A Proof Environment for Partial Specifications in OUN

Einar Broch Johnsen and Olaf Owe
Department of informatics, University of Oslo

Abstract

Aspect-oriented specifications and formal reasoning are often advocated for the design of complex, distributed systems. Commonly used design notations seem to lack the reasoning control of formal languages. OUN is an object-oriented language with facilities for aspect-oriented specifications through multiple viewpoints and for reasoning control. However, to be of practical use the language needs tool support. In this paper, we propose a formalization of aspects of the observable behavior of objects, using the PVS proof system. The formalism is extended with constructs to provide reasoning support for OUN specifications.

1 Introduction

The Oslo University Notation (OUN) is a high-level object-oriented design language specialized towards the development of open distributed systems. The language is designed in a restricted way so that reasoning control is based on static typing and proofs, and the generation of verification conditions is based on static analysis of pieces of programs or specification units [5]. In the language, different behavioral aspects of an object can be specified using OUN interfaces. An object may support a number of interfaces and this number may change dynamically. The semantics of the language is based on traces. Requirement specifications of interfaces are given as assumptions and invariants, by means of first-order predicates on traces. In this paper, we develop a proof environment for OUN specifications in the Prototype Verification System (PVS) [8].

We consider objects running in parallel and communicating by remote method calls. The observable behavior of an object at a point in time is given by its *trace*, i.e. the sequence of observable communication events, reflecting remote method calls both to and from the current object. At any given time, the number of such calls will be finite, so the trace is also finite. The set of traces that reflect all possible runs of an object, give us an observational description of the object. Internal activity is not captured directly in the observable communication traces, but such internal activity may manifest itself at the level of communication traces as non-deterministic choices, i.e. a trace may have several extensions in the trace set.

A specification of an object includes an alphabet of communication events and a set of traces over this alphabet. When an object is specified at a given level of abstraction, some lower level details concerning its behavior are ignored in order to focus the specification on a relevant aspect of the behavior of the object. By generalization, specifications can

describe the behavior of a component that encapsulates several objects. In particular, the composition of two specifications is itself a specification.

There may be several specifications of the same object, describing communication traces built over different subsets of the alphabet of that object. These so-called partial specifications correspond to different roles of the object. We employ a refinement relation that allows alphabet expansion, introduced in OUN [2, 5]. Thus, various (partial) specifications of an object may have a common refinement, although their alphabets differ.

PVS is a specification and verification environment with mechanized support for formal verification in a specification language based on higher-order logic [8]. The language has a rich type system including predicate subtypes and dependent types, as well as type constructors for functions, tuples, records and abstract datatypes [7]. PVS specifications are organized in theories. Modularity and reuse are supported by means of parameterized theories. PVS has a powerful verification tool which uses decision procedures to simplify and discharge proofs, and provides proof techniques such as induction, rewrite, backward and forward proofs, and proof by cases for interactive user intervention.

In this paper, a PVS proof environment is proposed for aspect-oriented object specifications based on finite communication traces. The proof environment is then extended with the syntactic sugaring of OUN, and we show by examples how OUN specifications can be translated in a fairly straightforward manner into PVS theories. The proof environment gives machine support for formal reasoning about OUN specifications. For brevity, only OUN interfaces are discussed here, other constructs such as contracts are treated in an extended version of this paper [4].

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, to be defined later. The finite sequences over T are defined as an abstract datatype in PVS.

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

This definition gives us selector functions `lr` and `rt` as well as identifiers `empty?` and `nonempty?` and constructors `empty` and `addr`. Several properties of abstract datatypes are available to the user, constructed by the PVS theorem prover itself [6], such as

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

Functions on the finite sequences can be defined by induction over the constructors of the abstract datatype. Let `length` be a predicate on sequences defined in this manner. The concatenation of two sequences is defined by a predicate

```
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)
```

Notice the definition by cases construct and the measure, which generates TCC's (type correctness conditions, see [8]) to guarantee that the definition is well-founded. The length of a sequence is used as a measure in recursive definitions. Finite sequences can be

extended at either end. Define by `addl` the complementary constructor to `addr`, with selectors `lt` and `rr`. We need projections `restrict` and `hide` on the sequences of type *T* to restrict the sequences to subtypes of *T*. The function `restrict` is defined as

```
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)
```

We define a prefix ordering on sequences:

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

The abstract datatype construct of PVS gives us a notion of equality, and we can prove properties of equality and distribution for the defined functions, for instance

```
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)
```

3 Encoding Partial Specifications

Specifications in OUN consist of behavioral constraints on communication patterns between object identifiers. When aspects of objects are specified, there may be several specifications of the same object identifier. In this section, we propose PVS constructs for the basic notions of aspect-oriented specifications with refinement and composition, using finite communication traces. Then we present restrictions to obtain a compositional refinement property for the formalism, following previous work [3].

A communication event in our setting is a structured event reflecting a remote method call from one object to another. Let `Object` and `Method` be types in PVS. Events are defined as structured triples that consist of two object identifiers and a method name. For structured triples, we use the PVS record type:

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

The fields of the triple are labeled `caller`, `to`, and `name`, respectively. These are used to access the different fields; if `e` is an event, `name(e)` accesses its method name.

Prefix closed sets of communication traces correspond to safety specifications [1]. Define

```
prefixclosure: [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)))
```

Consider a record type `Triple`, which consists of the fields `alpha:set[Event]`, `traces:(prefixclosure)`, and `obj:set[Objects]`. By `(prefixclosure)`, we denote the subtype of `set[seq[Event]]` which satisfies the prefix closure predicate. Now, a specification is a subset of `Triple`, 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))))}
```

Internal communication between the objects of a specification is not reflected in its alphabet. We call a specification Γ an *interface specification* if `obj(Γ)` is singleton.

OUN offers a notion of refinement with alphabet expansion, which is particularly suitable for aspect-oriented specifications, as the relation allows multiple inheritance of specifications through projection on alphabets. Thus, we can join different aspects of an object through refinement. For a discussion of refinement for aspect-oriented specifications using this formalism, see [3]. Refinement for specifications is defined as

```
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)))
```

Composition encapsulates object identifiers, hiding internal communication events. For a set of objects, the internal communication events are given by a function `int`, defined as

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

Define composition of specifications by hiding internal communication and by projection on the traces of the components:

```
comp(S1, S2: Specification): Specification =
 (# alpha := {x: Event |
            member(x, difference(union(alpha(S1), alpha(S2)),
                                 int(union(obj(S1), obj(S2)))))},
    traces := {t: seq[Event] | restrict(t, alpha) = 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 observable alphabet of the specifications. 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))))
```

Commutativity and associativity of composition holds for *composable* specifications. In a refinement step, new object identifiers can be introduced 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. This difficulty is avoided by a `properness` criterion:

```
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)))
```

We can now state and prove the following compositional refinement property in PVS:

```
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 Interface Declarations

In this section, we briefly present a syntax for OUN interfaces. For a motivational discussion of the language, the reader is referred to [5]. Objects support behavioral interfaces that correspond to different roles. Different objects may have the same role, so interfaces are declared independently of their objects and objects that support the same interfaces can be used in the same places.

4.1 Asynchronous Communication by Remote Method Calls

In distributed systems, communication between objects is asynchronous. Hence, in the language, each remote method call is represented by two distinct events in the traces, referred to as the initiation and the completion of the call, respectively. In OUN, events contain information about input and output values as well as about the kind of event we are dealing with, either initiation or completion, the identities of the calling and receiving objects, and the method name. In the syntax, an operation is declared as

$$\textbf{opr}\ m(\textbf{in}\ p_1{:}\,T_1,\ldots,p_i{:}\,T_i;\textbf{out}\ p_j{:}\,T_j,\ldots,p_n{:}\,T_n).$$

If an object o_1 offers this method m to its environment, and m 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, this 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 are no values after the semicolon. As communication events reflect remote method calls, we refer to o_2 as the caller and to o_1 as the receiver of both $o_2 \rightarrow o_1.m(\ldots)$ and $o_2 \leftarrow o_1.m(\ldots)$. Let $\textbf{caller}(o)$ be the set of events corresponding to remote calls that are called by an object o and $\textbf{receiver}(o)$ the corresponding events received by o. Internal activity in the objects are not directly observable. Hence, $\textbf{caller}(o) \cap \textbf{receiver}(o) = \emptyset$. Global traces are communication histories for the entire system we consider, whereas local traces are restricted to a subset of the communication events. As communication is asynchronous, the calling object need not wait for the completion of a call. Hence, in the (local) traces, other communication events may occur between the initiation and completion events of a call. Synchronous communication can be specified explicitly by a special event $o_2 \leftrightarrow o_1.m(p_1,\ldots,p_i;p_j,\ldots,p_n)$, here the completion immediately follows the initiation.

4.2 The Syntax and Semantics of Interfaces

An interface contains syntactic definitions of operations and semantic requirement specifications. The semantic requirement has the form of an assumption invariant specification: The invariant is guaranteed to hold when the assumption is respected by the objects in the communication environment.

At the semantic level, objects are typed by interfaces. The semantic requirements of an interface rely on the available communication history of an object offering the interface up to the present point in time; they are predicates on the finite traces. Let $F, F_1, \ldots, F_m$, and G represent interfaces. An interface declaration will then have the following general form:

interface F (<parameters>)
  inherits $F_1, F_2, \ldots, F_m$
begin
  with $x{:}G$
    opr $m_1(\ldots)$
      $\vdots$
    opr $m_n(\ldots)$
  asm <formula on local trace restricted to a calling object>
  inv <formula on local trace>
where <auxiliary function definitions>
end

The parameter list contains values (typed by datatypes) or objects (typed by interfaces) that describe the minimal environment needed by an object offering the interface at the point of creation. The **with** clause of the interface F is used to restrict the communication environment of an object o offering F, to objects offering some interface G. Thus, o has knowledge of some methods of the calling object, as specified in the interface G. When no such knowledge is required and access is open to all objects in the environment, the **with** clause is omitted. An interface declared in a **with** clause is referred to as a cointerface.

In OUN, $o{:}F$ denotes an object o which offers an interface F (as above, but for given parameters) to the environment. For $o{:}F$, we derive an alphabet $\alpha(o{:}F)$ of communication events from the syntactic declaration of F. To each method, associate initiation and a completion events, ranging over possible input and output values and possible callers. The alphabets of objects included as parameters to F, are included in $\alpha(o{:}F)$. If F has a cointerface G, we include the events of $\alpha(o'{:}G)$ that are available to o, i.e. $[\bigcup_{o'\in \texttt{Object}} \alpha(o'{:}G) \cap \mathbf{caller}(o)] \subseteq \alpha(o{:}F)$. The alphabet of $o{:}F$ is maximal, including all possible callers, although the communication environment of an object actually evolves over time, due to information about calling objects and object identifiers transmitted as input values to method calls.

The assumption predicate is a requirement on the environment, expected to hold for local traces restricted to one caller at a time. Hence, in the declaration of an interface, the assumption predicate ranges over traces as well as calling and receiving objects (of the methods declared in the interface). Since assumptions are the responsibility of the environment, these are only expected to hold for traces that end with input to the object offering the interface.

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) if it is not an input to o. 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 A in an interface F, offered by an object o, we define

$$A^{in}(h,o) = \forall x \neq o : A(\mathbf{in}_o(h/\alpha(o{:}F)/x), o, x),$$

and a similar predicate $A^{out}(h,o)$, hiding inputs to o by $\mathbf{out}_o(h)$. When we specify an assumption A in an interface offered by an object o, $A^{in}(h,o)$ is assumed to hold. Now define $I^{out}(h)$ for invariant predicates I as:

$$I^{out}(h,o) = I(\mathbf{out}_o(h/\alpha(o{:}F))).$$

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 we consider, 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 the same predicate as the invariant. If auxiliary functions or predicates are needed for specification purposes, these may be defined following the **where** keyword.

Traces are used explicitly in interface declarations to determine behavior at specific points in time. The behavior of an object offering the interface is then described by a set T of possible (finite) traces. For every trace h in the set T, all prefixes of h represent prior points in the life of the object and must also belong to T, so T is prefix closed and represents a safety specification in the sense of Alpern and Schneider [1].

When an interface F inherits interfaces $F_1,\ldots,F_m$ $(1 \leq i \leq m)$ 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. The trace set $T\,(o{:}F)$ of $o{:}F$, where A and I are the assumption and invariant predicates of F, is then the prefix closure of

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

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

5 Embedding OUN Specifications in PVS

In this section, a richer theory for communication events is introduced in PVS to reflect remote method calls in OUN. The OUN notion of a trace is formalized, and a framework for representing interface declarations in PVS is introduced. We discuss the representation of interface specifications and of objects offering OUN interfaces.

5.1 Communication Events for Remote Method Calls

The `Event` type of Section 3 needs to be modified to include all the information of OUN events. To commence, let `Interface` be a type for interface names in PVS. A type is introduced for different kinds of events, i.e. initiation and completion, `EvtKind: TYPE = {i,c}`. A uniform type is used for the possible data values transmitted as either input or output values to the communication events. Here, transmitted data values are either natural numbers or objects typed by interface. The (disjoint) union of two types cannot be constructed directly in PVS [6]. Instead, they are wrapped into a new type, using different constructors and identifiers for the different subtypes. Define an abstract datatype `Data`:

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

The modified type `Event` is a subtype of

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

such that the list of output data is null for initiations and that the caller and receiver of an event are different objects. Checking that the number and types of input as well as output values are correct for any actual method declaration is implicit in the alphabet declarations. All the results of Section 3 hold for the modified `Event` type.

5.2 A Theory of Traces

For OUN traces, 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`.) 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. 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 `h` over a set of events *S* is *causal* if, for every balanced subset of *S*, initiation events occur more often than not in all prefixes of the trace, formalized by

```
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`. 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, the *causal* subtype of `Seq[Event]`, as follows:

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

The empty sequence satisfies the causality predicate, so it inhabits `Trace`. The largest input and output prefixes are defined by recursion and denoted `in_prefix` and `out_prefix`, respectively.

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 $T(o{:}F)$. We can therefore interpret $o{:}F$ as an interface specification in the sense of Section 3. Whereas the alphabet may be translated into a PVS set of events 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. First, 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 corresponding subtype of `Specification`:

```
OUNspec: 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]
```

Furthermore, we want these predicates to be true for the empty trace. For each inherited interface, we assume that the alphabet is included by construction. The traces of the new interface must belong to the trace set of an inherited interface, after appropriate restriction. This requirement is defined as a predicate on traces:

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

Using 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[OUNspec]): 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 above:

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

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

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

An interface F that has not been associated with a particular object identifier, can be represented by the union of all (possible) objects offering F. Thus, F is represented by the specification

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

With this representation of interfaces, we can reason about refinement and composition directly, even before it is decided which objects shall actually offer an interface. Remark that other parameters to interface declarations can be treated in the same manner.

6 A Theory for Graphically Oriented Specifications

A convenient predicate for safety specifications is the predicate that defines prefixes of regular expressions, because the set defined by such a predicate is prefix closed. A theory for such predicates is now introduced and will be used in the examples of Section 7 in order to express assumption and invariant predicates. Regular expressions can be defined as an abstract datatype in PVS, parameterized by a 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 a predicate `prs(t,exp)` which is true if the trace `t` is a prefix of a trace described by the regular expression `exp`. To represent error situations in the recursion, the regular expressions are wrapped into another datatype:

```
ereg[T: TYPE]: DATATYPE
 BEGIN
  okreg(rexp: reg): okreg?
  emptyreg: emptyreg?
  nomatchreg: nomatchreg?
 END ereg
```

We can then define the function `prs` 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` has been removed (from the left). The `prs` predicate gives a graphical specification style, as demonstrated by its prefix closure:

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

7 Examples

We derive PVS specifications from two OUN interfaces *Writer* and *ReaderWriter*, that describe an object controlling read and write access to some shared data, following [2]. For *Writer*, access is restricted so that only one object in the environment may perform write operations at the time. The interface *ReaderWriter* also allows concurrent read operations. In PVS, we prove that *ReaderWriter* refines *Writer*.

7.1 A Writer Interface in PVS

The *Writer* interface limits write access to one object at the time. A calling object must first invoke a method *Open_write* in order to obtain write access. To return access control, the calling object invokes the method *Close_write*. In OUN, the *Writer* interface can be declared as follows:

interface $Writer[T$: Data-type$]$
begin
  opr $Open\_write()$
  opr $Write(d : T)$

opr $Close\_write()$
asm $A(h,o,x) = h$ **prs** $(\leftrightarrow.open\_write \leftrightarrow.write^* \leftrightarrow.close\_write)^*$
inv $I(h,o) = (h/\leftarrow)$ **prs** $(\leftarrow.open\_write \leftarrow.write^* \leftarrow.close\_write)^*$
end

In PVS, let `object` have type `Object` and let `Open_write`, `Write`, and `Close_write` be of type `Method`. Assume that `object` supports *Writer*, and define `AlphaW` of type `(balanced?)` to be the events with these method names and receiver `object`.

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, so define a type `AbstractEvent` that only includes these fields, import the regular expressions over this type into the current theory, and define a function `liftseq` that turns a trace into a sequence of `AbstractEvent`. The assumption and invariant predicates are similar, except that the invariant only considers completion events. It can be defined as

```
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 #)))))
```

The trace set `TracesW` can be obtained by `AsmInvTraceSet`, and we get:

```
ObjectW: OUNspec = (# alpha:= AlphaW, traces:= (TracesW),
                      obj:= singleton(object) #)
```

7.2 A ReaderWriter Interface in PVS

The *ReaderWriter* interface allows concurrent read access but only sequential write access. In addition, an object can perform read operations when granted write access to the shared data. The methods and behavior of *Writer* are inherited.

interface $ReaderWriter[T : \text{Data-type}]$
inherits $Writer[T]$
begin
opr $Open\_read()$
opr $Read(\text{out } d : T)$
opr $Close\_read()$
asm $A(h,o,x) = h$ **prs** $(\leftrightarrow.open\_write \leftrightarrow.write^* \leftrightarrow.close\_write$
$\mid \leftrightarrow.open\_read \leftrightarrow.read^* \leftrightarrow.close\_read)^*$
inv $I(h,o) = \sharp(h/\leftarrow.open\_read) - \sharp(h/\leftarrow.close\_read) = 0$
$\vee\ \sharp(h/\leftarrow.open\_write) - \sharp(h/\leftarrow.close\_write) = 0$
end

In PVS, we import the theory for the *Writer* interface and declare `Open_read`, `Read` and `Close_read` to be of type `Method`. Then, construct a set of events `AlphaR` corresponding to these method names. The alphabet of *ReaderWriter* becomes `AlphaRW: (balanced?) = union(AlphaR, AlphaW)`. 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(ObjectW))
```

We get an OUN specification for the `object` supporting the *ReaderWriter* interface, which is denoted `ObjectRW`. Finally, we can prove in PVS that

```
RWrefinesW: LEMMA refines(ObjectRW, ObjectW)
```

8 Conclusion

For any practical purposes, a formal specification language needs support by a proof environment in order to allow 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 is shown to support compositional refinement of specifications. We extend the proof environment with constructs to facilitate its use as a formal reasoning environment for OUN, a formal specification language for the development of open distributed systems.

The main difficulty associated with translating OUN specifications into the formalism of the proof environment now arise from the representation of assumption and invariant predicates. A theory for prefixes of regular expressions is added to the formalism as a convenient way to define predicates, allowing more graphically oriented specifications. 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] O.-J. Dahl and O. Owe. Formal methods and the RM-ODP. Technical Report 261, Department of Informatics, University of Oslo, 1998.

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

[4] E. B. Johnsen and O. Owe. A PVS proof environment for OUN. Technical Report 295, Department of informatics, University of Oslo, 2001. The PVS theories are available at `http://www.ifi.uio.no/~einarj/OunPvs.tar.bz2`.

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

[6] S. Owre and N. Shankar. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, June 1997.

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

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