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

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 objects), 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.

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 systems where intensive processing and continuous inter-process communication are intrinsic 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 objects 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]

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

Schemas

The ASL computational model is defined in terms of *schemas*[2], hierarchical concurrent objects, as 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*. 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.)

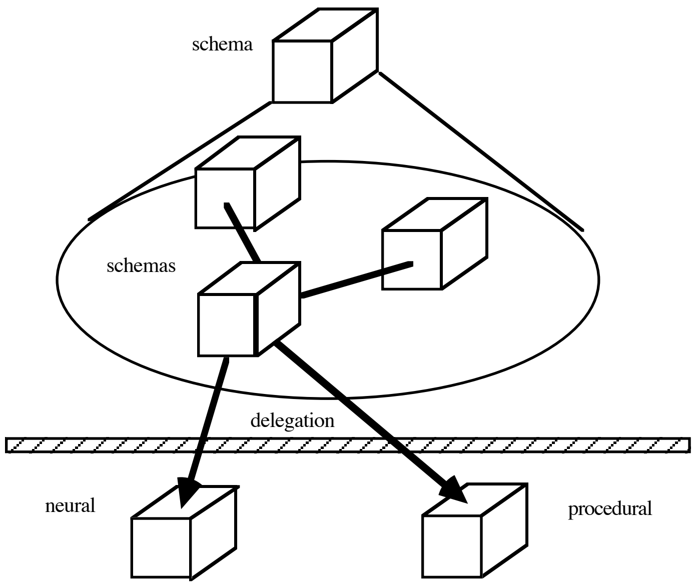

Figure 1. ASL schema model

There are several software design issues which the ASL schema model addresses:

- **Hierarchy**: System components may be partitioned into sub-components giving rise to top-down design methodology, and bottom-up design.
- **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**: Schemas can be distributed over a network of processors.
- **Concurrency**: Schemas are concurrent processes.
- **Communication**: Inter-schema communication is asynchronous, having "continuous" message passing.
- **Multi-granularity**: Schemas can be implemented as coarse-grain as well as fine-grain processes.

A functional description of the *stereopsis* vision problem is presented next illustrating ASL. The system's objective is to recognize a scene's object depth from available stereo information [House 1984].

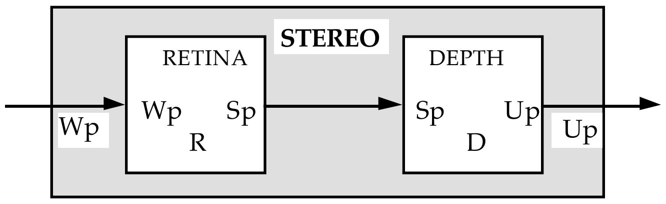

Figure 2. The basic stereo model.

Referring to the complete schema system as 'STEREO', it contains a 'RETINA' schema performing pre-processing on external images, and a 'DEPTH' schema obtaining depth information from general stereo cues. A diagram of the system is shown in Figure 2, which includes internal data path between the two sub-schemas, as well as external ones between the two sub-schemas and the external one.

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

RETINA Schema

Consider the 'RETINA' schema code as described in Figure 3 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' performs the actual schema task computing.

DEPTH Schema

The 'DEPTH' schema is shown in Figure 4. 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 RETINA {
external:
      input  Wp;
      output Sp;
internal:
      layer   W,S;
body:
      while (true) {
              Wp ? W;
              retina_procedure(W,S);
              Sp ! S;
      }
}
```

Figure 3. RETINA Schema

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

Figure 4. DEPTH schema

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 5. 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. The external schema section consists of the two external ports, 'Wp' and 'Up' for input and output, respectively. The body of the 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. Yet, a schema may terminate at any moment causing the termination of all its delegated schemas.

```
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 5. STEREO Schema

[3] The notation for reading and writing is similar to that of CSP [Hoare 1978], although the basic communication paradigms are distinct.

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[4].[5] Thus wrapping is defined as the integration of independently developed external programs to ASL schemas[6].

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 by an *actor* and 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. 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 retransmitted to the delegated one.

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.

Neural-Schemas

The ASL modeling methodology has been extended towards a domain specific neural networks model, giving rise to the Neural Schema Language (NSL), as shown in Figure 6, 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].

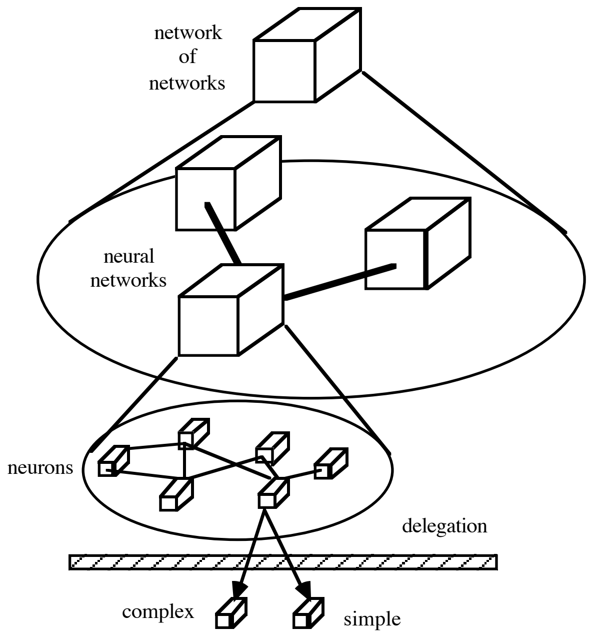

Figure 6. NSL model

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.

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

[5] 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 through 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.

[6] For further work on wrapping, refer to Bellman and Gillam [1990].

ASL basic mechanisms can be contrasted to other models. Multiple ports have been utilized in such computational models as CSP [Hoare 1978] and Port Automata [Steenstrup et al. 1983]. On the other hand, most concurrent object-oriented models follow a single port model, in particular the *actor* model [Agha 1986]. Contrasting ASL to languages derived from CSP, Ada [Ichbiah 1983], is based on 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, supporting point-to-point synchronous communication, although, 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 with actors, the multiple port paradigm avoids the need to search through single input queues when looking for a particular type of message. The 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 is required to compensate for the restrictions of single port models. When contrasting schemas with actors we distinguish the basic 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].

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.

In terms of ASL as a language, current research involves the incorporation of typing, schema signatures, and inheritance [Briot and Yonezawa 1990]. In parallel, research is under way in extending the theoretical work in defining an asynchronous model for ASL, work related to current thrust in defining semantics for COOP models 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, translating into C++ which is also the underlying implementation language.

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].

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 Artificial 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.

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 Information 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.

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.

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.

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, ASL: A Hierachical Computational Model for Distributed Heterogeneous Systems, TR USC-CS-93-552/CNE-93-02, Center for Neural Engineering, University of Southern California, Los Angeles, CA.

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

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.