(comp(x, y), z) =>
     (FORALL (t1, t2: seq[Event]):
        (restrict(t1, union(alpha(comp(x, y)), alpha(z))) = t1 AND
          restrict(t2, union(alpha(x), alpha(y))) = t2 AND
           restrict(t1, alpha(comp(x, y))) =
            hide(t2, int(obj(comp(x, y)))))
         =>
         (EXISTS (t3: seq[Event]):
            (restrict(t3, union(union(alpha(x), alpha(y)), alpha(z))) =
              t3
              AND
              restrict(t3, union(alpha(comp(x, y)), alpha(z))) = t1 AND
               restrict(t3, union(alpha(x), alpha(y))) = t2)))

CompTraceLemma2: LEMMA
  FORALL (x, y, z: Specification):
    composable?(comp(x, y), z) =>
     (FORALL (t: seq[Event]):
        (restrict(t, union(alpha(comp(x, y)), alpha(z))) = t AND
          member(restrict(t, alpha(comp(x, y))), traces(comp(x, y))) AND
           member(restrict(t, alpha(z)), traces(z)))
         =>
         (EXISTS (t2: seq[Event]):
            (restrict(t2, union(union(alpha(x), alpha(y)), alpha(z))) =
              t2
              AND
              member(restrict(t2, alpha(x)), traces(x)) AND
               member(restrict(t2, alpha(y)), traces(y)) AND
                member(restrict(t2, alpha(z)), traces(z)) AND
                 restrict(t2, union(alpha(comp(x, y)), alpha(z))) =
                  restrict(t, union(alpha(comp(x, y)), alpha(z))))))

SingletonCompAssoc: LEMMA
  FORALL (x, y, z: Singleton_Spec):
    (composable?(comp(x, y), z) AND composable?(x, comp(y, z))) =>
     comp(comp(x, y), z) = comp(x, comp(y, z))
```

```
  CompAssoc: LEMMA
    FORALL (x, y, z: Specification):
      (composable?(comp(x, y), z) AND composable?(x, comp(y, z))) =>
       comp(comp(x, y), z) = comp(x, comp(y, z))

  ComposabilityLemma: LEMMA
    FORALL (x, y, z: Specification):
      (refines?(x, y) AND composable?(x, z)) => composable?(y, z)

  PropernessLemma1: LEMMA
    FORALL (x, y, z: Specification):
      (refines?(x, y) AND proper?(x, y, z) AND composable?(x, z)) =>
       intersection(union(alpha(y), alpha(z)), int(obj(comp(x, z)))) =
        intersection(union(alpha(y), alpha(z)), int(obj(comp(y, z))))

  PropernessLemma2: LEMMA
    FORALL (x, y, z: Specification):
      (refines?(x, y) AND proper?(x, y, z) AND composable?(x, z)) =>
       difference(union(alpha(y), alpha(z)), int(obj(comp(x, z)))) =
        difference(union(alpha(y), alpha(z)), int(obj(comp(y, z))))

  AlphaCompRef: LEMMA
    FORALL (x, y, z: Specification):
      (refines?(x, y) AND proper?(x, y, z) AND composable?(x, z)) =>
       subset?(alpha(comp(y, z)), alpha(comp(x, z)))

  SpecificationCompRef: LEMMA
    FORALL (x, y, z: Specification):
      (refines?(x, y) AND proper?(x, y, z) AND composable?(x, z)) =>
       refines?(comp(x, z), comp(y, z))
 END compref
```

C Communication Events for OUN

These theories formalize the OUN notion of communication events representing remote method calls. Each method call is reflected in the traces by two distinct events, which we refer to as the initiation and completion of the call. In the examples, we profit from declaring the `Object` and `Method` types as non-empty.

```
Typeuniverse: THEORY
 BEGIN

  Object: NONEMPTY_TYPE

  Method: NONEMPTY_TYPE

  Interface: NONEMPTY_TYPE

  EvtKind: TYPE = {i, c}
```

```
 END Typeuniverse
```

In this model, transmitted values are either natural numbers or object identifiers. The latter are always typed by interface, so the transmitted reference is a pair.

```
Data: DATATYPE
 BEGIN
  IMPORTING Typeuniverse

  numb(x: nat): numb?
  ref(x: Object, y: Interface): ref?
 END Data
```

Communication events are represented by tuples that consist of named caller and receiver objects, a method name, the event kind (initiation or completion) as well as transmitted data values for input and output parameters. At this level, an object may not make calls to itself. (Such calls are not reflected in the traces.) Initiation events do not transmit output values.

```
Universe: THEORY
 BEGIN

  IMPORTING Data

  CommTuple: TYPE =
  [# caller: Object,
     to: Object,
     name: Method,
     kind: EvtKind,
     input: list[Data],
     output: list[Data] #]

  EvtReq(m: CommTuple): bool =
      (kind(m) = i IMPLIES output(m) = null) AND NOT (to(m) = caller(m))

  Event: TYPE = (EvtReq)
 END Universe
```

D OUN Traces in PVS

Traces that appear in the trace sets of OUN specification have some special characteristics, which are formalized in the `traces` theory. First, the notion of a balanced set of events is introduced. Then, we introduce an OUN `Trace` type as a sequence of events that obeys a causality predicate, which states that an event that reflects the completion of a remote call must be preceded by a corresponding initiation event, i.e. the trace is dominated by initiations when restricted to the events reflecting a particular call. Observing that the alphabet of an OUN specification is balanced, and so are the pair of events reflecting any particular call, domination must hold for every balanced set. When we

later consider inheritance and refinement by projection on traces, we obtain causality for the projections for free.

Special cases of restriction are introduced, restricting a trace to a particular set of method names, to a particular kind of events, and to a particular set of object identifiers. Finally, functions are defined that identify the largest prefix of a trace ending with an initiation and a completion event, respectively.

```
traces: THEORY
 BEGIN

  IMPORTING compref

  init(m: Event): Event =
      (# name := name(m),
         to := to(m),
         caller := caller(m),
         kind := i,
         input := input(m),
         output := null #)

  balanced?(s: set[Event]): bool =
      (FORALL (m: Event):
         kind(m) = c AND member(m, s) IMPLIES member(init(m), s))

  restrict_mtd(tr: seq[Event], o_set: setof[Method]): RECURSIVE
  seq[Event] =
    CASES tr
      OF empty: empty,
         addr(s, x):
           IF member(name(x), o_set)
             THEN addr(restrict_mtd(s, o_set), x)
           ELSE restrict_mtd(s, o_set)
           ENDIF
      ENDCASES
     MEASURE length(tr)

  restrict_kind(tr: seq[Event], e: EvtKind): RECURSIVE seq[Event] =
    CASES tr
      OF empty: empty,
         addr(s, x):
           IF kind(x) = e
             THEN addr(restrict_kind(s, e), x)
           ELSE restrict_kind(s, e)
           ENDIF
      ENDCASES
     MEASURE length(tr)

  causal?(tr: seq[Event]): bool =
      FORALL (s: set[Event]):
        balanced?(s) IMPLIES dom(restrict(tr, s), {x: Event | kind(x) = i})
```

```
  Trace: TYPE = (causal?)

  BalancedLemma1: LEMMA
    (FORALL (s1, s2: (balanced?)): balanced?(intersection(s1, s2)))

  restrict_balanced(t: Trace, s: (balanced?)): Trace = restrict(t, s)

  restrict_object(tr: Trace, O: Object): Trace =
      restrict_balanced(tr, {x: Event | to(x) = O OR caller(x) = O})

  restrict_objects(tr: Trace, o_set: set[Object]): Trace =
      restrict_balanced(tr,
                        {x: Event |
                           member(to(x), o_set) AND
                            member(caller(x), o_set)})

  in_prefix(tr: Trace, O: Object): RECURSIVE Trace =
    CASES tr
      OF empty: empty,
         addr(s, x):
           IF (kind(x) = i AND to(x) = O) OR
               (kind(x) = c AND caller(x) = O)
             THEN tr
           ELSE in_prefix(s, O)
           ENDIF
      ENDCASES
     MEASURE length(tr)

  out_prefix(tr: Trace, O: Object): RECURSIVE Trace =
    CASES tr
      OF empty: empty,
         addr(s, x):
           IF (kind(x) = c AND to(x) = O) OR
               (kind(x) = i AND caller(x) = O)
             THEN tr
           ELSE out_prefix(s, O)
           ENDIF
      ENDCASES
     MEASURE length(tr)
 END traces
```

The theory `trace_lemmas` considers properties of OUN traces, that are formalized and proved in the PVS theorem prover.

```
trace_lemmas: THEORY
 BEGIN

  IMPORTING traces
```

```
  CausalRestrictBalanced: LEMMA
    (FORALL (tr: Trace, s: (balanced?)): causal?(restrict(tr, s)))

  CausalEmpty: LEMMA causal?(empty)

  CausalLr: LEMMA
    (FORALL (t: seq[Event]):
       nonempty?(t) AND causal?(t) IMPLIES causal?(lr(t)))

  CausalPrefix: LEMMA
    FORALL (tr, tr2: Trace): causal?(tr) AND ord?(tr2, tr) IMPLIES causal?(tr2)

  CausalAddrInit: LEMMA
    (FORALL (tr: Trace, m: Event): kind(m) = i IMPLIES causal?(addr(tr, m)))

  RestrictMethod: LEMMA
    (FORALL (tr: Trace, m: Method):
       restrict(tr, {x: Event | name(x) = m}) =
        restrict_mtd(tr, singleton(m)))

  CausalRestrictMethod: LEMMA
    (FORALL (tr: Trace, m: Method):
       dom(restrict_mtd(tr, singleton(m)), {x: Event | kind(x) = i}))

  RestrictKindMethod: LEMMA
    FORALL (Tr: Trace, m: Method):
      restrict(Tr, {x: Event | name(x) = m AND kind(x) = i}) =
       restrict_kind(restrict_mtd(Tr, singleton(m)), i)

  HideInitRestrictCompl: LEMMA
    FORALL (Tr: Trace, m: Method):
      hide(restrict_mtd(Tr, singleton(m)), {x: Event | kind(x) = i}) =
       restrict_kind(restrict_mtd(Tr, singleton(m)), c)

  CausalInPrefix: LEMMA
    (FORALL (Tr: Trace, O: Object): causal?(in_prefix(Tr, O)))

  CausalOutPrefix: LEMMA
    (FORALL (Tr: Trace, O: Object): causal?(out_prefix(Tr, O)))

  OutPrefixOrdered: LEMMA
    FORALL (Tr: Trace, O: Object): ord?(out_prefix(Tr, O), Tr)
END trace_lemmas
```

E Interfaces and Contracts in PVS

In this theory, we introduce PVS constructs for representing OUN declarations as elements of type `Specification`. As the trace sets of OUN interfaces are

prefix closed, we insist that at least the empty trace belongs to the set. Hence, the assumption and invariant predicates must be true for the empty trace, and these are defined to be of types `AsmPred` and `InvPred`, respectively. The predicate `inheritanceReq` states that a trace, when appropriately restricted, belongs to all the trace sets given by the inherited interfaces. By `tracepred`, we represent the assumption and guarantee predicates combined with the inheritance relation. An actual trace set contains all the traces over a given alphabet such that all prefixes satisfy the `tracepred` predicate (for appropriate arguments). This trace set is captured by `AsmInvTraceSet`. Finally, we prove that the trace set of an OUN specification contains the empty trace and that all trace sets constructed by `AsmInvTraceSet` are prefix closed.

```
oun: THEORY
 BEGIN

  IMPORTING trace_lemmas

  OUNSpecification: TYPE = {s: Specification | balanced?(alpha(s))}

  AsmPredHelp: TYPE = [Trace, Object, Object -> bool]

  InvPredHelp: TYPE = [Trace, Object -> bool]

  EmptyAsm?(Pred: AsmPredHelp): bool =
      FORALL (o1, o2: Object): Pred(empty, o1, o2)

  EmptyInv?(Pred: InvPredHelp): bool =
      FORALL (O: Object): Pred(empty, O)

  AsmPred: TYPE = (EmptyAsm?)

  InvPred: TYPE = (EmptyInv?)

  inheritanceReq?(h: Trace, Interfaces: set[OUNSpecification]):
    bool =
      FORALL (s: OUNSpecification):
        member(s, Interfaces) =>
          member(restrict(h, alpha(s)), traces(s))

  tracepred?(h: Trace, Asm: AsmPred, Inv: InvPred, O: Object,
             Inherited: set[OUNSpecification]):
  bool =
      ((FORALL (x: Object): NOT (x = O) =>
        Asm(in_prefix(h, O), O, x)) =>
          (Inv(out_prefix(h, O), O) AND
            (FORALL (x: Object):
               NOT (x = O) => Asm(out_prefix(h, O), O, x))))
         AND inheritanceReq?(h, Inherited)
```

```
  AsmInvTraceSet(a: (balanced?), Asm: AsmPred, Inv: InvPred,
                 O: Object, Inherited: set[OUNSpecification]):
  set[Trace] =
      {h: Trace |
         restrict(h, a) = h AND
          (FORALL (prefix: Trace):
             (ord?(prefix, h) =>
                tracepred?(prefix, Asm, Inv, O, Inherited)))}

  EmptyTraceLemma: LEMMA
    FORALL (s: OUNSpecification): member(empty, traces(s))

  PrefixclosureLemma: LEMMA
    FORALL (a: (balanced?), Asm: AsmPred, Inv: InvPred, O: Object,
            Inherited: set[OUNSpecification]):
      prefixclosed?(extend[seq[Event], Trace, bool, FALSE]
                          (AsmInvTraceSet(a, Asm, Inv, O, Inherited)))
 END oun
```

F Regular Expressions in PVS

Regular expressions are defined as an abstract datatype in PVS. The common regular expresiion constructor a^+ is omitted as it equivalent to aa^*, which we represent by `AND(a, star(a))` in PVS. Define

```
reg[T: TYPE]: DATATYPE
 BEGIN
  rempty: rempty?
  single(elt: T): single?
  AND(fst: reg, snd: reg): and?
  OR(lft: reg, rgt: reg): or?
  star(body: reg): star?
 END reg
```

The regular expressions are wrapped into a datatype `ereg`, encapsulating the regular expressions in possible error messages. This datatype is used in the sequel for recursive functions where we want to end the computation in the erroneous cases.

```
ereg[T: TYPE]: DATATYPE
 BEGIN
  IMPORTING reg[T]

  okreg(rexp: reg): okreg?
  emptyreg: emptyreg?
  nomatchreg: nomatchreg?
 END ereg
```

Define the length of a regular expression by counting the number of atomic events and constructors. The function `pop` will remove the leftmost event from

a regular expression, if this event matches the given event `y`, and return what remains of the regular expression. In the case of disjunction, it will chose the branch where the leftmost event matches `y`, or return a new disjunction if there is a match in both branches. When we have a successful match, the new regular expression `r` is returned as an `ereg`, using the constructor `okreg(r)`. We distinguish the case where an empty regular expression is used as an argument, where `emptyreg` is returned, from the case where `y` does not match any branch of the regular expression and `nomatchreg` is returned.

The `prs` function is true for a trace `t` and a regular expression `r` if `t` is a prefix of a trace described by `r`. This is determined by means of the `pop` function.

```
reg_exp[T: TYPE]: THEORY
 BEGIN

  IMPORTING ereg[T], seq_lemmas[T]

  length(r: reg[T]): RECURSIVE nat =
    CASES r
      OF rempty: 0,
         single(x): 0,
         AND(e1, e2): length(e1) + length(e2) + 1,
         OR(e1, e2): length(e1) + length(e2) + 1,
         star(e): length(e) + 1
      ENDCASES
     MEASURE reduce_nat(0, (LAMBDA (x: T): 0),
                        (LAMBDA (n1, n2: nat): n1 + n2 + 1),
                        (LAMBDA (n1, n2: nat): n1 + n2 + 1),
                        (LAMBDA (n: nat): n + 1))

  pop(y: T, e: reg): RECURSIVE ereg =
    CASES e
      OF rempty: emptyreg,
         single(x): IF x = y THEN okreg(rempty)
                             ELSE nomatchreg
                    ENDIF,
         AND(e1, e2):
           LET f1 = pop(y, e1) IN
             IF emptyreg?(f1)
               THEN pop(y, e2)
             ELSIF nomatchreg?(f1)
                   THEN nomatchreg
                   ELSE okreg(AND(rexp(f1), e2))
             ENDIF,
```

```
       OR(e1, e2):
         LET f1 = pop(y, e1), f2 = pop(y, e2) IN
           IF emptyreg?(f1) OR nomatchreg?(f1)
             THEN f2
           ELSIF emptyreg?(f2) OR nomatchreg?(f2)
                 THEN f1
                 ELSE okreg(OR(rexp(f1), rexp(f2)))
           ENDIF,
       star(e1):
         LET f1 = pop(y, e1) IN
           IF emptyreg?(f1)
             THEN emptyreg
           ELSIF nomatchreg?(f1)
                 THEN nomatchreg
                 ELSE okreg(AND(rexp(f1), star(e1)))
           ENDIF
    ENDCASES
   MEASURE length(e)

prs?(s: seq, e: reg): RECURSIVE bool =
  CASES s
    OF empty: TRUE,
       addr(q, y):
         LET z = lt(s), q = rr(s), f = pop(z, e) IN
           IF emptyreg?(f) OR nomatchreg?(f)
             THEN FALSE
           ELSE prs?(q, rexp(f))
           ENDIF
    ENDCASES
   MEASURE length(s)

PRSdistr1: LEMMA
  FORALL (r: reg[T]): and?(r) IMPLIES AND(fst(r), snd(r)) = r

PRSdistr2: LEMMA
  FORALL (t: seq[T], r: reg[T]):
    (nonempty?(t) AND prs?(t, r)) IMPLIES NOT (rempty?(r))

PRSlemma1: LEMMA
  FORALL (t: seq[T], r: reg[T]):
    (nonempty?(t) AND prs?(t, r))
       IMPLIES prs?(rr(t), rexp(pop(lt(t), r)))

PRSlemma2: LEMMA FORALL (r: reg[T]): prs?(empty, r)

PRSlemma3: LEMMA
  FORALL (t: seq[T], r: reg[T]):
    (nonempty?(t) AND prs?(t, r))
       IMPLIES NOT (emptyreg?(pop(lt(t), r)))
```

```
  PRSlemma4: LEMMA
    FORALL (x: T, t: seq[T], r: reg[T]):
      (nonempty?(t) AND prs?(addr(t, x), r)) IMPLIES
       NOT (emptyreg?(pop(lt(t), r)))

  PRSlemma5: LEMMA
    FORALL (t: seq[T], r: reg[T]):
      (nonempty?(t) AND prs?(t, r))
         IMPLIES NOT (nomatchreg?(pop(lt(t), r)))

  PRSlemma6: LEMMA
    FORALL (x: T, t: seq[T], r: reg[T]):
      (nonempty?(t) AND prs?(addr(t, x), r)) IMPLIES
       NOT (nomatchreg?(pop(lt(t), r)))

  PrefixLemma: LEMMA
    FORALL (q: seq), (e: reg), (y: T): prs?(addr(q, y), e)
      IMPLIES prs?(q, e)

  PrefixLemma2: LEMMA
    FORALL (s, t: seq, r: reg): (ord?(s, t) AND prs?(t, r))
      IMPLIES prs?(s, r)
 END reg_exp
```

G Reader and Writer Interfaces

In this section, we present the PVS encoding of the examples of Section 7. To represent an actual OUN declaration in PVS, we commence by inhabiting the types `Object` and `Method`. We proceed by defining the events of the alphabet. The examples provided here present different predicates used for assumptions and invariants. When such predicates have been defined, the OUN specifications of all these examples are constructed schematically. In Appendix G.2, the PVS theory for regular expressions is imported to provide support for `prs`-predicates in the specifications.

G.1 The Reader Specification

```
Reader: THEORY
 BEGIN

  IMPORTING oun

  Read: Method

  object: Object

  AsmReader: AsmPred =
    (LAMBDA ((tr: Trace), (o1, o2: Object)): TRUE)
```

```
  InvReader: InvPred = (LAMBDA ((tr: Trace), (O: Object)): TRUE)

  AlphaReader: set[Event] =
      {e: Event | to(e) = object
                    AND name(e) = Read AND input(e) = null}

  TracesReader: set[Trace] =
      restrict({h: seq[Event] | restrict(h, AlphaReader) = h})

  TracesReader2: set[Trace] =
      AsmInvTraceSet(AlphaReader, AsmReader,
                       InvReader, object, emptyset)

  ObjectReader: OUNSpecification =
      (# alpha := AlphaReader,
         traces := extend[seq[Event], Trace, bool, FALSE](TracesReader2),
         obj := singleton(object) #)

  ReaderLemma: LEMMA TracesReader = TracesReader2
 END Reader
```

G.2 The Writer Specification

```
Writer: THEORY
 BEGIN

  IMPORTING oun

  Open_write: Method

  Write: Method

  Close_write: Method

  object: Object

  OpenWriteEvents: set[Event] =
      {e: Event |
         to(e) = object AND name(e) = Open_write
          AND input(e) = null AND output(e) = null}

  CloseWriteEvents: set[Event] =
      {e: Event |
         to(e) = object AND name(e) = Close_write
          AND input(e) = null AND output(e) = null}

  WriteEvents: set[Event] =
      {e: Event | to(e) = object AND
                     name(e) = Write AND output(e) = null}
```

```
 AlphaWriter: (balanced?) =
     union(union(OpenWriteEvents, WriteEvents),
           CloseWriteEvents)

 AbstractEvent: TYPE = [# name: Method, kind: EvtKind #]

 IMPORTING reg_exp[AbstractEvent]

 lift(e: Event): AbstractEvent =
   (# name := name(e), kind := kind(e) #)

 liftseq(t: seq[Event]): RECURSIVE seq[AbstractEvent] =
   CASES t OF empty: empty,
              addr(t1, m): addr(liftseq(t1), lift(m))
     ENDCASES
    MEASURE length(t)

 AsmWriter: AsmPred =
     (LAMBDA ((tr: Trace), (o1, o2: Object)):
        prs?(liftseq(tr),
             AND(AND(AND(AND(AND(single((# name := Open_write,
                                         kind := i #)),
                                 single((# name := Open_write,
                                         kind := c #))),
                             single((# name := Write, kind := i #))),
                         single((# name := Write, kind := c #))),
                     single((# name := Close_write, kind := i #))),
                 single((# name := Close_write, kind := c #)))))

 InvWriter: InvPred =
     (LAMBDA ((tr: Trace), (O: Object)):
        prs?(liftseq(restrict_kind(tr, c)),
             AND(AND(single((# name := Open_write, kind := c #)),
                     single((# name := Write, kind := c #))),
                 single((# name := Close_write, kind := c #)))))

 TracesWriter: set[Trace] =
     AsmInvTraceSet(AlphaWriter, AsmWriter,
                    InvWriter, object, emptyset)

 ObjectWriter: OUNSpecification =
     (# alpha := AlphaWriter,
        traces := extend[seq[Event], Trace, bool, FALSE](TracesWriter),
        obj := singleton(object) #)
END Writer
```

G.3 The ReaderWriter Specification

```
ReaderWriter: THEORY
 BEGIN

  IMPORTING Writer

  Open_read: Method

  Read: Method

  Close_read: Method

  OpenReadEvents: set[Event] =
      {e: Event |
         to(e) = object AND name(e) = Open_read
          AND input(e) = null AND output(e) = null}

  CloseReadEvents: set[Event] =
      {e: Event |
         to(e) = object AND name(e) = Close_read AND
          input(e) = null AND output(e) = null}

  ReadEvents: set[Event] =
      {e: Event | to(e) = object AND name(e) = Write AND output(e) = null}

  AlphaReader: (balanced?) =
      union(union(OpenReadEvents, ReadEvents), CloseReadEvents)

  AlphaRW: (balanced?) = union(AlphaReader, AlphaWriter)

  AsmRW: AsmPred =
      (LAMBDA ((tr: Trace), (o1, o2: Object)):
         prs?(liftseq(tr),
             OR(AND(AND(AND(AND(AND(single
                                    ((# name := Open_write, kind := i #)),
                                    single
                                    ((# name := Open_write, kind := c #))),
                                single((# name := Write, kind := i #))),
                            single((# name := Write, kind := c #))),
                        single((# name := Close_write, kind := i #))),
                    single((# name := Close_write, kind := c #))),
                AND(AND(AND(AND(AND(single
                                    ((# name := Open_read, kind := i #)),
                                    single
                                    ((# name := Open_read, kind := c #))),
                                single((# name := Read, kind := i #))),
                            single((# name := Read, kind := c #))),
                        single((# name := Close_read, kind := i #))),
                    single((# name := Close_read, kind := c #))))))
```

```
 InvRW: InvPred =
     (LAMBDA ((tr: Trace), (O: Object)):
        (length(restrict(tr, {e: Event | name(e) = Open_read})) -
          length(restrict(tr, {e: Event | name(e) = Close_read}))
          = 0)
         OR
         (length(restrict(tr, {e: Event | name(e) = Open_write})) -
           length(restrict(tr, {e: Event | name(e) = Close_write}))
           = 0))

 TracesRW: set[Trace] =
     AsmInvTraceSet(AlphaRW, AsmRW, InvRW,
                    object, singleton(ObjectWriter))

 ObjectRW: OUNSpecification =
     (# alpha := AlphaRW,
        traces := (extend[seq[Event], Trace, bool, FALSE]((TracesRW))),
        obj := singleton(object) #)

 RWrefinesW: LEMMA refines?(ObjectRW, ObjectWriter)
END ReaderWriter
```