PSL: Protocols and Pragmatics for Open Systems

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

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

Abstract

PSL is a framework for describing dynamic and architectural properties of component-centered 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 the *roles* of components, role attributes, and the issuance and reception of events. A specialized form, PSL/IDL supports design methods and engineering tools for describing protocols in CORBA systems.

1 Introduction

This report describes PSL, a framework for specifying dynamic and architectural properties of open systems. The remainder of this section provides background on the interface-based specification of open systems. Section 2 discusses the nature of protocol specification in open systems. Section 3 describes PSL concepts and notation, illustrated in more detail in Section 4. Section 5 surveys prospects for associated methods and tools. A discussion of related work appears in Section 6. The Appendix contains a few technical details omitted from the main presentation.

1.1 Open Systems

Open systems have been defined [29, 1, 67] 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 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.

1.2 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 [9]) on a family of components, not necessarily a complete or closed (nonextensible) description of any given component. Any implementation(s) that provide 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 [53]. One interface type may describe only certain aspects features that are listed more completely in various subinterfaces. 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 [52] 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.

1.3 Roles

While not always phrased as such, interface-based specification intrinsically relies upon the distinction between *roles* [68, 60, 6] and the components 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*). A role may also be thought of as a typed, abstract *access channel* [65, 4] providing a *view* [63] of one or more interaction participants [34], where each view is described by an interface that lists a set of supported message types. This concept of a role is nearly synonymous with that of a *subject* in the essential sense of Harrison & Ossher [26]. 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.

Interface specifications describe roles without commitment to the computational entity or entities that may implement them. 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.

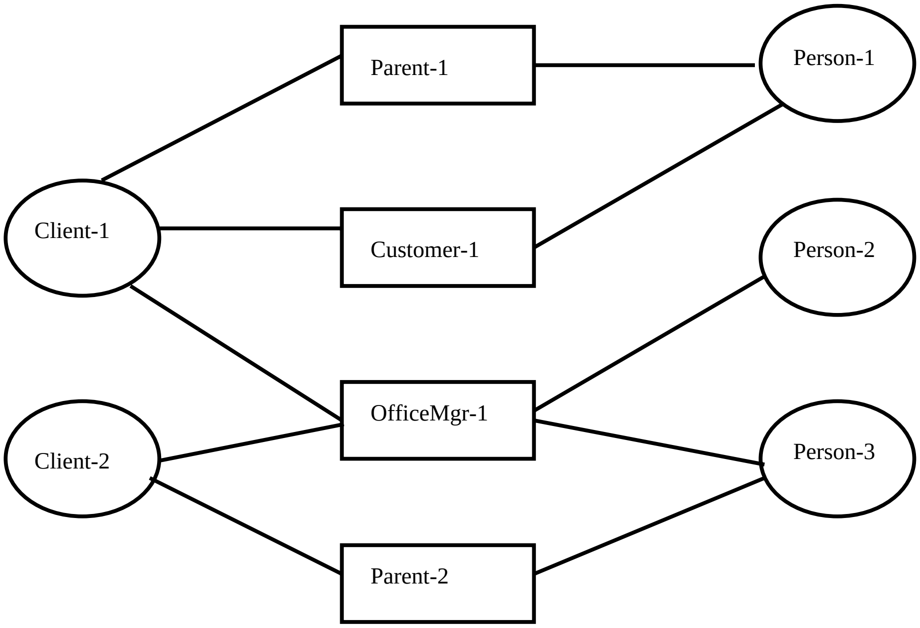
While roles are not the same as implementation objects, they bear several definitional similarities in terms of identity, state, and behavior [11, 19]:

- 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" [54]) 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.

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.
- One role may be implemented by several otherwise unrelated objects working together. Each may implement a subset of operations, or redundantly implement all of them, etc.

For example, an `OfficeManager` role might be implemented using a `Person` object, or a `Robot` object, or a set of job-sharing `Person`s, 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 interface instances, each seen from a different client, may access the same `Person` implementation:



Any of a number of programming constructs and infrastructure schemes can be used to support these kinds of mappings. For example, most CORBA systems reify some aspects of roles as "interface objects" that support some of the possible mappings by relaying messages to implementation objects. Other infrastructures (e.g., [10]) connect role-based messages via channels to multiple objects that together implement the role. In non-distributed frameworks, interfaces and implementations are (often incompletely) separated through programming language mechanisms and conventions. For example, Modula-3 and Ada9X contain interface types useful for open system development, and C++ and SmallTalk programs often use "abstract classes" and related constructs to distinguish interfaces from implementations. Finally, it is very possible to apply these concepts in systems in which roles are never explicit at the implementation level, and so are used only as abstract design constructs ([60]).

2 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. 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.

These kinds of dynamic and architectural 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 [22, 21, 28].

PSL is a framework for addressing such issues by extending the interface-based description of open systems to capture the questions of *who*, *what* and *when* across interactions, as well as the structural relations underlying these dynamics. The principal abstractions include:

Interface:	A view of a family of components in terms of supported operations.
Role:	An instance of an interface.
Handle:	An identifier of a role instance.
Class:	A description of a family of concrete components.
Object:	An instance of a class.
Situation:	An expression describing attribute values of one or more roles.
Realization:	An instance of a situation.
Value Type:	A type of any attribute, message argument, protocol parameter, etc.
Operation:	A procedure defined via a signature indicating request and reply arguments.
Message:	A communication carrying an operation request, reply, or exception.
Event:	An occurrence of issuing or receiving an individual message.
Attribute:	An abstract function with values that may vary over time.
Event Predicate:	An attribute denoting the occurrence of a single event.
Protocol Rule:	A temporal constraint relating the realizations of two or more situations.
Constraint Rule:	An expression constraining values of attributes in all possible realizations.

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. PSL does not assume any specific computational model or mechanics relating roles and implementation objects. Descriptions of how each instance of a role is mapped to one or more implementation objects (and vice versa) lie outside of PSL.

PSL captures the essential *incompleteness* of open systems by specifying 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 further decomposition; thus that no unmentioned events may exist.

Thus, anything that is not ruled out is assumed to be possible. However, stronger specifications may be introduced via *refinement.* Protocol refinement is the act of introducing new rules within a narrower context than more general rules, but without invalidating these more general rules. The opposite of refinement is generalization, in which a weaker set of rules is introduced in a broader context.

Additive refinement may take either of two forms. *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.

PSL is designed to support methods and tools for developing components in open systems; formal specification analysis is only a secondary goal. It is in principle possible to build or adapt inference tools (e.g., [43]) that check collections of PSL rules for consistency, and to answer questions about their properties. For example, one could ask which exceptions might be expected in a system from a given starting point. However the nature of open systems and their specifications preclude certain categories of analytic questions and answers. Specification systems (see Section 6) based on "closed world" assumptions (i.e., that the system implements only those features specified) are uniformly more powerful than PSL in addressing questions about liveness, deadlock, interference, and aliasing [49, 31]. Such questions often have no definitive answers in open systems. While a selected set of PSL rules can be analyzed in "closed system mode", under the assumption of complete knowledge, there is no use in doing so unless one has independent assurance that the relevant portion of the system is fixed and communication closed with respect to the rest of the system for all time (i.e., produces no outputs except replies to procedural requests). While closed-world models may have a place in open system development, they are perhaps best 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 [37]) to rely on fixed known implementation properties when constructing individual components.

3 PSL

As a basis for specification, PSL relies on any framework for declaring interface types that minimally include operation signatures in their declarations, and 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 role instances of the associated interfaces.

To extend interface-based specification 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 *ProtocolCharts*, a semigraphical form of PSL/IDL corresponding in simple ways to the textual representation.

PSL/IDL role types are synonymous with CORBA IDL interfaces. All PSL/IDL constructs appear as annotations 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 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 (otherwise equivalent to &&). 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 [69]. PSL/IDL does not define types for sets, multisets, or maps. Sequence types may be used to equivalent effect.

3.1 Handles

For purposes of specification, each distinct instance of a role is ascribed a unique *handle*. Handles are an abstraction of names [48], references [52], ports [65], static labels, etc., used to identify role instances and message destinations. Handles are *not* references to *objects*, nor are they 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.

PSL relies on the existence of one or more handle types, and handle values that may be used in any value context. There may be several distinct handle types in a system. 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 equality relation "`==`" among handle values denotes only value equality among handles (also known as "shallow" equality), not necessarily identity or equality of implementation-level components.

PSL/IDL handle values are expressed using the syntax of IDL `objref`s. These serve as both references to instances of IDL `interface`s (*not* necessarily their implementations) and destinations of CORBA messages. However, as discussed in Section 3.4 and the Appendix, PSL handle values need not bear a one-to-one relation with `objref` values in CORBA implementations.

3.2 Attributes

An attribute (also known as a *state function*[39, 34]) 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; 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 5) 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. For example:

```
interface File { /* ... */ };

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

PSL/IDL uses function syntax rather than IDL `attribute` syntax. In IDL, the keyword `attribute` is only a stylistic device for declaring accessor and mutator *operations* within interfaces. There is no PSL/IDL-G notation for declaring attributes, although OMT [61] or related object-oriented graphical notations can be used (see Section 4.4).

Among the most common forms of attributes used in PSL are "relational" or "connection" attributes that represent the values of handles that a participant 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.

3.3 Situations

The concept of a situation extends the common notion of abstract object state [44, 7]. 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 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, a situation might contain only the constraint that a given kind of `File` is open. The realizations of this situation in a particular system include all "snapshots" of the 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 5).

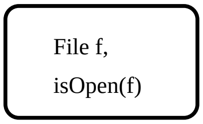
A realization is thus a partial description of an arbitrarily brief instance, an "actual world", in which the predicate describing a situation holds. Realizations represent individual observations, mappings, or inferences about concrete system behavior, characterized by expressions describing attribute 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, $r \propto S$ for realization r and situation S, holds if the situation predicate for S can be made true using the values in r (see Appendix). Two or more realizations may all match the same situation but in different ways because they differently instantiate free parameters listed in the situation.

A situation is represented as a parameterized predicate. We provide a full notation in the Appendix, 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 references to named arguments. They may also include in-line local declarations (which have the expression value of *true*). For example, assuming the definition of an `isOpen` attribute the following situation describes the state of a `File` being open:

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

PSL/IDL-G notation is the same except that situation expressions are demarcated by solid rounded boxes. They may be embedded within a `rules` declaration drawn with a dashed box with name and parameters in the corner. However, in PSL/IDL-G, it is usually more convenient to use "in-line" declarations in situation boxes rather than explicit `rules` parameterization:



3.4 Messages

Messages carry communications among participants, including operation requests, operation replies, asynchronous messages, exceptions, and so on. For each kind of message M possible in a system, PSL assumes 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.

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

PSL/IDL message types are abstractions of `CORBA::Request`, 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*. Bindings for arguments follow normal IDL rules.

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 [56, 27]. For example, channel-style `objref`s may be used. Channel values [65] 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. Other syntactic details and mappings to base PSL constructs may be found in the Appendix.

3.5 Events

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 and seldom needed. When otherwise unconstrained by context, 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; similarly for `in(`*m*`) by(`*p*`)`. Situations and rules may thus be phrased in terms of events that have occurred at any time (cf., [45]).

PSL/IDL event predicates use PSL/IDL messages as arguments to `in` and `out`. For example, assuming declarations of `aFile`, `c`, and `exc` within the scope of a given `rules` declaration or in referenced IDL `interface` declarations:

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

3.6 Protocols

PSL protocol rules describe conditions under which realizations of situations occur. Rules are collections of situations linked by temporal operators. These 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 a. (See Section 3.10 for a more formal characterization.) The means by which this relation is observed or arranged lies outside of the scope of PSL proper (see Section 5).

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

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

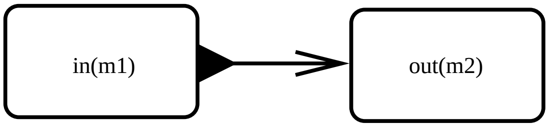
*A*s pair with *B*s: $A \blacklozenge B \overset{\text{def}}{=} (A \blacktriangleright B) \wedge (A \blacksquare B)$

In PSL/IDL-G, rules are expressed via arrows connecting situations through corresponding symbols drawn on the outside of the predecessor 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.

Leading. The "forward reasoning" operator $A \blacktriangleright B$ is akin to that of 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 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:

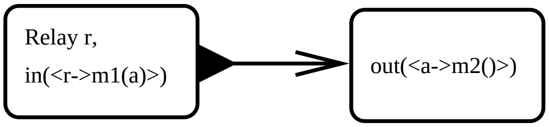
```
{in(m1)} lead {out(m2)}
```



In the scope of a particular PSL/IDL protocol, we would have to be more explicit about types and messages. PSL/IDL 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). All `rules` parameters implicitly serve as universal quantifiers for the enclosed expressions, and other inline declarations are implicitly existentially quantified. 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 (see Appendix).

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)>)} lead {out(<a->m2()>)}
};
```



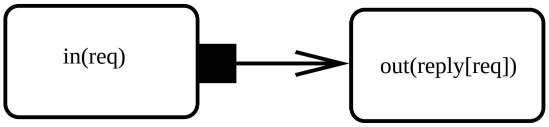
This rule does not in any way imply that `Relay` operations must be single-threaded. Because there are no constraints that indicate otherwise, two or more different "threads" of this rule may be active concurrently, each triggered by a realization corresponding to a different instance of an `m1` message. 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). For example, the first relay rule is equivalent to one explicitly mentioning `in(m1)` in the successor situation:

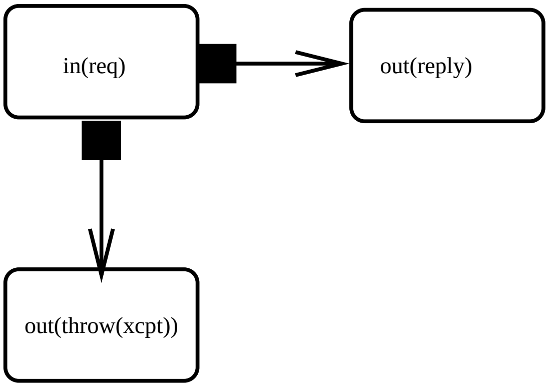
```
{in(m1)} lead {in(m1), out(m2)}
```

Enabling. The "backward reasoning" operator *A* ■ *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, saying 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:

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

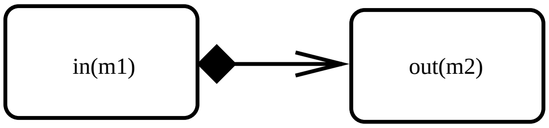


Separate rules may link multiple right-hand-sides to the same left-hand-side to describe multiple possible effects. For example, we could add another rule stating that `req` may lead to an exception. This may be pictured together with the first rule, reflecting the fact that rules are combined conjunctively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



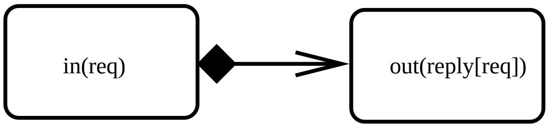
Pairing. The *A* ◆ *B* operator is used for if-and-only-if relations, in which both every *A* is followed by a *B*, and every *B* is preceded by an *A*; thus *A*s and *B*s occur only in pairs. For example, another desirable global rule is the *one-to-one delivery rule*, for any `m`, saying that all and only those messages that have been issued are ultimately received.

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



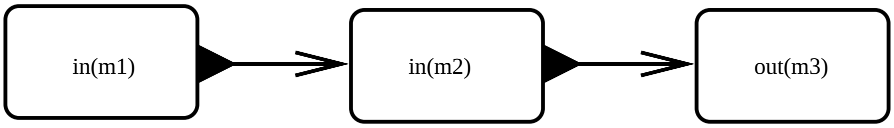
The ◆ relation may be used to provide guarantees about procedural operations. For example, we could strengthen the above ■ form to assert that a reply to `req` must be issued, thus precluding the possibility of exceptions or other non-procedural behavior:

```
{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* ▶ *B* ▶ *C*. This indicates *(A* ▶ *B)* ∧ *(B* ▶ *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)} 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 could add the rule:

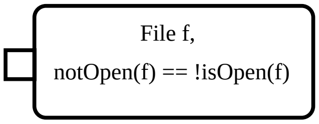
```
{in(m1)} enable {in(m2)}
```

3.7 Constraints

Some conditions must *always* hold, across all possible contexts. Constraint rules limit the set of possible system states to those in which the 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 [20, 66], operator □. The expression □ P indicates that P invariantly holds for all time, over all possible worlds. All realizations must obey □ constraints in addition to those explicitly listed in situations. In PSL/IDL `rules` declarations, brace-demarcated expressions describing necessary restrictions among attribute values are prefaced by `inv`. In PSL/IDL-G, they are listed with an unfilled box on their sides. For example:

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



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 implication constraints:

```
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 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. However, while convenient, and sometimes unavoidable, the use of unconstrained attributes is also notoriously troublesome in practice [44, 12, 19], 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. As illustrated in Section 4.5, any required variation or invariance in the values of unconstrained attributes across time must be explicitly tracked in individual protocol rules.

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 request `req`, but at most one of these, we could list:

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

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

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:

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

3.8 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 [19, 63]. 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*`)`.

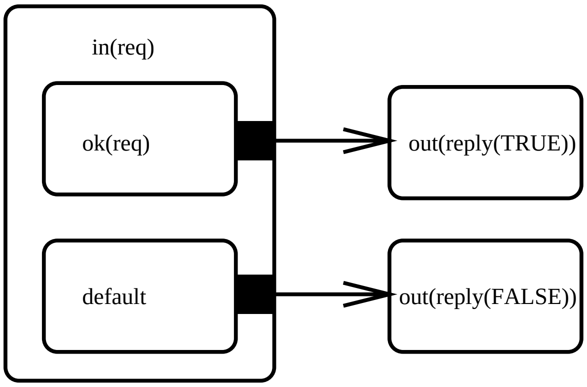
PSL does not use any special notation to *declare* that one situation is a subsituation of another except in the special case of disjoint union where $S = S_1 + S_2 + \ldots + S_n$. This represents a set of subsituations S_i that are constrained to completely partition a situation-space S. Partitioned subsituations must be exhaustive and mutually exclusive. They cannot simultaneously hold. Partitioning is thus one way to express the notion that one situation "disables" another [70].

In PSL/IDL, partitions are enclosed in cases of the form `{ case` S_1 `case` S_2 …`case {default} }`. 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 `case`s. The special trailing situation `default` matches only if all previous cases fail, thus acting as a generalized `else`. Properties common to all partitions may be expressed by linking a situation with the common expressions to all partitions via `&&` to the `case`s.

In PSL/IDL-G, partitioning is indicated via nested stateCharts [25]. 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.

Conditional protocol rules can be expressed using partitions. Ordering operators "distribute" through situation partitionings to describe conditional paths. For example, to indicate that a `TRUE` reply is enabled if some function `ok` holds, and conversely for `FALSE`:

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



3.9 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 constraints maintain consistency among these related rules and contexts.

In PSL, refinements are normally localized to new entities described by new interfaces bearing subinterface relations to the originals. PSL protocol declarations are parameterized with reference to interface types. Specialized protocol declarations may be attached to (parameterized by) instances of subinterfaces. 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.

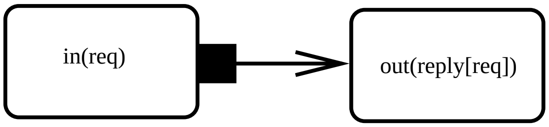
In PSL/IDL, when two kinds of roles differ in protocol but not operation signatures (for example, when protocols differ across “local” versus “remote” versions of components), this may require construction of “dummy” IDL interface types just to give the two types different names (cf., [5]).

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:

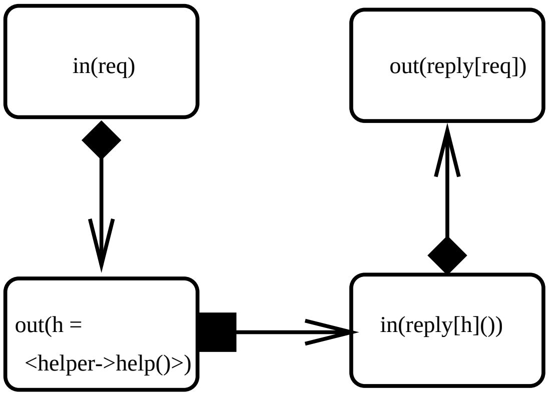
$A \blacktriangleright \{\ \texttt{TRUE}\ \}$	$A \blacksquare \{\ \texttt{FALSE}\ \}$
$\{\ \texttt{FALSE}\ \} \blacktriangleright A$	$\{\ \texttt{TRUE}\ \} \blacksquare A$
$A \blacktriangleright B, B \blacktriangleright C \vdash A \blacktriangleright C$	$A \blacksquare B, B \blacksquare C \vdash A \blacksquare C$
$A \blacktriangleright B, B \blacklozenge C \vdash A \blacktriangleright C$	$A \blacksquare B, B \blacklozenge C \vdash A \blacksquare C$
$A \blacklozenge B, B \blacktriangleright C \vdash A \blacktriangleright C$	$A \blacklozenge B, B \blacksquare C \vdash A \blacksquare C$
$A \blacktriangleright B \cap C \vdash A \blacktriangleright B$	$A \blacksquare B \cup C \vdash A \blacksquare B$
$B \cup C \blacktriangleright A \vdash B \blacktriangleright A$	$B \cap C \blacksquare A \vdash B \blacksquare A$
$B \blacktriangleright A, C \subseteq B \vdash C \blacktriangleright A$	$C \blacksquare A, C \subseteq B \vdash B \blacksquare A$
$A \blacktriangleright C, C \subseteq B \vdash A \blacktriangleright B$	$A \blacksquare B, C \subseteq B \vdash A \blacksquare C$

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 specified end-points. For example, a coarse-grained specification 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:



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 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 say:

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

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

```
{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 $\blacksquare$ 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 the $\blacklozenge$ operator instead of $\blacksquare$ 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 $\blacksquare$. These are situational analogs of type conformance [59], 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 *($B' \subseteq B$)*, thus logically subsuming 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.

3.10 Foundations

PSL constructs are based on structures describing possible worlds, as found in model theory and temporal and modal logic [32, 20, 66]. These serve as the basis for several frameworks for specifying possibly distributed systems [39, 42, 15, 35, 6], as well as related applied temporal reasoning systems in AI and object-oriented logic programming [44, 3, 17].

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 $\Box$ 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.

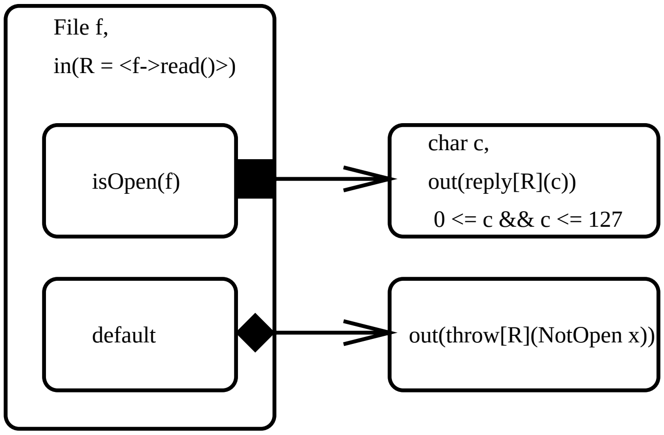
The relation $\mathbf{R}_\preceq$ serves as the basis for PSL ordering operators. This relation is simplest to describe formally when expressions are restricted to event predicates on fixed messages [24, 70, 58]. In this case, expressions in **V** are just characteristic predicates of sets of event occurrences, 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.

4 Examples

4.1 Roles

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(R = < f->read() >)} && {
   case {isOpen(f)} enable {char c, out(reply[R](c)), 0 <= c, c <= 127}
   case {default}   pair   {out(throw[R](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`. If for some reason we had 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.

4.2 Interactions

One-sided protocol descriptions can be useful even when interactions are focused upon fixed sets of communications partners. For example, consider a transaction framework in which `Coordinator`s help arrange the actions of `Transactor`s. The IDL interfaces are:

```
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 `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. As indicated by the use of interface inheritance, members may be other `Coordinator`s. 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.

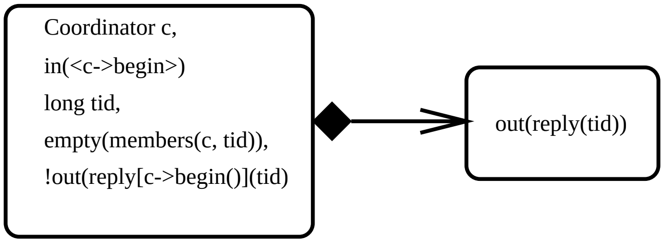
To capture some of this in PSL/IDL, we declare a `protocol module`, starting off with a declaration of attribute `members` to represent the handles of all members of a given transaction. There may be several sets of members maintained by each `Coordinator`, each referenced via its transaction identifier (`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`:

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

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

We next 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:

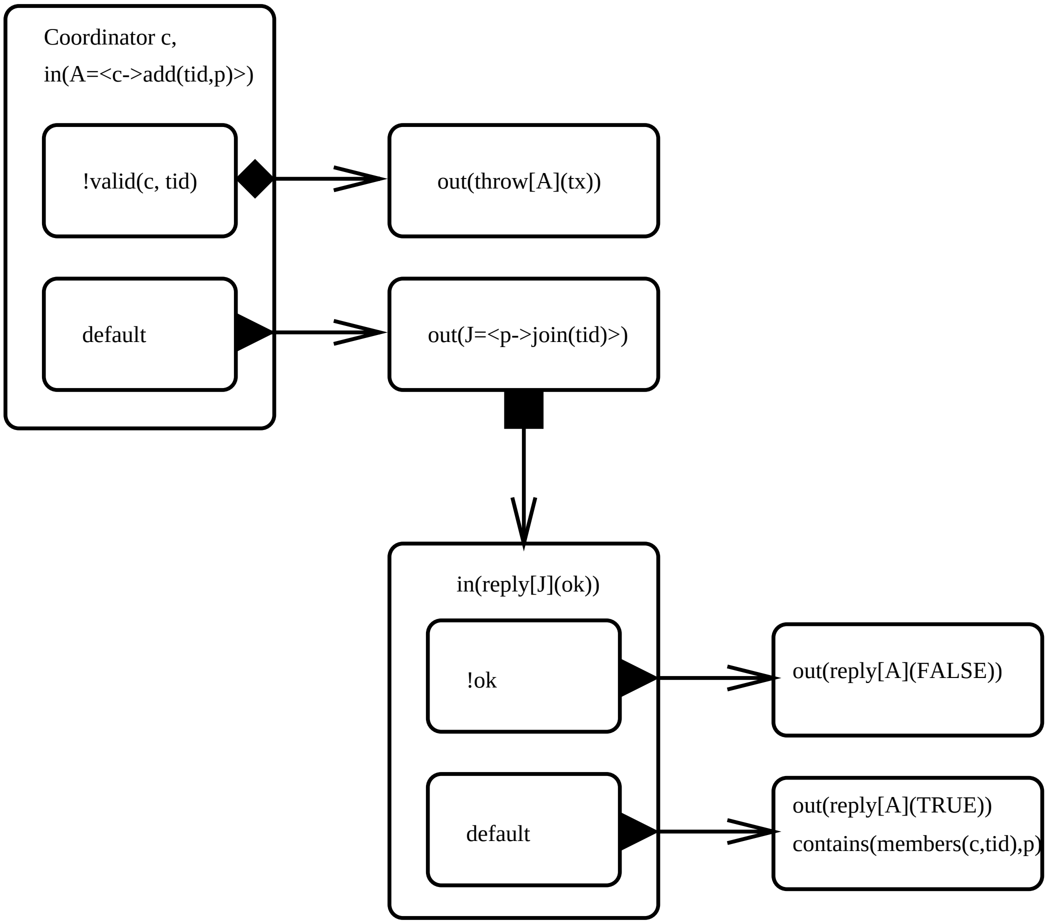
```
boolean validtid(Coordinator c,TID tid);

rules valid(Coordinator c, TID tid) {
 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 describe some of the dynamics. 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:



```
rules coord(Coordinator c) {
 {in(<c->begin()>), long tid,  empty(members(c, tid)),
  !out(reply[<c->begin()>](tid))}
 pair  {out(reply(tid))}
}
```

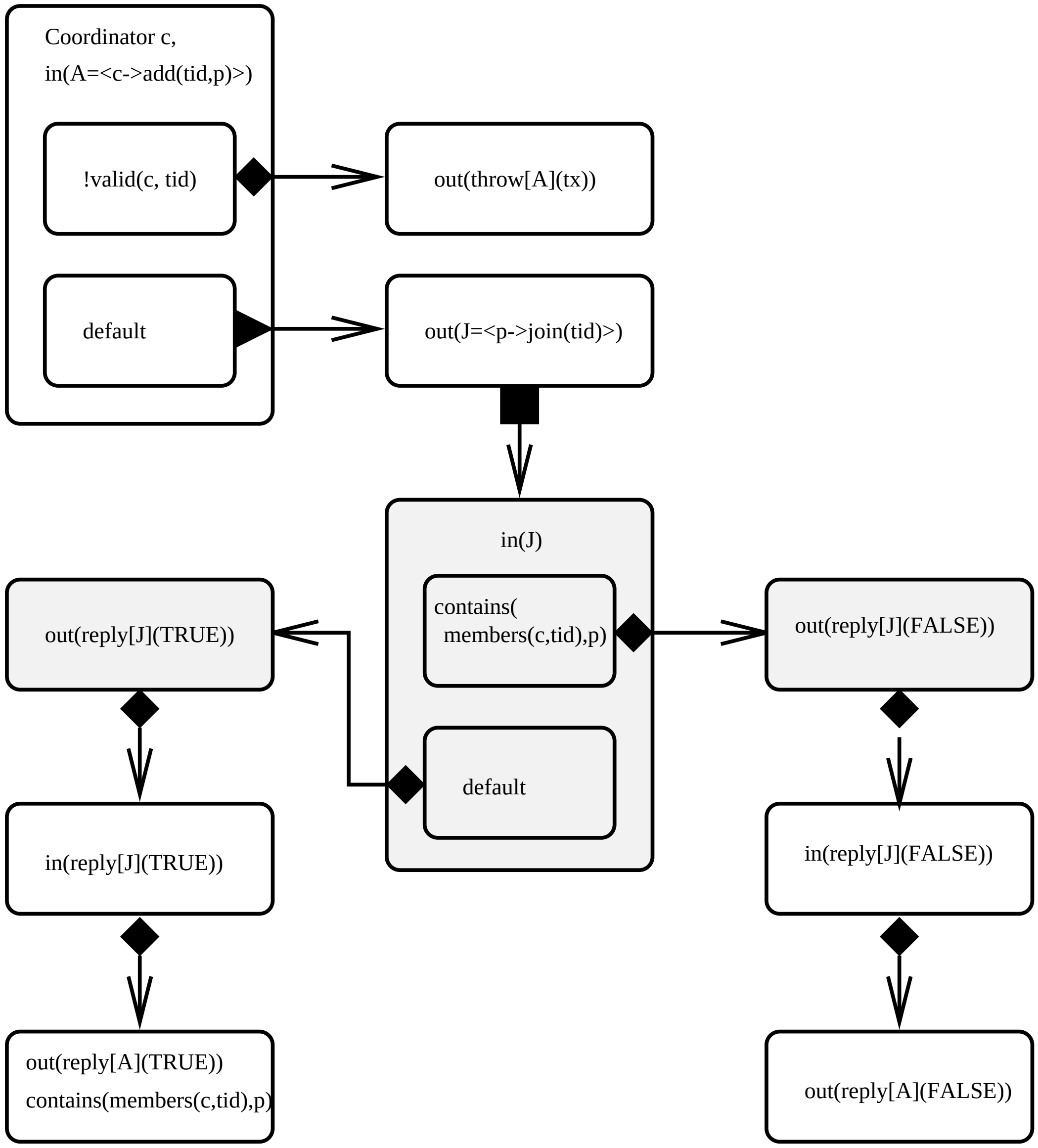
The rule for add contains a main "thread" saying that upon receiving an add request for a Transactor p with a valid tid, a coordinator invokes p's join operation. If it then receives a TRUE reply, p is then a member of members and the operation completes successfully. The other cases are "error paths"; 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 p were not a live handle, the use of timeouts, and so on.)



```
 {in(A = <c->add(tid, p)>)} && {
    case {!validtid(c,tid)} pair  {throw(tx)}
    case {default} lead {out(J = <p->join(tid)> )} enable
           {in(reply[J](ok))} && {
            case {!ok}     lead {out(reply[A](FALSE))}
            case {default} lead {out(reply[A](TRUE)),
                                 contains(members(c,tid), p)}}}
}
```

4.3 Multiple Participants

Protocols including situations describing multiple participants are used when the perspectives and responsibilities of all parties need to be taken into account simultaneously, often because one 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 [19]. For example, we may rework a more committal version of the `add` rule:



For clarity, situations describing the view of the `Transactor` are shaded. This rule assumes for simplicity that `p` replies `FALSE` to `join` only if it is already a member of the transaction. 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.

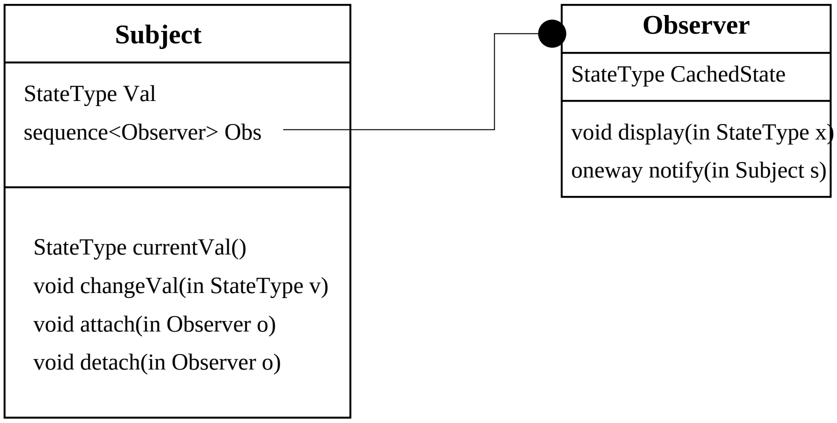
Rules in such PSL declarations often represent "snippets" [64] of a longer or fuller protocol. It is normally possible to show a more complete view of a protocol by linking situations described in one scope to those in others. This may entail exploitation of global axioms such as the *one-to-one delivery rule* when available to match `in`s with `out`s, along with special rules for matching the obligations and expectations of particular partners [18]. For example, the multi-participant view of `add` may be constructed given the single forms of `Coordinator ::add` and `Transactor::join`. The resulting *timethread* [13] is a path linking initial situations to terminal ones. Such timethreads can be valuable tools for informally validating protocols.

4.4 Design Patterns

The elements of many common object-oriented design patterns [21] consist of protocols common to all participants in a framework. For example, in the *Observer* pattern, `Subjects` maintain representations of things they are modeling, along with operations to reveal and change this represented state. `Observers` display (or otherwise use) the state represented by `Subjects`. When a `Subject`'s state is changed, it merely informs one or more `Observers` that it has changed. `Observers` are responsible for determining the nature of the changes, and whether they require re-display. The version illustrated here is one of the variants described in [21] in which the `Subject` sends a handle to itself as an argument to `notify` when it is changed. The `Observer` uses this handle to probe for values, and performs a redisplay only if the state actually differs from that held in a cache.

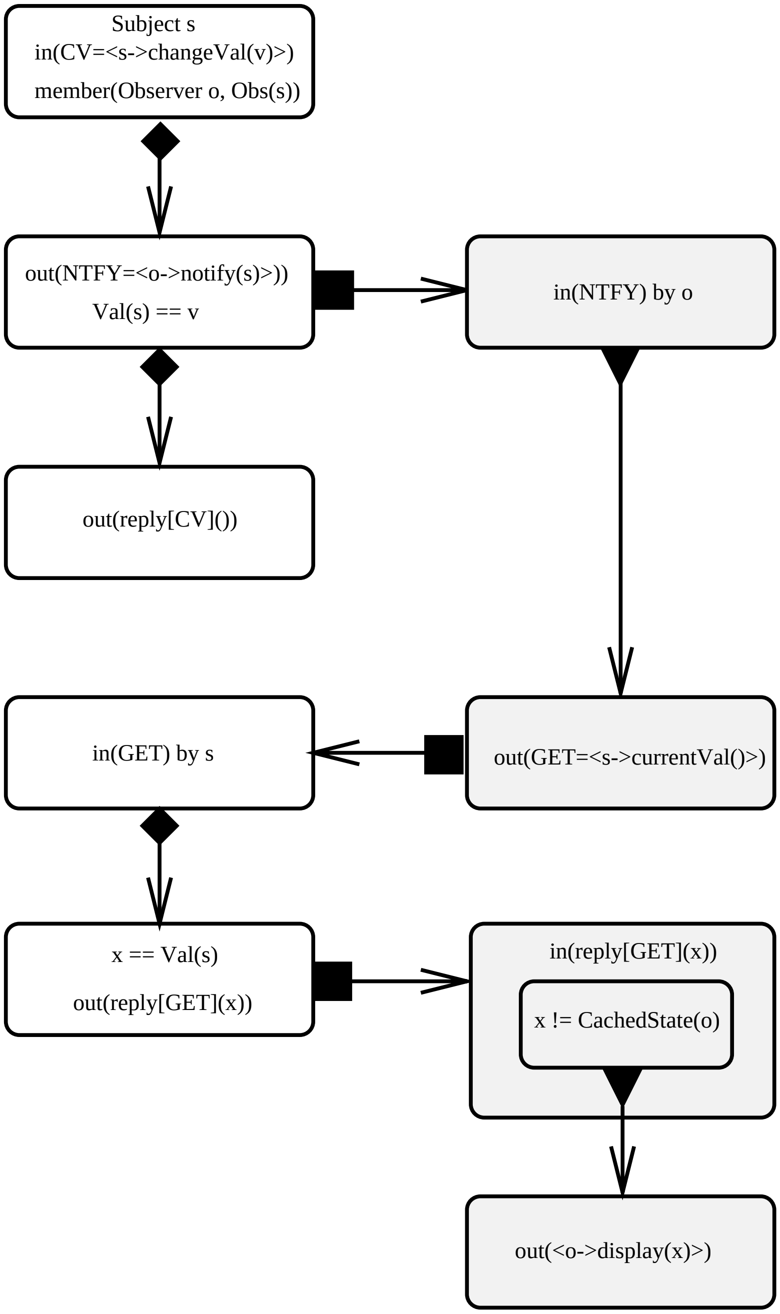
The static structure may be described in PSL/IDL, as well as in an IDL dialect of the OMT [61] notation often used for design patterns in which attributes are listed along with operations:

```
interface Subject {
  StateType   currentVal();
  void        changeVal(in StateType v);
  void        attach(in Observer ob);
  void        detach(in Observer ob);
}


interface Observer {
  void        display(in StateType v);
  oneway void notify(in Subject s);
};


protocol module {
  StateType          Val(Subject);
  sequence<Observer> Obs(Subject);
  StateType          CachedState(Observer);
};
```


Subject
StateType Val
sequence<Observer> Obs
StateType currentVal()
void changeVal(in StateType v)
void attach(in Observer o)
void detach(in Observer o)
Observer
StateType CachedState
void display(in StateType x)
oneway notify(in Subject s)

Informally, the main notification protocol is that when a `Subject s` receives a `changeVal` request, it makes sure that its abstract state `Val(s)` has been set, and for each member `o` in its set of `Observers`, sends a `notify` message to the observer, and then replies to whoever called `changeVal`. When the `Observer o` receives the notification, it probes `s` by asking for its `currentVal`. If the returned value differs from the value recorded as `CachedState`, the `Observer` sends itself a `display` request. The details of what happens when an Observer receives that `display` request aren't important to this aspect of the protocol.

In the corresponding PSL/IDL-G specification, situations primarily viewing an `Observer`'s role are shaded. Note that the declaration and use of `Observer o` in the first situation means that the `Observer` roles of the protocol are taken (perhaps concurrently) by all observers of `Subject s`:


Subject s
in(CV=<s->changeVal(v)>)
member(Observer o, Obs(s))
out(NTFY=<o->notify(s)>))
Val(s) == v
in(NTFY) by o
out(reply[CV]())
in(GET) by s
out(GET=<s->currentVal()>)
x == Val(s)
out(reply[GET](x))
in(reply[GET](x))
x != CachedState(o)
out(<o->display(x)>)

4.5 Constraints

Recall the equivalence of our first relay rules in Section 3.6:

```
{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)}
```

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 `Account`s 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 (AccountFactory f) {
  {in(<f->makeAcc(sn, initbal)>)}
  pair
  {out(reply(a)), live(a), bal(a) == initbal, serialNo(a) == sn}
 };

 rules (Account a) {
  {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))}
 };

};
```

Access. 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) {
 inv {(bal(a) == b) -->
         out(reply[<f->makeAcc(s,b)>](a)) ||
         out(reply[<a->setBalance(b)>]())}
};
```

Initialization. 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)  {
 inv {(serialNo(a) == s) --> out(reply[<f->makeAcc(s,b)>](a))}
 inv {unique(out(reply[<f->makeAcc(s,b)>](a))}
};
```

Single-threading. 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) {
 inv {unique(in(s = <a->setBalance(b)>), !out(reply[s]()))}
};
```

Timing. 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*. (The physical/temporal implementation properties of timers of course remain outside the scope of PSL.)

One way to impose such constraints is via "client-side" protocol rules. For example, assuming the existence of a `Timer` with attribute `ticks`, we could state that any `AccountUser` client issuing a `getBalance` receives a reply within `N` time units:

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

Interactions with timers may also be specified. For example, an `AccountUser` may invoke `getBalance` in conjunction with a time-out `alarm` set for `N` time units of a `Timer`. If the client receives the reply before the time-out, it somehow uses the value. One part of such a specification is:

```
rules (AccountUser client, Account a, Timer tm) {
 {out(to=<tm->alarm(N)>)by(client), out(gb=<a->getBalance()>)by(client)}
 enable
 {case {in(reply[gb](b)), use(client,b)} enable {in(reply[to]())}
  case {in(reply[to]())}                 enable {in(reply[gb](b))}
 }
};
```

These kinds of idiomatic constructions are obvious candidates for simpler expression and support in PSL.

5 Methods and Tools

PSL may be used to support a number of development practices and engineering tools:

Monitoring. Recordings of freely executing “live” components may check whether and how often a given set of actual components obey a particular protocol over some period of interest. Simple monitoring tools merely record event histories for later checks that they conform to stated protocols. More sophisticated monitoring agents perform conformance checks in real-time during the lifetime of the system. Monitors may also be attached to protocol visualization tools that allow developers to “see” the realizations of a protocol as they unfold within implementations. Interactive monitoring may be extended to create debuggers allowing users to construct new entities, issue new events, and/or alter existing ones in a live system.

Testing. Recordings of components from a fixed initial configuration check whether protocols are obeyed and/or particular terminal situations are reached. Protocol rules may also guide construction of a test suite that covers all listed paths and situations defined in a given protocol. Additionally, they may be used to dynamically evaluate existing or randomized tests, checking that all paths have been covered.

Verification. Analytic, symbolic checks (at various levels of formality) determine whether implementations possess required properties. Given a set of specifications and an implementation, verification efforts analytically decide whether the implementation conforms to the specification.

Design for testability. Specification-guided programming (also known as “reification” [19]) is the process of “deriving” concrete implementations from their specifications. Reification typically involves substantial refinement, narrowing down interfaces and protocols to reflect various design constraints. In addition to required operations, implementations may also compute or approximate attribute functions and expressions on attributes found in specifications. This provides a simpler basis for testing, enhances the likelihood that implementations of other operations are correct “by construction”, and enables “reflective” programming practices in which components inspect their own logical state. However, even the most detailed open specifications do not completely determine their implementations. For example, even when a PSL specification boils down to mandating that a given component rely on another with the properties of a simple `int` variable, this might not preclude an implementation using a special kind of C++ object that behaves as an `int` but also periodically logs its value to persistent media.

Design recovery. Recovery is the reverse of reification: retrospectively “deriving” specifications by abstracting over implementation choices. Design recovery may also involve translation from or to less formal design documentation, as well as validation efforts, comparing specifications against high-level requirements.

Simulation and prototyping. Semi-automated partial implementation, using a set of default strategies for minimally implementing those features of roles, events, and attributes of interest may be used to investigate system behavior.

5.1 Mappings

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. All of the above 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 event predicates and concrete communication occurrences.
3. Mappings between expressions defining situations and realizations observed or inferred in concrete code and/or its execution.

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

The need for, nature, and use of mappings differs across these kinds of applications. At one extreme lie formal verification efforts. Analytic verification requires a rigorous mapping to the underlying target programming languages, systems, and tools in order to construct an operational semantics [30]. Such a mapping is often impossible in practice, in large part because properties of implementation languages, libraries, and tools are not known in sufficient detail. Moreover, even when implementations are produced in well-behaved languages, full systems-level verification is often an unrealistic, intractable goal.

Simple visualization tools lie at the other extreme. A useful visualization tool might require only a mapping from events to observable messages, ignoring all other aspects of PSL expressions, at the price of not being able to distinguish two situations containing the same event predicates but different attribute predicates.

5.2 Mapping PSL/IDL

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++ [55]).
- 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 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 [71]. CORBA allows "listener" objects to be interposed along any communications path by transparently altering the mappings between roles and implementation objects [56]. However, even if attention is restricted to events, 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.

Any "listener" is only an approximation of an unobtainably ideal observer, and will have limitations in its ability to discriminate the occurrence of certain protocols. For example, it may be impossible in practice to obtain sufficient temporal granularity and synchronization of observations to recognize instances of, say, $A \blacktriangleright B$ when event predicates in A and B are mapped to occurrences on different sides of the planet. (An intrusive alternative to detection is enforcement, using tools [10] that guarantee that certain messages obey certain orderings.)

Similar empirical challenges are encountered in relating the observation of a "tapped" message to a reception predicate for a particular role, detecting the creation of a new instance and its corresponding handle, as well as detecting different realizations of the same situation in multithreaded implementations.

Mappings of attributes remain the least automated aspect of tool development. Unless attribute functions are implemented by programmers, defined entirely via expressions on mapped observable event predicates, or are inferable via protocol rules, developers of tests and monitors must themselves implement all computable or approximable expressions of interest and implement them as program instrumentation. For example, the simplest possible testing tool would require implementations of boolean functions corresponding to each defined situation, along with history mechanisms recording event occurrences that are referenced in situation expressions. These functions may be invoked whenever a listener intercept notices an event that might make a situation of interest hold or stop holding. A more intelligent tool could infer the values of certain attributes without the need for instrumentation.

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.

6 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 [30] and specification systems based upon them [36], 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, 62] 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 [61, 34, 19] 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, 60, 68, 26]. Unlike some other approaches [2, 71] 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 [51] 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 [47] 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 [45, 50, 33, 14, 23] 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 [57]. 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 [38, 57, 10] 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 [70, 24, 58], as adapted for use in interface-based specification.

7 Appendix

Here we describe relations between PSL/IDL conventions and underlying PSL constructs. We first introduce an expanded notation for representing situations as named, parameterized predicates of the form

Name(Parameters){ *Predicate*}.

For example, a situation describing the state of a `File` being open is:

```
OpenFileSituation(File f){ isOpen(f) }
```

`OpenFileSituation` names the situation. The argument list declares all variables and types used in the defining predicate.

The standard PSL/IDL form is abbreviated in that:

- Situations are not named.
- Situations are defined only within the scope of `rules` declarations, and implicitly take the handles declared in their `rules` declaration as parameters.
- Situation predicates may include in-line local declarations to name values inside expressions. These are added as implicit parameter declarations.
- Message arguments are implicit, with values bound according to IDL conventions, as described below.

A realization may be represented as a tagged set of values. A realization *r matches* situation *S* (i.e., $r \propto S$) if there exists a consistent assignment of the values in *r* to the arguments in *S* in accord with all $\Box$ constraints such that *S*'s predicate is true. Value assignments must be consistent across all situations in the same scope. Realizations obey one or more rules if the rules hold under applicable assignments of values to `rules` parameters.

Variants. 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 rather than the implementation objects that they are normally defined under. Handles of the form *Interface*`::`*Operation* denote abstract activations (executing instances) of the indicated operations. For example, an instance of the form `aFile::read` is a handle to an abstract activation of a given `File aFile`'s read operation as defined in the `File` interface. (Among other alternatives, this handle might be mapped to an implementation-level pointer to a stack frame.)

In addition to the message forms described in Section 3, PSL/IDL procedural messages may include `from(`*src*`)` designation indicating the “return address” of a sender:

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

Also, `in` and `out` event predicates may be annotated with `by(`*Handle*`)` to indicate the participant issuing or receiving a message. For example,
`in(<aFile->write(c)>)by(aFile::write w)`
allows a shift to an activation-level description focused on a single `write` operation. Note that `from` could be different than `by` in systems supporting implicit forwarding of delegated messages.

Message types may also be made explicit. They may be given names using a `typedef` convention analogous to the inline declaration style. Required fields have the same names as their arguments; sources and destinations are called `_src` and `_dest` respectively. For example, to represent the type of `File::write` messages:

```
 typedef  <File::write(in char c)> WriteMsg;
```

Bindings. Argument bindings in messages follow normal IDL conventions, which are related to underlying PSL constructs as follows. Any situation containing an `in` or `out` predicate implicitly takes a message of the corresponding type as a parameter, and message fields are implicitly equated with named variables according to their use. When `by` is omitted, it is also added an parameter simply of type `Object`, the base of the IDL interface hierarchy. For example, consider PSL/IDL situations:

```
 {in( <aFile->write(c)> ), c == ’a’}

 {out( <aFile->write(’a’)> )}
```

These may be expanded as:

```
typedef <File::write(in char c)> WriteMsg;


Sin(File aFile, WriteMsg m, char c, Object rcvr) {
  m._dest == aFile,
  in(m) by(rcvr),
  c == ’a’, c == m.c
}

Sout(File aFile, WriteMsg m, Object sndr) {
  m._dest == aFile, m.c == ’a’,
  out(m) by(sndr)
}
```

IDL-level `attributes` are also treated as operation declarations in PSL, with message selector names formed by prepending `get_` and `set_` for the “read” and “write” forms of the operation respectively, in accord with common language mapping conventions [52, 55].

Scopes. Pairs of situations related by ordering rules must be normalizable under a common set of symbols, each assuming a unique value in realizations. All successor situations carry all parameters from their `rule`s and predecessors as well as all non-negated event predicates in predecessors. Consider, for example:

```
rules x(P p) {
 {in( <p->op1(a, b)> ), long i == attr(p)}
 lead
 {out( <a->op2(b, i)> )}
};
```

The situations may be tediously expanded as:

```
typedef <P::op1(in A a, in B b)>    op1Msg;
typedef <A::op2(in B b, in long i)> op2Msg;


S1(P p, op1Msg mop1, long i, A a, B b, Object s1rcvr) {
 in(mop1) by(s1rcvr),
 i == attr(p),
 a == mop1.a,
 b == mop1.b
}


S2(P p, op1Msg mop1, long i, A a, B b, Object s1rvcr,
   op2Msg mop2, Object s2sndr) {
 in(mop1) by(s1rcvr),
 out(mop2) by (s2sndr),
 mop2._dest == a,
 mop2.b == b,
 mop2.i == i
}
```

Acknowledgments

Thanks to the members of 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., *Situations and Attitudes*, MIT Press, 1983.

[8] 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.

[9] Baumgartner, G., & V. Russo, "Signatures: A C++ Extension for Type Abstraction and Subtype Polymorphism", *Software—Practice and Experience*, 1994.

[10] Birman, K., & R. Van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, 1994.

[11] Booch, G., *Object-Oriented Analysis and Design*, Benjamin Cummings, 1993.

[12] 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.

[13] Buhr, R. & R. Casselman, "Architecture with Pictures", *Proceedings, OOPSLA '92*, ACM, 1992.

[14] Campbell, R. H., & A. N. Habermann, "The Specification of Process Synchronization by Path Expressions". *Lecture Notes in Computer Science 16*, Springer-Verlag, 1974.

[15] Chandy, K. & J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

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

[17] 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.

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

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

[20] Emerson, E., "Temporal and modal logic". J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Volume B*, MIT press, 1990.

[21] Gamma, E., R. Helm, R. Johnson, & J. Vlissides. *Design Patterns*, Addison-Wesley, 1994.

[22] 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.

[23] Gatziu, S., & K. Dittrich, "Events in an Active Object-Oriented Database System", *Proceedings, 1st International Workshop on Rules in Database Systems*, 1993.

[24] Gupta, V., "Concurrent Kripke Structures", *Proceedings of the North American Process Algebra Workshop* Cornell CS-TR-93-1369, 1993.

[25] Harel, D., "StateCharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, 8, 1987.

[26] Harrison, W., & H. Ossher, "Subject-Oriented Programming", *Proceedings, OOPSLA '93*, ACM, 1993.

[27] Harrison, W., *The Importance of Using Object References as Identifiers of Objects*, Document 94.6.12, Object Management Group, 1994.

[28] Helm, R., I. Holland, & D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", *Proceedings, OOPSLA '90*, ACM, 1990.

[29] Hewitt, C., P. Bishop, & R. Steiger, "A Universal Modular ACTOR Formalism for AI", *Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

[30] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, 12, 1969.

[31] Hogg, J., D. Lea, R. Holt, A. Wills, & D. de Champeaux, "The Geneva Convention on the Treatment of Object Aliasing", *OOPS Messenger*, April 1992.

[32] Hughes, G.E., & Cresswell, M.J. *An Introduction to Modal Logic*, Methuen, 1971.

[33] Jagadish, H., & O. Shmueli, “Composite Events in a Distributed Object-Oriented Database” *Distributed Object Management*, Morgan Kaufmann, 1994.

[34] Jarvinen, H., R. Kurki-Suonio, M. Sakkinnen, & K. Systa, “Object-Oriented Specification of Reactive Systems”. *Proceedings, International Conference on Software Engineering*, IEEE, 1990.

[35] Jarvinen, H. *The Design of a Specification Language for Reactive Systems*, Technical Report 95, Tampere University of Technology, 1992.

[36] Jones, C., *Systematic Software Development Using VDM*, Prentice Hall, 1986.

[37] Kiczales, G. *Open Implementations*, Forthcoming book.

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

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

[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] Manna, Z., A. Anuchitanukul, N. Bjorner, A. Browne, E. Chang, M. Colon, L. de Alfaro, H. Devarajan, H. Sipma, & T. Uribe, *STeP: The Stanford Temporal Prover*, Technical Report 94-1518, Stanford University, 1994.

[44] 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.

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

[46] 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.

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

[48] Milner, R., J. Parrow, & D. Walker, “A Calculus of Mobile Processes”, *Information and Computation*, vol 10, pp1-77, 1992.

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

[50] Nierstrasz, O. “Regular Types for Active Objects”, *Proceedings, OOPSLA ’93*, ACM, 1993.

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

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

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

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

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

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

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

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

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

[60] Reenskaug, T. *The Object Industry: The Large Scale Provision of Customized Software*, Addison-Wesley, forthcoming.

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

[62] 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. Also *Sun Microsystems Laboratories Technical Report SMLI TR 94-23* (April 1994).

[63] 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.

[64] Sproull, R., “Guide to the Trace Modeling Tools”. Internal Sun Microsystems Laboratories document, 1993.

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

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

[67] 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.

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

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

[70] 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.

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