An Overview of ASL: Hierarchy, Composition, Heterogeneity, and Multi-Granularity in Concurrent Object-Oriented Programming

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

Extended Abstract

The Abstract Schema Language (ASL) [Weitzenfeld 1992; Weitzenfeld and Arbib 1992] unifies *schema* modeling [Arbib 1992] with concurrent object-oriented programming (COOP). ASL extends the current state of the art in both areas by providing a hierarchical approach towards heterogeneous and multi-granular concurrent object design. Schemas in ASL are functional units which get implemented in an orthogonal fashion. This aspect not only encourages code reusability, but enables schema implementations as, e.g., procedural programs or neural network processes. ASL addresses the need to separate task specification from actual task implementation, by utilizing the concept of dynamic task delegation and static wrapping of external programs. ASL schemas define class templates from which active objects get dynamically instantiated. Schemas incorporate a dynamic interface made of multiple input and output ports, plus a body section which explicitly specifies the schema behavior. Communication is in the form of asynchronous message passing, hierarchically managed through anonymous port reading and writing, at one level, and dynamic port inter-connections and relabelings, specified at a higher level. ASL supports object composition, through the notion of schema assemblages, enabling top-down and bottom-up model designs.

Current research involving ASL includes the development of a Structural Operational Semantics (SOS) describing the language, as well as a multi-process machine implementation. As part of its prototyping, a domain specific schema language for neural networks simulation, the Neural Schema Language (NSL), has been developed based on ASL. NSL is based on previous work with the Neural Simulation Language [Weitzenfeld 1989; 1990; 1991]), and its goal has been the development of complex distributed applications in the areas of Brain Theory and Distributed Artificial Intelligence. The NSL model supports neural entities as multi-granular schemas where neurons can be described with different levels of internal detail, and neural networks which may be inter-connected as networks of networks [Weitzenfeld and Arbib 1991][1].

ASL is designed to enable translation into different languages, the most important aspects of the model and the language are overviewed next.

Schemas and the Parenting Tree

The Abstract Schema Language (ASL) distinguishes 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 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. A schema program terminates once all instantiated processes have de-instantiated. However, in many on-line applications, the program will not terminate.

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

The root of the parenting 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.

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, 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 with special values.

Independently of whether external or internal in their scope, allocation of ASL structures may be static or dynamic. The possible static **declarations** involve the following entities:

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

sn $si_1,...,si_n$;

where *sn* specifies the schema class type 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, 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, helping define sequences of any ASL structure.

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 an address for 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 at any point. Furthermore, arrays in ASL may not only be dynamically created, but they may also be dynamically extended.

Another issue 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 by a simple port identifier. **Remote ports**, on the other hand, are those ports belonging to a child schema instance and are referenced by prefixing the parent instance name to the port identifier, i.e., *si.p* where *si* is the schema instance identifier, and *p* is the port identifier.

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.

Schema Instantiation

Schemas can be dynamically instantiated and 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 below). 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 next section on port management).

Port Management

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

(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.

(iii) 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.

(iv) 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.

(v) 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, as discussed later. Relabelings are made in a dynamic fashion, analogous to connections. 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_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. 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.

(vi) 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.

(vii) The ASL communication model supports both **fan-in** and **fan-out** of both connections and relabels, where 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.

Communication

Communication is asynchronous and buffered, where port-interconnection have to be specified before messages may be sent between processes. The asynchronous nature of the model is characterized by that writing is *non-blocking*, while 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 may not be directly accessed for writing, only indirectly via port connections. 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 **!** *e*

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[2].

(iii) Reading data occurs independently from sending, and in contrast to synchronous communication. 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 reading 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 message from the port buffer. Reading blocks, wait indefinitely, until a message can be read. The read command is

ip **?** *v*

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

Delegation

The ASL model includes the notion of delegation which slightly differs from a similar notion in other systems, such as in the actor model [Agha 1987]. Delegation in ASL extends the functionality of children schema instances in that they become *delegated* schema instances and the parent the *delegating* schema instance by relabeling the delegated schema instance ports to those of the delegating schema instance. Furthermore, the delegating process cannot terminate before the delegated processes does so. This is to ensure that messages sent to the delegating process are actually 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.

Assemblages

A key concept evolving from the RS schema model [Lyons and Arbib 1989] is that of schema composition in the form of *schema assemblages*, enabling the building of complex hierarchical systems in an encapsulated fashion. Contrary to assemblages in RS, the notion of schema assemblage in ASL is built in terms of basic constructs, such as relabeling, connections and delegation, and not as a special class of schema assemblage classes. Assemblages provide a crucial mechanism for defining multi-granular systems and for describing schema composition through message passing and as first-class objects.

Wrapping

By delegating the task implementation from a parent schema to its child we define the notion of dynamic schema tasks. On the other hand an external program may be statically *wrapped* into a schema as an alternative to task implementation. Wrapping enables the development of independent programs, e.g., procedural or neural, giving rise to the development of heterogeneous schemas.

The integration of static schemas with dynamic schemas provides the expressiveness of the schema model. The decision on when to use static or dynamic schemas 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 defined, while a system which includes learning should involve dynamic schemas[3].

---

[2] New types may be derived from basic ones, including derived ports which can understand the new derived types. The basic port structure only supports primitive types, integers, floats, and characters. Ports and schemas as messages are not currently supported. 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, refer to Weitzenfedl [1992] for further details.

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

Conclusions

This extended abstract presented an overview of the main aspects of ASL. In particular, ASL incorporates hierarchy, composition, heterogeneity and multi-granularity in concurrent object-oriented programming design. The ASL model has been compared to other object-oriented models, refer to Weitzenfeld [1992]. ASL has already been prototyped as part of our ongoing research. Future work includes its implementation on a truly distributed environment, as well as the full implementation of the domain specific NSL as a distributed neural network simulation system based on the ASL model. There are many issues which have yet to be fully analyzed, in particular those aspects arising from the asynchrony and non-determinism present in truly parallel systems. As part of our on-going research, work is 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 model, for applications in Distributed Artificial Intelligence and Brain Theory, in which the distinction of schemas as functional units from their actual body implementation gives rise to schemas as meta-objects.

References

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

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.

Weitzenfeld, A., 1989, NSL: Neural Simulation Language, Version 1.0, CNE-TR 89-02, University of Southern California, Center for Neural Engineering, Los Angeles, CA.

Weitzenfeld, A., 1990, NSL: Neural Simulation Language, Version 2.0, CNE-TR 90-01, University of Southern California, Center for Neural Engineering, Los Angeles, CA.

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., Arbib, M.A., 1992, Operational Semantics for the Abstract Schema Language ASL, (in preparation).