Interface-Based Protocol Specification of Open Systems using PSL

Doug Lea
SUNY at Oswego / NY CASE Center
dl@g.oswego.edu

Jos Marlowe
Sun Microsystems Laboratories
jos.marlowe@eng.sun.com

November 30, 1994

Abstract

PSL is a framework for describing dynamic and architectural properties of open systems. 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 instances of interfaces, their attributes, and the issuance and reception of events. PSL accommodates refinement and extensibility of specifications along the three dimensions of interfaces, situations, and orderings. A specialized form, PSL / IDL describes protocols in CORBA systems.

1 Introduction

An *open system*, in the technical sense [28, 1, 63] (not necessarily the commercial sense) is encapsulated, reactive, spatially extensible, and temporally extensible:

Encapsulation. An open system is composed of possibly many components, each described by one or more public *interfaces* along with an otherwise inaccessible *implementation*. Each component relies only on the properties described in the interfaces of others.

Reactivity. Open systems do not just perform one service. Functionality of an open system is ''on-demand'', produced in reaction to a potentially endless series of requests.

Spatial Extensibility. Open systems are potentially distributed. Components need not bear any fixed connectivity relations among each other. They may interact via message passing mechanisms ranging from local procedure calls to asynchronous remote communication.

Temporal Extensibility. Components and functionality are not necessarily fixed across a system's lifetime. Typical characteristics include the ability to add new components and new component types (perhaps even while the system is running) as well as the structured evolution of 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 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.

1.1 Interface-Based Specification

Interfaces specify capabilities in open systems at varying levels of precision and formality. An interface describes only those services that clients 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 a complete or closed description of any given component. Any implementation that provides required functionality may be used.

Subtyping regimes over interfaces 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[50]. One interface type may describe only certain aspects of a role listed more completely in various subinterfaces. At an extreme, an interface might include only a single service operation.

Roles. Interface-based specification hinges upon the distinction between *roles*[64] and the *objects* that implement those roles. This distinction is similar to that 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*). In ordinary usage, 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. The concept of a role is nearly synonymous with that of a *subject* in the essential sense of Harrison & Ossher[25]. A role may also be thought of as an abstract *access channel*[61, 4] providing a *view*[59] of one or more components, where each view is described by an interface type that lists a set of related operations.

Implementation Objects. Interface specifications deal only with roles, not objects. They describe properties of abstract interaction participants[33], each defined via an interface that is described without commitment to the computational entity or entities that may implement it. Thus, the notion of a role is distinct from the programming-level notion of an *implementation object*. In fact, components need not be conceptualized as objects in the usual narrow sense of the term. Implementation objects may also consist of finer-grained instantiations (activations) of single procedures, or coarser-grained process-level components.

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 supports a *many-to-many* relationship between roles and objects, at both the type and instance level:

- One implementation object may support multiple interfaces concurrently, or different interfaces across time, and may be accessed via any of its roles.
- One role may be implemented via a group of otherwise unrelated objects.

For example, an `OfficeWorker` 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. Multiple interface instances, each seen from a different client, may access the same `Person` implementation.

CORBA. Systems constructed according to the OMG[49] CORBA model are among the best examples of the interface-based approach to 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.

However, pure interface-based specification does not assume any particular computational model or mechanics relating roles and objects. Descriptions of how each instance of a role is mapped to one or more implementation objects (and vice versa) lie outside of this framework. Any number of programming constructs and infrastructure schemes can be used to support at least certain subsets of the possible mappings. For example, most CORBA systems reify some aspects of roles as ''interface objects'' that support some of the above mappings by relaying messages to implementation objects. Other infrastructures (e.g., [9]) connect role-based messages via channels to multiple objects that together implement the role.

1.2 Extending Interfaces with Protocols

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. For example, 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 dynamic and architecture issues often overwhelm those surrounding service functionality. Descriptions of connection and communication patterns among components are more central in the characterization of many systems than are descriptions of services[21, 20, 27]. To address such issues, PSL adds descriptions of *protocols*. Protocols address the questions of *who*, *what* and *when* across interactions, as well as the structural relations underlying these dynamics. However, PSL differs from most other dynamic specification techniques (see Section 5) in order to reflect the characteristics of open systems:

Implementation Independence. PSL specifications describe dynamic and architectural matters without pinning down implementations, connections or communication mechanisms. PSL remains faithful to the distinctions between interfaces versus implementations that permit the development of open systems, while still providing a means to specify that implementations obey necessary interoperability constraints. The essential abstractions that support this include:

1. Separation of the notion of an instance of an interface from its implementation(s).
2. Separation of the notion of descriptions of transient states (generalized as *situations*) from their occurrences.
3. Separation of the notion that a message has been issued or received from concrete communication mechanics.

Openness. Anything that is not ruled out is assumed to be possible. PSL specifications are always ‘‘incomplete’’. They document sets of properties and constraints without claiming that these fully describe any component. In an open system, one cannot rule out the existence of components, situations, and events that are not explicitly excluded in a specification. Specification systems based on ‘‘closed world’’ assumptions (i.e., that the system implements only those features specified) are uniformly more powerful in addressing questions about liveness, deadlock, interference, and aliasing. Such questions often have no definitive answer in open systems[46, 30].

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. Here, a weaker set of rules is introduced in a broader context.

There are two styles of refinement. *Versioning*-based refinement is the act of inserting or extending rules within an existing context, adding greater specificity or enhancements in successive versions during the course of initial development and / or system evolution. Specification frameworks themselves cannot provide direct support for versioning, although compatible tools would facilitate use. *Specializations* are additions that are localized to new entities, described by new interfaces bearing subinterface relations to the originals. PSL supports specialization by extending common subtype-based tactics. New subinterfaces may extend interfaces, new subsituations may extend situations, and new ordering constraints on new situations may extend those described in existing rule sets.

2 PSL Concepts and Notation

In the course of defining concepts, we introduce *three* notations. Our primary PSL notation helps define basic abstractions. However, rather than establishing a general syntax for expressions, types, interfaces, and so on, we illustrate using PSL / IDL, a concrete syntax for PSL using CORBA IDL value and interface types and C / C++-style expressions, geared for use in CORBA systems. Also, because representations of even simple protocols stretch the limits of readability and writability in textual form, we simultaneously define PSL / IDL-G, a semigraphical form of PSL / IDL corresponding in simple ways to the textual representation.

2.1 Interfaces and Roles

As a basis for specification, we assume the existence of interface types that minimally include operation signatures in their declarations, where signatures are based on any reasonable type system on pure value types such as integers, booleans, reals, records, etc. PSL specifications provide additional constraints and rules that hold for all instances of the associated interfaces.

Roles are instantiations of interfaces. While roles are not the same as implementation objects, they bear several definitional similarities in terms of identity, state, and behavior[10, 18]:

- 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.

- A role may be ascribed abstract attributes that possibly vary across time. However, because roles do not necessarily bear a one-to-one relation to implementation objects, it is generally inappropriate to ascribe ''initial'' values of attributes to roles that hold across all instantiations. Specific operations (e.g., ''factories''[51]) may establish instances for which particular attribute values hold.
- A role may be said to issue and / or receive events. These events are realized by message passing, procedure calls, etc., among implementations.

Handles. For purposes of specification, each distinct instance of a role is referenced via a unique *handle*. Handles are an abstraction of names, pointers, static designations, etc., used as references to abstract participants and / or as message destinations. We assume the existence of one or more handle types, and handle values that may be used in any value context. Handles are *not* implementation-level pointers to concrete components, although there may be a one-to-one relationship between handles and pointers in a particular mapping to a programming language, tool, or infrastructure.

There may be several distinct handle types in a system. Handle types may include values that do not provide access to instances (as in the case of ''null'' and ''dangling'' references), in which case we say that the handle is not *live*. The equality operation ''=='' among handle values denotes only value equality among handles (also known as ''shallow'' equality), not necessarily identity or equality of implementation-level components.

Interfaces and Handles in PSL/IDL. PSL / IDL requires that each role type be described by a CORBA IDL interface. All PSL / IDL constructs appear within `protocol module` declarations that have the syntactic properties of IDL `modules` but contain only type declarations, attribute function declarations, and / or `rules` declarations. A PSL / IDL `rules` declaration is an optionally named scope, parameterized over one or more types, and containing protocol rules and / or constraint rules.

PSL / IDL uses C / C++ expression syntax over CORBA IDL value types. Predicates are `boolean` expressions. For convenience, we add logical implication ''`-->`'' and its left-sided version ''`<--`'' to the list of boolean operators. The fields of IDL `exception`, `struct` and `union` values are referenced using dot notation. To make them more useful in specifications, PSL / IDL predefines common pure value functions on the IDL `sequence` and `string` types. The functions `head`, `tail`, `empty`, `contains`, `prepend`, `append`, `concat`, `equal` and `remove` are predefined for sequences of all types for which there exists an equality operation. Definitions are identical to those on parameterized list types in functional programming languages such as ML[65]. PSL / IDL does not define types for sets, multisets, or maps. Sequence types may be used to equivalent effect.

PSL / IDL handle values are normally expressed using the syntax of IDL `objref`s. These serve as both references to instances of IDL `interface`s and destinations of CORBA messages. However, as discussed in Section 2.3, PSL handle values need *not* bear a one-to-one relation with `objref` values in CORBA implementations. Also, to accommodate the range of options available in CORBA systems, PSL / IDL rules may also be described at the level of individual activations of individual operations. Handles of the form *InterfaceHandle*`::`*OperationName* denote abstract activations (executing instances) of the indicated operations.

2.2 Situations

The concept of a situation extends the common notion of abstract object state[42, 6]. Situations describe *partial* views of possible system-wide states, factored with respect to particular roles. Thus, a situation may describe the abstract state of multiple components. Situations are defined declaratively, at any level of granularity. In PSL, situations are represented by parameterized expressions describing ''interesting'' commonalities of transient ''possible worlds''. These expressions classify aspects of worlds in terms of characteristic values of relevant attributes and event predicates with respect to one or more roles.

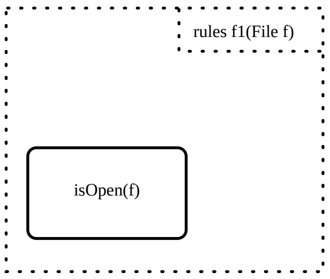
In the same sense that an interface is instantiated by a role, a situation is said to be instantiated by a *realization*. And just as interfaces describe the properties of an unbounded family of potential implementations, situations describe the properties of an unbounded family of potential realizations. However, realizations are not *explicitly* instantiated by programmers. They just unfold over the course of system execution. For example, a situation might contain only the constraint that a given `File` is open. The realizations of this situation in a particular system include all ''snapshots'' of the system in which this `File` is open. The manner in which such realizations are observed or inferred in executing systems is outside the scope of PSL (see

Section 4).

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 [38] for details). Two or more realizations may all match the same situation but in different ways because they differently instantiate free parameters listed in the situation.

Situations in PSL/IDL. A situation is represented as a parameterized predicate. We provide a full notation in [38], but use here an abbreviated form in which situations are defined within `rules` declarations as boolean C/C++-style expressions within braces. Expressions inside situations may reference attributes and event predicates, and may include in-line local declarations and references to named message arguments. PSL/IDL-G notation is the same except that situation expressions are demarcated by solid rounded boxes inside a `rules` declaration drawn with a dashed box with name and parameters in the corner. For example, the following situation describes the state of a `File` being open:

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



2.3 Events

An attribute is an abstract property ascribed to zero or more role instances, and possibly assuming different values in different realizations. In PSL, an 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, exceptions, and so on. (The declaration and use of attributes that are not directly tied to communications events are discussed in Section 2.5.)

Messages Types. 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). We will illustrate PSL usage in systems of fixed message types. However, at an extreme, all messages in a system might have the same selector, with an argument format requiring dynamic interpretation. Message types may vary over any number of dimensions. For example, all messages of a certain type may include fields suitable for use in routing over a particular topology of distributed components.

Directed Messages. Message types (and the underlying transport media) may support special ‘‘addressing modes’’. In particular, we illustrate using systems of *handle-directed* messages, where a ‘‘destination’’ handle describing the intended receiver(s) is a required part of any message. Similarly, the message type corresponding to a procedural operation that returns a reply (or perhaps just a ‘‘`void`’’ completion indication) includes a handle describing the ‘‘return address’’ of the caller. While the destination and return values in directed messages normally bear a one-to-one correspondence with actual receivers and senders, this need not be so, for example in systems supporting implicit forwarding of delegated messages.

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. As a notational convenience, a `reply` or `throw` without a bracketed message argument refers to the most closely associated message whenever this is unambiguous.

Event Predicates. Messages themselves are not used directly in PSL situation descriptions. Instead, PSL contains two families of special attribute functions, `in` and `out`. Inside situations, `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. Omission reflects lack of commitment about which participant issues or receives a message.

PSL event predicates are otherwise treated as attribute functions that happen to have predefined characteristics. In particular, events never "unhappen": *Once a predicate describing the occurrence of an individual event becomes true, it never becomes false.* 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; and similarly for `in(`m`) by(`p`)`. Situations and protocol rules may thus be phrased in terms of events that have occurred at any time (cf., [43]).

Messages in PSL/IDL. All messages in PSL / IDL are handle-directed. While PSL / IDL message conventions correspond closely to those used in CORBA, the relation need not be one-to-one. Mappings between handles and the values used in CORBA message destination fields may take several forms, even within the same CORBA implementation[53, 26]. For example, channel-style `objref`s may be used. Channel values [61] represent "paths" to role instances. Two or more messages sent with the same channel identifier reach the same instance, but two channel values that access the same interface instance may differ. Object-reference style `objref`s may also be used so long as messages are always delivered to implementation objects corresponding to the appropriate roles; for example when references are routed through proxies that relay messages to the appropriate implementations.

PSL / IDL message types are abstractions of `CORBA::Request`, with a shorthand handle-based message syntax delimited by angle-brackets:

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

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; in the case of procedural (reply-bearing) messages, src is an optional handle describing the destination of a reply 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. Examples:

```
in( <aFile->write(c)> )
out( reply[<aFile->read()>](c) )
out( throw(exc) )
```

The types of `aFile`, `c`, and `exc` would be established within the scope of some `rules` declaration or in referenced IDL `interface` declarations. Bindings for arguments follow normal IDL rules. Details on how these map to base PSL constructs may be found in [38].

If necessary, the particular role instances issuing or receiving messages may be indicated using `by(`*Handle*`)`. For example,

`in(<aFile->write(c)>)by(aFile::write w)`

allows a shift to an activation-level description focused on a single `write` operation.

2.4 Protocol Rules

PSL protocol rules describe conditions under which realizations of situations occur. Rules are collections of situations linked by temporal operators. PSL temporal operators 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 2.9 for a more formal characterization.) The means by which this relation may be observed or arranged are outside of the scope of PSL proper (see Section 4).

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:

$$A \blacktriangleright B \overset{\text{def}}{=} \forall a : a \propto A \Rightarrow \exists b : b \propto B \wedge a \preceq b$$
$$A \blacksquare B \overset{\text{def}}{=} \forall b : b \propto B \Rightarrow \exists a : a \propto A \wedge a \preceq b$$
$$A \blacklozenge B \overset{\text{def}}{=} (A \blacktriangleright B) \wedge (A \blacksquare B)$$

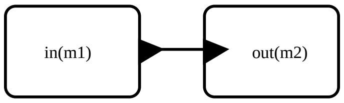
Rules in PSL/IDL. In PSL / IDL-G, rules are expressed via lines connecting situations through corresponding symbols drawn on the outside of the predecessor situation to the same symbol drawn on the inside of the successor situation. In PSL / IDL, these operators are designated as `|>`, `#` and `<>` respectively. The "earlier" situation is always to the left of the "later" one, and the choice of operator depends on the kind of relation.

PSL / IDL (and PSL / IDL-G) rules always lie within the scope of a given `rules` declaration. PSL / IDL situations are implicitly parameterized by the arguments listed in their `rules`, as well as all declarations in their predecessor situation(s). The PSL matching relation ("$\propto$") requires that unambiguous names be used in any two situations related by operators. To ensure this, any situation expression may reference, but not redeclare, any symbol declared in its `rules` and its predecessors. Thus, all value names in a set of ordered situations must be unique. Details may be found in [38]. In the remainder of this section, we usually illustrate with isolated rule fragments, ignoring most scoping issues. More complete examples are presented in Section 3.

The ▶ Operator. The "forward reasoning" operator $A \blacktriangleright B$ is used for relations in which A *leads to* B. The relation is akin to that of a state transition, applying to cases in which As always precede Bs. 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 or interleavings 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 considered to be aspects of the situations themselves.

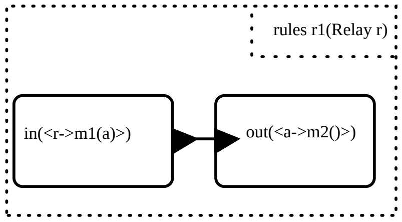
For example, a protocol rule for a relay operation that sends `m2` whenever `m1` is received takes the form:

```
{ in(m1) } |> { out(m2) }
```



In the scope of a particular PSL/IDL protocol, we would have to be more explicit about types and messages. For example, a relay that accepts `a` (where `a` is a handle of a role supporting operation `m2`) as an argument of `m1` and then issues `m2` to `a`:

```
rules r1(Relay r) {
 { in(<r->m1(a)>) } |> { out(<a->m2()>) }
};
```



This rule does not in any way imply that `Relay` operations must be single-threaded. Because there are no constraints that indicate 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.

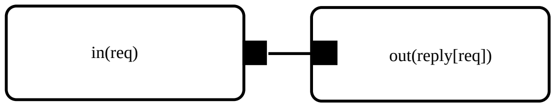
Once a predicate describing the occurrence of an individual event becomes true, it never becomes false. Thus, event predicates "latch" from left to right in PSL linked situations. For any event predicate e, if e holds in A, then it also holds in all successors of A. (This property does *not* necessarily hold for expressions on attributes that are not tied to event predicates; see Section 2.5). For example, the first relay rule is equivalent to one explicitly mentioning `in(m1)` in the successor situation:

```
{ in(m1) } |> { in(m1) && out(m2) }
```

The ■ Operator. The "backward reasoning" operator $A \blacksquare B$ is used for relations in which A *enables* B, applying to cases in which Bs are always preceded by As (or are "caused" by As, 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) } # { out(reply[req]()) }
```

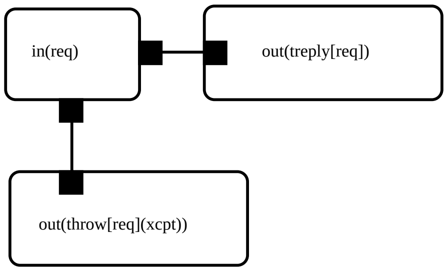


This rule says that replies are sent only if messages are received. It does not say that requests always lead to replies, only that replies are never sent unless preceded by requests. In PSL/IDL, this rule is considered to be predefined for all procedural requests, since it is enforced by CORBA.

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:

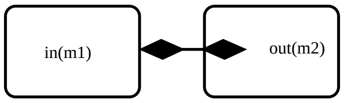
```
{ in(req) } # { out(throw[req](xcpt)) }
```

This may be pictured together with the first rule, reflecting the fact that rules are combined conjunctively (i.e., implicitly *and*ed via $\wedge$):



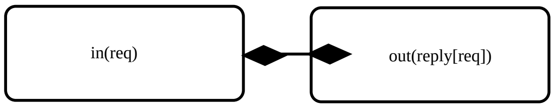
The ◀▶ Operator. The $A \blacktriangleleft\!\blacktriangleright B$ operator is used for if-and-only-if relations, in which both every A is followed by a B, and every B is preceded by an A. For example, another desirable global rule is the *one-to-one delivery rule*, for any `m`:

```
{ out(m) } <> { 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:

```
{ in(req) } <> { out(reply[req]()) }
```

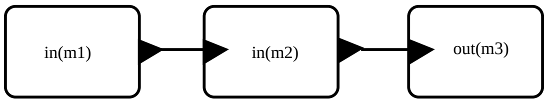


This strengthens the above $\blacksquare$ form to assert that a reply to `req` must be issued, thus precluding the possibility of exceptions or other non-procedural behavior.

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. Chains across the other operators are expressed similarly.

For example, a special protocol in which a relay outputs `m3` after receiving the ordered fixed messages `m1` followed by `m2` could include a rule of the form:

```
{ in(m1) } |> { in(m2) } |> { 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) } |> { out(m3) }
```

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

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

Operator Properties. In addition to their definitions in terms of $\preceq$, the principal PSL operators may be characterized in terms of inferences that they support. These include, for all A, B, C containing predicates meaningful in their scopes:

$$
\begin{array}{ll}
A \blacktriangleright \{\mathtt{TRUE}\} & A \blacksquare \{\mathtt{FALSE}\} \\
\{\mathtt{FALSE}\} \blacktriangleright A & \{\mathtt{TRUE}\} \blacksquare A \\
A \blacktriangleright B, B \blacktriangleright C \vdash A \blacktriangleright C & A \blacksquare B, B \blacksquare C \vdash A \blacksquare C \\
A \blacktriangleright B, B \blacklozenge C \vdash A \blacktriangleright C & A \blacksquare B, B \blacklozenge C \vdash A \blacksquare C \\
A \blacklozenge B, B \blacktriangleright C \vdash A \blacktriangleright C & A \blacklozenge B, B \blacksquare C \vdash A \blacksquare C
\end{array}
$$

Note that no simple relation between A and C can be derived given only $A \blacktriangleright B$ and $B \blacksquare C$ or vice versa.

2.5 Attributes

So far, we have illustrated protocol rules using only special attributes representing event reception and issuance. But PSL specifications may include declarations of attributes of any kind. In PSL, attributes are abstract functions of handle and/or other value type arguments; for example function `isOpen` takes a `File` handle as an argument.

As with other PSL constructs, the relationship between abstract attributes and their implementations (if any) is outside the scope of the framework. For example, the attribute `isOpen` is a hypothetical function that might actually be computed (perhaps only approximately; see Section 4) 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. We use function syntax in PSL/IDL rather than IDL `attribute` syntax. In IDL, the keyword `attribute` is only a stylistic device for declaring parameterless *operations* within interfaces. For example, assuming existence of an IDL `interface File`:

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

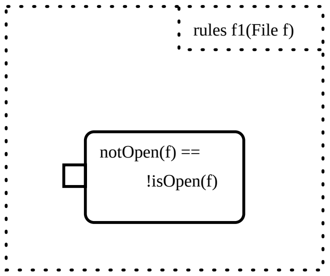
Among the most common forms of attributes used in PSL are "relational" or "connection" attributes that represent the values of handles that a role uses to communicate with others. For example, a `File` might be ascribed attribute `IODevice dev(File f)` representing a handle used in I/O requests. As is true for any attribute, the values of connection attributes may vary over time, subject to any other listed constraints.

2.6 Constraint Rules

Some conditions must *always* hold, across all possible contexts. 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, 62], operator $\Box$. The expression $\Box P$ indicates that P 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 `[]`. In PSL/IDL-G notation, they are listed with an unfilled box on their sides. For example:

```
rules f1(File f) {//...
 [] { notOpen(f) == !isOpen(f) }
};
```



Constraint rules limit the set of possible system states to those in which the expressions hold. All realizations must obey $\Box$ constraints in addition to those explicitly listed in situations. (All `rules` parameters implicitly serve as universal quantifiers for the enclosed logical expressions, and other ''inline'' declarations are implicitly existentially quantified.)

Relating attributes to events. One common use of constraints is to relate non-monotonic time-varying attributes to PSL ''latching'' event predicates. 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. Among other possibilities, we could write this via the pair of constraint expressions:

```
rules f1(File f) { //...

[] { isOpen(f) -->
     out(reply[<f->open()>]()) }

[] { out(reply[<f->close()>]()) -->
     !isOpen(f) }
};
```

Informally, 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 constraints might be buttressed with a description of a `FileFactory` operation that guarantees that `isOpen` is true upon reply of its `File` handle result.

While constraints are typically used to relate conceptual attributes to event predicates in this manner, it is not at all required, and sometimes not even possible to do so. For example, 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 above would not hold. More generally, attributes associated with ''base'' interfaces are often only weakly constrained. They are further constrained in declarations associated with different subinterfaces.

While convenient, and sometimes unavoidable, the use of unconstrained attributes is also notoriously troublesome in practice[42, 11, 18], and requires care in specification. 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 (see Section 3.4). As a matter of style, it is a good idea to minimize use of such attributes.

Restricting events. Constraint rules may also relate event predicates. PSL does not notationally distinguish constraints that are ''definitionally'' true versus those that are required to be true as a matter of system policy. For example, to reflect the common requirement that either a normal reply or an exceptional reply can be issued for a procedural request `req`, but at most one of these, we could (tediously) list constraints:

```
[] { out(reply[req]()) -->
     !out(throw[req](x)) }
[] { out(throw[req](x)) -->
     !out(reply[req]()) }
[] { out(r1 = reply[req]()) &&
     out(r2 = reply[req]()) -->
     (r1 == r2) }
[] { out(t1 = throw[req](x)) &&
     out(t2 = throw[req](y)) -->
     (t1 == t2) }
```

Because CORBA (like most systems) enforces these conventions for all messages, such rules are predefined for all message and exception types in PSL/IDL.

Uniqueness. The third and fourth rules above employ a standard logic trick for declaring uniqueness. The third rule says that if there are two values matching `reply[req]()` then they must be the same message. This kind of constraint is common enough that we define the PSL / IDL "macro" `unique(`$expr$`)`, which is false if there is more than one match. For example, we could rephrase the last two rules above as:

```
[] { unique(reply[req]()) }
[] { unique(throw[req](x)) }
```

2.7 Subsituations

Like interfaces, states and classes, situations may be specialized into *subsituations* that describe additional features. 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 operators $A \cap B$ and $A \cup B$ in terms of the corresponding boolean relations on their component expressions (cf., [18, 59]). In PSL / IDL $A \cap B$ is expressed as $\{expr_A\}$ `&&` $\{expr_B\}$, and $A \cup B$ as $\{expr_A\}$ `||` $\{expr_B\}$.

Subsituation relations are analogs of the subtype relations underlying interface inheritance. The simplest and most common means of constructing a subsituation is to "strengthen" an expression by adding an *and*'ed 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 `in(`m`)` might be strengthened by replacing it with the more committal `in(`m`)by(`p`)`.

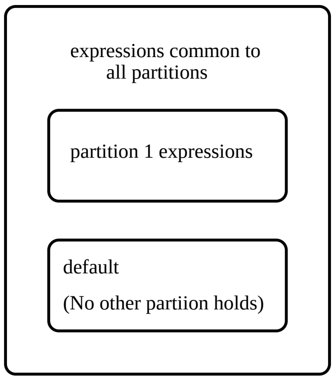
Valid inferences mixing situation relations and temporal operators include:

$$\begin{array}{ll}
A \blacktriangleright P \cap Q \vdash A \blacktriangleright P & A \blacksquare P \cup Q \vdash A \blacksquare P \\
P \cup Q \blacktriangleright A \vdash P \blacktriangleright A & P \cap Q \blacksquare A \vdash P \blacksquare A \\
P \blacktriangleright A, Q \subseteq P \vdash Q \blacktriangleright A & Q \blacksquare A, Q \subseteq P \vdash P \blacksquare A \\
A \blacktriangleright Q, Q \subseteq P \vdash A \blacktriangleright P & A \blacksquare P, Q \subseteq P \vdash A \blacksquare Q
\end{array}$$

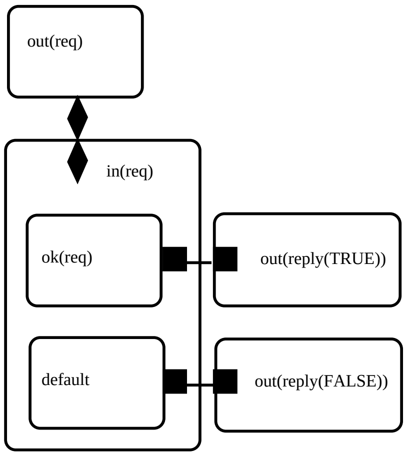
Partitioning. We do not use any special notation to *declare* that one situation is a subsituation of another except in the special case of disjoint union where $S = S_1 \oplus S_2 \oplus \ldots \oplus S_n$. This represents a set of subsituations S_i that are constrained to completely partition a situation-space S. Partitioned subsituations are mutually exclusive. They cannot simultaneously hold. Partitioning is thus one way to express the notion that one situation "disables" another[66]. In PSL / IDL, partitioning is expressed using:

$\{$ `case` S_1 `case` $S_2 \ldots$ `default` $\}$.

Successive `case`s are interpreted in the same way as `case`, `cond`, and `if` ...`elsif` statements in most languages, implicitly negating the expressions in all previous `case`s; `default` acts as a generalized `else`. In PSL / IDL-G, partitioning is expressed via nested stateChart-like [24] boxes that may contain expressions common to all partitioned subsituations outside of the nested boxes:


expressions common to all partitions
partition 1 expressions
default
(No other partiion holds)

Conditionals. The most common use of partitioning in PSL is to express conditional protocol rules. Ordering operators "distribute" through situation partitionings to describe conditional paths. For example, to indicate that a `TRUE` reply is enabled for a request if some function `ok` holds, and conversely for `FALSE`:



In PSL / IDL, ordering operators have lower precedence than the && operator combining situations, and expressions common to partitions expressions must be described using &&. Thus, the above example is expressed as:

```
{ out(req) }
<>
{ in(req) } && {
  case { ok(req) } # { out(reply(TRUE)) }
  default          # { out(reply(FALSE)) }
}
```

2.8 Refinement and Generalization

Refinement constraints maintain consistency among different rules and contexts. PSL protocol declarations are parameterized with reference to interface types. Specialized protocol declarations may be attached to (parameterized by) instances of subinterfaces. In PSL / IDL, when two kinds of roles differ in protocol but not operation signatures, this may require construction of ‘‘dummy’’ IDL interface types just to give the two types different names (cf., [5]. Attaching new rules to subinterfaces allows commitment to more specific protocols in special cases, without overcommitting in the general case. Similarly, placing generalizations of existing rules in a supercontext supports simpler high-level views and allows other alternative specializations.

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 refinement techniques include the following:

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 specified end-points. For example, a coarse-grained specification of a rule for a procedural operation might list only the request and reply:

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

A refinement may then specify internal structure such as an interaction with a helper:

```
 { in(req) } <>
  { out(h = <helper->help()>) } #
   { in(reply[h]()) } <>
    { 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 abbreviates the form $(A \blacklozenge B) \wedge (B \blacksquare C) \wedge (C \blacklozenge D)$ where A, B, C, and D represent the situations in each line of the above PSL / IDL. 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 say:

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

A refinement may split apart the conditions under which it replies true versus false:

```
 { in(req) } && {
   case { badstuff() } #
    { out(reply(FALSE)) }
   default #
    { 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 $\blacksquare$ 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 $\blacklozenge$ to make the stronger claim that this exception is *always* generated under this condition. Similarly, we could strengthen the previous example to use $\blacklozenge$ instead of $\blacksquare$ if we were sure that the listed situations represented the only ways in which the replies could occur.

Weakening and Strengthening Situations. If $A \blacktriangleright B$ 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[56], 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) } |> { 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 ($B' \subseteq B$), thus logically subsuming the original version:

```
{ in(m1) } |> { 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.

2.9 Foundations

A more formal account of PSL constructs can be provided from the perspective of model theory [31]. Structures for possible worlds are generally of the form $< \mathbf{W}, \mathbf{V}, \mathbf{R} >$ where **W** is a set of worlds. **V** is a set of expressions over some basis, with an associated function $\phi(p, w)$, which is true if expression p holds in world w.

R here stands for 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.

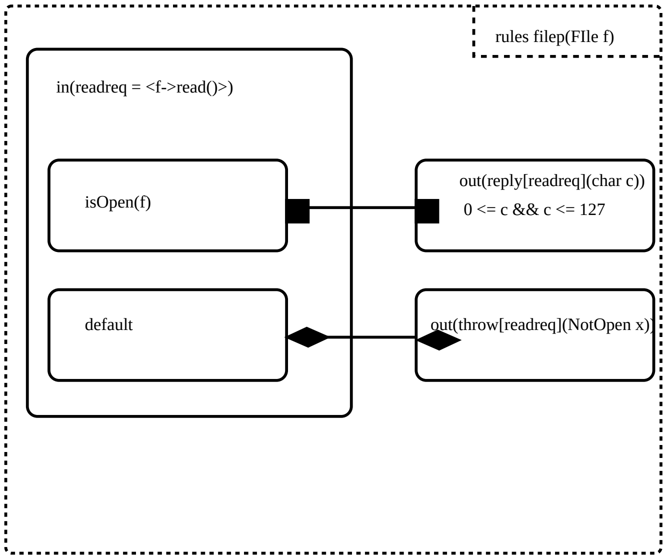
The relation $\mathbf{R}_\preceq$ serves as the basis for PSL ordering operators. This relation is simplest to describe formally when expressions are restricted to event predicates on fixed messages[23, 66, 55]. In this case expressions in **V** are just characteristic predicates of sets of event occurrences, 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 $e_b \Rightarrow e_a$. When attribute values are unconstrained with respect to events, the relation does not intrinsically reflect whether or how they vary.

3 Examples

3.1 Role Specifications

Even though protocols describe multi-participant interactions, PSL declarations are frequently "one-sided", describing a role with respect to a single interface. This captures the modularity of protocols in which a participant's interactions do not vary with respect to the types or states of others that it interacts with. For example, here is a fragment of a protocol for a simple `File`:


rules filep(FIle f)
in(readreq = <f->read()>)
isOpen(f)
out(reply[readreq](char c))
0 <= c && c <= 127
default
out(throw[readreq](NotOpen x))

Here, 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 ◆, 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`. Had we wanted to make the weaker claim that `NotOpen` is possible but not guaranteed, we would have used ■. 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 the ▶ operator.

3.2 Interactions

One-sided protocol descriptions can be useful even when interactions are focused upon fixed sets of communications partners. For example, consider the following fragments of a protocol for transactions in which a `Coordinator` helps arrange the actions of `Transactor`s, with IDL interfaces:

```
typedef long TID;

interface Transactor {
 boolean join(in TID tid);
 boolean vote(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 `Coordinator`s create (via `begin`) transaction identifiers (`TID`s) that are used to index transactions. Each transaction consists of a group of `Transactor` members, added via the `add` operation. The particular application operations that each perform within transactions (perhaps bank account updates) are not described in this set of interfaces. The use of interface inheritance indicates that members may be other `Coordinators`. Each `Transactor` may be asked to `join` a transaction, and `vote` on whether to `commit` versus `abort`.

To capture some of this in PSL / IDL, we first declare a `protocol module`:

```
protocol module CoordM { // ...
```

Inside this module, we declare an attribute that will 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`). PSL / IDL does not support any kind of *set* construct, but we can use a `sequence` to equivalent effect:

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

Next is the auxiliary function `validtid`:

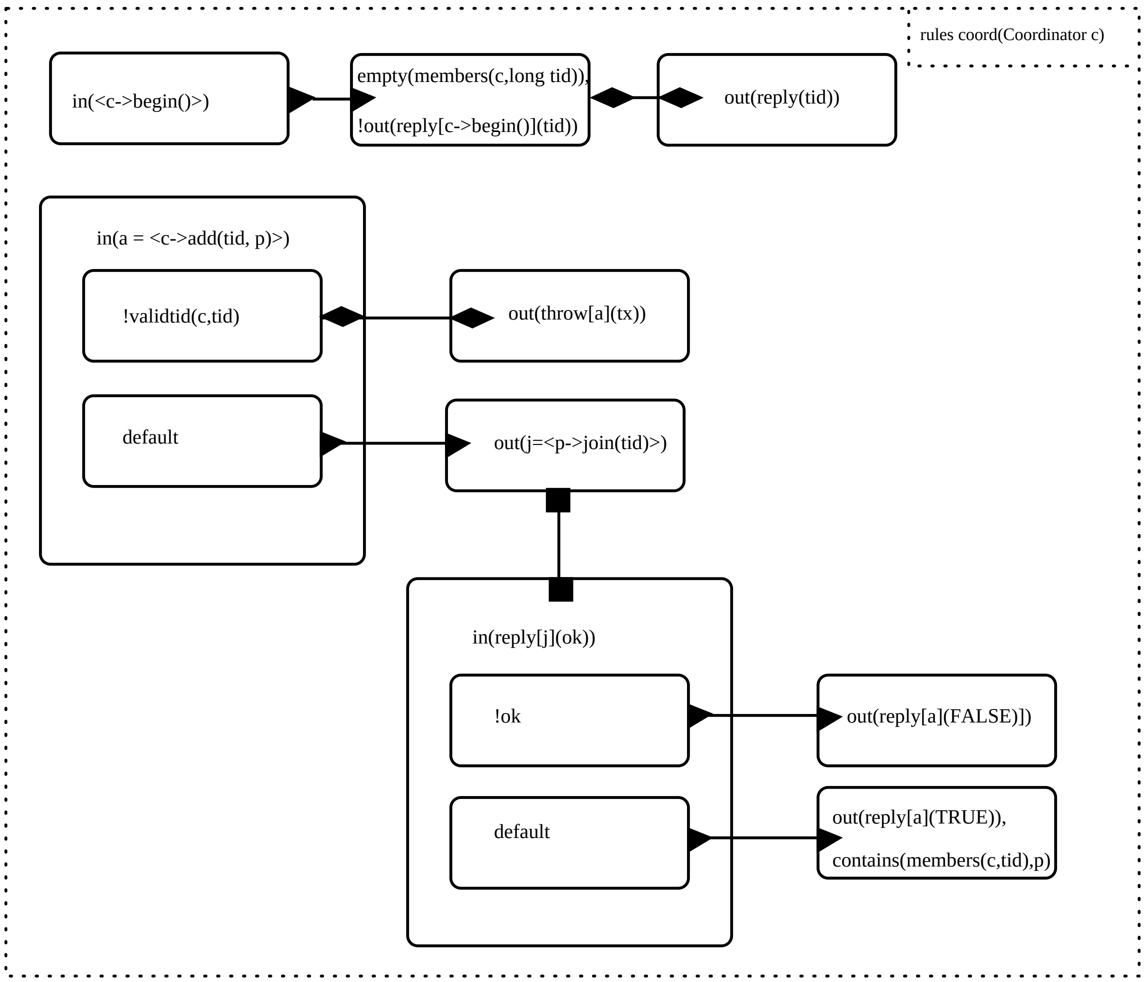
```
 boolean validtid(Coordinator c,
                  TID tid);
```

A corresponding constraint declaration shows how `validtid` 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. For simplicity, we can isolate these constraints in a separately named and scoped `rules` declaration:

```
rules valid(Coordinator c, TID tid) {
[] { validtid(c, tid) -->
     out(reply[<c->begin()>](tid)) }
[] { in(<c->abort(tid)>) -->
     !validtid(c, tid) }
[] { in(<c->commit(tid)>) -->
     !validtid(c, tid) }
};
```

Two sample rules are shown in the accompanying `rules coord(Coordinator c)` 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. In this way PSL / IDL 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.

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.

Finally, we may add an additional constraint precluding the existence of any additional operations or rules that cause p to become a member unless they somehow invoke add:

```
rules(Coordinator c,Transactor p,TID t){
[] { contains(members(c,t), p) -->
    out(reply[<c->add(p,t)>](TRUE)) }
};
```

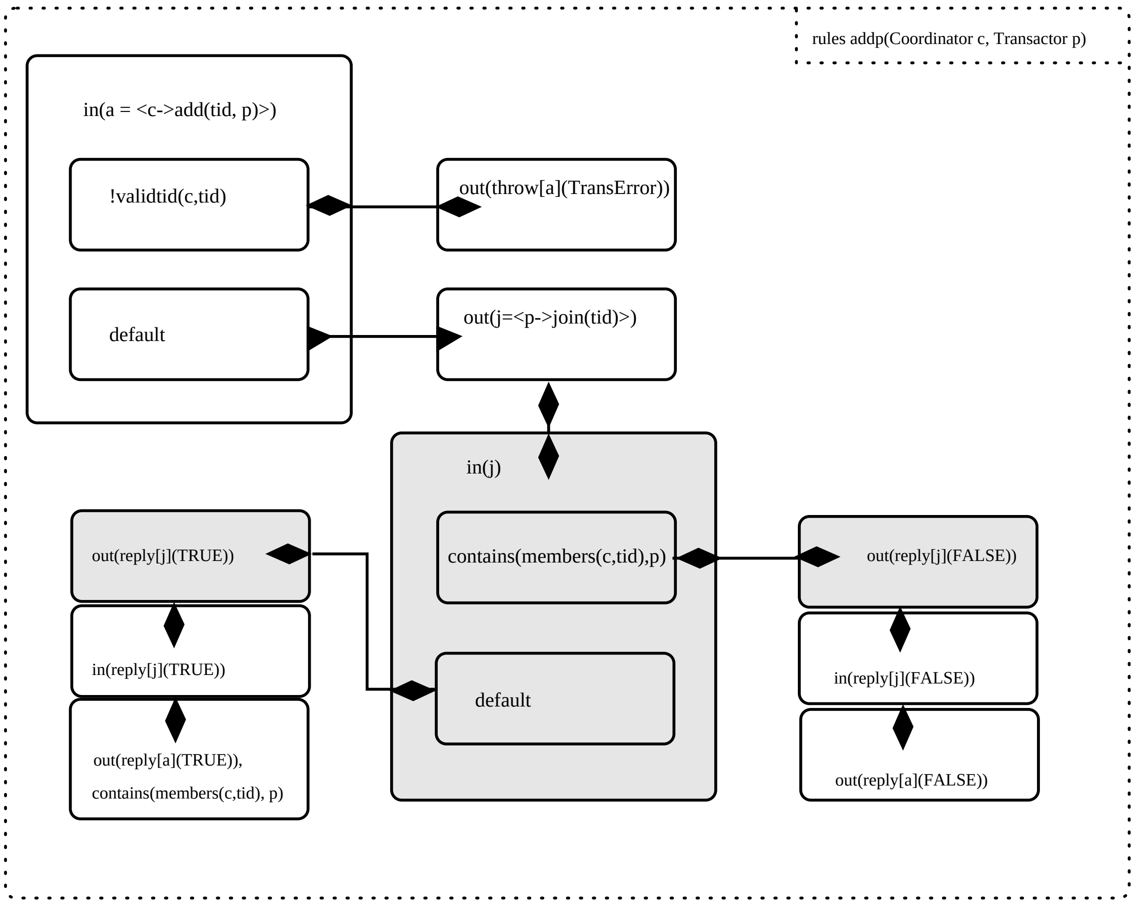
3.3 Multiple participants

Parameterization is used in PSL/IDL both to fix the frame of reference for a set of related rules, as well as to encourage modular specification. While situations in a given `rules` declaration may describe properties of any roles for which a handle is available, it is stylistically preferable to limit attribute and event expressions to those declared as parameters to a given `rules` declaration.

Parameterization over multiple participants allows the perspectives and responsibilities of all parties need to be taken into account simultaneously. This allows greater precision in specification, although with a concomitant loss of modularity. This is especially useful in those cases where any party responds in special ways to another that do not apply to interactions with other kinds of participants. Such rules may be seen as the specification analog of multimethods[18].

For example, we may rework a more committal version of the `add` rule illustrated in the accompanying partial PSL/IDL-G declaration of `rules addp(Coordinator c, Transactor p)`, which 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.

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.

Also, in this particular example, there is nothing in PSL/IDL stopping us from entering the rule in `addp` directly into the `coord` declaration in the previous section. Because `p` is available as a handle via the argument to `add`, we could have included the situation descriptions directly referencing `p` inside the original `add` rule, but with needless loss of modularity. Moreover, in any full protocol description including rules for `commit` etc., in which multiple rules would need to refer to the same `p` and `c`, we would have no alternative to multiple parameterization.

Tools can alleviate such trade-offs between modularity and specificity. Rules in modular PSL declarations often represent ‘‘snippets’’ [60] of a longer protocol. Tools may show a more complete view of a protocol by linking situations described in one scope to those in others. Such tools may rely on global axioms such as the *one-to-one delivery rule* when available to match `in`s with `out`s, along with special rules for matching the obligations and expectations of particular partners[17]. For example, a tool might generate a multi-participant view of `add` given the single-view forms of `Coordinator::add` and `Transactor::join`. The resulting *timethread*[12] is a path linking initial situations to terminal ones.

3.4 Additional Constraints

Recall the equivalence of the relay rules in Section 2.4:

```
{ in(m1) } |> { out(m2) }
{ in(m1) } |> { 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)} |> {out(m2) && ok(r)}
```

Constraint rules may be used to avoid such problems. Constraint rules add requirements that do not otherwise come ‘‘for free’’ in open protocol specifications. Consider, for example, interfaces describing Accounts that maintain balances:

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

interface AccountFactory { // ...
 Account makeAcc(long sn, float initbal);
};
```

As a start, we can declare a protocol module with abstract attributes bal and serialNo, along with simple postcondition-style rules stating that makeAcc initializes bal and serialNo, getBalance ‘‘reads’’ bal, setBalance ‘‘writes’’ bal, and getSerialNo reports serialNo:

```
protocol module accountm {
 float bal(Account a);
 long  serialNo(Account a);

 rules (Account a) {
  { in(<a->getBalance()>) }
  <>
  { out(reply(bal(a)) }

  { in(<a->setBalance(b)>) }
  <>
  { out(reply()) && bal(a) == b }

  { in(<a->getSerialNo()>) }
  <>
  { out(reply(serialNo(a)) }
 };
 rules (AccountFactory f) {
  { in(<f->makeAcc(sn, initbal)>) }
  <>
  { out(reply(a)) && live(a) &&
    bal(a) == initbal &&
    serialNo(a) == sn }
 };
};
```

Encapsulation Constraints. It is useful here to add a further constraint saying that the setBalance and makeAcc 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 may be expressed by relating bal to values associated with replies from either of the two operations:

```
rules (Account a, AccountFactory f, float b){
[] { (bal(a) == b) -->
     out(reply[<f->makeAcc(s, b)>](a)) ||
     out(reply[<a->setBalance(b)>]()) }
};
```

Initialization Constraints. A similar tactic may be used to describe attributes whose values are fixed forever upon initialization. For example, to claim that the serialNo is always the one established by the factory operation, and further claim that initialization occurs at most once per account:

```
rules (Account a, AccountFactory f, long s){
[] { (serialNo(a) == s) -->
     out(reply[<f->makeAcc(s,b)>](a)) }
[] { unique(
     out(reply[<f->makeAcc(s,b)>](a)) }
};
```

Single-Threading Constraints. We could further require that processing of setBalance requests is not subject to arbitrary interleavings (i.e., that setBalance operations proceed serially), thus precluding multithreaded implementations. Again, without such a constraint, there is nothing forcing this interpretation. The restriction that no two setBalance operations operate concurrently may be expressed by saying that any message that has been received but not replied to is unique:

```
rules (Account a, float b) {
[] { unique(
      in(s = <a->setBalance(b)>) &&
      !out(reply[s]())) }
};
```

Timing Constraints. The relative ordering approach to protocol specification does not directly admit the use of global timing constraints. However, it is very much possible to describe constraints with respect to one or more *timers*. (Although the physical/temporal properties of timers themselves remain outside the scope of PSL.) One way to impose such constraints is via ‘‘client-side’’ protocol rules. For example, to state that any client issuing a getBalance receives a reply within N time units of some Timer with attribute ticks:

```
rules (Object client, Account a, Timer tmr) {
 { out(m = <a->getBalance()>)
     by(client) &&
   long t1 == ticks(tmr) }
 <>
 { in(reply[m](b)) by(client) &&
   long t2 == ticks(tmr) &&
   t2 - t1 <= N }
};
```

4 Methods and Tools

While PSL represents the core, it is only one piece of a unified approach to the specification of open systems. A complete account requires models, languages, and / or tools that *map* these abstractions to concrete features of particular systems in order to construct corresponding design methods and tools; for example simulation, prototyping, verification, visualization, testing, and monitoring. Such applications rely upon mappings relating any given specification to code that may conform to it. These mappings naturally vary across the languages, tools, and infrastructures used to implement a system:

1. Mappings between roles and implementation objects (components).

2. Mappings between expressions defining situations and realizations observed or inferred in concrete code and / or its execution.

3. Mappings between events and concrete communication occurrences.

Additionally, several of these applications require descriptions of certain initial conditions of the system of interest.

We illustrate some general mapping issues with PSL / IDL. While the use of PSL in some systems requires 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 (e.g., C++[52]).

- PSL / IDL message types map directly to those used in CORBA. Observations of messages may be used to establish instantiation of corresponding event predicates.

- CORBA Object Request Brokers (ORBs) and repositories dynamically 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.

- Typically, the initial conditions of a CORBA application 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[67]. 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 [38]). 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.

5 Related Work

The ways in which PSL constructs support interface-based specification of open systems distinguish it from most other approaches to protocol specification and architectural description. While all such approaches may be related at some level, they differ significantly in their theoretical bases, definitional primitives, and range of usability. The following brief comparison with some well-known formalisms highlights their differences.

Preconditions and Postconditions (e.g., in Hoare Logic [29]) 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, like most other specification systems (e.g., [35]), does not include any language-specific operational semantics, and omits reference to s. PSL additionally differs in its scoping and parameterization of situation predicates.

Abstract Data Types (ADTs) (e.g., [39, 58]) describe functional properties of ‘‘black box’’ components (e.g., 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 (e.g.,[40, 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 reflecting the general purpose communication substrate of their infrastructures; for example, ORB-mediated communication in CORBA systems.

Object-Oriented Analysis notations, at varying degrees of formality (e.g., [57, 33, 18]) describe classes of objects in terms of attributes, relations, states, operations, and messaging. PSL generalizes, extends and reworks the dynamic aspects of such concepts to apply to interfaces of components in open systems. Unlike some other approaches (e.g., [2, 67]) 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.

Process Calculi (e.g., CCS[45]) and specification languages based upon them (e.g., LOTOS) 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 (e.g., [43, 47, 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 $\prec$ relation that may be seen by any particular implementation *object*. Thus PSL history patterns cannot be described as languages or regular expressions[54]. 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 (e.g., [36, 54, 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[66, 23, 55], as adapted for use in interface-based specification. Derivation of more complete ties to such frameworks remains the subject of further study.

Temporal and Modal Logic. As discussed in Section 2.9, PSL/IDL is an application of temporal and modal logic[19, 62], akin to other frameworks (e.g., [37, 41, 14, 34]) that adapt temporal logic for specifying possibly distributed systems, as well as related applied temporal reasoning systems used in AI and object-oriented logic programming (e.g., [42, 3, 16]). While nearly all such efforts rely on a $\Box$ operator or its equivalent to describe necessary constraints over possible worlds, there is considerable variation in how other temporal relations are defined and used. The choice of relational operators $\blacktriangleright$ and $\blacksquare$ as a basis for PSL appears to be unique in system specification, but has parallels in tense logic in linguistics[62] and situation theory in philosophy[6, 7].

A notable difference between PSL and most adaptations of temporal logic for system specification (e.g., TLA[37]) is that PSL omits any kind of *step* operator. In temporal logic, *step* asserts that one predicate occurs at the ‘‘next time step’’ after another. The omission of *step* weakens analytic properties, but makes way for refinement and extensibility properties necessary in the development of open systems. The lack of *step* is analogous to the lack of a *leaf* assertion for IDL interfaces or PSL/IDL situations. A *leaf* directive for interfaces would assert that an interface has no possible extensions (i.e., can support no additional operations). A similar directive on situations would assert that a situation has no possible subsituations (i.e., can support no further state decomposition; hence no unmentioned event predicates). A *step* operator would further assert that there are no possible intervening situations. All of these assertions might be useful or even necessary when analyzing partially closed subsystems, but are at best problematic in the description of open systems.

Acknowledgments

Thanks to the members of the Sun PrimaVera and Vantage groups, and to Desmond D'Souza, Dennis de Champeaux, Peter O'Hearn, 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] Barwise, J., *Situations and Attitudes*, MIT Press, 1983.

[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 Processs 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] Coad, P. & E. Yourdon, *Object-Oriented Analysis*, Yourdon Press, Prentice-Hall, 1990.

[16] 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.

[17] de Champeaux, D., Verification of Some Parallel Algorithms, *Proceedings, 7th Annual Pacific Northwest Software Quality Conference*, Portland, OR, 1989.

[18] de Champeaux, D., D. Lea., & P. Faure, *Object-Oriented System Development*, Addison-Wesley, 1993.

[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] Lamport, L., ''Time, Clocks, and the Ordering of Events in Distributed Systems'', *Communications of the ACM*, 21(7), 1978.

[37] Lamport, L., *The Temporal Logic of Actions* SRC Research Report 79, Digital Equipment Corp, 1991.

[38] Lea, D., & J. Marlowe, *PSL: Protocols and Pragmatics for Open Systems*, Technical Report, Sun Microsystems Laboratories, 1994.

[39] Liskov, B., & J. Guttag, *Abstraction and Specification in Program Development*, MIT Press, 1986.

[40] 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.

[41] Manna, Z., & A. Pneulli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.

[42] 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.

[43] McCarthy, J. *Elephant 2000: A Programming Language Based on Speech Acts*, Unpublished Manuscript, Stanford University, 1994.

[44] 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.

[45] Milner, R., *Communication and Concurrency*, Prentice Hall International, 1989.

[46] Mullender, S. (ed.) *Distributed Systems*, 2nd ed., Addison-Wesley, 1993.

[47] Nierstrasz, O. ''Regular Types for Active Objects'', *Proceedings, OOPSLA '93*, ACM, 1993.

[48] Newmeyer, F. *Linguistics: The Cambridge Survey*, Cambridge University Press, 1988.

[49] OMG, *Common Object Request Broker Architecture and Specification*, Document 91.12.1, Object Management Group, 1991.

[50] OMG, *Response to the Object Management Group Object Services Task Force Request for Information*, Document 91.11.6. Object Management Group, 1992.

[51] OMG, *Common Object Services Specification*, Document 94.1.1, Object Management Group, 1994.

[52] OMG, *IDL C++ Language Mapping Specification*, Document 94.8.2, Object Management Group, 1994.

[53] Powell, M., *Objects, References, Identifiers and Equality*, Document 93.7.5, Object Management Group, 1993.

[54] Pratt, V.R., ''Modeling Concurrency with Partial Orders'', *International Journal of Parallel Programming*, 15 (1), 1986.

[55] Pratt, V.R., *Chu Spaces: Complementarity and Uncertainty in Rational Mechanics*. Technical Report, Stanford University, 1994.

[56] Raj, R., E. Tempero, H. Levy, A. Black, N. Hutchinson, & E. Jul, ''Emerald: A General Purpose Programming Language'', *Software---Practice and Experience*, 1991.

[57] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[58] 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.

[59] 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.

[60] Sproull, R., ''Guide to the Trace Modeling Tools''. *Technical Memo, Sun Microsystems Laboratories*, 1993.

[61] Strom, R., D. Bacon, A. Goldberg, A. Lowry, D. Yellin, & S. Yemeni, *Hermes: A Language for Distributed Computing*, Prentice Hall, 1991.

[62] von Benthem, J. *The Logic of Time*, Kluwer, 1991.

[63] Wegner, P., ''Tradeoffs between Reasoning and Modelling'', in G. Agha, P. Wegner, & A. Yonezawa (eds.) *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.

[64] Wieringa, R., & W. de Jonge, ''The Identification of Objects and Roles'', Technical Report TR-267, Faculty of Mathematics and Computer Science, Vrije Universiteit, 1993.

[65] Wikstrom, A., *Functional Programming Using Standard ML*, Prentice Hall International, 1987.

[66] 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.

[67] Yellin, D., & R. Strom. ''Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors'', *Proceedings, OOPSLA '94*, ACM, 1994.