Interface-Based Protocol Specification of Open Systems using PSL

Doug Lea[1] and Jos Marlowe[2]

[1] SUNY at Oswego / NY CASE Center
[2] Sun Microsystems Laboratories

Abstract. PSL is a framework for describing dynamic and architectural properties of open systems of components. PSL extends established interface-based tactics for describing the functional properties of open systems to the realm of protocol description. PSL specifications consist of logical and temporal rules relating *situations*, each of which describes potential states with respect to the *roles* of components, role attributes, and the issuance and reception of events. A specialized form, PSL/IDL supports protocol description in CORBA systems.

1 Open Systems

Open systems have been defined[28, 1, 65] as encapsulated, reactive, and spatially and temporally extensible systems: They are composed of possibly many *encapsulated* components, each of which is normally described as an *object*. Each component supports one or more public *interfaces*. The inner workings of each component are separately specified via a normally inaccessible *implementation* description (often a *class*). Open systems are *reactive* in that they do not just perform a single one-time service; most functionality is provided "on-demand", in reaction to a potentially endless series of requests. Open systems are *spatially extensible* when components need not bear fixed connectivity relations among each other. They may interact via message passing mechanisms ranging from local procedure calls to multi-threaded invocations to asynchronous remote communication. And open systems are *temporally extensible* to the extent to which their components and functionality may be altered across a system's lifetime, typically by adding new components and new component types (perhaps even while the system is running) and/or updating components to support additional functionality.

For example, an open system supporting financial trading might include a set of components that "publish" streams of stock quotes, those that gather quotes to update models of individual companies, those serving as traders issuing buys and sells of financial instruments, and so on. Such a system possesses each of the above features: It relies on interfaces (e.g., `QuotePublisher`) with multiple implementations. It contains reactive components handling a constant influx of messages. It is intrinsically distributed in order to deal with world-wide markets. And it may evolve in several ways across time; for example to accommodate a new company listed on an exchange, or to replace components computing taxes with interoperable versions reflecting tax law changes.

2 Interfaces

Interface declarations provide a basis for specifying capabilities in open systems at varying levels of precision and formality. An interface describes only those services that other components may depend on, in terms of a set of constraints (e.g., a collection of required operation signatures[8]) on a family of components, not necessarily a complete or closed (nonextensible) description of any given component. Any implementation(s) that provide requisite functionality may be used. Subtyping regimes allow one interface to be described as an extension, refinement, or specialization of one or more superinterfaces. Conversely one interface may abstract a subset of the functionality described in one or more subinterfaces. Typically, base interfaces describe only those operations that are involved in a set of related interactions, and "fatter" interfaces are derived via multiple inheritance[52]. At an extreme, an interface might include only a single service operation. Interface subtyping allows systems to include multiple interoperable implementations of common services, and sets of component types that specialize on common core features.

Systems constructed according to the OMG CORBA [51] model are among the best examples of interface-based development. In CORBA, component interfaces are described in CORBA IDL, an interface description language indicating only the functional properties of provided services in terms of operation signatures. Implementations are often process-level components distributed across machines, communicating via message passing mechanisms mediated by an object request broker (ORB) that directs requests to their destinations.

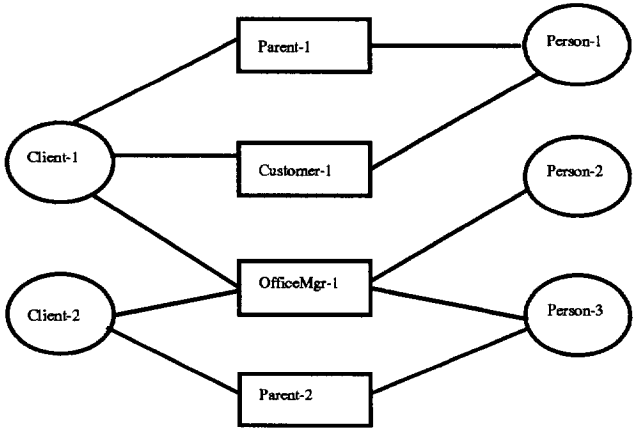
Roles. While not always phrased as such, interface-based specification intrinsically relies upon the distinction between *roles*[66, 59, 6, 18] and the components that implement those roles. This notion follows ordinary usage distinguishing between a role in a particular play performance (e.g., *Hamlet*) and the actor or actors playing the role (e.g., *Richard Burton* or *Lawrence Olivier*). While the term "role" is sometimes applied to a description of a role rather than its instantiation, we reserve the term "interface" for descriptions, and just "role", or for emphasis, "role instance" for instantiations. A role may be thought of as a typed, abstract *access channel*[63, 4] providing a *subjective view*[62, 25] of one or more interaction participants[33], where each view is described by an interface.

Interface specifications deal only with roles, not *implementation objects*. In fact, implementations need not be conceptualized as objects in the usual narrow sense of the term. They may also consist of finer-grained instantiations (activations) of single procedures, or coarser-grained process-level components. Role-based development methods enforce an opaque layering of roles on top of implementation-level objects, in just the same sense that common object-oriented methods in turn enforce an opaque layering of objects on top of machine-level memory cells, CPUs, etc.

In classic modular development methods, there is typically exactly one implementation type per interface type and vice versa. In most object-oriented

languages, there may be multiple implementation subclasses per interface. But in general, interface-based specification admits a *many-to-many* relationship between roles and objects: One implementation object may support several otherwise unrelated interfaces concurrently, or different interfaces across time. Conversely, one role may be implemented by several otherwise unrelated objects connected to the same channel. Each may implement a subset of operations, or redundantly implement all of them. CORBA systems support such mappings by reifying some aspects of roles as "interface objects" that relay messages to implementations (see Section 11).

For example, an `OfficeManager` role might be implemented using a `Person` object, or a `Robot` object, or a set of job-sharing `Persons`, or even a mechanism causing a new `SpecialtyWorker` object to be constructed to handle each incoming service request. A `Person` object might additionally play the roles (hence "export" the interfaces) of `Parent`, `Driver`, `Customer`, etc. perhaps all at the same time. Multiple roles, each seen from a different client, may be associated with the same `Person` implementations; for example (using boxes for roles and ellipses for objects):



3 Extending Interface Specification

Common interface-based specification techniques are limited in their ability to deal with dynamics, architecture, and quality of service. An interface alone defines only "syntactic" conformance: Implementations must respond to the listed operations. An interface adorned with descriptions of the nature of operation arguments and results defines per-operation functional conformance. But even this does not always provide sufficient information for designers and implementors to ensure that components in open systems behave as desired. A new interface-compatible component may not actually interoperate with an existing one if it fails to send the same kinds of messages to other components in the course of providing services. Such concerns often overwhelm those surrounding service functionality[21, 20, 27].

The remainder of this paper describes PSL, a framework for specifying the dynamic and architectural properties of open systems. An associated technical report [39] contains additional syntactic details, examples, and elaborations omitted from this paper because of space limitations.

PSL adds four kinds of specification constructs to interfaces: Abstract *attributes* are descriptions of properties that may vary across time. These include the special properties `in` and `out` describing the reception and issuance of messages. *Situations* are contexts in which attributes assume particular values. *Protocols* describe required temporal relationships among the instances ("realizations") of situations. *Constraints* describe invariant relationships among the values of attributes.

PSL differs from most other approaches to specification (see Section 12) by mirroring the intrinsic *incompleteness* of open systems. PSL specifications document sets of properties and constraints without claiming that these fully describe a system, or ruling out the existence of unmentioned components, situations, or events. Thus, PSL lacks some familiar constructs found in "closed-world" formalisms. In particular: (1) PSL does *not* support the use of *frame axioms* asserting that properties that are not otherwise mentioned in one situation are not changed from those in predecessors. (2) PSL does *not* contain *step operators* asserting that given states occur at the "next time step" after others; thus that there are no intervening situations. (*Step* is also implicit in the description of transitions in most state models.) (3) PSL does *not* support *leaf interfaces* asserting that interfaces have no possible extensions; thus that instances do not support additional operations. (4) PSL does *not* contain any notion of a *leaf state* asserting that a situation has no possible subsituations; thus that no further state decomposition or unmentioned events exist. These omissions make way for the kinds of refinement and extensibility necessary in the development of open systems.

Types and Interfaces in PSL. As a basis for specification, PSL relies on any framework for declaring interfaces, operations, and pure value types such as integers, records, etc. Rather than establishing a novel syntax, we illustrate using PSL/IDL, a concrete syntax for PSL with CORBA IDL value and interface types and C++-style expressions. All PSL/IDL constructs appear as annotations within `protocol module` declarations. Protocol modules have the syntactic properties of IDL `modules` but contain only type declarations, attribute function declarations, and/or `rules` declarations that consist of an optionally named parameterized scope containing one or more rules. Because textual representations of even simple protocol rules stretch the limits of readability and writability, we simultaneously define PSL/IDL-G, a semigraphical form corresponding in simple ways to the textual version.

PSL/IDL uses C++ expression syntax over CORBA IDL value types. Predicates are `boolean` expressions. For convenience, we add the logical *implies* operator ("`-->`") and its left-sided version, *is implied by* ("`<--`") to the list of boolean operators. Within expressions, commas may be used as low-precedence

and separators. To make them more useful in specifications, PSL/IDL predefines common pure value functions (`head`, `tail`, `empty`, `contains`, `prepend`, `append`, `concat`, `equal`, `remove`) on IDL `sequence` and `string` types.

Handles. A role may have finite temporal existence. However, the lifetime of a role need not be coextensive with that of any given implementation object. Instead, lifetime properties are described with respect to the *liveness* of *handles* used to access instances. In PSL, each distinct instance of a role is ascribed a unique handle. Handle types are abstractions of names[47], references[51], ports[63], and static labels used to identify role instances and message destinations. Handle types may include values that do not provide access to implementations (as in the case of "null" and "dangling" references), in which case we say that the handle is not `live`. The relation "==" among handle values denotes only ("shallow") value equality, not necessarily identity or equality of implementation-level components. PSL/IDL handle values are expressed using the syntax of IDL `objrefs`. These serve as both references to instances of IDL `interfaces` (*not* necessarily their implementations) and destinations of CORBA messages. However, as discussed in [39], PSL handle values need not bear a one-to-one relation with CORBA `objref` implementations.

4 Attributes

An attribute (also known as a *state function*[38, 33]) is an abstract property ascribed to zero or more role instances, and possibly assuming different values across time. In PSL, attributes are abstract functions of handle and/or other value type arguments. The relationship between abstract attributes and their implementations (if any) is outside the scope of PSL proper. For example, the attribute `isOpen` is a hypothetical function that might actually be computed or approximated as a side-effect-free procedure, a function whose value is deduced by an analytic tool, and/or a "derived" function that is symbolically definable or otherwise constrained in terms of other attributes. The use of an attribute in PSL does not commit implementations to "know" its values in any computational sense, and even if implemented, does not mandate that the value be computed by any component it describes.

In PSL/IDL, attributes are declared as auxiliary functions within the scope of a `protocol module` using normal IDL/C++ function syntax, and where all function arguments are explicit. (The IDL keyword `attribute` is not used in PSL to avoid conflict with its IDL usage in implicitly declaring "get" and "put" *operations*.) For example:

```
protocol module fm { boolean isOpen(File f); /*...*/ };
```

Event Predicates. A PSL *event predicate* is a special kind of attribute representing the issuance or reception of any kind of explicit communication among participants, including operation requests, operation replies, asynchronous messages, and exceptions. For each kind of message M possible in a system, we assume the

existence of a corresponding record type $MessageType_M$ that minimally includes fields describing the message "selector" (e.g., operation name) and arguments (if any). All messages are assumed to be *handle-directed*. A handle describing the intended receiver role is a required part of any communication. Similarly, messages corresponding to a procedural operation that returns a reply (or perhaps just a "`void`" completion indication) include a handle describing the "return address" of the caller. We also assume an abstract function `reply[`*msg*`]` that represents a reply message (not its issuance) of the appropriate type for a given procedural request *msg*, and a function `throw[`*msg*`]` that represents an exception reply message for a request.

Messages themselves are not used directly in PSL. Instead, PSL contains two families of special attribute functions, `in` and `out`. The construct `in(`*m*`) by(`*p*`)`, where *m* is of some type $MessageType_M$ and *p* is a handle value, denotes a context in which a particular message instance *m* has been received by participant *p*, and `out(`*m*`) by(`*p*`)` denotes a context in which *m* has been issued by *p*. The `by(`*p*`)` suffixes are optional and seldom needed. When otherwise unconstrained, omission reflects lack of concern about which participant issues or receives a message.

PSL event predicates are otherwise treated as attribute functions that happen to have predefined characteristics. Event predicates are monotonic attributes, behaving as persistent "latches". Once `out(`*m*`) by(`*p*`)` is true for a particular *m* and *p*, it is true forever more; similarly for `in(`*m*`) by(`*p*`)`. This reflects the fact that events never "unhappen". Once a predicate describing the occurrence of an event becomes true, it never becomes false. Situations and rules may thus be phrased in terms of events that have occurred at any time [44].

PSL/IDL event predicates use IDL messages as arguments to `in` and `out`. PSL/IDL message types are abstractions of the `CORBA Request` type [51], with a shorthand handle-based message syntax delimited by angle-brackets:

m = < *dest*`->`*op*`(`*args*`)` >

Here, *m* is an instance of an implicitly "pattern-matched" message type corresponding to the form of the message expression; *dest* is a handle indicating the destination role instance of the message; *op* is an operation name literal; and *args* are arguments, each of some value type as defined in a corresponding interface. Messages need not be named, and values are referenced directly rather than through the implicit fields of *m*. As a notational convenience, a `reply` or `throw` without a bracketed message argument refers to the most closely associated message whenever this is unambiguous. Examples:

```
in( <aFile->write(c)> )           // write req received by aFile
out( reply[<aFile->read()>](c) ) // c sent as reply to read
out( throw(exc) ) by(aFile)       // exc thrown by aFile
```

5 Situations

The concept of a situation extends the common notion of abstract object state[43, 7]. Situations describe *partial* views of possible system-wide states, factored

with respect to possibly several roles. In PSL, situations are represented by parameterized declarative expressions describing "interesting" commonalities of transient *possible worlds* in terms of relevant attribute and event predicate values. In PSL/IDL, a situation is represented as a parameterized predicate. We provide a full notation in [39], but use here an abbreviated form in which situations are defined within `rules` declarations as boolean C++ expressions within braces. (PSL/IDL-G notation is the same except that situations are demarcated by solid rounded boxes inside a dashed `rules` box.) Expressions may reference attributes and event predicates, named arguments and in-line local declarations. For example, the following situation describes the state of a `File` being open:

```
rules f1(File f) { { isOpen(f) } };
```

Realizations. In the same sense that a class is instantiated by an object, and an interface is instantiated by a role, a situation is said to be instantiated by a *realization*. And just as interfaces describe the properties of an unbounded family of potential implementations, situations describe the properties of an unbounded family of potential realizations. However, realizations are not *explicitly* instantiated by programmers. They just unfold over the course of system execution. For example, the realizations of the above `isOpen` situation include all "snapshots" of a system in which such a `File` is open. The manner in which such realizations are observed or inferred in executing systems is outside the scope of PSL proper (see Section 11).

A realization is thus a partial description of an arbitrarily brief instance, an *actual world* in which the predicate describing a situation holds. Realizations represent individual observations, mappings, or inferences about concrete system behavior, characterized by expressions describing attribute and event predicate values. A realization is said to *match* one or more situations if all features required in the situation(s) are present. The matching relation, $s \propto S$ for realization s and situation S, holds if the situation predicate for S can be made true using the values in s (see [39]). Two or more realizations may all match the same situation but in different ways because they differently instantiate free parameters listed in the situation.

6 Protocol Rules

PSL temporal operators describe conditions under which realizations occur. They are defined in terms of an underlying temporal dependency relation $a \preceq b$ among two *realizations* a and b. If $a \preceq b$, then a happens no later than b (see Section 11). But since it would not be very productive to describe protocols via orderings among individual occurrences, PSL protocol rules are instead described at the level of situations (classes of realizations). Each of the following operators relates the occurrences of realizations of a predecessor situation A and successor situation B:

As lead to *B*s: $A \blacktriangleright B \stackrel{\text{def}}{=} \forall a : a \propto A \Rightarrow \exists b : b \propto B \wedge a \preceq b$

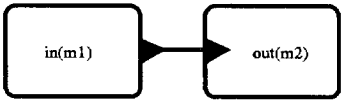
As enable *B*s: $A \blacksquare B \stackrel{\text{def}}{=} \forall b : b \propto B \Rightarrow \exists a : a \propto A \wedge a \preceq b$

As pair with *B*s: $A \blacklozenge B \stackrel{\text{def}}{=} (A \blacktriangleright B) \wedge (A \blacksquare B)$

Rules are collections of situations linked by temporal operators. In PSL/IDL-G, rules are expressed via lines connecting situations through corresponding symbols drawn on the outside of the predecessor situation to the same symbol drawn on the inside of the successor situation. In PSL/IDL, these operators are designated as `lead`, `enable` and `pair` respectively. In PSL/IDL (but not necessarily in PSL/IDL-G), the "earlier" situation is always to the left of the "later" one. PSL/IDL situations are implicitly parameterized by the arguments listed in their `rules`, as well as all declarations in their predecessor situation(s).

Leading. The "forward reasoning" operator $A \blacktriangleright B$ is akin to a state transition, applying to cases in which *A*s always precede *B*s. However, unlike a state transition, $A \blacktriangleright B$ does not indicate that instances of *B* form the "next" situations after those of *A*. Instead, an instance occurs at some unspecified time after an instance of *A* occurs, in a manner that neither requires nor precludes concurrency with respect to any other non-conflicting rules. Also, unlike state transitions, the orderings are not explicitly "triggered" by events. Instead predicates on events are listed as aspects of the situations themselves. For example, a protocol rule for a relay operation that sends `m2` whenever `m1` is received takes the form (in PSL/IDL and PSL/IDL-G respectively):

```
{in(m1)} lead {out(m2)}
```



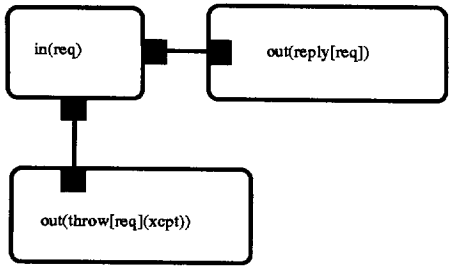
This rule does not in any way imply that `Relay` operations must be single-threaded. Because there are no constraints indicating otherwise, two or more different "threads" of this rule may be active concurrently, each triggered by a realization corresponding to a different instance of an `m1` message. On the other hand, this specification does not preclude implementation via a single-threaded relay object either.

Enabling. The "backward reasoning" operator $A \blacksquare B$ is used for relations in which *A enables B*, applying to cases in which *B*s are always preceded by *A*s. (Or are "caused" by *A*s, under some interpretations of this overloaded term.) For example, a desirable rule in most systems is the *no spurious replies* rule; for any procedural message `req`:

```
{in(req)} enable {out(reply[req]())}
```

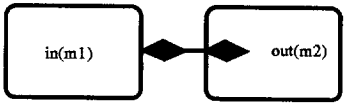
This rule says that replies are sent only if messages are received. It does not say that requests always lead to replies, only that replies are never sent unless preceded by requests. In PSL/IDL, this rule is considered to be predefined for all procedural requests, since it is enforced by CORBA.

Separate rules may link multiple right-hand-sides to the same left-hand-side to describe multiple possible effects. For example, we could add another rule stating that `req` may lead to an exception, and link it to the first rule, reflecting the fact that rules are combined conjunctively (i.e., implicitly ∧'ed):



Pairing. The $A \blacklozenge B$ operator is used for if-and-only-if relations, in which both every A is followed by a B, and every B is preceded by an A; thus As and Bs occur only in pairs. For example, another desirable global rule is the *one-to-one delivery rule*, for any `m`:

```
{out(m)} pair {in(m)}
```



This says that all and only those messages that have been issued are ultimately received. The $\blacklozenge$ relation may be used to provide guarantees about procedural operations. For example, we could strengthen the above ■ form to assert that a reply to `req` must be issued, thus precluding the possibility of exceptions or other non-procedural behavior:

```
{in(req)} pair {out(reply[req]())}
```

Sequences. A single rule may include chains of situations connected by temporal operators to describe sequencing. We express sequences of, for example, the leads-to operator as $A \blacktriangleright B \blacktriangleright C$. This indicates $(A \blacktriangleright B) \wedge (B \blacktriangleright C)$ while also extending scoping so that expressions in C may reference terms in A. For example, a special protocol in which a relay outputs `m3` after receiving message `m1` followed by `m2` could include a rule of the form:

```
{in(m1)} lead {in(m2)} lead {out(m3)}
```

If we did not care about the ordering of `m1` and `m2`, we would have written this using simple conjunction of event predicates:

```
{in(m1), in(m2)} lead {out(m3)}
```

On the other hand, if we wanted to claim that this `m1-m2` ordering were the only one possible, we would add the rule:

```
{in(m1)} enable {in(m2)}
```

7 Constraints

Some conditions must *always* hold, across all possible contexts. PSL constraint rules limit the set of possible system states to those in which listed expressions hold. For a trivial example, if we declared attribute `notOpen(File f)`, we would want to claim that its value is always equal to `!isOpen(f)`. The PSL notation for static constraints is based on the "□"*necessity* operator of temporal and modal logic[19, 64]. The expression □*p* indicates that *p* invariantly holds for all time, over all possible worlds. In PSL/IDL `rules` declarations, brace-demarcated expressions describing necessary restrictions among attribute values are prefaced by `inv`. For example:

```
rules f1(File f) { inv {notOpen(f) == !isOpen(f)} };
```

Constraints may be used to describe system-wide messaging policies. For example, CORBA (like most systems) enforces the policy that either a normal reply or an exceptional reply can be issued for procedural request `req`, but at most one of these. Using the PSL construct `unique` (*expr*), which is false if there is more than one match to *expr*, these constraints may be expressed as:

```
inv { out(reply[req]()) -->   !out(throw[req](x)) }
inv { out(throw[req](x)) -->  !out(reply[req]()) }
inv { unique(reply[req]()) }
inv { unique(throw[req](x)) }
```

Relating Attributes to Events. Once a predicate describing the occurrence of an individual event becomes true, it never becomes false. Thus, event predicates "latch" from left to right in PSL linked situations. For any event predicate *e*, if *e* holds in *A*, then it also holds in all successors of *A*. For example, the following two rules for simple relays are equivalent:

```
{in(m1)} lead {out(m2)}
{in(m1)} lead {in(m1), out(m2)}
```

This "latching" property of event predicates does *not* necessarily hold for arbitrary attributes. For example, if there were an unconstrained attribute `ok(Relay r)`, and we required that `ok(r)` persist as true across these two situations, we would have to write:

```
{in(m1), ok(r)} lead {out(m2), ok(r)}
```

Such unconstrained attributes are notoriously troublesome in practice [43, 11, 17]. When attributes are not tied to events, there are no global rules stating how values change as a function of events. Any required variation or invariance in the values of unconstrained attributes across time must be explicitly tracked in individual protocol rules. Constraint rules may be used to avoid such problems, by linking the values of arbitrary attributes to the occurrence particular events. For example, supposing that our unrealistically simple `File` may be opened and closed only once, we might supply constraints indicating how attribute `isOpen` varies with `open` and `close` events:

```
rules f1(File f) {
 inv { isOpen(f) -->   out(reply[<f->open()>]()) }
 inv { out(reply[<f->close()>]()) -->  !isOpen(f) }
};
```

Here, the first constraint says that if a file is open, then it must have at some time replied to an `open` request (but not necessarily vice versa). The second says that if the file has ever replied to a `close` request, then it must not be open. These might be buttressed with a description of a `FileFactory` operation that guarantees that `isOpen` is true upon reply of its result. On the other hand, if a `File` could be implemented by a special kind of object that is initially open upon construction without requiring an explicit `open` operation, then the first constraint would have to be weakened or eliminated.

To further illustrate, consider an `Account` interface, along with a `protocol module` declaring abstract `bal` and `serialNo`, along with simple postcondition style rules stating that `mkAcc` initializes them, `getBalance` returns `bal`, `setBalance` changes `bal`, and `getSerialNo` reports `serialNo`. A set of `inv` rules adds further constraints:

```
interface Account {
 void  setBalance(in float b);
 float getBalance();
 long  getSerialNo();
};

interface AccountFactory {
 Account mkAcc(in long s, in float b);
};

protocol module accountm {
 float bal(Account a);
 long  serialNo(Account a);
 rules (Account a, AccountFactory f, long s, float b) {
  {in(<f->mkAcc(s, b)>)}  pair
   {out(reply(a)), live(a), bal(a) == b, serialNo(a) == s}
  {in(<a->getBalance()>)}  pair {out(reply(bal(a)))}
  {in(<a->setBalance(b)>)} pair {out(reply()), bal(a) == b}
  {in(<a->getSerialNo()>)} pair {out(reply(serialNo(a)))}

  inv {(bal(a) == b) -->
       out(reply[<f->mkAcc(x,b)>](a)) ||
       out(reply[<a->setBalance(b)>]()) }
  inv {(serialNo(a) == s) -->
      out(reply[<f->mkAcc(s,b)>](a)) }
  inv {unique(out(reply[<f->mkAcc(s,b)>](a)) }
  inv {unique(in(s = <a->setBalance(b)>), !out(reply[s]())) }
 };
};
```

The four invariants establish the following constraints:

1. The `setBalance` and `mkAcc` operations are the *only* ones that affect the value of attribute `bal`. Without such a constraint, there is no requirement that this reasonable and often implicitly assumed encapsulation property holds. This is expressed by relating `bal` to values associated with replies from either of the two operations.
2. The `serialNo` is always the one established by the factory operation. Note that because roles need not bear a one-to-one relation to implementation objects, it is not possible in PSL to ascribe such "initial" properties to roles that hold across all instantiations. Instead, factory[53] operations are used to establish instances for which particular properties hold.
3. Initialization occurs at most once per account.
4. The processing of `setBalance` requests is not subject to arbitrary interleavings (i.e., that they proceed serially), thus precluding multithreaded implementations. The restriction that no two `setBalance` operations operate concurrently can be expressed by saying that any message that has been received but not replied to is `unique`.

8 Subsituations

Like interfaces, states and classes, situations may be specialized into *subsituations*. Subsituation relations are analogs of the subtype relations underlying interface inheritance. Mechanics follow those for ordinary sets defined via predicates. For example, situation Q: {`in(readrequest) && isOpen(f)`} is a subsituation of P: {`in(readrequest)`} in which case we say that $Q \subseteq P$.

If $Q \subseteq P$, then fewer possible realizations match Q than P. Every situation is a subsituation of the empty situation {} (or equivalently {*true*}), which is matched by all realizations. The situation {*false*} (matched by no realizations) is a subsituation of all others. We say that expression *expr holds* in situation S if $S \subseteq \{ expr \}$. Conversely, we define set-like union and intersection operators in terms of the corresponding boolean relations on their component expressions [17, 62]. In PSL/IDL $A \cap B$ is expressed as $\{expr_A\}$ `&&` $\{expr_B\}$, and $A \cup B$ as $\{expr_A\}$ `||` $\{expr_B\}$.

The simplest and most common means of constructing a subsituation is to "strengthen" an expression by *and*'ing an independent predicate p, since $A \cap \{p\} \subseteq A$. Strengthening may also occur by replacing a predicate with one that implies it. For example, a situation including `out` (*m*) might be strengthened by replacing it with the more committal `out` (*m*) `by` (*p*).

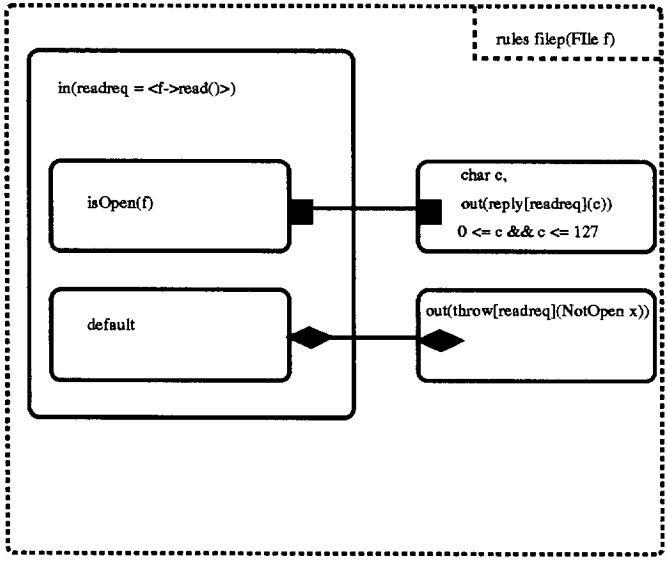
Partitions. PSL does not use any special notation to *declare* that one situation is a subsituation of another except in the special case of disjoint union where $S = S_1 \oplus S_2 \oplus \ldots \oplus S_n$. This represents a set of subsituations that are constrained to completely partition a situation-space S. Partitioned subsituations must be exhaustive and mutually exclusive. Partitioning is thus one way to express the notion that one situation "disables" another[67].

In PSL/IDL, partitions are enclosed in `case`. Successive cases are interpreted in the same way as `case`, `cond`, and `if ...elsif` statements in most languages, implicitly negating the expressions in all previous `cases`. The special trailing situation `default` matches only if all previous cases fail, thus acting as a generalized `else`.

Conditional protocol rules can be expressed using partitions. Ordering operators "distribute" through situation partitionings to describe conditional paths. Since ordering operators have lower precedence than the `&&` operator combining situations, expressions common to all partitions can be described in a situation linked by `&&` to the `cases`. For example, to indicate that a `TRUE` reply is enabled if some function `ok` holds, and conversely for `FALSE`:

```
{ out(req) } pair
{ in(req) } && {
 case { ok(req) } enable { out(reply(TRUE)) }
 case { default } enable { out(reply(FALSE)) } }
```

In PSL/IDL-G, partitioning is indicated via nested StateCharts [24]. Each enclosed box represents a partition. The special trailing "else" box contains only the expression `default`. Expressions common to all partitioned subsituations may be listed outside of the nested boxes. The accompanying fragment of a PSL/IDL-G protocol for a simple `File` illustrates some of the interplay between conditionals and temporal operators.



In this example, the use of ■ linking the "normal" `read` reply indicates that a situation in which a reply is generated occurs only when a file is open and receives a read request, but may not occur at all so far as can be determined

from the perspective of the roles parameterized within the current `rules` declaration. For example, there may be "downstream" errors stemming from internal invocations that are not visible at this scope or level of specification. However, if a reply occurs, the return value `c` is subject to the listed constraints that amount to a guarantee that the return value is a 7-bit character value.

In contrast, the "exceptional" reply situation is linked via $\blacklozenge$, indicating that (only) when a read request is received by a file that is not open, an exception reply to the request is always generated. This does *not* indicate that this is the only context in which the `NotOpen` exception is thrown. It says instead that this is the only context in which it is thrown *as a reply to* `read`. If for some reason we had wanted to make the weaker claim that `NotOpen` is possible but not guaranteed, we would have used $\blacksquare$. Had we wanted to make the differently weaker claim that `NotOpen` is always issued not only here, but perhaps also in some other context (i.e., even if the file is open) we would have used $\blacktriangleright$.

9 Refinement

Protocol refinement is the act of introducing new rules that apply in more specific situations than do general rules, without invalidating these more general rules. Refinement is thus an additive process, where rules accumulate, each adding specificity in a narrower context. The opposite of refinement is generalization, in which a weaker set of rules is introduced in a broader context. In PSL, refinements are normally localized to new interfaces bearing subinterface relations to the originals. Specialized protocol declarations may be attached to (parameterized by) instances of subinterfaces. Similarly, generalizations may be attached to superinterfaces.

PSL/IDL-annotated interfaces are always refinements of those described in IDL only. When two kinds of PSL/IDL interfaces differ in protocol but not operation signatures (for example, when protocols differ across "local" versus "remote" versions of components), it may be necessary in practice to construct "dummy" IDL interface types just to give the two types different names [5].

Of course, not all reasonable modifications are valid refinements. For example, instances of a protocol description could differ in that one corrects an error, or removes unwanted behavior, or describes a subtly different protocol, or imposes additional constraints due to changed or overlooked requirements. Valid refinements are generally limited to the following techniques, that may also be applied in reverse in the course of generalization. These techniques reflect properties of PSL that hold for all situations A, B, C containing predicates meaningful in their scopes:

$A \blacktriangleright \{ \mathit{true} \}$	$A \blacksquare \{ \mathit{false} \}$
$\{ \mathit{false} \} \blacktriangleright A$	$\{ \mathit{true} \} \blacksquare A$
$A \blacktriangleright B, B \blacktriangleright C \vdash A \blacktriangleright C$	$A \blacksquare B, B \blacksquare C \vdash A \blacksquare C$
$A \blacktriangleright B, B \blacklozenge C \vdash A \blacktriangleright C$	$A \blacksquare B, B \blacklozenge C \vdash A \blacksquare C$
$A \blacklozenge B, B \blacktriangleright C \vdash A \blacktriangleright C$	$A \blacklozenge B, B \blacksquare C \vdash A \blacksquare C$

$A \blacktriangleright B \cap C \vdash A \blacktriangleright B$ $\quad$ $A \blacksquare B \cup C \vdash A \blacksquare B$
$B \cup C \blacktriangleright A \vdash B \blacktriangleright A$ $\quad$ $B \cap C \blacksquare A \vdash B \blacksquare A$
$B \blacktriangleright A, C \subseteq B \vdash C \blacktriangleright A$ $\quad$ $C \blacksquare A, C \subseteq B \vdash B \blacksquare A$
$A \blacktriangleright C, C \subseteq B \vdash A \blacktriangleright B$ $\quad$ $A \blacksquare B, C \subseteq B \vdash A \blacksquare C$

Adding Rules. New rules relating new situations, as well as new constraints, may be added so long as they do not conflict with existing ones. For example, if `ReadWriteFile` is defined as a subinterface of `ReadFile`, new rules applicable to `write` operations may be defined in `rules` for `ReadWriteFile`. The `rules` for `ReadFile` would also still hold for all `ReadWriteFile` instances.

Splicing Situations. A new situation S may be spliced among existing ordered situations A and B, so long as the original relation between A and B is maintained. Thus, $A \blacktriangleright B$ may be extended to $A \blacktriangleright S \blacktriangleright B$, or to $A \blacktriangleright B \blacktriangleright S$, or to the separate $(A \blacktriangleright S) \wedge (A \blacktriangleright B)$, and so on. Splicing allows arbitrarily complex subprotocols to be inserted between existing end-points. For example, an original version of a rule for a procedural operation might list only the request and reply. A refinement may then specify internal structure such as an interaction with a helper:

```
{in(req)} enable
   {out(reply[req]())}

{in(req)} pair {out(h = <helper->help()>)} enable
   {in(reply[h]())} pair {out(reply[req]())}
```

Note that even though the $\blacksquare$ was juggled around, the original sense of the relation is maintained. If necessary, this may be checked formally. For example here, the refined rule is of the form $(A \blacklozenge B) \wedge (B \blacksquare C) \wedge (C \blacklozenge D)$. From the first two clauses we see that $A \blacksquare C$. Then applying the last clause, we obtain $A \blacksquare D$, as required by the original rule.

Subdividing Situations. A situation may be split into subsituations, so long as all ordering relations are maintained across all paths along all subsituations. For example, an initial rule for a boolean operation might only list that a reply occurs, while a refinement splits apart the conditions under which it replies true versus false:

```
{in(req)} enable {out(reply(b))}

{in(req)} && {
  case {badstuff()} enable  {out(reply(FALSE))}
  case {default}    enable  {out(reply(TRUE))} }
```

Strengthening Relations. The relation $A \blacktriangleright B$ or $A \blacksquare B$ may be strengthened to $A \blacklozenge B$ when this does not conflict with other existing rules. For example, a

preliminary version of a rule may use ■ to indicate that a particular exception may result from a certain request in a certain condition. Assuming that no other existing rules indicate otherwise, a refinement may instead use ◆ to make the stronger claim that this exception is *always* generated under this condition. Similarly, we could strengthen the previous example to use the ◆ operator instead of ■ if we were sure that the listed situations represented the only ways in which the `TRUE` and `FALSE` replies could occur.

Weakening and Strengthening Situations. If relation $A \blacktriangleright B$ holds in an original specification, a refinement may add a new rule $A' \blacktriangleright B'$, where $A \subseteq A'$ and $B' \subseteq B$. The reverse relation holds for ■. These are situational analogs of type conformance and substitutability[58], ensuring that rules applying in the original versions continue to hold even when refined. For example, consider an $A \blacktriangleright B$ rule for a `Relay r` with attribute `broken`:

```
{in(m1), !broken(r)} lead {out(m2)}
```

In a refined version $A' \blacktriangleright B'$, we could have a weaker left-hand-side ($A \subseteq A'$) and a stronger right-hand-side ($B' \subseteq B$), leading to a stronger rule that subsumes the original version:

```
{in(m1)} lead {out(m2) by(r)}
```

The opposite maneuver would be either superfluous or an error: If the second rule had been the original specification, then it would have already covered the first rule. And if we had wanted to *restrict* the second rule to the first, the relation would not be a refinement; we would create an unrelated (on this dimension) protocol and/or interface.

10 Describing Interactions

Consider a set of transaction components in which `Coordinators` help arrange the actions of `Transactors`, with IDL interfaces:

```
typedef long TID;

interface Transactor {
 boolean join(in TID tid);
 boolean commit(in TID tid)
 void    abort(in TID tid)
};

interface Coordinator : Transactor {
 TID     begin();
 boolean add(in TID tid,in Transactor p) raises (TransError);
};
```

The overall design is that Coordinators create (via begin) transaction identifiers (TIDs) that are used to index transactions. Each transaction consists of a group of Transactor members, added via the add operation. As indicated by the use of interface inheritance, members may be other Coordinators. Each Transactor may be asked to join a transaction, and to commit or abort it. The application operations that each component performs within transactions (perhaps bank account updates) are not described in this set of interfaces.

To capture some of this in PSL/IDL, we declare a protocol module, starting off with a declaration of attribute members to represent the handles of all members of a given transaction. There may be several sets of members maintained by each Coordinator, each referenced via its transaction identifier (tid). The associated constraint precludes the existence of any additional operations or rules that cause p to become a member unless they somehow invoke add. We also declare auxiliary function validtid and corresponding constraint to show how the notion of a valid transaction id is related to event predicates. A transaction identifier tid is valid if it was created as the result of a begin operation, but becomes invalid when the subject of any abort or commit request:

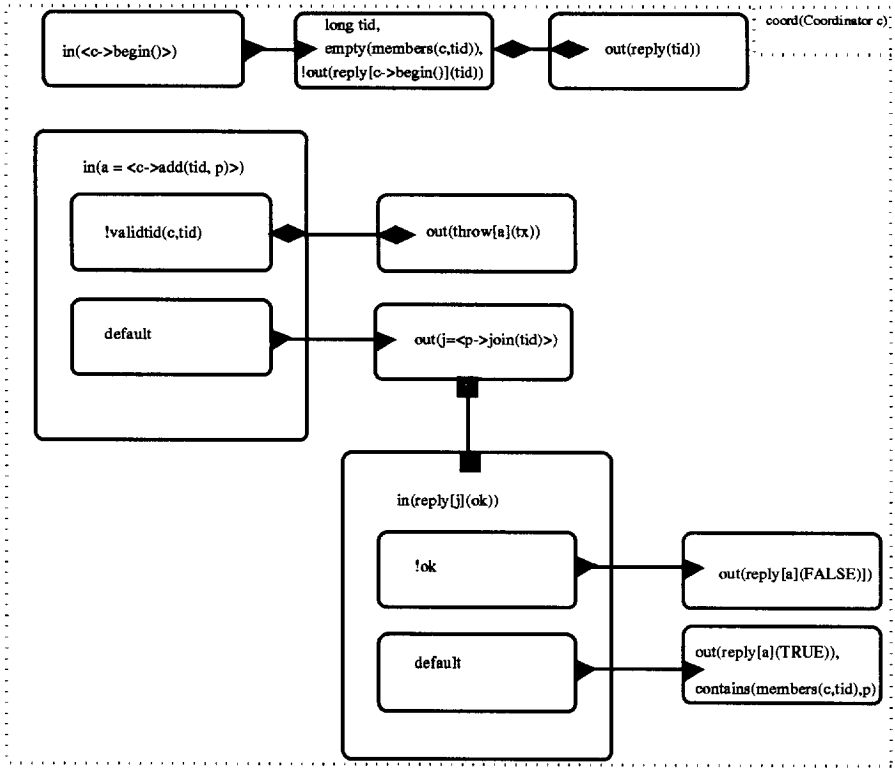
```
sequence<Transactor> members(Coordinator c, TID tid);
boolean validtid(Coordinator c,TID tid);

rules(Coordinator c, Transactor p, TID t){
 inv { contains(members(c,t), p) -->
       out(reply[<c->add(p,t)>](TRUE)) }
 inv { validtid(c, tid) -->  out(reply[<c->begin()>](tid)) }
 inv { in(<c->abort(tid)>) -->  !validtid(c, tid) }
 inv { in(<c->commit(tid)>) --> !validtid(c, tid) }
};
```

Two sample protocol rules are also shown in the accompanying PSL/IDL-G declaration. The first rule says that on receiving a begin request, the Coordinator replies with a tid value that has never been used before. This statement, along with the above constraints on validtid amount to a promise that each tid value returned by begin is unique and valid for the length of the transaction.

The main "thread" in the second rule says that upon receiving an add request for a Transactor p with a valid tid, a coordinator invokes p's join operation. If it then receives a TRUE reply, p is then a member of members and the operation completes successfully. The other cases are "error paths"; one causing an exception, and the other a simple FALSE reply. Additional situations and relations would surely be included in a more realistic specification. For example, it may describe cases dealing with the possibility that !live(p) (i.e., if p were not a live handle), the use of timeouts, and so on.

We may also rework a more specialized version of the add rule by parameterizing over both the Coordinator and the Transactor roles. The accompanying partial PSL/IDL-G declaration of rules addp(Coordinator

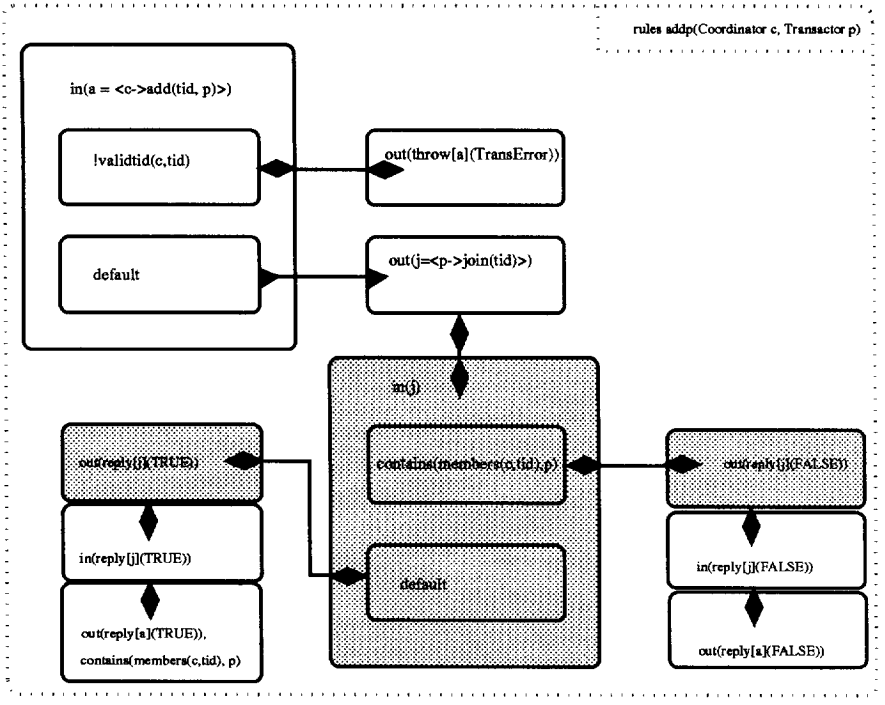


`c, Transactor p)` assumes for simplicity that `p` replies `FALSE` to `join` only if it is already a member of the transaction. For emphasis, situations describing the view of the `Transactor` are shaded. One sense in which this protocol is more committal is that rather than relying on a *one-to-one delivery rule* to match the `p->join` request with its reception (and similarly for the `join` reply), this version directly connects the associated situations for these particular communication partners. Such rules may be seen as the specification analog of multimethods[17].

Along a different dimension, we could have presented a less committal version by omitting various situations if we happened not to care about them for the sake of this protocol declaration, and then perhaps inserted them later as refinements. For example, the `join` reply and its acceptance might have been elided without changing the ordering requirements of the remaining situations.

11 Foundations

PSL constructs are based on structures describing possible worlds, as found in model theory and temporal and modal logic[31, 19, 64]. These serve as the basis for several frameworks for specifying possibly distributed systems



[38, 42, 14, 34, 6], as well as related applied temporal reasoning systems in AI and object-oriented logic programming [43, 3, 15].

Possible world structures are of the form **<W, V, R>**. **W** is a set of worlds. **V** is a set of value expressions over some basis, with an associated function $\phi(p, w)$, which is true if expression p holds in world w. **R** represents any number of defined relations among the worlds in **W**. Chief among them is the relation generated by constraints. PSL □ rules define the set of all worlds that are possible, and a corresponding relation containing every pair of possible worlds.

PSL situations define a another family of "static" equivalence relations $\mathbf{R}_\sigma$. Situation S describes that set of worlds for which its defining predicate P_S holds given the values in the world (i.e., $\{w \mid \phi(P_S,w)\}$). In PSL this is expressed in terms of the matching relation, $\propto$, between values holding in worlds and situation predicates. The corresponding relation $\mathbf{R}_S$ contains all pairs of worlds that are members of this set.

The relation $\mathbf{R}_\preceq$ serves as the basis for PSL ordering operators. This relation is simplest to describe formally when expressions are restricted to event predicates on fixed messages[23, 67, 57]. In this case, expressions in **V** are just characteristic predicates of sets of event occurrences, and $\phi(e,w)$ is true if w contains the events of e. For example, suppose a satisfies e_a = `out(m1)` for some message `m1`, and b satisfies e_b = `out(m1) && in(m1)`. The relation $a \preceq b$ states that $e_b \Rightarrow e_a$. The `out(m1)` event has not "gone away" in B. In fact, if $a \preceq b$, `out(m1)` holds no later than the realization in which `in(m1)` holds as

well. Thus, when restricted to events, the relation $\mathbf{R}_{\preceq}$ contains all *(a,b)* such that the set of events described by e_a must be be a subset of that described by e_b. When expressions are liberalized to allow reference to arbitrary attributes, the $\preceq$ relation is no longer definable in this semi-automatic manner, since it is not necessarily the case that $e_b \Rightarrow e_a$. Constraint rules must be supplied to describe how arbitrary attributes vary with respect to events.

Application of these constructs to particular systems relies on *mappings* from PSL to concrete implementation code and/or observations. These include (1) mappings between roles and implementation objects; (2) mappings between event predicates and concrete communication occurrences; (3) mappings between expressions defining situations and realizations observed or inferred in concrete code and/or its execution; and in some cases (4) mappings of the initial conditions of the system of interest. These generate an operational semantics that in turn allows construction of corresponding design methods and tools; for example simulation, prototyping, verification, visualization, testing, and monitoring. This also allows PSL to be used as a scripting language in which prococol specifications are directly compiled into default implementations.

While the use of PSL in some systems would require development of auxiliary configuration languages and tools to establish mappings, the particular features of PSL/IDL along with those of CORBA permit simpler tactics: PSL/IDL uses the same value type system as CORBA IDL. OMG standards in turn already map IDL value types to those of various implementation languages [54]. PSL/IDL message types map directly to those used in CORBA. Observations of messages may be used to establish instantiation of corresponding event predicates. Also CORBA Object Request Brokers (ORBs) and repositories maintain information relating values that are used as message destinations and the locations of concrete implementation components. These may be relied on to maintain implicit mappings between interface instance handles and implementation objects. And the initial conditions of most CORBA applications amount only to the initialization of a small number of components, avoiding the need for extensive description of static configuration properties.

CORBA also supports development of the instrumentation needed for dynamic execution tools. Event monitoring may be accomplished through interpositioning; the placement of intercepts between communicating components to tap communication traffic[68]. However, even if attention is restricted to event predicates, mapping communications to event predicates, and in turn realizations of particular situations, and in turn rule instantiations is not a trivial matter in a distributed open system (see [39]). However, provided that such observational apparatus is available, one could create, for example, a monitoring tool reporting whether realizations matching listed situations occurred and whether the corresponding ordering rules were observed.

12 Related Work

Protocol specification, architectural description, and approaches to dynamics in general have a long history. While all such approaches may be related at some level, they differ significantly in their theoretical bases, definitional primitives, and range of usability. The ways in which PSL constructs support interface-based specification of open systems distinguish it from other frameworks:

Preconditions and Postconditions [29] and specification systems based upon them [35], employ the construct $\{A\}\ s\ \{B\}$, asserting that program fragment s brings a program from a state obeying A to one obeying B. The PSL constructs $A \blacktriangleright B$ and $A \blacksquare B$ have similar usages, but split the different senses of this relation when applied to ordered events. PSL may be used to express the kinds of assertions typically associated with operation postconditions, but applies them to arbitrary "evaluation points" rather than necessarily only upon issuance of a reply. PSL does not include any language-specific operational semantics, and omits reference to s. PSL also differs in its scoping and parameterization of predicates.

Abstract Data Types (ADTs) [40, 61] describe functional properties of "black box" components via preconditions, postconditions, and invariants, without describing the nature of their dynamic dependencies or interactions. PSL attributes and constraints share a similar basis, but are used primarily to describe interaction constraints.

Architecture Description Languages (ADLs), module interconnection languages, and related approaches [41, 4] usually extend an ADT-style basis to describe static configuration and communication properties of sets of components. This focus on statics varies in degree across languages. PSL may be construed as variant ADL best suited for systems with few fixed configuration properties beyond those of their general purpose communication substrates; for example, ORB-mediated communication in CORBA systems.

Object-Oriented Analysis notations [60, 33, 17] describe classes of objects in terms of attributes, relations, states, operations, and messaging, at varying degrees of formality. PSL generalizes, extends and reworks the dynamic aspects of such concepts to apply to interfaces of components in open systems, in a manner compatible with other role-based frameworks [6, 59, 66, 25]. Unlike some other approaches [2, 68] that add protocol specifications to object-oriented interfaces, PSL does not assume any particular model or mechanism relating these interfaces and roles to classes and objects.

Linguistic Approaches to Pragmatics [50] address the context-specific dynamics of communication (speech acts) among participants, while semantics abstracts over situations to address context-independent meanings. This distinction provides a useful conceptual basis for approaching these qualitatively different aspects of interaction, and serves as a guide to the kinds of phenomena that PSL is intended to capture.

Process Calculi [46] and specification languages based upon them model systems as collections of abstract processes communicating via messages, where each process and communication act obeys a particular abstract computation

model. In contrast, PSL specifications are non-constructive. They do not rely on a particular computational model beyond that implied by minimal assumptions about message passing in open systems. PSL specifications contain sets of constraints on behavior that may be implemented by any kind of component meeting the constraints.

History-based Frameworks [44, 49, 32, 13, 22] specify actions that occur under given patterns of event histories. These patterns are most often described in terms of regular expressions or variants thereof. Because PSL deals with roles in potentially distributed systems, events as seen by a given instance are not necessarily totally orderable. They can be ordered only by $\preceq$, not the strict $\prec$ relation that may be seen by any particular implementation *object*. Thus PSL history patterns cannot be described as languages or regular expressions[56]. They are instead indicated by linked situations. Also, an event occurrence is construed in PSL as just one kind of attribute (although one with a special interpretation) ascribable to a role. Other kinds of attributes can be defined as well. For example, one set of instances of a `File` interface may be "born" in an `isOpen` state, while others are not.

Event-based Frameworks [37, 56, 9] are typically based on orderings defined over raw events. The PSL $\preceq$ relation serving as the basis for protocol operators is defined in a fashion similar to such orderings, but ranges over abstract instances of situations described via event predicates and other arbitrary attributes, not instances of events themselves. When restricted to event predicates, these are related in a simple way under the intended mapping to raw events: If the instances of two events are ordered, then so are the corresponding instances of event predicates. PSL operators, situations, and partitionings are more closely related to corresponding constructs in *event structures* and its variants[67, 23, 57], as adapted for use in interface-based specification.

The most important differences between PSL and other approaches reflect instrinsic properties of open systems development. But they also weaken analytic properties. Closed-world models are uniformly more powerful than PSL in addressing questions about liveness, deadlock, interference, and aliasing [48, 30], which often have no definitive answers in open systems. Closed-world models may still have a place in open system development, but only when applied to descriptions of classes and objects. Object implementation code does what it does, and nothing more. While interface-level open system development cannot exploit this fact, it is sometimes profitable (but see [36]) to rely on fixed known implementation properties when constructing individual components.

Acknowledgments. This work was supported in part by a Sun Microsystems Labs collaborative research grant to the first author. Thanks to the Sun PrimaVera and Vantage groups, and to Alistair Cockburn, Dennis de Champeaux, Desmond D'Souza, Peter O'Hearn, Alan Pope, Doug Schmidt, Bob Sproull, and Carolyn Talcott for helpful discussions and comments.

References

1. Agha, G., *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.
2. Aksit, M., L. Bergmans, & S. Vural, "An Object-Oriented Language - Database Integration Model: The Composition-Filters Approach", *Proceedings, ECOOP '92*, LNCS 615, Springer-Verlag, 1992.
3. Alexiev, V., *Mutable Object State for Object-Oriented Logic Programming: A Survey*, Technical Report TR 93-15, Department of Computing Science, University of Alberta, 1993.
4. Allen, R., & D. Garlan, "Formal Connectors", Technical Report CMU-CS-94-115, Carnegie Mellon University, 1994.
5. America, P., "A Parallel Object-Oriented Language with Inheritance and Subtyping", *Proceedings, OOPSLA '90*, ACM, 1990.
6. Arapis, C., *Dynamic Evolution of Object Behavior and Object Cooperation*, Thesis, University of Geneva, 1992.
7. Barwise, J., "Constraints, Channels, and the Flow of Information", in j. Peters (ed.) *Situation Theory and its Applications, Volume 3*, CSLI Lecture Notes, Stanford University, 1993.
8. Baumgartner, G., & V. Russo, "Signatures: A C++ Extension for Type Abstraction and Subtype Polymorphism", *Software---Practice and Experience*, 1994.
9. Birman, K., & R. Van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, 1994.
10. Booch, G., *Object-Oriented Analysis and Design*, Benjamin Cummings, 1993.
11. Borgida, A., J. Mylopoulos, & R. Reiter, "...And nothing else changes: The frame problem in procedure specifications". *Proceedings Fifteenth International Conference on Software Engineering*, IEEE, 1993.
12. Buhr, R. & R. Casselman, "Architecture with Pictures", *Proceedings, OOPSLA '92*, ACM, 1992.
13. Campbell, R. H., & A. N. Habermann, "The Specification of Process Synchronization by Path Expressions". *Lecture Notes in Computer Science 16*, Springer-Verlag, 1974.
14. Chandy, K. & J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.
15. Davison, A., "A Survey of Logic Programming Based Object-Oriented Languages", in G. Agha, P. Wegner, & A. Yonezawa (eds.) *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.
16. de Champeaux, D., Verification of Some Parallel Algorithms, *Proceedings, 7th Annual Pacific Northwest Software Quality Conference*, Portland, OR, 1989.
17. de Champeaux, D., D. Lea., & P. Faure, *Object-Oriented System Development*, Addison-Wesley, 1993.
18. D'Souza, D., & A. Wills, *Catalysis -- Practical and Rigorous Object Development*, Techincal Report, ICON Computing, 1995.
19. Emerson, E., "Temporal and modal logic". J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Volume B*, MIT press, 1990.
20. Gamma, E., R. Helm, R. Johnson, & J. Vlissides. *Design Patterns*, Addison-Wesley, 1994.
21. Garlan, D., & M. Shaw, "An Introduction to Software Architecture". In V. Ambriola and G. Tortora (eds.) *Advances in Software and Knowledge Engineering*, vol II, World Scientific Publishing, 1993.
22. Gatziu, S., & K. Dittrich, "Events in an Active Object-Oriented Database System", *Proceedings, 1st International Workshop on Rules in Database Systems*, 1993.

23. Gupta, V., "Concurrent Kripke Structures", *Proceedings of the North American Process Algebra Workshop* Cornell CS-TR-93-1369, 1993.
24. Harel, D., "StateCharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, 8, 1987.
25. Harrison, W., & H. Ossher, "Subject-Oriented Programming", *Proceedings, OOPSLA '93*, ACM, 1993.
26. Harrison, W., *The Importance of Using Object References as Identifiers of Objects*, Document 94.6.12, Object Management Group, 1994.
27. Helm, R., I. Holland, & D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", *Proceedings, OOPSLA '90*, ACM, 1990.
28. Hewitt, C., P. Bishop, & R. Steiger, "A Universal Modular ACTOR Formalism for AI", *Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.
29. Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, 12, 1969.
30. Hogg, J., D. Lea, R. Holt, A. Wills, & D. de Champeaux, "The Geneva Convention on the Treatment of Object Aliasing", *OOPS Messenger*, April 1992.
31. Hughes, G.E., & Cresswell, M.J. *An Introduction to Modal Logic*, Methuen, 1971.
32. Jagadish, H., & O. Shmueli, "Composite Events in a Distributed Object-Oriented Database" *Distributed Object Management*, Morgan Kaufmann, 1994.
33. Jarvinen, H., R. Kurki-Suonio, M. Sakkinnen, & K. Systa, "Object-Oriented Specification of Reactive Systems". *Proceedings, International Conference on Software Engineering*, IEEE, 1990.
34. Jarvinen, H. *The Design of a Specification Language for Reactive Systems*, Technical Report 95, Tampere University of Technology, 1992.
35. Jones, C., *Systematic Software Development Using VDM*, Prentice Hall, 1986.
36. Kiczales, G. *Open Implementations*, Forthcoming book.
37. Lamport, L., "Time, Clocks, and the Ordering of Events in Distributed Systems", *Communications of the ACM*, 21(7), 1978.
38. Lamport, L., *The Temporal Logic of Actions* SRC Research Report 79, Digital Equipment Corp, 1991.
39. Lea, D., & J. Marlowe, *PSL: Protocols and Pragmatics for Open Systems*, Technical Report, Sun Microsystems Laboratories, 1994.
40. Liskov, B., & J. Guttag, *Abstraction and Specification in Program Development*, MIT Press, 1986.
41. Luckham, D., L. Augustin, J. Kenney, J. Vera, D. Bryan, & W. Mann, "Specification and Analysis of a System Architecture Using Rapide", *IEEE Transactions on Software Engineering*, 1994.
42. Manna, Z., & A. Pneulli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.
43. McCarthy, J. & P.J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence", in D. Michie and B. Meltzer (eds.), *Machine Intelligence 4*, Edinburgh University Press, 1969.
44. McCarthy, J. *Elephant 2000: A Programming Language Based on Speech Acts*, Unpublished Manuscript, Stanford University, 1994.
45. Meseguer, J., "A Logical Theory of Concurrent Objects and its Realization in the Maude Language", in G. Agha, P. Wegner, & A. Yonezawa (eds.) *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.
46. Milner, R., *Communication and Concurrency*, Prentice Hall International, 1989.

47. Milner, R., J. Parrow, & D. Walker, "A Calculus of Mobile Processes", *Information and Computation*, vol 10, pp1-77, 1992.
48. Mullender, S. (ed.) *Distributed Systems*, 2nd ed., Addison-Wesley, 1993.
49. Nierstrasz, O. "Regular Types for Active Objects", *Proceedings, OOPSLA '93*, ACM, 1993.
50. Newmeyer, F. *Linguistics: The Cambridge Survey*, Cambridge University Press, 1988.
51. OMG, *Common Object Request Broker Architecture and Specification*, Document 91.12.1, Object Management Group, 1991.
52. OMG, *Response to the Object Management Group Object Services Task Force Request for Information*, Document 91.11.6. Object Management Group, 1992.
53. OMG, *Common Object Services Specification*, Document 94.1.1, Object Management Group, 1994.
54. OMG, *IDL C++ Language Mapping Specification*, Document 94.8.2, Object Management Group, 1994.
55. Powell, M., *Objects, References, Identifiers and Equality*, Document 93.7.5, Object Management Group, 1993.
56. Pratt, V.R., "Modeling Concurrency with Partial Orders", *International Journal of Parallel Programming*, 15 (1), 1986.
57. Pratt, V.R., *Chu Spaces: Complementarity and Uncertainty in Rational Mechanics*. Technical Report, Stanford University, 1994.
58. Raj, R., E. Tempero, H. Levy, A. Black, N. Hutchinson, & E. Jul, "Emerald: A General Purpose Programming Language", *Software---Practice and Experience*, 1991.
59. Reenskaug, T. *The Object Industry: The Large Scale Provision of Customized Software*, Addison-Wesley, forthcoming.
60. Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
61. Sankar, S. & R. Hayes "ADL: An Interface Definition Language for Specifying and Testing Software", in *Proceedings of the Workshop on Interface Definition Languages*, ACM SIGPLAN Notices, 1994.
62. Scholl, M., C. Laasch, & M. Tresch, "Updatable Views in Object Oriented Databases", in C. Delobel, M. Kifer & Y. Masunaga (eds.) *Deductive and Object-Oriented Databases*, Springer-Verlag, 1991.
63. Strom, R., D. Bacon, A. Goldberg, A. Lowry, D. Yellin, & S. Yemeni, *Hermes: A Language for Distributed Computing*, Prentice Hall, 1991.
64. von Benthem, J. *The Logic of Time*, Kluwer, 1991.
65. Wegner, P., "Tradeoffs between Reasoning and Modeling", in G. Agha, P. Wegner, & A. Yonezawa (eds.) *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.
66. Wieringa, R., & W. de Jonge, "The Identification of Objects and Roles", Technical Report TR-267, Faculty of Mathematics and Computer Science, Vrije Universiteit, 1993.
67. Winskel, G., "An Introduction to Event Structures", *REX'88: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency* Lecture notes in Computer Science 354, Springer-Verlag, 1988.
68. Yellin, D., & R. Strom. "Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors", *Proceedings, OOPSLA '94*, ACM, 1994.