Operational Semantics for the Abstract Schema Language ASL

A. Weitzenfeld and M.A. Arbib
Center for Neural Engineering
University of Southern California
Los Angeles, CA 90089-2520
alfredo@usc.edu and arbib@usc.edu

Abstract

The Abstract Schema Language (ASL) defines a hierarchical computational model for the development of distributed heterogeneous systems. ASL extends the capabilities of concurrent object-oriented programming to enable the construction of highly complex multi-granular systems. ASL provides a basic set of concurrency constructs, distinguishing between a *schema* as the template for a process, and a *schema instance* (SI) as an active copy of that process. The ASL model supports aggregation (*schema assemblages*), and both top-down and bottom-up system designs. ASL encourages code reusability by enabling the integration of heterogeneous components, e.g., procedural and neural network programs. ASL schemas are designed and implemented in an orthogonal fashion; integrated, either statically, through *wrapping*, or dynamically, via (task) *delegation*. Schemas include a dynamic interface, made of multiple unidirectional input and output ports, and a body section where schema behavior is specified. Communication is in the form of asynchronous message passing, hierarchically managed, internally, through anonymous port reading and writing, and externally, through dynamic port inter-connections and relabelings. The operational semantics of the language is constructed based on *transition systems*. An example is given illustrating the model and its semantics.

Keywords

schema, instance, distributed, heterogeneous, multi-granular, hierarchical, concurrent, encapsulation, reusability, communication, asynchronous, assemblage, port, connection, relabeling, wrapping, task delegation, transition systems, configuration.

Introduction

The Abstract Schema Language (ASL) [27][29] describes a computational model for the development of distributed heterogeneous systems. ASL presents a hierarchical approach for the design and implementation of computational models where intensive processing and continuous inter-process communication are intrinsic system properties. ASL unifies *schema* modeling [3] with concurrent object-oriented programming (COOP) [31]. Generally speaking, COOP integrates concurrency with object-

oriented design, where an object-oriented language can be analyzed in terms of objects, instantiation, inheritance, and message passing [8]. In a concurrent world, some of these concepts become more complex, especially when designing inheritance schemes [7], where, for example, as an alternative to inheritance, the notion of delegation [17] has been suggested.

The ASL computational model is defined in terms of *schemas*[1], autonomous computational agents which cooperate with each other in a hierarchical fashion. The ASL model hierarchy is shown in Figure 1. At the top of the diagram a schema is shown decomposed into other schemas. This decomposition gives rise to schema aggregation, or schema *assemblages*, where schemas may be composed into complex schema networks. Schemas are specified and implemented in an orthogonal fashion, either through *wrapping*, which enables static integration of heterogeneous external programs (e.g. procedural and neural), or through *delegation*, which enables dynamic integration of schemas as both, specification and implementation tasks. (Simple lines between boxes represent connections between objects, while arrows represent task delegation. The barrier separates the higher level schema specifications from the lower level schema implementations.)

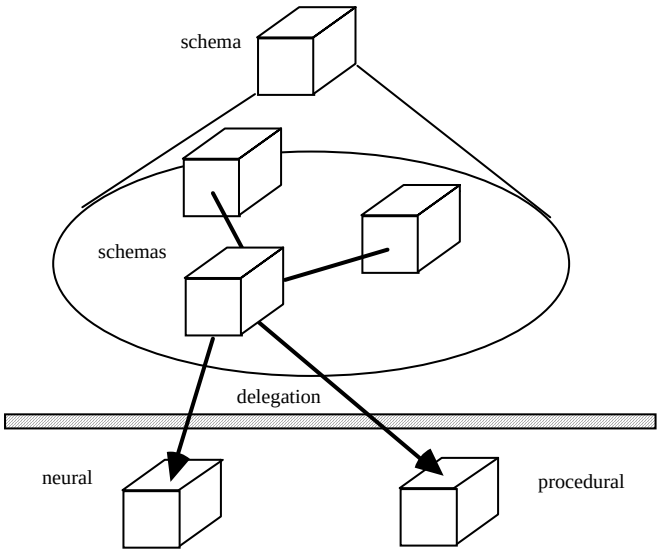


Figure 1. ASL schema model

The ASL communication model is asynchronous, based on dynamic multiple input and output *ports*, *connections* and *relabelings*. The ASL communication model is hierarchically managed, where messages are sent and received anonymously internally to schemas, while actual communication paths between schema ports are externally set. The hierarchical port management methodology enables the development of distributed systems where modules may be designed and implemented independently and without prior knowledge of their final execution environments. Furthermore, dedicated port interconnections avoids the overhead of direct process naming between continuously communicating entities. The ASL communication model is expressive enough to simulate other communication paradigms, such as client/server and blackboards (see [27]).

ASL extends previous work of two different schema systems, Robot Schemas (RS) [18] and Vision Schemas, VISIONS [9]. ASL extends the COOP model by providing a hierarchical approach towards

[1] The concept of *schema*, as presented in this paper, has no relation to the *schema* terminology used in database systems.

heterogeneous and multi-granular concurrent object design. In particular, ASL addresses the development of complex systems integrating developments in Distributed Artificial Intelligence (DAI), Robotics, as well as Brain Theory (BT) and Cognitive Psychology.

The integration of ASL with the Neural Simulation Language (NSL) system [28], a simulation system extensibly used by the neural networks research community, gives rise to the Neural-Schema Language [27] (also referred to as NSL).

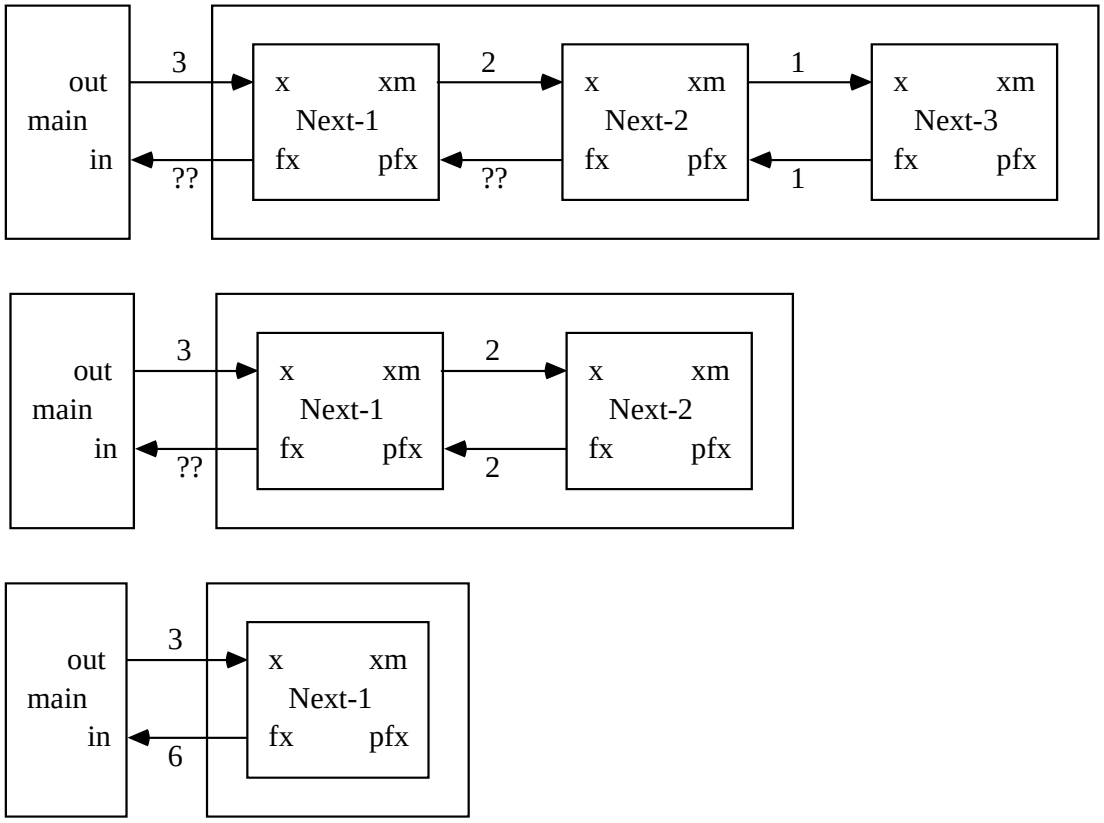


Figure 2. Some of the global configurations that arise during the computation of 3!

An Example of a Schema Program: The Factorial

The schema program for the factorial problem exemplifies ASL constructs. The schema called 'main' initializes the schema program, passing the value for which a factorial is to be computed, while waiting until a result is returned. The schema called 'factorial' provides the recursive definition for calculating the factorial of any number. To convey the essential ideas, we provide snapshots of the global configurations which occur during the computation of 3!, shown in Figure 2. Basically, the program starts by 'main' sending '3' to an initial 'Next' factorial schema instance (SI). Since 3 > 1, the initial 'Next' SI will instantiate another 'Next' SI. Each 'Next' SI will repeat the procedure, getting the 'n' value through its 'x' port for which it is to calculate a corresponding factorial to be returned through 'fx'. If the number 'n'

delivered through 'x' is bigger than '1' then 'n-1' is passed to a newly created 'Next' SI through 'xm' until the last 'Next' process receives the value of '1' and returns the corresponding factorial of value '1'. 'pfx' serves as the return path for intermediate factorial results. Each SI will wait for the successful calculation and delivery of the intermediate factorial value provided by the 'Next' SI to its right, which will de-instantiate once the intermediate factorial result has been delivered.

The factorial program contains only two schemas, although the execution of the program can create indefinitely many schema instances. Figures 3 and 4 contain the code for the two schemas, 'main' (for which only one schema instance is created) and 'factorial' (for which there are many instances, all named 'Next'). Strings in **bold** letters correspond to ASL language keywords, '//' are utilized to precede comments (ending at the end of the line), and all statements are terminated with a ';' (semi-colon). We will briefly go through this code to provide the reader with sufficient intuition to set the stage for the detailed formal semantics given below.

```
schema main
{
external:
internal:
      input in;          // internal input port
      output out; // internal output port

      int n;             // variable for n
      int pn;            // variable for !n

      factorial Next;    // instance reference
body:
      out >=> Next.x;    // connection
      Next.fx >=> in;    // connection
      sin ? n;           // system input
      out ! n;           // send n
      in ? pn;           // get !n
      sout ! pn;         // system output
      stop self
}
```

Figure 3. 'Main' schema code

Execution starts by creating an instance of 'main'. ASL includes a set of pre-defined ports dedicated for system input and output. These structures provide interactive input and output to the programmer. In terms of input, the system port is known as **sin** and in terms of output the system port is known as **sout**.

In the factorial schema example, the following call enables the programmer to interactively assign a value to 'n',

```
      sin ? n;           // system input
```

while the result is printed out on the screen with,

```
      sout ! pn;        // system output
```

```
schema factorial
{
external:
      input x;          // external input port
      output fx;        // external output port
internal:
      input pfx;        // internal input port
      output xm;        // internal output port

      int n;            // variable for n
      int pn;           // variable for !n

      schema* Next;     // instance reference
body:
      x ? n;            // wait until n is read
      if  (n <= 1)
            fx ! 1;
      else
      {
            Next = new factorial;   // Next instance
            xm >=> Next.x;          // connection
            Next.fx >=> pfx;        // connection
            xm ! n-1;               // send n-1
            pfx ? pn;               // get (n-1)!
            fx ! pn*n;              // send n!
      }
      stop self
}
```

Figure 4. 'Factorial' schema code

The external declaration is empty - 'main' has no explicit communication with the outside world. However, the internal declarations set up both the input and output ports, 'in' and 'out', with which it will communicate locally - with the instance of factorial that it creates as a child SI - and creates two integer variables, 'n' to hold the initial data, and 'pn' to hold the final result. Finally, this declaration creates a new instance of the 'factorial' schema and gives it the internal name 'Next'. Once the structure of the schema instance (in this case unique) is thus established, execution of the body begins immediately. 'out **>=>** Next.x' connects the output port out of the 'main' SI to the input port 'x' of its child SI 'Next', while 'Next.fx **>=>** in' connects output port 'Next.fx' to the local schema port 'in'. The command 'out **!** n' then causes it to send the current value of internal variable 'n' through its output port 'out'. The final instruction 'in **?** pn' will read into 'pn' whatever value the input port 'in' receives - but once the SI comes to this instruction it blocks, in the fashion made clear by the question marks '??' in Figure 1, until such a value is indeed received.

We now turn to the way in which the schema code of Figure 4 controls the behavior of each 'Next' SI. The external declaration makes the ports 'x' and 'fx' external, i.e., they can be referenced by the external

environment in which the particular SI of 'factorial' is created, whereas 'pfx' and 'xm' are declared internally and thus cannot be addressed directly by the external environment. Similarly, 'n' and 'pn' are declared as integer variables internal to the new SI. The final part of the declaration is dynamic - rather than creating a new instance of factorial statically, as was done in 'Main', we here simply create a pointer to 'Next', which will point to an SI whose schema-class has not yet been specified. The body then waits for a value to arrive on input port 'x' and then reads it into variable 'n', 'x **?** n'.

Once n has been given a value, the conditional '**if (**n **<=** 1**)** A **else** B' tests to see if this value is less or equal to '1'. If it is, the value '1' is emitted through output port 'fx' and the SI deinstantiates ('**stop** self'). However, if the value is larger than '1', then a new instance is created and "hooked up" appropriately ('Next = **new** factorial; xm **>=>** Next.x; Next.fx **>=>** pfx'), and sent the value of 'n' decremented by '1' through port 'xm' ('xm **!** n-1'). The SI then blocks until it can read a value from input port 'pfx' into variable 'pn' ('pfx **?** pn'), whereupon it multiplies the contents of 'pn' (which will have been set to '(n-1)!') and 'n', and sends the result (which thus equals 'n!') over output port 'pn', to be received by the internal input port of the parent SI, which can thus unblock. Note that after either branch of the conditional has been completed, the 'factorial' SI executes the '**stop** self' command, i.e., this instance of 'factorial' de-instantiates itself. (The 'stop self' command is actually implicitly called when the schema completes its execution.)

The Schema Interface and the Parenting Tree

A schema template is composed of a header containing the schema name and a set of optional instantiation parameters, external and internal declarations, and the schema body. In Backus-Naur form, a schema class definition *sc* takes the form given on the right-hand side of

```
sc ::= schema sn (xp-decl)opt
       {
            external: xv-decl
            internal: iv-decl
            body:     s
       }
```

where *sn* is the schema class name; *xp-decl* is the optional external instantiation parameters; *xv-decl* is the external variable declaration (visible both inside and outside the scope of the schema); *iv-decl* is the declarations visible only inside the scope of the schema; and *s* is the local schema program executing upon instantiation.

All schema programs are initialized from a 'main' schema which gets initially instantiated by the system[2]. During the execution of a specific schema instance (SI), a new SI may be created by instantiating from a schema definition. Similarly, one possible command for an SI is the '**stop**' which de-instantiates

[2] This is somewhat analogous to the situation in POOL [2] where a program is started by instantiating one object out of the existing object class definitions.

it. A schema program terminates once all instantiated processes have de-instantiated. However, in many on-line applications, the program will not terminate.

In Figure 1, we showed the changing pattern of connections between input and output ports as SIs were created and destroyed. The relation between currently active schema instances can also be shown in an abstract "parenting" diagram, as in Figure 5, where a line links each schema instance (*LConf*) to all the "child" instances of which it is a "parent". *GConf* correponds to the total processing environment of all instantiated schemas, while *LConf* corresponds to a single schema process.

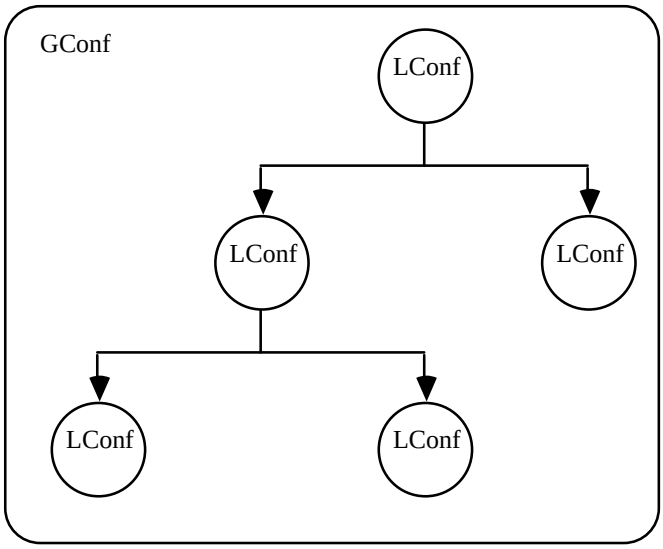


Figure 5. Parenting tree for schema instances

The root of the tree corresponds to the initial SI in the system, while nodes and leaves represent schema processes which are instantiated during the on-going execution of the program. Internal nodes in the tree correspond to parent SIs, while leaves in the tree correspond to SIs having no children processes. In our factorial example (Figure 2), 'main' corresponds to the root process of the program. Processes to the right are children processes while processes to the left correspond to parent processes. (The tree is basically one single branch.)

Encapsulation is accomplished by stipulating that internal (local) variables can only access other internal variables in the SI environment. The **external section** of the schema interface specifies which variables may be accessed both from the external environment as well as locally. If a schema is to be able to interact with the outside world, in particular through port connections, the basis for communication in ASL, it is crucial that the schema contain external ports. The **internal section** of the schema interface specifies which variables may be accessed only from the local schema environment. **Instantiation parameters** provide an optional alternative for initializing schema instance variables with special values.

Independently of whether external or internal in their scope, the possible static **declarations** in ASL apply to the following entities:

Input and Output Ports: Ports are unidirectional. Output ports are used for sending messages from one schema instance to another. Input ports are used for reading messages from other schemas. Input ports have to be connected from other output ports before messages may be actually received. A string of output ports is declared as:

output $op_1,...,op_n$;

where $op_1,...,op_n$ are output port identifier, while a string of input ports is declared by:

input $ip_1,...,ip_n$;

where $ip_1,...,ip_n$ are input port identifiers.

Schemas: Schemas define the active types in ASL. (Ports, on the other hand, are passive types since they execute as part of a schema process and not independently.) A set of schemas is declared as follows

schema-name $si_1,...,si_n$;

where *schema-name* specifies the schema type to be instantiated, and $si_1,...,si_n$ the corresponding schema instance identifiers. This declaration allocates a new n new instances of the given schema which immediately starts execution.

Primitive Types: ASL supports basic types, in particular **int**, **char** and **float**. These types provide completeness to the language. In general a set of primitive types are declared as follows

p-type $v_1,...,v_n$;

where *p-type* is either **int**, **float** or **char**, and $v_1,...,v_n$ the corresponding variable identifiers.

Arrays: One important data abstraction in ASL is the array. Arrays in ASL, analogous to other languages, help define a sequence of structures with a single declaration while enabling the programmer to access the individual array elements through indexing. The general array definition is given by

type *id*[*N*]

where *type* corresponds to any type in ASL, including schemas, ports and primitive types, while *id* is the name identifier for the array, and *N* is the number of elements in the array.

Arrays may have more than one dimension, in which case the basic declaration is defined with as many *N*s as needed,

type *id*$[N_1]...[N_n]$

where N_i corresponds to the *i*th dimension in the *i*th of *n* dimension array. (Array dimensions may be dynamically specified [27].)

By contrast with the above static declarations, **dynamic** declarations involve a two step process. First a pointer, similar to that in C, gets declared. This pointer does not allocate any memory for the particular kind of structure which it represents. It simply specifies a pointer serving as future reference to a corresponding structure allocation. While dynamic allocation is possible for any structure type, the most

interesting possibilities are given by dynamic allocation of ports and schemas. The dynamic alternative for port allocation permits the incorporation of new ports into an already compiled schema structure. Dynamic allocation in schemas provide the most important abstraction in ASL where processes may be instantiated and start executing at any point.

For example, in the factorial program, besides the initial static instantiation of a 'Next' factorial schema from the 'main' schema, a dynamic two step instantiation process was used:

schema* Next;

declares a pointer called 'Next' pointing to a future schema instantiation, then later the actual 'Next' schema instantiation in the schema body is executed as

Next = **new** factorial;

Extended flexibility was achieved here by declaring a pointer to a generic schema, allowing the programmer to actually decide the specific schema type during run-time, instead of during compilation.

A final issue in the schema interface is how different structures are referenced according to their locality. For example, in terms of ports, besides distinguishing between input and output ports, and between external and internal ports, we distinguish between local and remote port references. **Local ports** are those ports which are referenced by the schema instance of which they are part and are usually referenced simply by the simple identifier. (A local port may also be referenced by prefixing the port identifier with the local schema instance self reference, i.e., '**self**.*id*', where *id* is the port identifier.) **Remote ports**, on the other hand, are those ports belonging to a child schema instance and referenced by its parent instance with the syntax *si.port* where *si* is the name identifier for a schema instance, and *port* is the name identifier for a port belonging to that particular schema.

In the ASL port model, data can only be read or written directly to local ports. Reading and writing to remote ports can only be accomplished indirectly via connections and relabelings, as will be explained in the next sections.

Local and Global Configurations

We will approach the definition of ASL semantics in terms of local and global configurations. Local configurations together define the global configuration, with each local configuration representing the state of a single schema instance. We thus need a unique way of labeling SIs so we can distinguish their (local) configurations. In models of serial languages, a configuration of a machine in its simplest form consists of a program and a store, $Conf = Syn \infty Sto$, where programs are taken from *Syn*, the set of syntactically correct language programs, and Sto is the set of all stores. A store is defined as a function from variable names to values, where variables that are not in the store are undefined. In our ASL semantics, the local configuration must specify not only the state in the local $Syn \infty Sto$, but must also show how the SI is connected to other SIs, and what values are stored in its internal variables.

At any time in the execution of a schema program, there will be a parenting tree (Figure 5 for an example) linking the schema instances. The global configuration will then provide a local configuration for each node of that tree. Formally, a typical element c of the set *GConf* (or *Conf*) of global configurations is given by a set of local configurations $\{ c_1, \ldots , c_n \}$, with one c_i in *LConf* (the set of local configurations) for each element of the parenting tree. When any SI executes an instruction it may simply change internal state, it may read or write over its ports it may change the topology of the SI network, adding or deleting nodes (instantiating or deinstantiating SIs) or, as we shall see, changing the connectivity structure or its labeling. In order to define configurations, the following sets are defined:

- *SName* represents the set of **schema class names**, *sn* □ *SName*,
- *SVDecl* represents the set of **schema internal and external variables and parameter declarations**, *sv-decl* □ *SVDecl*, *SVDecl* = *XPDecl* ≈ *XVDecl* ≈ *IVDecl*, where *XPDecl* represents the set of schema instantiation parameters, *xp-decl* □ *XPDecl*; *XVDecl* represents the set of schema external variables, *xv-decl* □ *XVDecl*; and *IVDecl* represents the set of schema internal variables, *iv-decl* □ *IVDecl*.
- *SBody* represents the set of **schema bodies** corresponding to the schema instance statements, where a typical element *s* □ *SBody*, defines an ASL language statement.
- *SDef* = *SName* ∞ *XPDecl* ∞ *XVDecl* ∞ *IVDecl* ∞ *SBody* represents the set of **schema class definitions**, *sd* □ *SDef*.
- *SIR* represents the set of **schema instance labels**, where each different SI is represented by a unique global identifier α □ *SIR* = N^+, where '0' corresponds to the root schema instance in the configuration hierarchy ('main'), and $\alpha.n$ corresponds to the n^{th} SI created as a child of SI α.
- The set of **schema instance variable states** is $\Sigma = N^{SVDecl} \infty N^+$ with elements $\sigma = < \sigma_1, \sigma_2 >$. σ_1 is the local store with typical element $\sigma_1(v)$ □ N (assuming integer values for simplicity) for v □ *SVDecl*. σ_2 □ N is the local process instantiation counter which is used to create unique process labels. σ_2 is initially set to '0'. When the given SI α creates a new SI through instantiation, the new SI is given the label $\alpha = \alpha_p.\sigma_2$ and σ_2 is incremented by '1'.
- The set of **children processes**, *SIC* ∏ *SIR*, has typical element ι, representing the set of all locally instantiated children processes.

Ports, Connections, Relabels, and Buffering

When describing communication, it is necessary to keep track of ports, connections and relabelings. Thus, we define the following sets:

- The set 2^{Conn} of **schema instance port connections** has typical element $\kappa = \{ \kappa_1, \ldots , \kappa_n \}$ where each κ_s is an ordered pair of the form $\kappa_s = [(\alpha_i,p_i), (\alpha_j,p_j)]$□ *Conn*, where α_i, α_j are schema instance labels, p_i, p_j are port identifiers, and port p_i of SI α_i is connected to port p_j of SI α_j (the first element must be an output port, while the second must be an input port).

- The set 2^{Rel} of s**chema instance port relabelings** has typical element $\xi = \{ \xi_1, ... , \xi_n \}$ where each ξ_S is of the form $\xi_S = [(\alpha_i,p_i), p_j] \in Rel$, where α_i is an SI label, and p_i, p_j are port identifiers, where port p_i of SI α_i is relabeled to be referred to by the port identifier p_j.
- The set 2^{Port} of **schema instance ports** has typical element $\pi = \{ \pi_1, ... , \pi_n \}$ with typical list element $\pi_S = < p, ptype, buf, bstate, \kappa\pi in, \kappa\pi out, \xi\pi in, \xi\pi out > \in Ports$, where

p is the port identifier,

$ptype \in$ { **input**, **output** } is the port type,

buf is the port's buf fer, where we use the notation of $buf \varnothing x$ for reading a message out of the buffer, and $buf \blacklozenge x$ for writing a message x into the buffer. The change in the buffer state is represented by *buf* '.

bstate $\in$ { **true**, **false** } is the state of the port buffer given by **true** if a message is waiting in the buffer and **false** if the buffer is empty.

$\kappa\pi in = \{ \kappa\pi in_1, ... , \kappa\pi in_n \}$ represents the set of connected ports to which p is to send data (only applies to p being an output port), where $\kappa\pi in_i = (\alpha_i,p_i)$,

$\kappa\pi out = \{ \kappa\pi out_1, ... , \kappa\pi out_n \}$ represents the set of connected ports from which p is to receive data (only applies to p being an input port), where $\kappa\pi out_i = (\alpha_i,p_i)$,

$\xi\pi in = \{ \xi\pi in_1, ... , \xi\pi in_n \}$ represents the set of relabeled ports to which p is to forward its data (only applies to p being input port), where $\xi\pi in_i = (\alpha_i,p_i)$,

$\xi\pi out = \{ \xi\pi out_1, ... , \xi\pi out_n \}$ represents the set of relabeled ports from which p is to receive its data (only applies to p being an output port), where $\xi\pi out_i = (\alpha_i,p_i)$,

It should be noted that while it may seem that the port connection and relabeling information is unnecessarily duplicated, this information is replicated since ports may be in distributed processes, and it would become inefficient to have to ask for this information every time a message is sent or received. Furthermore, the presence of this extra information enables routing optimization when sending or receiving messages through multiple relabels.

Since the schema model is hierarchical, and based on the notion of "parenting", the intuition behind relabeling is that a port belonging to child schema instance may be accessible through its parent schema instance in order to enable external communication outside the local parent schema instance environment. In terms of output ports, a child schema instance communication will be sent, indirectly, to the destination specified by the inter-connections of the parent schema instance port to which the relabeling is made. On the other hand, in terms of input ports, any communications received by the port belonging to the parent schema instance will be forwarded to the child schema instance port to which a relabeling has been specified. A parent schema instance port forwarding messages according to local relabeling specification does not use its local buffer for intermediate storage, all messages are immediately sent to its new destination.

Both new and old labels can be used simultaneously. Furthermore, since the communication model supports fan-in and fan-out of both connections and relabels, a port may be simultaneously relabeled and also have connections to other ports. Each connection or relabel specifies an independent communication path which does not affect the other ones. Fan-out specifies how each message is copied into multiple communication paths, while fan-in simply sequentializes messages arriving from multiple communication sources.

In ASL, each *input* port has a local buffer. Sending a message involves delivering a message to its destination, where it is placed in the buffer of the input port. Buffers are unbounded theoretically, so messages are never lost (besides, one of the assumptions in the model is guaranteed message delivery, an assumption common to many asynchronous models, like actors [1]). Buffers in ASL are considered unbounded. (In practice, buffers are bounded by the available memory.) The buffer is a first-in-first-out (FIFO) device, where messages are read according to their arrival order. Once read, messages are retrieved from the buffer. ASL supports checking the state of the buffer, as well as checking the current message to be read, without actually reading it. This gives the programmer extended flexibility in deciding when to read from an input port, thus avoiding possible blocking. Reading is simply done by retrieving the first message from a local buffer without having to know its source. The only precondition on message reading is that the local process buffer has a message waiting to be read. Thus, for each *input* port p of SI α, the local configuration must include the state π of that port.

Finally, the following set of elements are defined on ports:

- *IPort* represents the set of internal ports, $ip \in IPorts \prod IVDecl$
- *XPort* represents the set of external ports, $xp \in XPorts \prod XVDecl$
- *LPort* represents the set of local ports, $lp \in IPorts \approx XPorts$
- *RPort* represents the set of remote ports, $(si,rp) \in SIR \infty (IPorts \approx XPorts)$.

Internal ports are those ports which can only be referenced by the schema body common to the schema owning the ports. External ports are those ports which may be referenced outside the local schema body, enabling a parent schema instance to reference the port of its children schema instances.

Local ports are those ports referenced from the schema body belonging to the same schema instance to which the port belongs. Remote ports, on the other hand, are those ports belonging to a child schema instance as referenced from its parent schema instance body.

The distinction between local and remote ports is made to emphasize the restriction in the ASL port model, where data can only be read or written directly to local ports. Reading and writing to remote ports can only be accomplished indirectly via connections and relabelings. There are two reason for these restrictions in disallowing direct reading and writing to remote ports. First, the possibility would exist where messages could be read and written to both local and remote ports, losing the distinction between input and output ports (i.e. an input port from which data is locally read may be sent remote data as if it

were an output port, making the distinction between input and output ports irrelevant). Second, and most important, the intended anonymous communication hierarchy would be lost through direct naming of remote ports, making the notion of connections and relabels irrelevant since any process would be able to directly write (or read) to any port belonging to any other process.

Delegation

The concept of delegation defines how a parent schema, as part of the delegation scheme, will forward its messages to the delegated (children) schema instances (through port relabeling), which may be many (multiple delegation). Delegation provides a dependency between the delegated instance and the delegating one, through the explicit "delegate" command, telling the system that the delegating schema has to stay alive until all its delegated schemas complete their task an de-instantiate. Without delegation no dependency exists between parent and children instances, except a hierarchical relation in the global referencing scheme. Delegation differs from the general notion of children instances existing in a parent instance's environment in that the delegated schema instance serves as the body of the delegating schema - and thus should have its ports relabeled to those of the parent or delegating schema instance.[3] Furthermore, the delegating process cannot terminate before the delegated process does. This is to ensure that messages sent to the delegating process are actually transmitted to the delegated one. Delegation may be defined on multiple children schema instances as follows

delegate $si_1,...,si_n$

where the schema instance executing the above expression will not terminate before *si* does.

The following set is defined for delegation:

- The next component is the set 2^{Del} of **delegated schema instances**, with typical element $\delta = \{ \delta_1, ... , \delta_n \} \in 2^{Del}$, with $\delta_s \in Del \prod SIC$.

In general terms, a schema is considered a delegating schema if the following three conditions are met, where the child process is considered

i) the schema instantiates a process (at least one process),

ii) the schema relabels its ports to those of the process,

iii) the schema terminates only after the instantiated process has completed its task (de-instantiated).

Defining a Local Configuration

We can now define a local configuration:

[3] By delegating the task implementation from a parent schema to its child we define the notion of *dynamic schema tasks*. On the other hand if a task implementation is specified into the parent schema at compilation time, we then define *wrapping* of implementation programs into schemas. Wrapping enables the development of independent programs, which may then be integrated into the schema model. Wrapping programs into schema classes provides a way to integrate programs written with different languages constructs. This aspect is more of an implementation issue, since different compilers have different restrictions, and sometimes there may be some conflicts when linking object files created through different compilers.

$$LConf = SIR \infty\ SBody \infty \Sigma \infty SIC \infty 2^{Conn} \infty 2^{Rel} \infty 2^{Del} \infty 2^{Port} \infty SName$$

with typical element $c = < \alpha, s, \sigma, \iota, \kappa, \xi, \delta, \pi, sn >$.

Structural Operational Semantics

The semantics for ASL will be given in terms of a Structural Operational Semantics (SOS) [22][11], which serves to describe the basic operations of a language by going over the transitions that a machine might take during the execution of a program. Related studies have provided an SOS for CSP [22], POOL [2], and RS [4].

To describe observable inter-process communication in the form of message passing, [23] introduced the notion of *labeled transition systems* described by a three element tuple •*Conf*, ∅, *Λ*®, where *Conf* is the set of possible configurations, *Λ* is the set of possible communication labels, and ∅ is a subset of the set $Conf \infty \Lambda \infty Conf$. The set of possible communication labels Λ defines the observable messages which a system can have, and for our particular language, are given by,[4]

- ε which indicates no communication (or externally unobservable communication).
- p ! x which indicates a message x sent to p.
- p ? x which indicates a message x received from p.

We use the notation $C_i \xrightarrow{\lambda_i} C_{i+1}$ to indicate that $(C_i, \lambda_i, C_{i+1}) \in \varnothing$, i.e., that the semantics allows a transition from C_i to C_{i+1} in which the communication denoted by λ_i may be observed. We may use $C_i \varnothing C_{i+1}$ or $C_i \not\varnothing C_{i+1}$ as alternative representations.

The transition relation, defining the possible transitions between configurations, is inductively defined over the structure of the language using a collection of axioms and rules. Axioms are defined by

$$C_i \xrightarrow{\lambda_i} C_{i+1},$$

which describes a single transition from configuration C_i into configuration C_{i+1}, having communication label $\lambda_i \in \Lambda$ and $C_i, C_{i+1} \in Conf$. An inference rule

$$\frac{C_a \xrightarrow{\lambda} C_b}{C_c \xrightarrow{\lambda'} C_d}$$

allows us to infer the validity of transition $C_c \xrightarrow{\lambda'} C_d$ from that of transition $C_a \xrightarrow{\lambda} C_b$.

A valid computation is then given by a (finite or infinite) sequence of configurations

$$C_0 \xrightarrow{\lambda_0} C_1 \xrightarrow{\lambda_1} \ldots C_i \xrightarrow{\lambda_i} C_{i+1} \xrightarrow{\lambda_{i+1}} \ldots,$$

starting from some initial configuration C_0, and with $C_i \xrightarrow{\lambda_i} C_{i+1}$ a valid transition for some λi for each i. The actual meaning of the program is abstracted from this sequence. It might be the transformation of certain variables from the initial state of the store to the values of (other) variables in the state of the store in some final configuration, or it might be the stream of external communications

[4] Similar labels are used in describing the transition system for CSP [23].

throughout program execution. Here our concern will be to formalize the legal transitions. It should be noted that some axioms involve single local configuration while others involve sets of them. When a transition affects only a single local configuration brackets will be omitted from the corresponding axioms and rules.

We start with few basic rules, and then systematically work through an interleaved exposition of the syntax and semantics of most of the ASL constructs. The axiom for the empty statement[5]

$< \alpha, \varepsilon;s, \sigma, \iota, \kappa, \xi, \delta, \pi, sn > \not\Rightarrow < \alpha, s, \sigma, \iota, \kappa, \xi, \delta, \pi, sn >$

shows that an empty statement has no effect on the configuration state. The next rule describes sequential statement composition for any arbitrary schema body statement,

$$\frac{< \alpha, s_1, \sigma, \iota, \kappa, \xi, \delta, \pi, sn > \not\Rightarrow < \alpha, s_1', \sigma', \iota', \kappa', \xi', \delta,' \pi', sn >}{< \alpha, s_1;s_2, \sigma, \iota, \kappa, \xi, \delta, \pi, sn > \not\Rightarrow < \alpha, s_1';s_2, \sigma', \iota', \kappa', \xi', \delta,' \pi', sn >}$$

where if one configuration changes state by going over one statement in the language, we infer that the configuration changes state by going over that statement while the next statement is still pending.

Schema Body

The schema body specifies how schema instances actually execute. The language expressions, those constructs which return a value, and the language statements, those constructs returning no values, causing only side effects, are described next. We first examine assignment, conditionals, and iteration, which only involve changes to the local configuration of the SI of the body to which they belong.

(i) The general **assignment** syntax is given by $v = e$ where e is a value-returning expression, including a variable, and v is the variable in which the value is to be stored. The value returned by e and the variable v should be of the same type. Assignment is described by the transition,

$< \alpha, v=e, \sigma, \iota, \kappa, \xi, \delta, \pi, sn > \not\Rightarrow < \alpha, \varepsilon, \sigma[\sigma(e)/v], \iota, \kappa, \xi, \delta, \pi, sn >$

where $\sigma(e)$ represents the value returned by expression e, and $\sigma[\sigma(e)/v]$ represents the storing of that value in v.

(ii) Conditionals are defined in terms of the 'if-else' statement **if (***e***) { ***s*$_1$ **} else { ***s*$_2$ **}** where e is the conditional expression, which if true enables the execution of s_1, with s_2 executing otherwise. The 'else' part of the conditional is optional. The corresponding transition is

$< \alpha,$ **if (***e***) { ***s*$_1$ **} else { ***s*$_2$ **}**$, \sigma, \iota, \kappa, \xi, \delta, \pi, sn > \not\Rightarrow < \alpha, s_1, \sigma, \iota, \kappa, \xi, \delta, \pi, sn >$ if $\sigma(e) =$ **true**

$\not\Rightarrow < \alpha, s_2, \sigma, \iota, \kappa, \xi, \delta, \pi, sn >$ if $\sigma(e) =$ **false**

where $\sigma(e)$ represents the value returned by expression e for store σ.

(iii) Iteration is defined in terms of the 'while' statement **while (***e***) { ***s* **}** where e is the conditional expression, which if true enables the execution of s. This corresponds to

$< \alpha,$ **while (***e***) { ***s* **}**$, \sigma, \iota, \kappa, \xi, \delta, \pi, sn > \not\Rightarrow < \alpha,$ **if (***e***) { ***s***;** **while (***e***) { ***s* **} } else { ***ε* **}**$, \sigma, \iota, \kappa, \xi, \delta, \pi, sn >$

[5] In the following axioms and rules, ε represents either an empty statement or an empty label, accordingly.

where, depending on the value returned by expression *e* for σ, either *s* will be computed and a new iteration will take place or the statement will be completed, as signified by ε.

(iv) When we turn to **instantiation**, we must not only update the local configuration of the SI which contains the instantiation request but must also add a new local configuration representing the initial state of the new instantiation.

A **static schema instantiation** takes place in the schema declaration section with

schema-name sid or *schema-name sid*(*init-pars*)

where *sid* is the schema reference, and *schema-name* is a previously defined schema name. A **dynamic schema instantiation** takes place by first having a schema pointer declaration in the schema declaration sections, complemented by an instantiation construct in the schema body, where the declarations is given by

schema* *sid* or *schema-name** *sid*

depending on whether the generic **schema** pointer declaration or the specific schema-name pointer declaration is used. The instantiation construct takes the form, for either of the above declarations

sid = **new** *schema-name* or *sid* = **new** *schema-name*(*init-pars*)

where the **new** expression returns a reference to a *schema-name* object. In terms of transitions, the net effect of these statements is to add a new element to the set of local configurations

$< \alpha,$ new $sn, \sigma, \iota, \kappa, \xi, \delta, \pi, sn_O > \otimes < \alpha, \varepsilon, \sigma', \iota', \kappa, \xi, \delta, \pi, sn_O > < \alpha_C, s, \sigma'', \iota'', \kappa'', \xi'', \delta'', \pi'', sn >$

The right hand side has two terms. The first corresponds to the parent SI α, while the second represents the initial state of the *LConf* for the new SI, which is labeled α_C. This label is created by concatenating the label of the parent schema instance, α, with a new unique number, according to the local counter σ_2 in α, which gets incremented every time a new child schema gets instantiated. Thus, for σ_2 taken from $\sigma = < \sigma_1, \sigma_2 >$, the new label is created by $\alpha_C = \alpha \sqsupseteq \sigma_2$ and the internal counter σ_2 is then incremented to yield $\sigma' = < \sigma_1, \sigma_2+1 >$. The set of children schema instances is updated in the parent schema instance α with the label of the new child schema instance, $\iota' = \iota \approx \{ \alpha_C \}$.

For the local configuration of the new SI α_C, *s* is the body of schema sn where execution will immediately proceed. The store for the newly instantiated schema, $\sigma'' = < \sigma_1'', \sigma_2'' >$, is reset with all variables set to '0'[6] and all pointers set to 'nil', and where the new local counter for creating unique local labels, σ_2'', is also set to '0'. The set of local schema instances, connections, relabels, and delegated schema instance for the newly instantiated α_C is reset, represented by the empty set, □: $\iota'' = \kappa'' = \xi'' = \delta'' =$ □. (Of course, subsequent instructions in either α or α_C may immediately set up new values for any or all of these). The set of ports is also reset with $\pi'' = \{ \pi_1, \ldots, \pi_n \}$, $\pi_S = < p, ptype, \varepsilon,$ **false**$,\varepsilon,\varepsilon,\varepsilon >$, where π_S corresponds to an arbitrary port. The port *p* is of type *ptype*, while if an input port its buffer is

[6] When instantiating a new schema with '**new** sn(x_1,...,x_n)', which includes instantiation parameters, the only difference is that $\sigma'' = < \sigma_1'', \sigma_2'' >$, with $\sigma_1''(\bar{x} \varnothing \bar{v}) = \{ \lambda v_i, x_i \mid v_i = x_i, v_i \in$ XPDecl $\}$, where $\bar{x} = \{ x_1, \ldots, x_n \}$, $\bar{v} = \{ v_1, \ldots, v_n \}$.

reset to ε, and the corresponding buffer state is set to **false**, while all local connections and relabels are initially empty. If an output port the buffer and its state are irrelevant.

In order to define the initial set of local configurations, it is necessary to construct an initialization function which depends on a schema's static declarations. This initialization function should be recursively defined, since a static schema instantiation may in itself contain further static schema instantiations[7]. For such purposes a special inference rule is defined based on the addition of a new local configuration to the set of local configurations every time a static schema instantiation is encountered. Through the recursive application of this rule, an infinite hierarchy of static schema instantiations may be taken care of. Furthermore, this initialization function is applied not only during initialization of a complete schema program, but every time a schema gets instantiated throughout the program.

In general, for a set of local configurations C_0^{α} representing the state of the program after the addition of a new local configuration c_α corresponding to a newly instantiated schema α, the initialization function 'IFunc(α)' will be defined by

$$\text{IFunc}(\alpha) = \begin{cases} \bigcup_{\alpha_i} C_0^{\alpha_i} & \alpha_i \in SVDecl_\alpha \\ \varepsilon & \alpha_i \notin SVDecl_\alpha \end{cases}$$

where $SVDecl_\alpha$ corresponds to the schema interface of schema instance α, and α_i corresponds to every static schema instantiation in α. So, basically, the function when applied to a newly instantiated schema α will look at any statically declared schema α_i in α. For each one, their own initial configuration C_0^{α} will be calculated by,

$C_0^{\alpha} = \{ c_\alpha \} \approx \text{IFunc}(\alpha)$,

and thus recursively defining the initial configuration for α (note that α is a parameter in the function for any schema instance which is currently initialized).

In order to deal with static schema instantiations as part of dynamic schema creation, the following instantiation rule is given,

$$\frac{C \xrightarrow{\varepsilon} C' \cup \{c_\alpha\}}{C \xrightarrow{\varepsilon} C' \cup C_0^{\alpha}}$$

where $C \xrightarrow{\varepsilon} C' \cup \{c_\alpha\}$ represents the transition from a set of local configurations C to C' plus the instantiation of a new schema α and $C \xrightarrow{\varepsilon} C' \cup C_0^{\alpha}$ represents the transition from the set of local configurations C to C' which now includes any static instantiations due to α and given by C_0^{α}.

(v) A schema instance is **de-instantiated** either implicitly when its body finishes execution, or explicitly by stopping the schema instance. There is a slight difference between these. In the implicit way, a schema instance will only die after all its delegated schema instances have themselves died (refer to the delegation section below). An explicit schema de-instantiation is accomplished by **stop** si where si is a

[7] [4] described such an initialization procedure for assemblages in RS where he introduced the notions of *unfold* and *collect*, yet their definition involves great complexity, in particular the static assemblage initialization procedure.

schema instance and **stop** is a statement executed in the body of the schema process to be de-instantiated. When a schema process is stopped all its relabels and connections are deleted. To stop a schema instance from its local body the following expression is used, **stop self**, where **self** corresponds to the local schema instance reference.

Implicit de-instantiation is described by the following two axioms, where the first case corresponds to a schema instance having no delegated schema instances, while the second case corresponds to a schema instance which does have delegated ones. If δ_c is empty, $\delta_c' = \emptyset$, then

$\{< \alpha_c, \varepsilon, \sigma_c, \iota_c, \kappa_c, \xi_c, \delta_c, \pi_c\ sn_c>, < \alpha, s, \sigma, \iota, \kappa, \xi, \delta, \pi, sn >\} \not\otimes \quad < \alpha, s, \sigma, \iota', \kappa', \xi', \delta, \pi', sn>$

where α is the label of the parent SI of α_c. The local configuration of α_c is removed from the global configuration, while all reference to the child instance α_c are removed from the parent SI α:

$\iota' = \iota - \{ \alpha_c \}$; $\kappa' = \kappa - \{ (\alpha_i,p,q) \mid [(\alpha_c,p), (\alpha_i,q)] \Delta [(\alpha_i,q),(\alpha_c,p)] \}$; $\xi' = \xi - \{ (p,q) \mid [(\alpha_c,p), q] \}$; and $\delta' = \delta - \{ \alpha_c\}$. (Children schema instances of a stopped schema instance keep on executing.)

All the references to the local port relabel list in π are removed,

$\pi' = \{ \pi_p', \ldots, \pi_{pn} \}$

$\pi_p' = < p$, output, n/a, n/a, $\kappa\pi in_p$, $\kappa\pi out_p$, $\xi\pi in_p'$, $\xi\pi out_p' >$

$\xi\pi in_p' = \xi\pi in_p - \{ (\alpha_c, p) \mid (\alpha_c,p) \square \xi\pi in_p \}$

$\xi\pi out_p' = \xi\pi out_p - \{ (\alpha_c, p) \mid (\alpha_c,p) \square \xi\pi out_p \}$

Since α_c may include connections to other schema instances, then besides the parent schema having its local configuration modified, it is necessary to reset those references from the child instance α_i set of ports π. The following rule is applied to as many children schemas as necessary.

$$\frac{\{\alpha_c, \varepsilon, \sigma_c, \iota_c, \kappa_c, \xi_c, \delta_c, \pi_c, sn_c\rangle, \langle \alpha, s, \sigma, \iota, \kappa, \xi, \delta, \pi, sn\rangle\} \xrightarrow{\varepsilon} \{\alpha, s, \sigma, \iota', \kappa', \xi', \delta', \pi', sn\rangle\} \wedge \alpha_c, \alpha_i \in \alpha.\iota}{\{\alpha_i, s_i, \sigma_i, \iota_i, \kappa_i, \xi_i, \delta_i, \pi_i, sn_i\rangle\} \xrightarrow{\varepsilon} \{\alpha_i, s_i, \sigma_i, \iota_i, \kappa_i, \xi_i, \delta_i, \pi_i', sn_i\rangle\}}$$

where all the references to other child instance port connections in $\alpha_i.\pi$ are removed,

$\pi_i' = \{ \pi_p', \ldots, \pi_{pn} \}$

$\pi_p' = < p$, *ptype, buf, bstate*, $\kappa\pi in_p'$, $\kappa\pi out_p'$, $\xi\pi in_p$, $\xi\pi out_p$ >

$\kappa\pi in_p' = \kappa\pi in_p - \{ (\alpha_c, p) \mid (\alpha_c,p) \square \kappa\pi in_p \}$

$\kappa\pi out_p' = \kappa\pi out_p - \{ (\alpha_c, p) \mid (\alpha_c,p) \square \kappa\pi out_p \}$

where connections may be to either input or output ports in α_c.

If δ_c' is not empty, $\delta_c \neq \emptyset$, then,

$< \alpha_c, \varepsilon, \sigma_c, \iota_c, \kappa_c, \xi_c, \delta_c, \pi_c, sn_c > \not\otimes \quad < \alpha_c, \varepsilon, \sigma_c, \iota_c, \kappa_c, \xi_c, \delta_c, \pi_c, sn_c >$

and no change in the configuration occurs.

In terms of explicit schema de-instantiation, there are two different alternatives. The first one is used for de-instantiating the executing schema instance through the use of '**stop self**',

$\{< \alpha_c,$ **stop self**, $\sigma_c, \iota_c, \kappa_c, \xi_c, \delta_c, \pi_c\ sn_c>, < \alpha, s, \sigma, \iota, \kappa, \xi, \delta, \pi, sn >\} \not\otimes \quad < \alpha, s, \sigma, \iota', \kappa', \xi', \delta, \pi', sn>$

The general stop command is defined for any child SI — any other SIs would be outside the scope of the locally executing schema.)

{< α,**stop** α_c, ι, κ, ξ, δ, π, sn >,< α_c, s, σ_c, ι_c, κ_c, ξ_c, δ_c, π_c sn_c>} $\Rightarrow$ < α, s, σ, ι', κ', ξ', δ, π', sn>

where now the schema that gets de-instantiated is the one referred to by α_c, and not the one executing the stop command. Yet, the updates to the list of connections, relabels, instantiations and delegations are treated as before, save that the parent SI is the one issuing the stop command.

Port Management

We now give the syntax and semantics for the various ways of handling the ports of the SIs.

(i) Adding Ports: Ports may be dynamically instantiated, or added to the schema interface, by the use of the **new** *p* command, analogous to schema instantiation. The transition is

< α, **new** p, σ, ι, κ, ξ, δ, π, sn > $\Rightarrow$ <α, ε, σ, ι, κ, ξ, δ, π', sn >

i.e., a new port *p* is added to the set π, $\pi' = \pi \approx$ { <*p*,*ptype*,*buf*,**false**> } with the state of *p* set to **false**, and *buf* set to empty.

(ii) Deleting Ports: Only dynamically instantiated ports may be deleted. The syntax is **delete** id, where id is the port identifier, and the expression is general for any type of port. The transition is< α, **delete** p, σ, ι, κ, ξ, δ, π, sn > $\Rightarrow$ <α, ε, σ, ι, κ, ξ, δ, π', sn >

where a port *p* is deleted from the set π, $\pi' = \pi$ - { <*p*,*ptype*,*buf*,*bstate*> }.

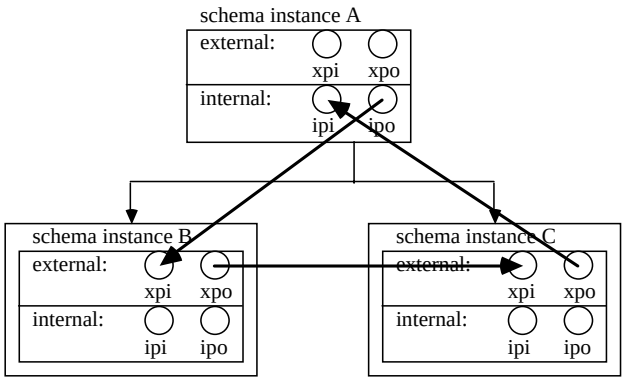
(iii) Connections: In order for communication to take place connections have to be established between ports. Connections are made exclusively between output and input ports. Ports may be connected and dis-connected in a dynamic fashion.[8] Connections not only provide the functionality for linking different schema instances, but they serve as basis for the next generation of schema learning models, where the dynamic nature of port connections becomes critical in describing evolving network topologies. The syntax for connecting an output port to an input port is

output_port **>=>** *input_port* or *input_port* **<=<** *output_port*

where output_port is the identifier of the output port and input_port is the identifier of the input port.

To further describe the notion of connections, consider Figure 6, where connections are made by 'A' between the two remote ports 'xpo' and 'xpi' belonging to 'C' and 'B', respectively. Connections are also made between the two local ports in 'A' and two of its remote ports. The first of those connections is between 'opo' in 'A' and 'xpi' in 'B', while the second connections is between 'opi' in 'A' and 'xpo' in 'C'.

[8] Multiple port inter-connections are allowed, providing fan-in and fan-out. See [27] for details.




```
// port inter-connections
schema schemaA
{
external:
      input xpi;
      output xpo;
internal:
      input ipi;
      output ipo;
      schemaB B;
      schemaC C;
body:
      B.xpo >=> C.xpi;
      C.xpo >=> ipi;
      B.xpi <=< ipo;
}
```

Figure 6 (Left) Schema port connections diagram. Thin arrows indicate parent-child relations; thick arrows indicate port connections. (Right) The code for 'schemaA', from which process 'A' is instantiated

No connections are allowed internally to local external ports, where the possible connection combinations are either between two remote external ports, or between a local internal port and a remote external port (remote internal ports are externally invisible)[9]. Thus, no port connections are allowed from ports 'xpi' and 'xpo' in schema instance 'A' to any other ports in either 'A', 'B', or 'C' in the above schema network.

The connection p **>=>** q is specified by the transition

$$< \alpha, p \text{ >=> } q, \sigma, \iota, \kappa, \xi, \delta, \pi, sn > \otimes < \alpha, \varepsilon, \sigma, \iota, \kappa', \xi, \delta, \pi, sn >$$

where ports p and q are specified by $p = \alpha_1.p_1$, $q = \alpha_2.q_1$ and where α_1 and α_2 are schema instances. and the set of connections is updated to $\kappa' = \kappa \approx \{ [(\alpha_1, p_1), (\alpha_2, q_1)] \}$

While the above axiom defines the port inter-connections between, possibly, two children schema instances, the ports involved have to be notified where they are connected to; in particular the output port needs to know where to send its outgoing messages. This is achieved by the following inference rule which updates the state of the involved children instance ports,

$$\frac{\langle \alpha, p \text{ >=> } q, \sigma, \iota, \kappa, \xi, \delta, \pi, sn \rangle \xrightarrow{\varepsilon} \langle \alpha, \varepsilon, \sigma, \iota, \kappa', \xi, \delta, \pi, sn \rangle}{\{\langle \alpha_1, s_1, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn_1 \rangle, \langle \alpha_2, s_2, \sigma_2, \iota_2, \kappa_2, \xi_2, \delta_2, \pi_2, sn_2 \rangle\} \xrightarrow{\varepsilon} \{\langle \alpha_1, s_1, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1', sn_1 \rangle, \langle \alpha_2, s_2, \sigma_2, \iota_2, \kappa_2, \xi_2, \delta_2, \pi_2', sn_2 \rangle\}}$$

where for

$p = \alpha_1.p_1$

$q = \alpha_2.q_1$

and given that

[9] Connections between local internal ports are also allowed for consistency in the port model.

$\pi_1 = \{ \pi_{p1}, \ldots, \pi_{pn} \}$ and $\pi_2 = \{ \pi_{q1}, \ldots, \pi_{qm} \}$

then

$\pi_1' = \{ \pi_{p1}', \ldots, \pi_{pn} \}$ and $\pi_2' = \{ \pi_{q1}', \ldots, \pi_{qm} \}$

where

$\pi_{p1}' = <$ p_1, **output**, ε, ε, $\kappa\pi in_{p1}'$, $\kappa\pi out_{p1}'$, $\xi\pi in_{p1}$, $\xi\pi out_{p1}$ $>$

$\pi_{q1}' = <$ q_1, **input,** *buffer*, *bstate*, $\kappa\pi in_{q1}'$, $\kappa\pi out_{q1}'$, $\xi\pi in_{q1}$, $\xi\pi out_{q1}$ $>$

and

$\kappa\pi in_{p1}' = \kappa\pi in_{p1} \approx \{ (\alpha_2, q_1) \}$

$\kappa\pi out_{q1}' = \kappa\pi out_{q1} \approx \{ (\alpha_1, p_1) \}$

where ε corresponds to the empty state.

The following possible port connections are allowed:

- $\alpha_1, \alpha_2 \in \iota \in p_1, q_1 \in$ *XPort*, (remote to remote ports)
- $\alpha_1 = \alpha \in p_1 \in$ *IPort* $\in \alpha_2 \in \iota \in q_1 \in$ *XPort*, (local internal to remote ports)
- $\alpha_2 = \alpha \in q_1 \in$ *IPort* $\in \alpha_1 \in \iota \in p_1 \in$ *XPort*, (remote to local internal ports)
- $\alpha_1, \alpha_2 = \alpha \in p_1, q_1 \in$ *IPort*. (local internal to local internal ports)

(iv) Disconnection: Ports which have been previously connected may be disconnected by the command port1 **>=<** port2 where port1 and port2 are output and input ports, respectively.

$< \alpha, p$ **>=<** $q, \sigma, \iota, \kappa, \xi, \delta, \pi, sn > \otimes < \alpha, \varepsilon, \sigma, \iota, \kappa', \xi, \delta, \pi, sn >$

where ports p and q are specified by $p = \alpha_1.p_1$, $q = \alpha_2.q_1$, and where α_1 and α_2 are schema instances. The set of connections is updated to $\kappa' = \kappa - \{ [(\alpha_1, p_1), (\alpha_2, q_1)] \}$

Analogous to port inter-connections, an inference rule describes the change in state of the corresponding ports, as described by

$$\frac{\langle \alpha, p \succ\!\prec q, \sigma, \iota, \kappa, \xi, \delta, \pi, sn \rangle \xrightarrow{\varepsilon} \langle \alpha, \varepsilon, \sigma, \iota, \kappa', \xi, \delta, \pi, sn \rangle}{\{\langle \alpha_1, s_1, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn_1 \rangle, \langle \alpha_2, s_2, \sigma_2, \iota_2, \kappa_2, \xi_2, \delta_2, \pi_2, sn_2 \rangle\} \xrightarrow{\varepsilon} \{\langle \alpha_1, s_1, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1', sn_1 \rangle, \langle \alpha_2, s_2, \sigma_2, \iota_2, \kappa_2, \xi_2, \delta_2, \pi_2', sn_2 \rangle\}}$$

where for

$p = \alpha_1.p_1$

$q = \alpha_2.q_1$

and given that

$\pi_1 = \{ \pi_{p1}, \ldots, \pi_{pn} \}$ and $\pi_2 = \{ \pi_{q1}, \ldots, \pi_{qm} \}$

then

$\pi_1' = \{ \pi_{p1}', \ldots, \pi_{pn} \}$ and $\pi_2' = \{ \pi_{q1}', \ldots, \pi_{qm} \}$

where

$\pi_{p1}' = <$ p_1, **output**, ε, ε, $\kappa\pi in_{p1}'$, $\kappa\pi out_{p1}'$, $\xi\pi in_{p1}$, $\xi\pi out_{p1}$ $>$

$\pi_{q1}' = <$ q_1, **input**, *buffer*, *bstate*, $\kappa\pi in_{q1}'$, $\kappa\pi out_{q1}'$, $\xi\pi in_{q1}$, $\xi\pi out_{q1}$ $>$

and

$\kappa\pi in_{p1}' = \kappa\pi in_{p1} - \{ (\alpha_2, q_1) \}$

$\kappa\pi out_{q1}' = \kappa\pi out_{q1} - \{ (\alpha_1, p_1) \}$

where ε corresponds to the empty state.

(v) Relabeling: Port relabeling complements the functionality of port connections, and corresponds to local external ports referencing lower hierarchy ports of the same type, either input or output. Relabeling permits remote ports to receive and send messages with the external environment as if they were the actual local external ports in such an environment. This is specially important in defining delegation and composition in schemas. Relabelings and de-labelings are made in a dynamic fashion, analogous to connections and disconnections.[10] The intuition behind the relabeling notion is that those ports which are relabeled behave as relay ports forwarding their message according to the specified relabels. In general, a port is relabeled by p **===** q where p and q are both of the same type, either input or output ports.

No relabelings are allowed other than from local external ports to either remote external ports or to local internal ports, where all ports are of the same type. The transition is

$< \alpha, p \text{ === } q, \sigma, \iota, \kappa, \xi, \delta, \pi, sn > \otimes < \alpha, \varepsilon, \sigma, \iota, \kappa, \xi', \delta, \pi, sn >$

where ports p and q are specified by $p = \alpha_1.p_1$, $q = q_1$, and where α_1 is a schema instance., and the set of relabels is updated to $\xi' = \xi + \{ [(\alpha_1,p_1), q_1] \}$ where the port relabel is from q_1 to $\alpha_1.p_1$.

The change of state in π corresponds to a relabeled port having its forwarding address modified according to the relabeling specification. It should be noted that while a relabeled input port forwards its messages from a parent instance to a child instance, the direction of output port forwarding is exactly the opposite. The following inference rule will describe these considerations for changing the state of relabeled ports,

$$\frac{\langle \alpha, p === q, \sigma, \iota, \kappa, \xi, \delta, \pi, sn \rangle \xrightarrow{\varepsilon} \langle \alpha, \varepsilon, \sigma, \iota, \kappa, \xi', \delta, \pi', sn \rangle}{\{\langle \alpha_1, s_1, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn_1 \rangle\} \xrightarrow{\varepsilon} \{\langle \alpha_1, s_1, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1', sn_1 \rangle\}}$$

where for

$p = \alpha_1.p_1$

$q = q_1$ or $(\alpha.q_1)$

and given that

$\pi_1 = \{ \pi_{p1}, \ldots, \pi_{pn} \}$ and $\pi = \{ \pi_{q1}, \ldots, \pi_{qm} \}$

then

$\pi_1' = \{ \pi_{p1}', \ldots, \pi_{pn} \}$ and $\pi' = \{ \pi_{q1}', \ldots, \pi_{qm} \}$

where two cases may arise,

i) Relabeling is defined on output ports:

$\pi_{p1}' = < p_1,$ **output**, $\varepsilon, \varepsilon, \kappa\pi in_{p1}, \kappa\pi out_{p1}, \xi\pi in_{p1}, \xi\pi out_{p1}' >$

[10] Fan-in and fan-out allow a single local external port to reference many remote ports in a single environment.

$\pi_{q1}' = < q_1$, **output**, ε, ε, $\kappa\pi in_{q1}$, $\kappa\pi out_{q1}$, $\xi\pi in_{q1}'$, $\xi\pi out_{q1}$ >

and

$\xi\pi out_{p1}' = \xi\pi out_{p1} \approx \{ (\alpha, q_1) \}$

$\xi\pi in_{q1}' = \xi\pi in_{q1} \approx \{ (\alpha_1, p_1) \}$

Since message forwarding in relabeled output ports is from a child instance to a parent instance, the child instance port, in this case p_1, will include the port destination address to which its messages are to be sent, while the parent instance port, in this case q_1, will include the port destination address from which its messages are to be received.

ii) Relabeling is defined on input ports:

$\pi_{p1}' = < p_1$, **input**, *buffer, bstate*, $\kappa\pi in_{p1}$, $\kappa\pi out_{p1}$, $\xi\pi in_{p1}'$, $\xi\pi out_{p1}$ >

$\pi_{q1}' = < q_1$, **input**, *buffer, bstate*, $\kappa\pi in_{q1}$, $\kappa\pi out_{q1}$, $\xi\pi in_{q1}$, $\xi\pi out_{q1}'$ >

and

$\xi\pi in_{p1}' = \xi\pi in_{p1} \approx \{ (\alpha, q_1) \}$

$\xi\pi out_{q1}' = \xi\pi out_{q1} \approx \{ (\alpha_1, p_1) \}$

Since message forwarding in relabeled input ports is from a parent instance to a child instance, the parent instance port, in this case q_1, will include the port destination address to which its messages are to be sent, while the parent instance port, in this case p_1, will include the port destination address from which its messages are to be received.

The following possible port relabels are allowed:

- $\alpha_1 \in \iota \in p_1,q_1 \in$ *XPort*, (remote to local external ports)
- $\alpha_1 = \alpha \in p_1 \in$ *IPort* $\in q_1 \in$ *XPort,* (local internal to local external ports).

(vi) Delabeling: Ports which have been previously relabeled may be de-labeled by the command p **=|=** q where p and q are both either input or output ports. The transition is

$< \alpha, p$ **=|=** $q, \sigma, \iota, \kappa, \xi, \delta, \pi, sn >$ $\Rightarrow$ $< \alpha, \varepsilon, \sigma, \iota, \kappa, \xi', \delta, \pi, sn >$

where ports p and q are specified by $p = \alpha_1.p_1$, $q = q_1$, and where α_1 is a schema instance. The set of relabels is updated to $\xi' = \xi - \{ [(\alpha_1,p_1), q_1] \}$.

Analogous to port relabels, an inference rule describes the change in state of the corresponding ports, as described by

$$\frac{\langle \alpha, p =|= q, \sigma, \iota, \kappa, \xi, \delta, \pi, sn \rangle \xrightarrow{\varepsilon} \langle \alpha, \varepsilon, \sigma, \iota, \kappa, \xi', \delta, \pi', sn \rangle}{\{\langle \alpha_1, s_1, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn_1 \rangle\} \xrightarrow{\varepsilon} \{\langle \alpha_1, s_1, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1', sn_1 \rangle\}}$$

where for

$p = \alpha_1.p_1$

$q = q_1$ or $(\alpha.q_1)$

and given that

$\pi_1 = \{ \pi_{p1}, \ldots, \pi_{pn} \}$ and $\pi = \{ \pi_{q1}, \ldots, \pi_{qm} \}$

then

$\pi_1' = \{ \pi_{p1}', \ldots, \pi_{pn} \}$ and $\pi' = \{ \pi_{q1}', \ldots, \pi_{qm} \}$

where the following two cases may arise.

The case where relabeling is defined on output ports:

$\pi_{p1}' = < p_1$, **output**, ε, ε, $\kappa\pi in_{p1}$, $\kappa\pi out_{p1}$, $\xi\pi in_{p1}$, $\xi\pi out_{p1}' >$

$\pi_{q1}' = < q_1$, **output**, ε, ε, $\kappa\pi in_{q1}$, $\kappa\pi out_{q1}$, $\xi\pi in_{q1}'$, $\xi\pi out_{q1} >$

and the respective references have to be deleted,

$\xi\pi out_{p1}' = \xi\pi out_{p1} - \{ (\alpha, q_1) \}$

$\xi\pi in_{q1}' = \xi\pi in_{q1} - \{ (\alpha_1, p_1) \}$

The case where relabeling is defined on input ports:

$\pi_{p1}' = < p_1$, **input**, *buf, bstate*, $\kappa\pi out_{p1}$, $\xi\pi in_{p1}'$, $\xi\pi out_{p1} >$

$\pi_{q1}' = < q_1$, **input**, *buf, bstate*, $\kappa\pi in_{q1}$, $\kappa\pi out_{q1}$, $\xi\pi in_{q1}$, $\xi\pi out_{q1}' >$

and the respective references have to be deleted,

$\xi\pi in_{p1}' = \xi\pi in_{p1} - \{ (\alpha, q_1) \}$

$\xi\pi out_{q1}' = \xi\pi out_{q1} - \{ (\alpha_1, p_1) \}$

Communication

Port Writing

Writing data is allowed only through local ports, either internal or external. Remote ports may not be directly accessed for writing, only indirectly via port connections. The syntax of writing data is given by an expression which returns 'true' when writing has been successful, and 'false' otherwise. There is no blocking on sending messages out (based on the unbounded buffer assumption) — independently of whether the receiver gets the message or a connection exists, the body of the SI will continue execution.[11] The writing command takes the form

output_port **!** expr

where output_port is the name of a local output port, and expr is any value-returning expression. The message may be of any type, including primitive types, char, int, float, or derived types[12].

[11] Since writing to an output port which is not connected will cause the data to be lost, the programmer should check for the existence of connections. ASL includes special constructs, which are not part of the basic language, for this purpose - see [27] for details.

[12] New types may be derived from basic ones, including derived ports which can understand new derived types. The basic port structure only supports basic types, integers, floats, and characters. Ports and schema as port messages are not currently supported by the basic language. Writing also supports multiple messages sent as one block through a single port. Remote procedure calls are simulated by transforming the actual function call into a message list corresponding to the function name and the sequence of the function's arguments, but we will not go into the details here.

Communication is achieved by sending messages to local output ports, where transitions include a pending communication labeled in general by $p!x$, where x represent any message. In particular for an expression e written through port p,

$$\langle \alpha, p!e, \sigma, \iota, \kappa, \xi, \delta, \pi, sn \rangle \xrightarrow{p!\sigma(e)} \langle \alpha, \varepsilon, \sigma, \iota, \kappa, \xi, \delta, \pi, sn \rangle$$

where $p!\sigma(e)$ specifies the value returned by expression e, according to the local store σ, to be sent through port p. There is no blocking on sending messages out (based on the unbounded buffer assumption).

The actual communication will only take place if there exists a connection or relabel on the corresponding port. The next inference rule corresponds to the actual sending of a message between two inter-connected process α_1 and α_2, according to the reference list stored in π_1, referencing a port in α_2. This rule is applied to all port references in π_1 so the same message is sent to all destinations.

$$\frac{\langle \alpha_1, p!e, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn_1 \rangle \xrightarrow{p!\sigma(e)} \langle \alpha_1, \varepsilon, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn_1 \rangle \wedge (\alpha_2, q) \in \pi_1.\kappa\pi out_p \wedge (\alpha_1, p) \in \pi_2.\kappa\pi in_q}{\{\langle \alpha_1, p!e, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn_1 \rangle, \langle \alpha_2, s_2, \sigma_2, \iota_2, \kappa_2, \xi_2, \delta_2, \pi_2, sn_2 \rangle\} \xrightarrow{p!\sigma(e)} \{\langle \alpha_1, \varepsilon, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn_1 \rangle, \langle \alpha_2, s_2, \sigma_2, \iota_2, \kappa_2, \xi_2, \delta_2, \pi_2', sn_2 \rangle\}}$$

where

$\pi_1 = \{ \pi_p, \ldots, \pi_{pn} \}$

$\pi_p = <$ p, output, n/a, n/a, $\kappa\pi in_p$, $\kappa\pi out_p$, $\xi\pi in_p$, $\xi\pi out_p$ $>$

$(\alpha_2, q) \in \kappa\pi out_p$

and

$\pi_2' = \{ \pi_q', \ldots, \pi_{qn} \}$

$\pi_q' = <$ q, input, buf', $bstate'$, $\kappa\pi in_q$, $\kappa\pi out_q$, $\xi\pi in_q$, $\xi\pi out_q$ $>$

$(\alpha_1, p) \in \kappa\pi in_q$

the actual communication will take place by changing the state of port q in α_2 according to

$buf' = buf \blacklozenge \sigma(e)$ storing the message in the local buffer,

$bstate'$ = **true** since a message has been received.

In case where $\xi\pi out$ is not empty then relabelings exist for q and the message has to be forwarded on according to the following relabeling rules.

When considering inter-process communication through port relabeling the following rule is used,

$$\frac{\langle \alpha_1, p!e, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn_1 \rangle \xrightarrow{p!\sigma(e)} \langle \alpha_1, \varepsilon, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn_1 \rangle \wedge (\alpha, q) \in \pi_1.\xi\pi out_p \wedge (\alpha_1, p) \in \pi.\xi\pi in_q}{\{\langle \alpha_1, p!e, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn_1 \rangle, \langle \alpha, s, \sigma, \iota, \kappa, \xi, \delta, \pi, sn \rangle\} \xrightarrow{q!\sigma(e)} \{\langle \alpha_1, \varepsilon, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn_1 \rangle, \langle \alpha, s, \sigma, \iota, \kappa, \xi, \delta, \pi, sn \rangle\}}$$

where α_1 is the child of α, and it has its port p relabeled to q in α.

Furthermore,

$\pi_1 = \{ \pi_p, \ldots, \pi_{pn} \}$

$\pi_p = <$ p, **output**, ε, ε, $\kappa\pi in_p$, $\kappa\pi out_p$, $\xi\pi in_p$, $\xi\pi out_p$ $>$

$(\alpha_2, q) \in \xi\pi out_p$

and

$\pi = \{ \pi_q, \ldots, \pi_{qn} \}$

π_q = < *q*, **output**, ε, ε, $\kappa\pi in_q$, $\kappa\pi out_q$, $\xi\pi in_q$, $\xi\pi out_q$ >

(α_1,*p*) □ $\xi\pi in_q$

the actual communication will take place by port *q* in α relaying the message according to its own list of connections or further relabels.

The two rules, the one for port inter-connections and the output port relabeling rule, together define the message sending communication path. In particular, the relabeling rule may be recursively applied to a single message when involving relabels of relabels.

Port Reading

In ASL, receiving occurs independently from sending, in contrast to synchronous communication. The important aspect when contrasting reading with writing, is that, due to the asynchronous nature of the communication, sending a message involves most of the work of actually delivering a message to its destination, while on the other hand reading is simply done by retrieving the message from a local buffer without having to know its source. The read command is

p **?** *var*

where *p* is the name of a local input port, and *var* is the variable where the message received is to be stored. All write command are *non-blocking*, while read commands *block* until there is something to be read from the queue. Thus, the only requirement is for port *p* having *bstate* = **true** in π. In case *bstate* = **false**, the configuration enters a waiting state, where no reading will take place until bstate becomes **true**.

if $\pi.bstate_p$ = **true** then

$$\langle \alpha, p?v, \sigma, \iota, \kappa, \xi, \delta, \pi, sn \rangle \xrightarrow{p?v} \langle \alpha, \varepsilon, \sigma', \iota, \kappa, \xi, \delta, \pi, sn \rangle$$

else

$$\langle \alpha, p?v, \sigma, \iota, \kappa, \xi, \delta, \pi, sn \rangle \xrightarrow{\varepsilon} \langle \alpha, p?v, \sigma, \iota, \kappa, \xi, \delta, \pi, sn \rangle$$

p?*v* is the communication label and specifies the reception of a message into variable *v* through port *p*.

After reading the message, the state of the store (buffer) is updated with the new value for *v*,

σ' = < σ_1 [*p.buf* ∅ *v*], σ_2 >

where *p.buf* ∅ *v* represents the reading of a buffer message from *p* into *v*.

When considering inter-process communication through port relabeling the following rule is used,

$$\frac{\langle \alpha_1, p?v, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn \rangle \xrightarrow{p?v} \langle \alpha_1, \varepsilon, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1', sn \rangle \wedge (\alpha, q) \in \pi_1.\xi\pi out_p \wedge (\alpha_1, p) \in \pi.\xi\pi in_q}{\{\langle \alpha_1, p?v, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1, sn \rangle, \langle \alpha, s, \sigma, \iota, \kappa, \xi, \delta, \pi, sn \rangle\} \xrightarrow{q?v} \{\langle \alpha_1, \varepsilon, \sigma_1, \iota_1, \kappa_1, \xi_1, \delta_1, \pi_1', sn \rangle, \langle \alpha, s, \sigma, \iota, \kappa, \xi, \delta, \pi, sn \rangle\}}$$

where α_1 is the child of α, and is the process having its port *p* relabeled to *q* in α.

Furthermore,

π_1' = { π_p', ... , π_{pn} }

π_p' = < *p*, **input**, *buf'*, *bstate'*, $\kappa\pi in_p$, $\kappa\pi out_p$, $\xi\pi in_p$, $\xi\pi out_p$ >

(α_2,*q*) □ $\xi\pi out_p$

buf = *buf* ♦ $\sigma(e)$ storing the message in the local buffer,

bstate = **true** since a message has been received.

and

$\pi = \{ \pi_q, \ldots, \pi_{qn} \}$,

$\pi_q = < q,$ **input**, *buf'*, *bstate*, $\kappa\pi in_q$, $\kappa\pi out_q$, $\xi\pi in_q$, $\xi\pi out_q >$,

$(\alpha_1, p) \square \xi\pi in_q$

further communication will take place if port *p* includes its own relabels.

The above rule defines the message receiving communication path which may be recursively applied to a single message when involving relabels of relabels.

Delegation

The delegation scheme is defined by an axiom describing the semantics of the **delegate** expression

$< \alpha,$ **delegate** *si*, σ, ι, κ, ξ, δ, π, *sn* $> \xrightarrow{\xi} <\alpha, \varepsilon, \sigma, \iota, \kappa, \xi, \delta', \pi, sn >$

where the set of delegated schema instances is updated with $\delta' = \delta \approx \{ si \}$.

Assemblages

A key concept evolving from the RS schema model [18] is that of schema composition in the form of *schema assemblages*, enabling the building of complex hierarchical systems in an encapsulated fashion, with the following characteristics

- an schema assemblage instantiate instantiates more than one internal schema instance, and
- internal schema instance ports are connected to other internal schema instance ports, or
- schema assemblage instantiate ports are relabeled to some of the internal schema instances ports.

The schema assemblage in RS is a special construct described by

[*N*(*ip*)(*op*)(*v*)(*s*)(*ib*)(*p*)(n)]

where

- *N* is the assemblage name
- *ip* is the list of input ports
- *op* is the list of output ports
- *v* is the list of variables
- *s* is the list of component schemas
- *ib* describes the component schema instantiations
- *p* describes the port relabeling
- *n* defines the internal port connections

Contrary to assemblages in RS, the schema assemblage abstraction in ASL is built from other basic ASL constructs, such as relabeling, connections and delegation, and not as a special class of schema assemblage classes. A general schema assemblage composed of two subschemas is shown in Figure 7, we can see the following equivalence

- the schema name *N* corresponds to *schemaA*
- the list of input ports *ip* corresponds to *xpi,ipi*
- the list of output ports *op* corresponds to *xpo,opo*
- the list of variables *v* is in this case empty
- the list of component schemas *s* corresponds to *schemaB,schemaC*
- the component schema instantiations *ib* corresponds to *schemaB,schemaC*
- the port relabeling *p* corresponds to *xpi*♦*schemaB*(*xpi*), *xpo*♦*schemaC*(*xpo*)
- the internal port connections *n* corresponds to *schemaC*(*xpi*)♦*schemaB*(*xpo*)

Note that since no variables are included as part of the schema definitions, *ib* looks exactly the same as *s*. The ASL schema delegation is implicit in the RS schema assemblage construct, while the body section is absent in the RS schema assemblage. This aspect contrasts the extended flexibility of the ASL's assemblage notion, while keeping homogeneity with a single general schema definition type.

```
// schema assemblage
schema schemaA
{
external:
      input xpi;        // external input port
      output xpo; // external output port
internal:
      input ipi;        // internal input port
      output ipo; // internal output port
      schemaB B;        // schema instance reference
      schemaC C;        // schema instance reference
body:
      B.xpo >=> C.xpi;
      C.xpo === xpo;
      B.xpi === xpi;
      delegate B,C;
}
```

Figure 7. Assemblage example code

Comparisons

The following sections contrast the ASL model most important characteristics to other models, including other concurrent object-oriented systems.[13]

[13] For a more extensive comparison refer to [27].

Instantiation

In concurrent object-oriented systems, there are basically two different paradigms for object creation, *class*-based and *prototype*-based. The concept of classes, basic in sequential object-oriented systems, such as Smalltalk [10], defines a special class *template* in creating class objects. In some concurrent object-oriented (object-based) models, such as actors, object creation is through *prototyping*, where an object makes a copy of itself in creating a new object.[14]

Multiple Ports

Multiple ports have been utilized in such computational models as CSP [12] and Port Automata [24]. Yet most concurrent object-oriented models follow a single port model, in particular the *actor* model [1]. (Some models based on Concurrent Logic Programming are also based on multiple ports, such as Vulcan [16].)

Contrasting ASL to languages derived from CSP, we have Ada [14], having synchronous communication and multiple ports, where ports define *entry* queues in remote procedure calls ('entry-per-procedure'), and data paths are set through direct naming. On the other hand Occam [15] is based on communication channels, supports point-to-point synchronous communication, yet, not allowing multiple inter-connections, i.e. fan-in nor fan-out. Some concurrent object-oriented languages, such as POOL [2], incorporate synchronous communication and remote procedure calls similar to Ada.

In contrast to single port models, where communication is asynchronous, such as actors, the multiple port paradigm avoids the need to search through single input queues when looking for a particular type of message. This allows to avoid special communication modes, such as the *express* mode in ABCL [30] in addition to the *ordinary* communication mode), and the special *reply port*, in addition to the regular *message port*, which are mainly designed to compensate for the restrictions of single port models.

Message Passing

As previously described, communication in ASL is asynchronous. Messages may be received through any input port and sent through any output port. Messages in ASL could stand for method invocation, in the way of message patterns activating scripts or as simple data values. The notion of message patterns and scripts is somewhat similar to that used in ABCL, where method arguments are passed as separate message entries in the pattern, and the script is activated when a message pattern is matched. (It is important to note that since messages may be sent and received through different ports, reading and writing in ASL is explicitly managed as opposed to other models, particularly those following the

[14] Refer to [7] for an anlysis their shortcomins in the context of inheritance.

client/server model, where remote procedure calls are implicitly serviced.) In the asynchronous communication paradigm, a schema sending a message doesn't have to wait for an acknowledgment or for the actual reception and servicing of the message. Yet, synchronous communication is possible with the help of a 'wait-for-reply' mechanism, similar to Ada's *rendezvous*. The general asynchronous communication paradigm also permits the *past*, *now*, and *future* modes of communication in ABCL. The paradigm supports multi-party communication, where a single output port may send messages to many other schema's input ports, and similarly many input ports may receive messages from a single output port.

Aggregation

Basic schemas may be composed together into schema *assemblages* in building complex systems. In contrast to ASL, in the actor model, this composition notion corresponds to *configurations* where *receptionist* actors and *external* actors are integrated together with 'regular' actors; yet contrary to assemblages, which are themselves schemas, an actor configuration is not considered a 'first-class' actor. This is partially due to the fact that schemas are multiple port entities while actors are single port abstractions. Moreover, when contrasting aggregation in both models, receptionist actors could correspond to assemblage input ports while external actors could correspond to assemblage output ports, whereas if we consider a basic schema as an actor configuration, then schema assemblages would correspond to configurations of configurations, which points out to the higher level abstraction and the multi-granularity of the schema model.

Multi-granularity

When contrasting schemas with actors we have distinguished the difference in granularities between the two models. Yet the schema model also supports fine-grained object models, such as neurons in neural networks systems [26]. This is similar to *domains* in Hybrid [21], which may be of different granularity to match hardware processing characteristics. A schema system may also be designed to match the particular machine environment, from coarse-grain to fine-grain schemas.

Implementation

The language has been implemented in C++ under Unix[15], where concurrency is simulated by using the Unix Light Weight Process (LWP) library[16]. The parsing of the language has been developed using Lex

[15] Unix is a registered name.
[16] LWP is a registered product.

and Yacc[17]. We are currently developing a truly distributed implementation where a number of workstations are networked, and schema processes can be mapped to Unix processes.

Conclusions

The ASL modeling methodology has been applied as a domain specific schema language to neural networks simulation, giving rise to the Neural Schema Language (NSL)[18], a system for the describing modular neural networks. Figure 8 shows the basic neural model, where neural networks correspond to schemas, and networks of neural networks correspond to schema assemblages. NSL exploits the notions of delegation and wrapping, by enabling a neural schema to recruit any number of neural networks for its implementation. Similarly a single neural network may be recruited by different schemas. Such an approach enables the encapsulation of neural networks into schema classes and the composition of hierarchical networks. Furthermore, at a lower level neurons may have their task delegated by neural implementations of different levels of detail, from the very simple neuron models to the very complex ones [26]. (It is interesting to note, that the neuron model is best modeled also as a multi-port entity.)

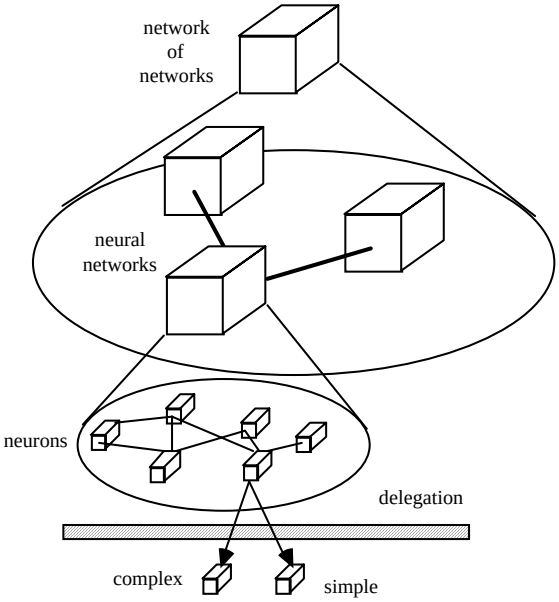


Figure 8. NSL model

Future Research

This paper has presented the Abstract Schema Language (ASL) computational model and its main characteristics, hierarchy, composition, heterogeneity and multi-granularity. ASL notion of schemas, assemblages, wrapping, and delegation extend the current state of concurrent object-oriented programming.

[17] Both Lex and Yacc are registered products.

[18] Not to confuse with the Neural Simulation Language. Basically, both systems merge under the upcoming NSL3.0 system at the end of 1993.

ASL is part of on-going research in the development of schema systems. In terms of ASL as a language, current research involves the incorporation of typing, in the form schema signatures. Other issues yet to be fully analyzed, include aspects arising from asynchrony and non-determinism present in truly parallel systems. Furthermore, there is the issue of how to deal with inheritance [7]. In parallel, research is under way in extending the theoretical work in defining an asynchronous model for ASL in particular, and COOP in general [20][13].

In terms of implementation, ASL has been prototyped on a multi-processing system, and current thrust is in its distributed, parallel, and heterogeneous implementation. ASL is a machine independent language, which translates into other high level languages. In particular, C++ [25] is currently both the underlying prototyping language for interpretation and system implementation.

An application of ASL, as previously discussed, is the development of the domain specific schema language for neural networks simulation, Neural Schema Language (NSL), based on previous work with the Neural Simulation Language [28]). An interesting concept in the integration of ASL as a computational model, and NSL as a simulation system is that of wrapping [5] where existing code may be interfaced to the ASL model without requiring a major rewriting of the system. The goal is the development of complex distributed applications in the areas of Brain Theory and Distributed Artificial Intelligence (DAI), where schemas may integrate with hardware processing. These developments integrate with current work in defining a common ground between COOP and DAI [6].[19]

Work is also under way in extending the basic schema model in two different directions. One thrust is in the extension of the model into the real-time domain, for applications in robotics and vision. The other thrust is the incorporation of learning capabilities into the schema model, through the introduction of *computational reflection* [19].

Appendix

Schema Definition

```
sd ::= schema sn (xp-decl)opt
       {
            external: xv-decl
            internal: iv-decl
            body:     s
       }
```

[19] For further discussions on the Neural Schema Language refer to [27].

Declarations

```
// schema types
Sdecl ::=          schema      // generic schema ref.
            |      sn          // specific schema ref.

SRdecl ::=         Sdecl*      // schema reference ptr
```

```
// port types
Pdecl ::=          input // input port reference
            |      output      // output port reference

PRdecl ::=         Pdecl*      // port reference pointer
```

```
// schema/port types
SPdecl ::=         Sdecl       // schema
            |      Pdecl       // port

// schema/port ptr types
SPRdecl ::=        SRdecl      // schema ptr
            |      PRdecl      // port ptr
```

```
// primitive (variable) types
Vdecl ::=          int         // integer
            |      float // float
            |      char        // character
            |      const // constant

VRdecl ::=         Vdecl*      // primitive type ptr
```

```
// instantiation parameter:
xp-decl ::=        ε                        // empty
            |      Vdecl id1,...,idn
            |      xp-decl1; xp-decl2       // sequence
```

```
// external declaration:
xv-decl ::=        ε                        // empty
            |      SPdecl      id1,...,idn // schema/port
            |      SPRdecl id1,...,idn     // schema/port ptr
            |      xv-decl1; xv-decl2      // sequence
```

```
// internal declaration:
iv-decl ::=        ε                        // empty
            |      SPdecl      id1,...,idn // schema/port
            |      SPRdecl id1,...,idn     // schema/port ptr
            |      Vdecl id1,...,idn       // primitive
            |      VRdecl      id1,...,idn // primitive ptr
            |      iv-decl1; iv-decl2      // sequence
```

```
// all variable declaration:
sv-decl ::=       xp-decl     // inst. parameter
            |     xv-decl     // external
            |     iv-decl     // internal
```

Expressions

```
e::=        v                       // variable
      |     new sn                  // dynamic schema inst.
      |     self                    // self reference
      |     p?v1,..., vn            // message reception
      |     p!e1,..., en            // message delivery
      |     p >=> q1,..., qn        // connect ports
      |     p <=< q1,..., qn        // alternative syntax
      |     p >=< q1,..., qn        // disconnect ports
      |     p === q1,..., qn        // relabeling ports
      |     p =|= q1,..., qn        // de-labeling
      |     stop si1,..., sin // de-instantiate schema
      |     delegate si1,..., sin   // delegation
      |     f(e1,..., en)           // function call
```

Statements

```
s ::=       ε                       // empty statement
      |     e                       // expr. as statement
      |     v = e                   // assignment
      |     if (e) then { s1 } else { s2 } // if-else
      |     while (e) { s }   // while-loop
      |     s1; s2                  // sequential composition
```

References

1. Agha, G., *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, *1986*.
2. America, P., POOL-T: A Parallel Object-Oriented Language, *Object-Oriented Concurrent Programming*, edited by A. Yonezawa and M. Tokoro, MIT Press, 1987.
3. Arbib, M.A., Schema Theory, In the *Encyclopedia of Artificial Intelligence*, 2nd. Edition, edited by Stuart Shapiro, 2:1427-1443, Wiley, 1992.
4. Baldamus, M., Formal Semantics of Robot Scheme Programs, Technical Report, Institute of Software and Theoretic Information, Technische Universitat Berlin, Germany, 1991.
5. Bellman, K.L., Gillam, A., Achieving Openness and Flexibility in VEHICLES, In *AI and Simulation*, Edited by W. Webster and R. Uttansingh, Simulation Series, 22(3), Society for Computer Simulation, 1990.
6. Briot, J.-P., Gasser, L., From Objects to Agents: Connections between Object-Based Concurrent Programming and Distributed Artificial Intelligence, *IJCAI '91 Workshop on Objects and AI*, 1991.

7. Briot, J.-P., Yonezawa, A., Inheritance and Synchronization in Object-Oriented Concurrent Programming, *ABCL: An Object-Oriented Concurrent System*, edited by A. Yonezawa, MIT Press, 1990.
8. Cointe, P., Implementation et Interpretation des Langages Objets, Application aux Langages Formes, ObjVlisp et Smalltalk, (these d'Etat), LITP Research Report, No. 85-55, LITP -Iniversite Paris-Vi - IRCAM, Paris, 1984.
9. Draper, B., Collins, R., Brolio, J., Hanson, A., Riseman, E., The Schema System, *Int. Journal of Computer Vision*, 2:209-250, 1989.
10. Goldberg, A., Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison Wesley, 1984.
11. Henessy, M., Plotkin, G.D., Full Abstraction for a Simple Parallel Programming Language, Proc. 8th MFCS, LNCS 74:108-120, Springer, 1979.
12. Hoare, C.A.R., Communicating Sequential Processes, *Communications of the ACM* Vol. 21 No. 8, pp 666-677, August, 1978.
13. Honda, K., Tokoro, M., An Object Calculus for Asynchronous Communication, *Proc. ECOOP '91*, Geneve, Switzerland, 1991.
14. Ichbiah, J., *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, 1983.
15. INMOS, *Occam Programming Manual*, London, Prentice-Hall, 1984.
16. Kahn, K., Tribble, E., Miller, M., Bobrow, D., *Research Directions in Object-Oriented Programming: Functions, Relations and Equations*, chapter Vulcan: Logical Concurrent Objects, pp. 75-112, MIT Press, 1987.
17. Lieberman, H., Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, *OOPSLA '86, Conference Proceedings*, 1986.
18. Lyons, D.M., Arbib, M.A., A Formal Model of Computation for Sensory-Based Robotics, *IEEE Trans. on Robotics and Automation*, 5:280-293, June, 1989.
19. Maes, P., Concepts and Experiments in Computational Reflection, *Proc. OOPSLA '87*, :147-155, Orlando, FL, Oct. 4-8, 1987.
20. Milner, R., Functions as Processes, In Automata, Language, and Programming, *LNCS* 443:167-180, Springer-Verlag, 1990.
21. Nierstrasz, O., 1987, Active Objects in Hybrid, *OOPSLA '87*, Conference Proceedings.
22. Plotkin, G.D., A Structural Approach to Operational Semantics, DAIMI FN-19, Computer Science Dept, Aarhus University, Aarhus, Denmark, Sept, 1981.
23. Plotkin, G.D., An Operational Semantics for CSP, *Formal Description of Programming Concepts II*, D. Bjørner Editor, North-Holland, 1983.
24. Steenstrup, M., Arbib, M.A., Manes, E.G., Port Automata and the Algebra of Concurrent Processes, *J. Computer Syst. Sci.*, Vol. 27, no. 1, pp. 29-50, Aug, 1983.
25. Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1987.

26. Weitzenfeld, A., Arbib, M., A Concurrent Object-Oriented Framework for the Simulation of Neural Networks, *Proceedings of ECOOP/OOPSLA '90 Workshop on Object-Based Concurrent Programming*, OOPS Messenger, 2(2):120-124, April, 1991.
27. Weitzenfeld, A., *A Unified Computational Model for Schemas and Neural Networks in Concurrent Object-Oriented Programming*, PhD Thesis, Center for Neural Engineering, University of Southern California, Los Angeles, CA, 1992.
28. Weitzenfeld, A., Arbib, M.A., NSL: Neural Simulation Language, *Neural Network Simulation Environments*, Ed. J. Skrzypek, Kluwer, 1993 (in press).
29. Weitzenfeld, A., An Overview of ASL: Hierarchy, Composition, Heterogeneity, and Multi-Granularity in Concurrent Object-Oriented Programming, *Proceedings of OOPSLA '92 Workshop on Next Generation Computing*, Vancouver, Canada, 1993 (in press).
30. Yonezawa, A., Briot, J-P., Shibayama, E., Object-Oriented Concurrent Programming in ABCL/1, *OOPSLA '86*, Conference Proceedings, 1986.
31. Yonezawa, A., Tokoro, M., Eds., *Object-oriented concurrent programming*, MIT Press, 1987.