

Growing Agents - An Investigation of Architectural Mechanisms for the Specification of “Developing” Agent Architectures

Virgil Andronache Matthias Scheutz

University of Notre Dame

Notre Dame, IN 46556

e-mail: vandrona,mscheutz@nd.edu

Abstract

In this paper we investigate various aspects of “developing agent architectures”, i.e., architectures that change over time according to their specification, in the framework architecture APOC. After a brief overview of APOC, we examine and discuss several ways of specifying developing architectures, and present examples of definitions of such architectures in APOC (e.g., Grossberg’s ART networks). We conclude with a brief discussion of the advantages of being able to specify developing architectures within an architecture framework at the level of the architecture itself.

Introduction

Architectures for intelligent agents typically do not define (as part of the architecture specification) the limitations of their components and how these components can change over time (if they can change at all). For example, architectures that include memory components usually do not specify how many items can be stored in memory, or whether memory is extensible (and if so, to what extent). Yet, making resource limitations and possible modifications of components explicit as part of the architecture specification can help in the design of agent architectures in many ways: for one, it will allow designers to explicitly take into account the limiting cases (e.g., where computational load or resource consumption is high). It will also permit them to capture some of the dynamics of instantiated components of the architecture in the running virtual machine at the level of architecture specification. But most importantly, it will allow them to specify various kinds of development and learning processes using architectural mechanisms instead of algorithms or mechanisms external to the architecture specification. That way learning and adaptation mechanisms, being part of the architecture and as such being implemented in certain components of the architecture, can be modified, adjusted, and generally altered in the same way that other parts of an architecture can be modified.

In this paper, we present the architecture framework APOC that provides architecture-level mechanisms to support investigations of architecture extension and modification as part of the architecture description. Following a brief

Copyright © 2003, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

overview of the components of APOC, we will describe in detail mechanisms in APOC that can be used to define developing agent architectures. We will demonstrate these mechanisms with several examples including a succinct specification of the development of fully-connected feed-forward neural networks in APOC. We conclude with a discussion of developing architectures and their advantages for the design of complex agents.

The APOC Architecture Framework

Basic Components

APOC is an architecture framework which provides a variety of capabilities for the specification of complex agent architectures. It can be used to implement commonly used architectures, such as subsumption and SOAR. The framework also provides mechanisms for run-time architecture modification and resource management that allow for specifications of simple architectures which can develop into complex architectures at run time. The APOC architecture framework consists of heterogeneous computational components called “nodes”, which can have any of the following four kinds of links among them: (1) Activation, (2) Priority, (3) Observer, and (4) Component links (hence the name “APOC”).

An APOC node is defined as a tuple :

$\langle priority, activation, inst_{default}, inst_{max}, links_{in}, links_{out}, maintenance, action \rangle$, where

- *priority* is the numeric priority of the node,
- *activation* is the numeric activation of the node,
- *inst_{default}* is the number of instances of the node which are instantiated automatically by the system. $inst_{default} \geq 0$
- *inst_{max}* is the maximum number of instances of the node which can exist simultaneously in the system. $inst_{max} \geq inst_{default}$ and $inst_{max} \geq 0$.
- *links_{in}* are incoming links from other nodes
- *links_{out}* are links going out to other nodes in the architecture
- *maintenance* is an ongoing computation used for updating the state of the node (e.g., priority calculation)

- *action* is a computation that is relevant to the system as a whole and is executed only if certain conditions are satisfied (e.g., if could consist of a series of motor commands).

The APOC Links

The first type of link, the A-link, is an activation passing link. Activations are the most general means through which a node becomes active. Activations are determined by the values passed via incoming activation links. How these activation link values are used to compute the final activation of a node is a decision to be made on a case by case basis (e.g., as in [0]). In most cases the simple and straightforward addition of incoming values will be the best way to determine the activation of a node.

The second type of link is a priority link (P-link). Unlike activation values, which are used in a combinatory fashion to determine an 'overall best' action, priorities are used to bias the system towards performing an action favored by one subsystem (or functional unit), e.g., if a global alarm mechanism is active, as described by Sloman [0].

The observer link (O-link) is another link supported by the architecture. The purpose of O-links is to provide a means of communication among the nodes in the architecture graph.

The C-link (component link) is used to instantiate nodes at run-time. Since it needs to determine resource availability at instantiation time, C-links play an important part in resource allocation and arbitration. If A-links or P-links are used in conjunction with a C-link, activation and priority based mechanisms can be used to trigger the action of the newly instantiated node.

Link Configuration

This section focuses on the specification of link behavior in the run-time machine. Link configuration can be specified in a recursive manner for each individual link. In this manner, any overall link configuration can be obtained in an APOC-based architecture. Conceptually, APOC link behavior is defined in terms of a basic two node unit as described in Figures 1 and 2.

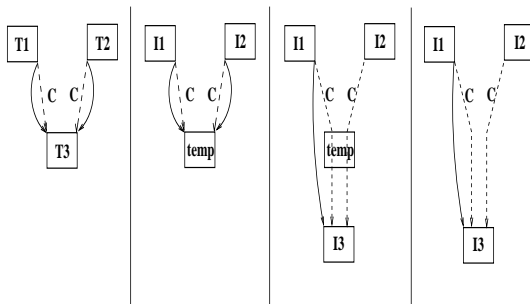


Figure 1: Link management with $access_{own}$ being false

Consider the case in Figure 1. Types $T1$ and $T2$ both use type $T3$ in their computation. They connect to $T3$ via C-links, for instantiation purposes, and via one of the other three types of link in order to influence the execution of $T3$

instances. In the run-time machine, when $I2$ needs to create an instance of type $T3$, a temporary node, *temp*, is created through the related C-link. The sole function of *temp* is the management of A-, P-, and O-links. All type-level links from $T1$ and $T2$ to $T3$ now connect to *temp*.

The C-link which causes the instantiation of *temp* - and, thereafter, of $I3$ - determines the access that other nodes have to $I3$, by specifying the boolean parameter $access_{own}$. If $access_{own}$ is false, $I3$ is created for the exclusive use of $I1$. Otherwise, the other nodes connected to *temp* are allowed to connect to the new instance. The decision of connecting to $I3$ is then made by each node individually and specified through the $access_{use}$ parameter of the corresponding C-link. The dynamic management of links when $access_{use}$ is true is illustrated in Figure 2.

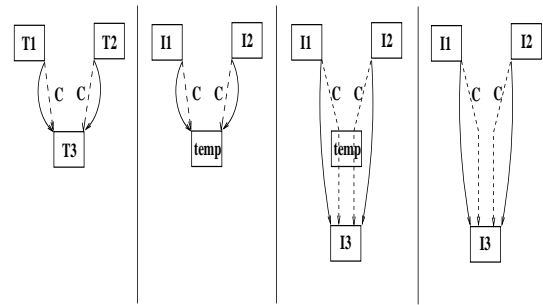


Figure 2: Link management with $access_{own}$ being true

In order to specify the link configuration for an arbitrary number of incoming links, a new *temp* node is created for each incoming link, as can be seen in Figure 3. In that figure, the following parameters are set:

- For $I1$ - $access_{own} = true$, $access_{use}$ is not used; can be either true or false
- For $I2$ - $access_{use} = false$, $access_{own}$ is not used; can be either true or false
- For $I3$ - $access_{use} = true$, $access_{own}$ is not used; can be either true or false

At the completion of this recursive process, any combination of links going to node $I3$ can be generated.

It should be noted that this process represents the theoretical foundation behind APOC's support of all possible link structures. Practical implementations can deviate from this strict modality of generating run-time links, as long as the overall functionality is preserved.

The following link behaviors are examples of the specifications that could be given to links "accompanying" C-links in the APOC framework

- Connect to all instances to which the C-link connects
- Connect to a single instance of the instances to which the C-link connects.
- Connect to some of the instances to which the C-link connects. It should be noted that for this case, link behavior

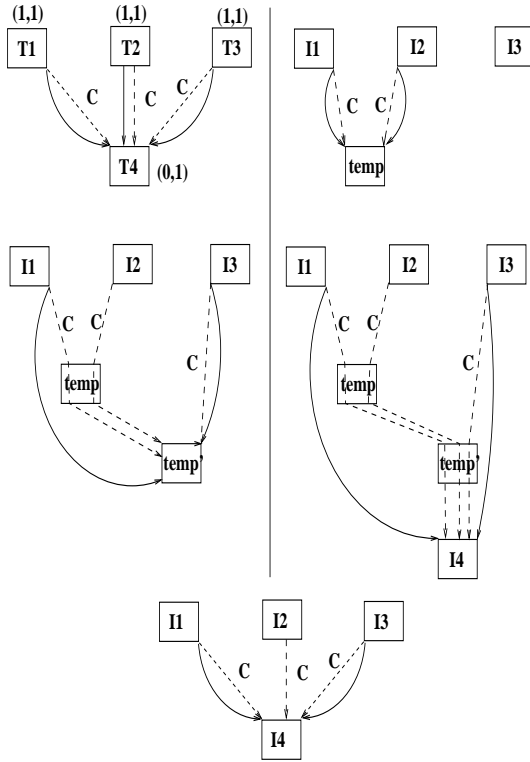


Figure 3: Recursive link management

needs to be specified in quite some detail, as the overall configuration may change on both the addition and removal of an instance from the run-time machine.

Two basic examples of APOC link configurations and their uses are presented in the next section.

Building architectures

In APOC, architectures are specified in terms of type relationships among components, i.e., APOC nodes, which provides a direct way of specifying abstract structures and modules. Tokens of these types are then instantiated in the running virtual machine. The following is a sampling of the possibilities exhibited by this mechanism.

This is the basic, non-resource-conflicting example of run-time node instantiation.

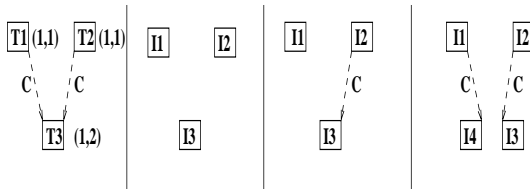


Figure 4: Type diagram, initial instantiation, state after first request, final state

In Figure 4, T1, T2, and T3 represent three node types,

with types T1 and T2 utilizing the action performed by type T3. The numbers in parentheses in the type diagram indicate *inst_default* and *inst_max* for each of the three types. Nodes I1 and I2 are instances of types T1 and T2 respectively, while nodes I3 and I4 are instances of type T3. When the architecture is first instantiated, nodes I1, I2, and I3 are instantiated, since all three types have *inst_default* set to 1. When the first explicit request for execution comes from I1 or I2 (in this case, I2), that node is connected to the existing instance (I3). The next request results in the instantiation of I4, with the requesting node, I1, using I4 to perform its operation.

Another use of C-links results in self-replicating units, as can be seen in Figure 5.

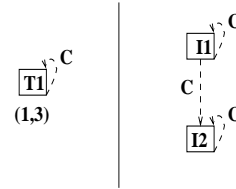


Figure 5: Type diagram, final state

The scenario in Figure 5 is both dynamic and reversible, i.e., the structures are created and can be destroyed at run-time. It is also possible to specify C-link behaviors which produce permanent structures.

Given the capabilities built into the APOC architecture via C-links, various types of learning can be directly modeled at the architecture level. For example, ART networks ([0]) could be developed during the life-time of an agent in a very similar way to the neural network above.

Unfolding and Copying

The C-link can be viewed as having two different types of functionality: duplicating and unfolding. Consider the situation in Figure 6.

The process shown in Figure 6 illustrates both uses of the C-link. After the architecture is initialized, the C-link is used to unfold the structure needed for executing instances of type T1. In the second stage of the process, the link is used to duplicate a node of type T1. While functionally identical, conceptually the two uses of the C-link give rise to different run-time properties.

Since APOC operations are performed at the level of individual instances, duplication by itself would have limited practical use. Little more than the creation of a set of instances (e.g., a network of nodes) that could be used in various places in the architecture could be achieved by duplication alone. Similarly, unfolding taken by itself could serve for little more than improving resource use. However, used in combination, the two C-link uses allow for the re-creation of entire structures. This opens the door for the creation of specialized structures, which can be 'plugged into' the architecture wherever appropriate. It also allows the system to adapt to changing resources, for example by duplicating an abstract system to speed up computation once the resources become available.

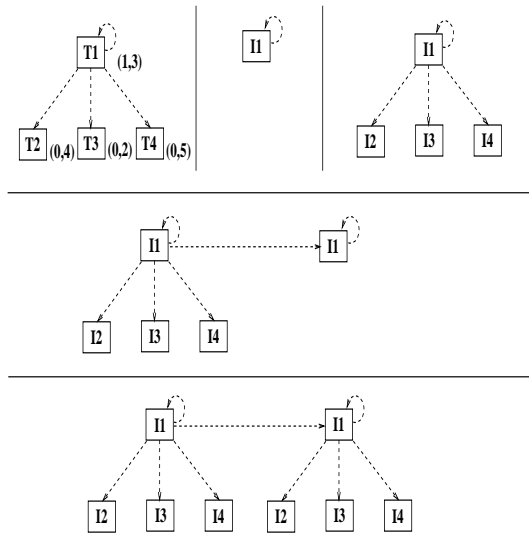


Figure 6: Duplicating and unfolding

Consider for example the scenario in Figure 6 to represent part of an agent, with node *I1* receiving information (data, activation, or priority) directly from the environment. In that case, the duplication of the structure can be a direct result of the agent's interaction with its surroundings. Different architectures can then develop from the same specification, each being appropriate for the circumstances in which it was developed. Figure 7 provides an example of the use such developing structures could have in an agent. In this figure, the network being grown is an ART network presented in [0]. This example will be presented in more detail later on.

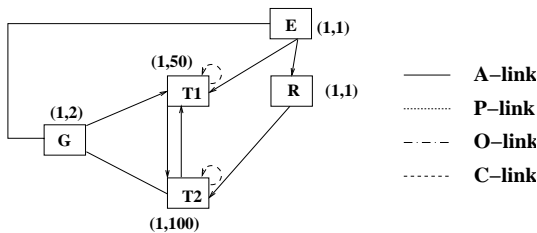


Figure 7: Sensory processing example

Examples

In this section we examine two cases in which the properties of the APOC framework are applied to developing structures that could be used in agents.

Fully Connected Feed-Forward Neural Network The properties of the C-link were presented in a previous section. Perhaps more interesting, however, are applications that involve combinations of links, such as those in Figure 8.

In Figure 8, an instance of type *T1* can create 100 instances of type *T2*. Similarly, instances of type *T2* can cre-

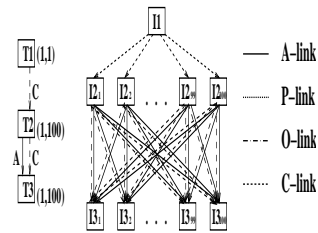


Figure 8: Type diagram, final state

ate 100 instances of type *T3*. With the proper link behavior specification, a run-time structure like the one represented by the final state can be obtained. What this amounts to is the possibility of “growing” such structures (e.g., a layered network that will eventually develop a fully connected layer).

ART network development As previously mentioned, APOC descriptions are made at the type-level. The correspondence of the nodes in Figure 7 to the ART network described by Carpenter and Grossberg is as follows:

- *E* represents the external inputs, in this case coming from a sensory node
- *G* represents gain control.
- *T1* represents the node type for nodes in the input layer
- *T2* represents the node type for nodes in the category representation layer, and
- *R* represents the reset of short term memory

The relation of the type-level links to the run-time machine depends on the behavior of each C-link. With the proper C-link definitions (e.g., C-links to instances of type *T1* copy over all links from the parent node), the final structure is the one of Figure 9, which mirrors the example presented in [0]. In the figure, the actual link structure for incoming links to the *T1* and *T2* groups is determined by the how the behavior of the C-links in the type-structure is defined. In this case, it would be appropriate to have connections to each of the nodes in those groups. However, since any link structure can be obtained in APOC, the links are represented in a more abstract manner.

One advantage provided by the APOC framework is that the number of categories can be made to vary according to the resources which the system is able to allocate to the process. For example, the network could start with 10 nodes of type *T1* and vary that number up to the maximum of 50 according to environmental circumstances.

Having looked at some of the ways in which complex agent architectures can be developed within the APOC framework, we will now briefly consider APOC in the general context of agent design.

Discussion

Agent architecture specification and instantiation are two distinct processes. However, in practice, the distinction between the two is often obscured by the fact that each element

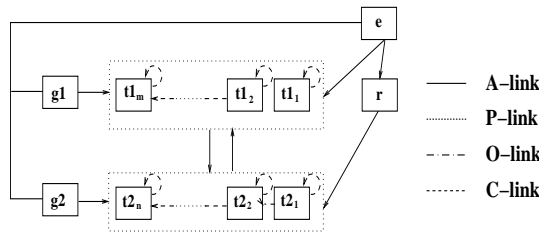


Figure 9: Sensory processing example - final architectural state

of the architecture specification (type) typically has exactly one instance (token) in the run-time virtual machine. That limitation of the architecture specification is removed in the APOC framework through the explicit specification of the maximum number of tokens that can be instantiated for each type. This makes run-time node recruitment for specialization a feasible option for APOC based architectures. It also gives agent designers the possibility of providing an agent with a variety of adaptation mechanisms, while allowing the agent to ‘choose’ which mechanisms are actually instantiated. Overall, the type-token differentiation opens the door to defining an entirely new class of ‘developing architectures,’ as illustrated in the previous sections.

Another asset of the framework is the built-in resource management mechanism. The type-level specification of the system allows the user to specify the default and maximum number of instances of each type that are going to be present in the run-time machine. Thus, nodes in the run-time machine perform their own resource management, through outgoing C-links. This allows the system to automatically add or delete nodes if changes occur in the available resources. The dynamic self-administration system, coupled with the four links which define all APOC communication also allows for the implementation of highly parallel systems at the architecture level.

It is characteristic to APOC that architectures in APOC can be specified at various levels of abstraction: an APOC type can represent a neural network unit, or something as abstract as a node performing a search operation (e.g., in long term memory). Therefore, on one hand, APOC architecture specification can be viewed as a direct mapping from a Use Code Map description of the system. This allows the analysis of top-down decomposition models such as [0]. On the other hand, a one-to-one correspondence can be made between the description of an architecture and the nodes instantiated in the run-time virtual machine, thus allowing the user to have complete control over all instances of components of the architecture at run time. User control can extend to specifying, at the architecture level, any of the following:

- models of learning algorithms
- adaptation mechanisms and changes in these mechanisms
- self-replicating architectures
- self-organizing architectures

as well as several other properties of the architecture.

Conclusion

In this paper we started an investigation of architecture development using the APOC architecture framework. Developing architectures provide new avenues for agent development, as they allow for investigations of the relationships between architecture specification and instantiation.

The characteristics which give the APOC framework its flexibility (i.e, the explicit handling of type-token relationships and C-links as means of dynamic control) were discussed and several examples of their application were presented. As a result of separating types from tokens, resource limitations are intrinsically taken into account in APOC by limiting the number of tokens which can be instantiated. The type-token distinction (in conjunction with the use of C-links) can be used in the development of complex structures in the run-time machine. Examples of applications of APOC features included: generation of fully connected layers of neural networks, self-replication, and ART network generation.

Future work with the APOC framework will include further investigations of the interplay between C-links and the other links in the framework, a generalization of the current implementation to support all previously described features of the C-link ¹, and applying developing architectures to software and embodied agents.

References

- R. J. A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski. High level, multi-agent prototypes from a scenario-path notation: A feature-interaction example. In *Proceedings of the 3rd International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-98)*, pages 255–276, London, UK, 1998.
- G.A. Carpenter and S. Grossberg. The art of adaptive pattern recognition by a self-organizing neural network. *IEEE Computer*, pages 77–88, March 1988.
- S. Grossberg. Adaptive pattern classification and universal recoding: I. parallel development and coding of neural feature detectors. *Biological Cybernetics*, pages 23:121–134, 1976.
- P. Maes. How to do the right thing. *Connection Science Journal*, 1:291–323, 1989.
- A. Sloman and B.S. Logan. Architectures and tools for human-like agents. In F. Ritter and R. M. Young, editors, *Proceedings of the 2nd European Conference on Cognitive Modelling*, pages 58–65, Nottingham, 1998. Nottingham University Press.

¹An APOC simulator is under development. Currently, A-, P-, and O-links are fully implemented. A limited-functionality version of the C-link is also available