ASL
A Hierarchical Computational Model for Distributed Heterogeneous Systems

Alfredo Weitzenfeld

Technical Report CNE-93-02, CS-93-552

May 1993

Brain Simulation Laboratory
Center for Neural Engineering
University of Southern California

ASL: A Hierarchical Computational Model for Distributed Heterogeneous Systems

Alfredo Weitzenfeld
Computer Science Department
University of Southern California
Los Angeles, CA 90089-2520
alfredo@usc.edu
tel: 213/740-6345

Abstract

The Abstract Schema Language (ASL) defines a hierarchical computational model for the development of distributed heterogeneous systems. ASL extends the capabilities and methodologies of concurrent object-oriented programming to enable the construction of highly complex multi-granular systems. The ASL model is described in terms of *schemas* (concurrent agents), supporting 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.

Keywords

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

Introduction

The Abstract Schema Language (ASL) [Weitzenfeld 1992; 1993] describes an evolved 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 [Arbib 1992] with concurrent object-oriented programming (COOP) [Yonezawa and Tokoro 1987]. 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 [Cointe 1984]. In a concurrent world, some of these concepts become more complex, especially when designing inheritance schemes [Briot and Yonezawa 1990], where as an alternative to inheritance, the notion of delegation [Lieberman 1986] has been suggested. ASL extends the current state of the art in both schema research and COOP while providing a hierarchical approach towards 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 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 inter-connections avoids the overhead of direct process naming between continuously communicating entities. Yet, ASL communication model is expressive enough, making it possible to simulate other communication paradigms, such as client/server and blackboards (see Weitzenfeld [1992]).

The integration of ASL with the Neural Simulation Language (NSL) system [Weitzenfeld 1991], a simulation system extensibly used by the neural networks research community, gives rise to Neural-Schema Language (Weitzenfeld [1992] (also referred to as NSL), a comprehensive simulation system for applications in DAI , Robotics and Brain Theory.[1]

Schemas

The ASL computational model is defined in terms of *schemas*[2], 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 are 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 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.)

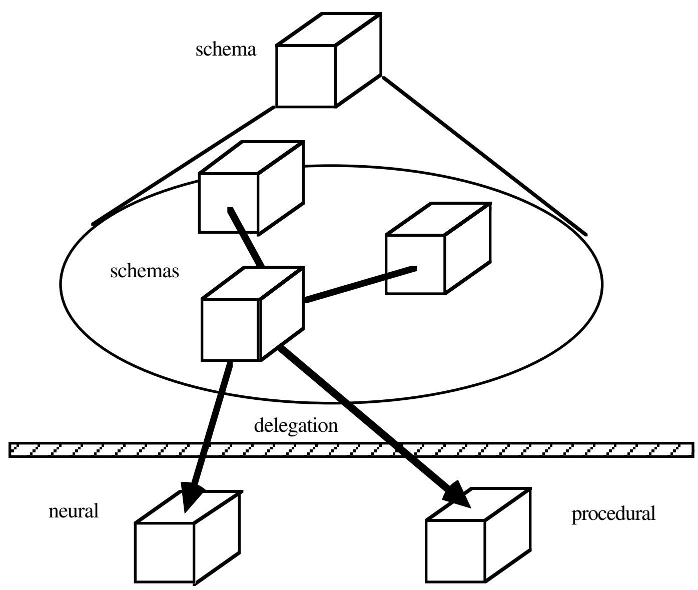

Figure 1. ASL schema model

[1] The ASL operational semantics are described in Weitzenfeld and Arbib [1993], and a multi-process implementation can be found in Weitzenfeld [1992].

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

Example

Let us introduce the ASL model and its special characteristics through a domain example. Consider a robotics system, as shown in Figure 2, having a vision component which continually processes external images. The robot will perform appropriate actions according to the particular input scenes, e.g. object avoidance or object grasping.

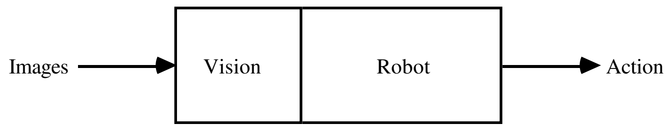

Figure 2. Robotics Example

There are several aspects in such a robotics system, which may be generalized to other domains. In particular, let us highlight the following concepts (using ASL terminology):

- **Hierarchy**: System components may be divided into sub-components giving rise to top-down design methodology, where the main problem is partitioned into smaller sub-problems; and bottom-up design where sub-components are developed first and then put together into more complex systems.
- **Assemblages**: A set of sub-components may be treated as a single component.
- **Heterogeneity**: Particular schema tasks may be implemented by different programming paradigms, e.g., procedural and neural networks programs.
- **Wrapping**: Independently developed heterogeneous programs may be integrated under a common schema interface.
- **Encapsulation**: Schemas specify the basic means for program encapsulation in ASL.
- **Reusability**: Schemas provide the basis for component reusability.
- **Task Delegation**: Schema tasks may choose their implementation in a dynamic way.
- **Distributed**: The system should enable distributed processing of components.
- **Concurrency**: The system should enable concurrent processing of components.
- **Communication**: Components performing intensive computations which require continuous message passing between corresponding processes.
- **Multi-granularity**: Processes should be able to efficiently map into a multi-granular processing environment.

Continuing the example exposition, and considering that either the area of robotics, or even that of vision, are extremely complex, a very simplified example of a vision sub-problem will be presented here, that of stereopsis, in order to illustrate ASL. The system's objective is to compute object depths from available stereo information, and can most basically be represented by two components, a retina, performing pre-processing on external images, and a depth component obtaining depth information from general stereo cues.

Referring to the complete system as 'STEREO', and its two sub-components as a 'RETINA' schema, and a 'DEPTH' schema, a diagram of the functional system is shown in Figure 3. The diagram includes internal data path between the two sub-schemas, as well as external ones between the two sub-schemas and the outside world.

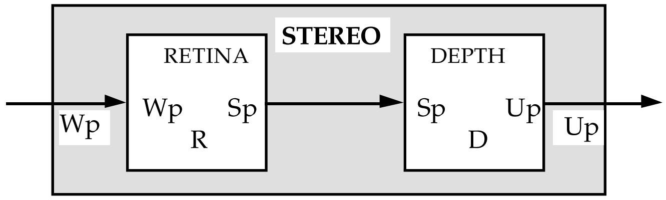

Figure 3. The basic stereo model.

RETINA Schema

Before giving the formal definition of a schema, consider the 'RETINA' schema as described in Figure 4 consisting of a header and a body. The header includes the schema name, 'RETINA', an external section containing an input port 'Wp' and an output port 'Sp', and an internal section containing two layers, 'W' and 'S' (layers are basically arrays of numbers). The body contains the schema code describing the schema behavior, in this case, an endless loop describing the continuous reading (Wp ? W) of external image data, the processing of this data (retina_procedure), and the output of the resulting data (Sp ! S)[3]. The 'RETINA' schema code, as well as other schema code presented in this paper is simplified for exposition purposes. The function 'retina_procedure' is called for the actual schema task computing.

```
schema RETINA
{
external:
        input   Wp;
        output  Sp;
internal:
        layer   W,S;
body:
        while (true) {
                Wp ? W;
                retina_procedure(W,S);
                Sp ! S;
        }
}
```

Figure 4. RETINA Schema

[3] The notation for reading and writing is similar to that of CSP [Hoare 1978].

DEPTH Schema

The 'DEPTH' schema is shown in Figure 5. Analogous to the 'RETINA' schema, the header includes an external section which consists of the two ports 'Sp' and 'Up', for input and output, respectively, and an internal section which consists of two layers, 'S' and 'U'. The body consists of an endless loop describing the continuous reading of data (Sp ? S), the processing of this data (depth_procedure), and the output of the resulting depth maps (Up ! U).

```
schema DEPTH
{
external:
        input    Sp;
        output   Up;
internal:
        layer    U,S;
body:
        while (true) {
                Sp ? S;
                depth_procedure(S,U);
                Up ! U;
        }
}
```

Figure 5. DEPTH schema

Schema Definition

Since ASL is a class-based language, we distinguish between a *schema* as the template for a process, and a *schema instance* (SI) as an active copy of that process. A schema template is composed of a header and a body. The header contains the schema name, a set of optional instantiation parameters, and external and internal declarations. The schema body describes the execution of the schema. 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 set of external variable declarations (visible both inside and outside the scope of the schema); *iv-decl* is the set of declarations visible only inside the scope of the schema; and *s* is the local schema program executing upon instantiation.

Encapsulation is accomplished by stipulating that internal (local) variables can only access other internal variables in the schema instance (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, a schema must 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.

The schema body gets executed immediately after schema instantiation and proceeds concurrently with other schema instances. It may contain communication expressions, instantiation commands, and other constructs such as iteration and conditionals. A schema instance may also spawn other schema instances, thus increasing the level of concurrency in the system. Schema instances are organized in a tree structure according to their instantiation history. The root of the parenting tree corresponds to the initial SI in the system, while nodes and leaves represent schema processes which get 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 general, all schema programs are initialized from a 'MAIN' schema which gets initially instantiated by the system. A schema program terminates once all instantiated processes have de-instantiated. However, in many "on-line" applications, the program will not terminate.

Independently of whether external or internal in their scope, ASL structures may be static or dynamic. Static declarations, such as those used in the 'RETINA' schema, are defined and allocated at the same time. The possible types are the following:

Schemas: Schemas define the active objects in ASL. (Ports, on the other hand are passive types.) A set of schemas is declared as follows

$sn \quad si_1,...,si_n;$

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

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.

Primitive Types: ASL supports basic types, particularly **int**, **char** and **float**. These types provide completeness to the language. 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. (Derived types include **layer**, which is equivalent to an array of **float**. ASL also supports array declarations of any ASL structure.)

In contrast to static declarations, dynamic declarations involve a two step process. First a pointer, similar to that in C [Kernighan and Ritchie 1978], is declared, which does not allocate actual memory for the particular structure type. It simply specifies an address for future reference. Then, the actual allocation is done as part of the schema body dynamic execution. While dynamic allocation is possible for any ASL type, the most interesting possibilities are given by dynamic allocation of ports and schemas. Dynamic port allocation permits the incorporation of new ports into an already compiled schema structure. Dynamic allocation of schemas provides the most powerful abstraction in ASL where processes may be instantiated at any point during the execution of a program. Other features in ASL include dynamically extensible arrays, and local and remote function calling simulating remote procedure calls (RPC).

Dynamic Schema Instantiation

As mentioned above, schemas can be dynamically instantiated, as well as de-instantiated, in the evolution of a schema program.

(i) Schema instantiation takes place by first having a schema pointer declaration in the schema interface sections, complemented by an instantiation construct in the schema body, where the declarations is given by

schema* *sid* or *sn** *sid*

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

sid = **new** *sn* or *sid* = **new** *sn*(*init-pars*)

where the **new** expression returns a reference to a *sn* object.

(ii) Schema de-instantiation can be accomplished, either implicitly, when its body finishes execution, or explicitly by stopping the schema instance. There is a slight difference between both kinds of "deaths". In the implicit way, a schema instance will only die after all its *delegated* schema instances have themselves died (refer to the delegation section). An explicit schema de-instantiation is accomplished by

stop *sid*

where *sid* is a schema instance and **stop** is a statement executed in the body of its schema parent process. When a schema process is stopped all its *relabels* and *connections* are deleted (refer to the section on port management).

Dynamic Port Instantiation

ASL greatly enhances the schema model by allowing dynamic port instantiations, as well as de-instantiations.

(i) Port instantiation may be dynamically accomplished by the use of the **new** *p* command, analogous to schema instantiation.

(ii) Ports de-instantiation is accomplished through **delete** *p*, where *p* is the port identifier, and the expression is general for any type of port. Only dynamically instantiated ports may be deleted.

Communication

Communication is asynchronous and buffered, characterized by that port writing is *non-blocking*, while port reading *blocks* until messages may be retrieved from the input buffer queue.

(i) Buffers are included in every input port, where incoming messages are stored until read. The ASL communication model assumes that input port buffers are unbounded and that a guarantee of message delivery exists. 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. Although reading *blocks*, the programmer may check the state of the buffer without *blocking*. This gives the programmer extended flexibility in deciding when to read from the input port, and thus avoiding possible blocking.

(ii) Writing data is allowed only through *local* ports, either internal or external. *Remote* ports, those ports belonging to other schemas, may not be directly accessed for writing, only indirectly via port connections. (Port-interconnections have to exist in order for communication to actually take place.) The syntax for 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. The writing command takes the form

$op \; \mathbf{!} \; e_1,\ldots,e_n;$

where *op* is the name of a local output port, and *e* is any value-returning expression. The message may be of any type, including primitive types, char, int, float, or derived types[4].

(iii) Reading data occurs independently from sending, in contrast to synchronous communication. Due to the asynchronous nature of communication, sending a message involves most of the work of actually delivering a message to its destination. Reading, on the other hand, requires only the retrieval of the message from the local buffer without having to know the message source. The only precondition on message reading is that a message exists in the local buffer, waiting to be read. A connection or relabel is not actually required at the time of reading. The read command is

$ip \; \mathbf{?} \; v_1,\ldots,v_n;$

where *ip* is the name of a local input port, and *v* is the variable where the message received is to be stored.

Wrapping

Both 'RETINA' and 'DEPTH' schemas are good examples of *wrapping*. The two schemas only provide an interface to external, possibly independently developed, programs. The two internal schema procedures, 'retina_procedure' and 'depth_procedure', may be developed as, e.g., procedural or neural networks programs[5].[6] Thus wrapping is defined as the integration of independently developed external programs to ASL schemas[7].

---

[4] The basic port structure only supports primitive types, integers, floats, and characters. Ports and schemas as messages are not currently supported. New port types may be derived by the programmer from basic ones, in particular passing derived data types. Remote procedure calls are simulated by transforming port message blocks into function names and their corresponding sequence of arguments (refer to Weitzenfeld [1992] for further details).

[5] Neural network systems may be developed in environments such as NSL2.1 [Weitzenfeld 1991].

[6] The key to successfully interfacing an external program to the schema model is more of an implementation issue than of a theoretical one. It requires that the external program be defined thorugh an external entry point, permitting external reading and writing of data passed as arguments to the program. Furthermore, this program may be executed more than once.

[7] For further applications of *wrapping*, refer to Bellman and Gillam [1990].

STEREO Schema

After having defined both 'RETINA' and 'DEPTH', it is necessary to define the 'STEREO' schema assemblage, providing composition and encapsulation of the two schemas. The 'STEREO' schema shown in Figure 6. Since schema assemblages are also schemas, the only difference with the 'STEREO' schema definition is in its instantiation of internal SIs 'R' and 'D', corresponding to 'RETINA' and 'DEPTH' schemas respectively (refer to the diagram in Figure 3). The external schema section consists of the two external ports to 'STEREO', 'Wp' and 'Up' for input and output, respectively. The body of the 'STEREO' schema includes port inter-connections between 'R.Sp' and 'D.Sp' (R.Sp >=> D.Sp), relabelings between 'R.Wp' and 'Wp' (W === R.Wp) and between 'D.Up' and 'Up' (Up === D.Up). The last entry in the body corresponds to the delegation command, the dependency of 'STEREO' on 'R' and 'D' (delegate R,D). This command implies that when instantiated, a 'STEREO' schema will not complete execution until both 'R' and 'D' have themselves completed. This is done in order to keep external communication paths to 'R' and from 'D'.

```
schema STEREO
{
external:
        input    Wp;
        output   Up;
internal:
        RETINA R;
        DEPTH  D;
body:
        Wp === R.Wp;
        Up === D.Up;
        R.Sp >=> D.Sp;
        delegate R,D;
}
```

Figure 6. STEREO Schema

Port Management

In order to understand the communication abstraction in ASL, it is important to understand the notion of *locality*. Basically, besides distinguishing between input and output ports, and between external and internal ports, we distinguish between local and remote port. **Local ports** are those ports which are referenced by the schema instance to which they belong (as specified through their declaration), and are usually referenced by a simple port identifier, i.e., *p*, where *p* is the port identifier. An example of a local port reference is 'Wp'. **Remote ports**, on the other hand, are those ports belonging to another child schema instance and are referenced by prefixing the schema instance name to the port identifier, i.e., *si.p* where *si* is the schema instance identifier, and *p* is the port identifier. An example of a remote port reference is 'R.Wp'. 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.

(i) Connections between ports have to be established in order for communication to take place. Connections are made exclusively between output and input ports. Ports may be connected and dis-connected in a dynamic fashion. 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

op **>=>** *ip* or *ip* **<=<** *op*

where *op* is the identifier of the output port and *ip* is the identifier of the input port.

Possible connection combinations are between remote external ports, or between a local internal port and a remote external port.

(ii) Disconnections can be made between ports which have been previously connected by using the command

p_1 **>=<** p_2

where p_1 and p_2 are output and input ports, respectively.

(iii) 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 parent schema instance environment. This is specially important in defining delegation and composition in schemas. Relabelings are made in a dynamic fashion, analogous to connections. In general, a port is relabeled by

p_1 **===** p_2

where p_1 and p_2 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.

Since the schema model is hierarchical and based on the notion of "parenting", the intuition behind relabeling is that a port belonging to a child schema instance may be accessible through its parent schema instance in order to enable communication beyond the parent schema environment behaving as message relays. In terms of output ports, a child schema instance communication will be send, indirectly, to the destination specified by the inter-connections of the parent schema instance port to which the relabeling is made. 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.

(iv) Delabeling enables ports, which have been previously relabeled, to be de-referenced by the command

p_1 **=|=** p_2

where p_1 and p_2 are both either input or output ports.

(v) The ASL communication model supports both **fan-in** and **fan-out** of both connections and relabels. Furthermore, a port may be simultaneously relabeled and also have connections to other ports. Each connection or relabel specifies an independent communication message path. Fan-out specifies how each message is copied into multiple communication paths, while fan-in sequentializes messages arriving from multiple communication sources.

Delegation

The notion of *delegation*, as used in ASL, differs from the notion used in other systems, in particular in the actor model, where delegation refers to either data or methods shared from an *actor* to its *proxy* [Lieberman 1986]. Delegation in the ASL sense, extends the functionality of children schema instances in that *delegated* schema instances behave more as *continuations* of the parent's tasks. This is also similar to the delegation notion in Hybrid [Nierstrasz 1987], where activities are delegated and not behaviors. On the other hand, the parent schema instance, or *delegator*, plays a continued role by forwarding all its external messages to those of the internal delegated schemas through appropriate port relabelings. This requires the delegating process not to terminate before the delegated processes does so, ensuring that messages sent to the delegating process are constantly transmitted to the delegated one. The construct ensuring such dependency is given by

delegate $si_1,...,si_n$

where $si_1,...,si_n$ correspond to the delegated schema instances.

(The integration of statically wrapped schemas, together with dynamically delegating schemas provides the expressiveness of the schema model. The decision on when to use static or dynamic abstractions depends on the desired tradeoff between processing efficiency vs. flexibility. A complex design would include a combination of both. For example, in a real-time distributed system, most schemas would be statically described, while a system which includes (dynamic) learning should involve schema delegation.)

Assemblages

'STEREO' is considered a schema assemblage composed of 'RETINA' and 'DEPTH'. The notion of *schema assemblage* enables aggregation and the building of complex hierarchical systems in an encapsulated fashion. This notion has directly evolved from a similar concept in the RS schema model [Lyons and Arbib 1989]. Yet, contrary to assemblages in RS, which are static in nature, assemblages in ASL are dynamic entities. Furthermore, schemas are abstractions in ASL, and not special syntactic entities as in RS.

MAIN Schema

Lastly, a complete ASL program requires a 'MAIN' schema which gets implicitly instantiated during system initialization. Thus, in this case, 'MAIN' schema, shown in Figure 7, contains the instantiation of 'STEREO', allowing the execution of the system. The 'MAIN' schema has no external ports since it is a the top of the schema tree hierarchy. Its internal section consists of 'S', a schema instance of 'STEREO' and built-in input port 'sin' (corresponding to the standard input in C) and output port 'sout' (corresponding to the standard output in C). The schema body contains port relabelings and its delegation to 'S'. Basically, both input and output may be in the form of external files, where input could correspond to data from a camera. When the task is finished, all the schemas in the program implicitly de-instantiate.

```
schema MAIN
{
internal:
        STEREO          S;
        input           sin;
        output          sout;
body:
        sin === S.Wp;
        sout === S.Up;
        delegate S;
}
```

Figure 7. MAIN Schema

Up until know a basic stereo system has been presented to illustrates the basics of ASL. In order to appreciate more the power and expressiveness of the schema model, an extension to this basic example will be given.

Extended Example

Let us extend the basic depth perception model, which only considers stereo information, to a more realistic one which takes into consideration lens focusing, or accommodation. This extended stereo model will incorporate a second 'DEPTH' schema. This system corresponds to one where 'DEPTH' is mapped to the 'Dev' neural network for basic stereopsis [Dev 1975]. The extended model corresponds then to the 'House' model [House 1984], where two 'Dev' neural networks are integrated to simulate both lens accommodation and binocular disparity cues. A diagram of the extended system is shown Figure 8. Again, the motivation behind this extensions is to illustrate the properties of the ASL model, where systems may be easily expanded as their designs and implementations evolve in time. Such an approach would require a total re-implementation of the system in other models. Yet, ASL hierarchical and modular structure enables a smooth transition between generations of systems.

In order to accommodate for a second 'DEPTH' schema, there are slight modifications done to the 'STEREO' schema, and to the 'RETINA' schema and the 'DEPTH' schema. These modification are necessary to enable inter-connections between the two 'DEPTH' schemas as well as to enable inter-connections from the 'RETINA' schema to the two 'DEPTH' schemas. It is emphasized, that externally, the 'STEREO' schemas looks exactly the same as before, as can also be seen by observing that the 'MAIN' schema remains unchanged.

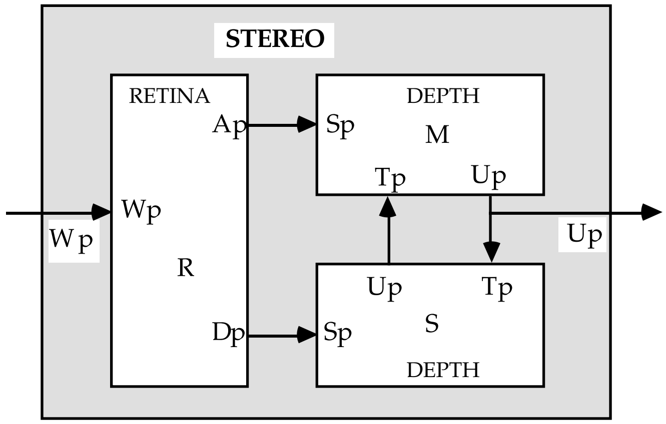

Figure 8. The extended stereo model

RETINA Schema

The extended 'RETINA' schema is shown in Figure 9. The schema is modified to include two output ports 'Ap' and 'Dp', instead of the previous single 'Sp' port. This modification enables transmission of both disparity (stereo) information through port 'Dp' to 'DEPTH' schema 'M', and accommodation information through port 'Ap' to 'DEPTH' schema 'S'. The 'retina_procedure' is also extended ('retina_procedure_ext').

```
schema RETINA
{
external:
        input    Wp;
        output   Ap,Dp;
internal:
        layer    W,SA,SD;
body:
        while  (true)  {
                Wp ? W;
                retina_procedure_ext(W,SA,SD);
                Ap ! SA;
                Dp ! SD;
        }
}
```

Figure 9. Extended RETINA schema

DEPTH Schema

The extended 'DEPTH' schema is shown in Figure 10. It consists of basically the same code as before except that an extra input port, 'Tp', has been added to enable communication between the two 'DEPTH' schemas. The 'depth_procedure' is also extended ('depth_procedure_ext') to reflect the new change.

```
schema DEPTH
{
external:
        input    Sp,Tp;
        output   Up;
internal:
        layer    U,S,T;
body:
        while  (true)  {
                Sp ? S;
                Tp ? T;
                depth_procedure_ext(S,T,U);
                Up ! U;
        }
}
```

Figure 10. Extended DEPTH schema

```
schema STEREO
{
external:
        input    Wp;
        output   Up;
internal:
        RETINA R;
        DEPTH          M,S;
body:
        Wp === R.Wp;
        R.Ap >=> M.Sp;
        R.Dp >=> S.Sp;
        S.Up >=> M.Tp;
        M.Up >=> S.Tp;
        M.Up === Up;
        delegate  R,M,S;
}
```

Figure 11. Extended STEREO schema

STEREO Schema

The extended 'STEREO' schema is shown in Figure 11. It includes two 'DEPTH' schema instances, 'M' and 'S', corresponding inter-connections between 'RETINA' and the two 'DEPTH' schema instances, and two port inter-connections between the two 'DEPTH' schema instances, from 'M.Up' to 'S.Tp', and from 'S.Up' to 'M.Tp'. The external relabeling is exactly the same, reflecting that theses changes have not affected the interaction of 'STEREO' with its external world. Also note that delegation is now extended from two to three schema instances.

MAIN Schema

The most important aspect of the extended model is that 'MAIN' schema stays exactly the same as before, as shown in Figure 12. This reflects that externally 'STEREO' has not changed.

```
schema MAIN
{
internal:
        STEREO          S;
        input           sin;
        output          sout;
body:
        sin === S.Wp;
        sout === S.Up;
        delegate S;
}
```

Figure 12. MAIN Schema

Neural-Schemas

The ASL modeling methodology has been applied to neural networks simulation, giving rise to the Neural Schema Language (NSL)[8], a system for the describing modular neural networks. Figure 13 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 [Weitzenfeld and Arbib 1991]. (It is interesting to note, that the neuron model is best modeled also as a multi-port entity.)

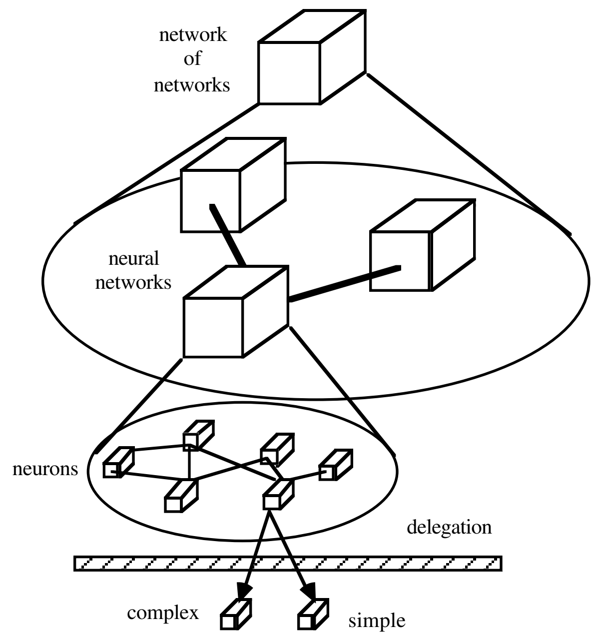

Figure 13. NSL model

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

Comparisons

The following sections contrast the ASL model most important characteristics to other models, including other concurrent object-oriented systems.[9]

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 [Robson and Goldberg 1984], 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.[10]

Multiple Ports

Multiple ports have been utilized in such computational models as CSP [Hoare 1978] and Port Automata [Steenstrup et al. 1983]. Yet most concurrent object-oriented models follow a single port model, in particular the *actor* model [Agha 1986]. (Some models based on Concurrent Logic Programming are also based on multiple ports, such as Vulcan [Kahn 1987].)

Contrasting ASL to languages derived from CSP, we have Ada [Ichbiah 1983], 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 [INMOS 1984] 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 [America 1987], 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 [Yonezawa et al. 1986] 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

9 For a more extensive comparison refer to Weitzenfeld [1992].
10 Refer to Briot and Yonezawa [1990] for an anlysis their shortcomins in the context of inheritance.

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 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 [Weitzenfeld and Arbib 1991]. This is similar to *domains* in Hybrid [Nierstrasz 1987], 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.

Conclusions and 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.

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, providing 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 [Briot and Yonezawa 1990]. In parallel, research is under way in extending the theoretical work in defining an asynchronous model for ASL in particular, and COOP in general [Milner 1990, Honda and Tokoro 1990].

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++ [Stroustrup 1987] 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 [Weitzenfeld 1991]). Its goal is the development of complex distributed applications in the areas of Brain Theory and Distributed Artificial Intelligence (DAI). These developments integrate with current work in defining a common ground between COOP and DAI [Briot and Gasser 1990].[11]

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* [Maes 1987].

References

Agha, G., 1986, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press.

America, P., 1987, POOL-T: A Parallel Object-Oriented Language, Object-Oriented Concurrent Programming, edited by A. Yonezawa and M. Tokoro, MIT Press.

Arbib, M.A., 1992, Schema Theory, In the Encyclopedia of Artificial Intelligence, 2nd. Edition, edited by Stuart Shapiro, 2:1427-1443, Wiley.

Bellman, K.L., Gillam, A., 1990, 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.

Briot, J.-P., Gasser, L., 1991, From Objects to Agents: Connections between Object-Based Concurrent Programming and Distributed Artificila Intelligence, IJCAI '91 Workshop on Objects and AI.

Briot, J.-P., Yonezawa, A., 1990, Inheritance and Synchronization in Object-Oriented Concurrent Programming, ABCL: An Object-Oriented Concurrent System, edited by A. Yonezawa, MIT Press.

Cointe, P., 1984, 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.

Dev, P., 1975, Perception of Depth Surfaces in Random-dot Stereograms: A Neural Model, Int. J. Man-Machine Studies, 7:511-528.

Goldberg, A., Robson, D., 1984, Smalltalk-80: The Language and its Implementation, Addison Wesley.

[11] For further discussions on the Neural Schema Language refer to Weitzenfeld [1992].

Hoare, C.A.R., 1978, Communicating Sequential Processes, Communications of the ACM Vol. 21 No. 8, pp 666-677, August.

Honda, K., Tokoro, M., 1991, An Object Calculus for Asynchronous Communication, Proc. ECOOP '91, Geneve, Switzerland.

House, D., 1984, Neural models of depth perception in frog and toad, PhD dissertation, Dept. of Computer and Informatin Science, U. of Massachusetts at Amherst.

Ichbiah, J., 1983, Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A.

INMOS, 1984, Occam Programming Manual. London: Prentice-Hall.

Kahn, K. Tribble, E., Miller, M., Bobrow, D., 1987, Research Directions in Object-Oriented Programming: Functions, Relations and Equations, chapter Vulcan: Logical Concurrent Objects, pp. 75-112, MIT Press.

Kernighan, B.W., Ritchie, D.M., 1978, The C Programming Language, Prentice-Hall.

Lieberman, H., 1986, Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, OOPSLA '86, Conference Proceedings.

Lyons, D.M., Arbib, M.A., 1989, A Formal Model of Computation for Sensory-Based Robotics, IEEE Trans. on Robotics and Automation, 5:280-293, June.

Maes, P., 1987, Concepts and Experiments in Computational Reflection, Proc. OOPSLA '87, :147-155, Orlando, FL, Oct. 4-8.

Milner, R., 1990, Functions as Processes, In Automata, Language, and Programming, LNCS 443:167-180, Springer-Verlag.

Nierstrasz, O., 1987, Active Objects in Hybrid, OOPSLA '87, Conference Proceedings.

Steenstrup, M., Arbib, M.A., Manes, E.G., 1983, Port Automata and the Algebra of Concurrent Processes, J. Computer Syst. Sci., Vol. 27, no. 1, pp. 29-50, Aug.

Stroustrup, B., 1987, The C++ Programming Language, Addison-Wesley.

Weitzenfeld, A., 1991, NSL: Neural Simulation Language, Version 2.1, CNE-TR 91-05, University of Southern California, Center for Neural Engineering, Los Angeles, CA.

Weitzenfeld, A., Arbib, M., 1991, 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.

Weitzenfeld, A., 1992, 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.

Weitzenfeld, A., 1993, An Overview of ASL: Hierachy, Composition, Heterogeneity, and Multi-Granularity in Concurrent Object-Oriented Programming, Proceedings of OOPSLA '92 Workshop on Next Generation Computing, Vancouver, Canada (in press).

Weitzenfeld, A., Arbib, M.A., 1993, Operational Semantics for the Abstract Schema Language ASL, (in preparation).

Yonezawa, A., Briot, J-P., Shibayama, E., 1986, Object-Oriented Concurrent Programming in ABCL/1, OOPSLA '86, Conference Proceedings.

Yonezawa, A., Tokoro, M., Eds., 1987, Object-oriented concurrent programming, MIT Press.

Appendix - ASL Syntax

Schema Definition

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

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* $id_1,\ldots,id_n$	// schema/port
	|	*SPRdecl* $id_1,\ldots,id_n$	// schema/port ptr
	|	*xv-decl*$_1$; *xv-decl*$_2$	// sequence

// internal declaration:

iv-decl ::=		ε	// empty
	|	*SPdecl* $id_1,\ldots,id_n$	// schema/port
	|	*SPRdecl* $id_1,\ldots,id_n$	// schema/port ptr
	|	*Vdecl* $id_1,\ldots,id_n$	// primitive
	|	*VRdecl* $id_1,\ldots,id_n$	// primitive ptr
	|	*iv-decl*$_1$; *iv-decl*$_2$	// 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***?**$v_1,\ldots, v_n$	// message reception
	|	*p***!**$e_1,\ldots, e_n$	// message delivery
	|	*p* **>==>** $q_1,\ldots, q_n$	// connect ports
	|	*p* **<==<** $q_1,\ldots, q_n$	// alternative syntax
	|	*p* **>==<** $q_1,\ldots, q_n$	// disconnect ports
	|	*p* **===** $q_1,\ldots, q_n$	// relabeling ports
	|	*p* **=|=** $q_1,\ldots, q_n$	// de-labeling
	|	**stop** $si_1,\ldots, si_n$	// de-instantiate schema
	|	**delegate** $si_1,\ldots, si_n$	// delegation
	|	*f***(**$e_1,\ldots, e_n$**)**	// function call

Statements

s ::=		ε	// empty statement
	|	*e*	// expr. as statement
	|	*v* **=** *e*	// assignment
	|	**if (***e***) then {** s_1 **} else {** s_2 **}**	// if-else
	|	**while (***e***) {** *s* **}**	// while-loop
	|	s_1 **;** s_2	// sequential composition