APOC - a Framework for Complex Agents

Virgil Andronache **Matthias Scheutz**
Artificial Intelligence and Robotics Laboratory
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
e-mail: {vandrona,mscheutz}@cse.nd.edu

Abstract

In this paper we use the APOC–an agent architecture framework intended for the analysis and implementation of complex agent architectures–to investigate mechanisms of runtime architecture modification and resource management that allow for the specification of simple architectures which can develop into complex architectures at run time. After a brief overview of APOC, we show how a particular way of connecting components in APOC can be used to model various kinds of growing structures and, furthermore, how these structures can form modules to achieve higher level functionality. We also define a preliminary notion of "level of abstraction" for an APOC architecture and present a simple algorithm that can automatically assess the level of abstraction of APOC components in the architecture.

Introduction

Research in complex agents has produced a wide variety of architectures. These architectures range from competitive (e.g., subsumption) to cooperative (e.g., schema-based) and from strictly hierarchical (e.g., contention scheduling [Cooper & Shallice2000]) to virtually no hierarchy (e.g., the agent network architecture [Maes1989]. This paper presents APOC - an architecture framework for the development of complex agent architectures. APOC can be used as a common framework to implement, analyze, and compare commonly used architectures, such as subsumption and SOAR. It 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. We then explore ways of identifying and extracting functionality from the developed architectures. This framework can thus be used to investigate topics such as composition of modules to achieve high level functionality or the advantages of one top-down decomposition method over another. An example of the analysis capabilities provided by APOC can be seen in [Andronache & Scheutz2002].

The Architecture

APOC is an architecture framework which provides one basic component type and a set of basic mechanisms for connecting basic and derived components to allow for the specification of complex agent architectures. An APOC architecture consists of (possibly heterogeneous) computational components called "nodes", which can have any of the following four kinds of links among them: (1) *A*ctivation, (2) *P*riority, (3) *O*bserver, and (4) *C*omponent links (hence the name "APOC").

An APOC node is defined as a tuple [1]:
$< priority, activation, inst_{default}, inst_{max}, links_{in}, links_{out}, maintenance, action >$, 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 $inst_{max} \leq 0$.
- $links_{in}$ 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., it could consist of a series of motor commands).

The APOC Links

The A-Link 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. A-links are defined as:

$(V_s, V_e, act, , t, op)$, where
V_s is the component controlling the link,
V_e is the component acted upon by the link,
act is a unit of information passed through the link,
t is the time needed for information to traverse the link, and
op is an operation performed on act in the link

Activations are determined by the values passed via incoming activation links. How these activation link values are

[1]APOC nodes are an elaboration of "behavioral nodes" of [Scheutz2001]

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 [Maes1989]). In most cases the simple and straightforward addition of incoming values will be the best way to determine the activation of a node.

The P-Link The second type of link is a priority link (P-link). P-links are defines as:

$(V_s, V_e, pri, c, t, [S])$, where
V_s is the component controlling the link,
V_e is the component acted upon by the link, and
pri is the numerical value passed through the link, $pri \in [0,1]$
c is the activation cost per unit change in priority in V_e
t is the time needed for information to traverse the link, and
S is a signal sent to V_e. This signal can be an interrupt, which in turn can be either a reset or a suspend, or a directive to begin executing the action associated with V_e.

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 [Sloman & Logan1998].

The O-Link The observer link (O-link) is another link supported by the architecture and is defined as follows:

$(V_s, V_e, t, i_1[, i_2, ..., i_n]), n \geq 1, n \in \mathbb{N}$ where
V_s is the observer component,
V_e is the observed component,
i_1 is information mandatorily passed through the link from V_e to V_s,
t is the time needed for information to traverse the link, and
$i_2, ..., i_n$ are optional pieces of information passed through the link from V_e to V_s

The purpose of O-links is to provide a means of communication among the nodes in the architecture graph. Through an O-link a component supervises the execution of another component and is automatically informed any time a change occurs in the quantified entities being recorded by the observed component. Examples of these entities are speed, position, hunger level, etc.

The C-Link The C-link (component link) is used to instantiate nodes at run-time. It is defined as a 3- or 4- tuple:

$(V_s, V_e, t, [S])$, where
V_s is the controlling component of the link,
V_e is a component or group of components which are used in achieving the action of V_s,
t is the delay between the time the C-link is 'triggered' and the time $V - e$ is instantiated, and
S is a signal sent to V_e, usually triggerring the action of the component.

Since C-links need to determine resource availability at instantiation time, they 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 for a set of instances of a particular type can be specified in a recursive manner. Overall, APOC provides the functionality necessary to specify any link pattern. 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 connnects
- 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 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.

Three 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. In the following, we will present several examples of how modifiable architectures can be specified in APOC using the mechanisms provided by C-links.

Fully connected layered neural networks

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

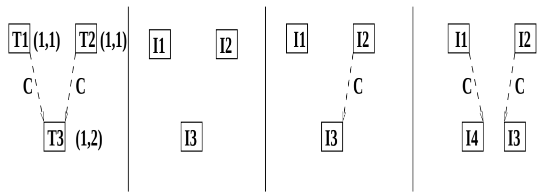

Figure 1: Type diagram, initial instantiation, state after first request, final state

In Figure 1, 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 2.

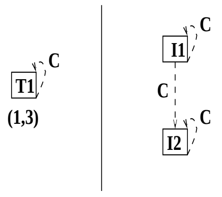

Figure 2: Type diagram, final state

Perhaps more interesting, however, are applications that involve combinations of links, such as those in Figure 3.

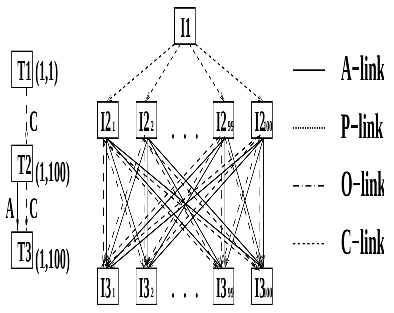

Figure 3: Type diagram, final state

In Figure 3, an instance of type T1 can create 100 instances of type T2. Similarly, instances of type T2 can create 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).

The scenarios in Figures 2 and 3 are 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. APOC nodes have the capability of instantiating other APOC nodes through the C-link, as well as making all the necessary connections. In most circumstances, the instantiating nodes also use the C-link to delete the instantiated structure. However, it is also an option to delete the instantiating C-link and allow the newly instantiated nodes to function independently (e.g., performing a specialized function such as recognizing a specific stimulus).

ART networks

Structures which can perform a categorization of their inputs are useful in a variety of applications, e.g. computer vision. In this section we show how APOC mechanisms can be used to grow an ART network. Figure 4 provides the basic structure needed for its development.

The correspondence of the nodes in Figure 4 to the ART network described by Carpenter and Grossberg [Carpenter & Grossberg1988] is as follows:

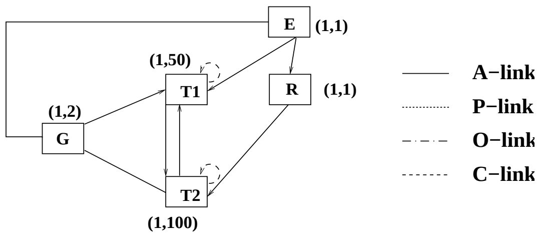

Figure 4: Sensory processing example

- 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 5, which mirrors the example presented in [Carpenter & Grossberg1988]. 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.

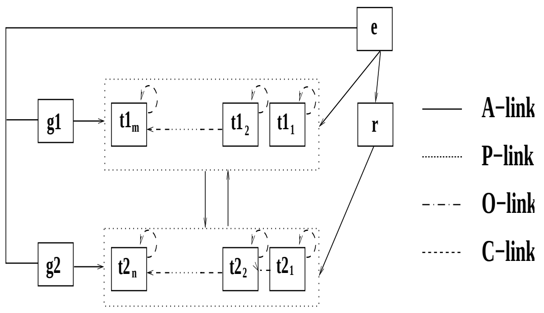

Figure 5: Sensory processing example - final architectural state

In the above example, the external inputs are defined by a single unit, E. However, it is reasonable to generalize upon the example presented and consider the possibility of subjecting the external inputs to resource constraints. This can be done with relative ease in APOC given the broad definition given to APOC nodes. Since each APOC node can be broken down to a minimal complexity, E can be viewed as being composed of several units, each of which takes care of one environmental unit. The quantity of information used for categorization can then be modified dynamically based on resource constraints and/or environmental complexity. The resulting instantiated architecture is shown in Figure 6.

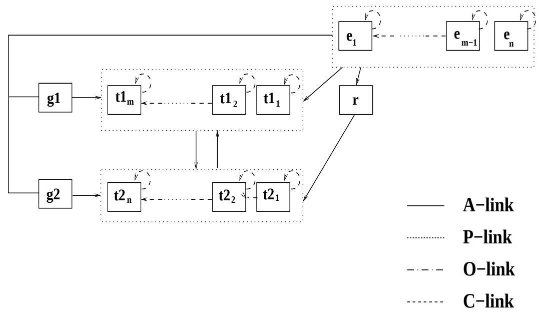

Figure 6: Sensory processing example - final architectural state

Incremental, resource constrained planning

Another situation where APOC architecture development capabilities can be successfully used involves planned sequences of actions. Consider the example in Figures 7 and 8, where a physical agent has a choice of three different actions: "move right", "move left", and "move straight". The agent's goal is to move from its current location at point A to a point B, say. To support the planning process, the agent has an environment simulation module, in which plan actions can be simulated instead having to carry them out in the real world.

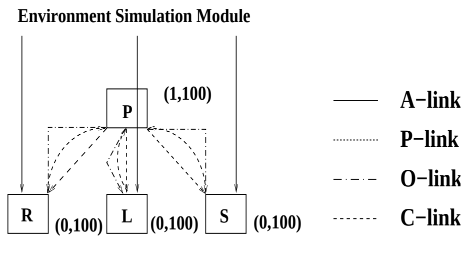

Figure 7: Planning example - specification

Figure 7 contains a four node assembly which defines the type relationships in the architecture:

- Node P represents a planning type
- Node R represents a type which simulates a right turn in the environment simulation module
- Node L represents a type which simulates a left turn in the environment simulation module
- Node S epresents a type which simulates a finite forward move in the environment simulation module

The component links in the figure indicate that nodes of type P can instantiate nodes of types $R, L,$ and S. Conversely, nodes of types $R, L,$ and S can instantiate nodes of type P. The implication of this circular C-link structure is that nodes of type P can instantiate nodes for all actions which are currently feasible.

In the example of Figure 7 above, the run-time system changes are described below.

The P-node checks for completion of the system-wide goal (e.g., reaching B). If the goal has not been reached, the P-node makes a decision on which of the $L-, R-,$ and S-nodes should be instantiated and adds the necessary incoming activation links from the "environment simulation module". Each instantiated node receives activation inputs from an environment simulator and computes its activation level. The P-node then uses the information gathered through the O-links to decide which node is most likely to lead to goal completion and therefore which node will carry out its action (a process similar to the arbitration found in contention scheduling [Cooper & Shallice2000]). The winner then gets to instantiate another P-node, and the process continues until the goal is completed.

It should be noted that the run-time evolution of the system is dependent on the internal computations of the nodes and the environmental input. With a different definition of P-nodes, the system could be used to explore the state space in a limited depth first search manner with backtracking. For example, the R-node always instantiates a P-node first. This, in turn, instantiates another set of the three nodes, where the R-node instantiates a P-node, etc. Here too, each P-node checks whether the goal is satisfied, continuing the recursive instantiation process until the depth limit (i.e., the maximum number of nodes of type P that can be instantiated) is reached. If the goal is not reached, the P-node finish its processing, hence, its parental R-node will terminate as well, and the P-node that instantiated the terminated R-node will continue with the instantiation of the S-node, and later possible the L-node until a path to the goal is found.

C-link behavior already provides a benefit in resource allocation, in that an L-type node would not be instantiated if the agent finds itself next to a wall on its left side. Additionally, once the action associated with a particular step in the planning process is executed, C-links allow the releasing of those resources associated with branches in the planning tree which were not followed.

If actions, such as "turn right" are known not to fail, it is beneficial to remove all nodes of the graph once the physical agent has performed the actions associated with those nodes. If on the other hand, actions are susceptible to failure, keeping the nodes along the execution path in use (up to the resource limitations of the system) can lead to additional advantages. For example, if an action fails, the backtracking process only involves allowing the node one C-link up from the failing node to re-execute its action (e.g., if a "turn right" fails, the planner may delete the R node and try to find an alternate path by either moving forward or turning left).

Another advantage relates to the system's knowledge of the world. Planning processes usually create plans for several steps in advance of the related physical execution. In the physical world, conditions are changeable. Consider, for example, the case where point *B* is the known location of a resource the agent needs to obtain. If, at the conclusion of the physical actions the resource is not found (either because its location has changed or the planning process was erroneous), it is now easy, to retrace the last few steps that the agent has performed, until the state of the environment matches the expected state of the planning process - perhaps even returning to point *A*. With a system like the one in Figure 8, this retrace can easily be done by following the C-links backwards through the graph.

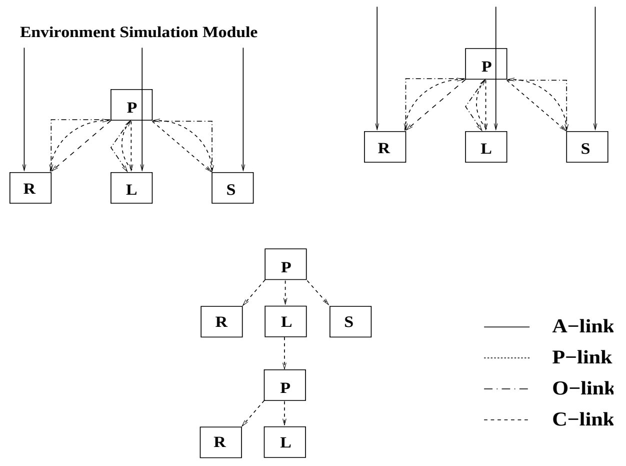

Figure 8: Planning example - sample instantiation

In Figure 8 the first three steps of a planning planning process developed from the structure of Figure 7. The figure illustrates two instances in which nodes were left uninstantiated.

Use Case Map example

By allowing the architecture to be specified at various levels of abstraction, the APOC architecture specification can be as abstract as to be a direct mapping from a *Use Case Map* (UCM) description of the system. This allows the analysis of top-down decomposition models such as [Buhr *et al.*1998].

Use case maps are defined through six types of components: paths, waiting places (for stimuli or events, denoted with circles in UCM diagrams), timers (waiting places with an upper bound on waiting time; denoted with "clock faces"), bars (markers for ends of paths, as well as beginnings and ends of concurrent paths), basic paths (paths starting at a waiting place and ending at a bar) and directions. For a direct (though not necessarily most efficient) mapping from UCMs to APOC, each waiting place, timer, and bar is mapped to a separate APOC node.

The two examples in Figures 9 and 10, taken from [Buhr *et al.*1998] illustrate how such mappings may be performed. The UCM diagram is a generic example, but it can be applied to the following scenario of action selection:

- The left waiting place represents inputs from external stimuli
- The left bar represents computation based on the internal state of the agent
- The right waiting place synchronizes the data based on internal state with data coming from a higher level planner
- The right bar represents the end of computation where all three pieces of data have been gathered and an action has been selected for the agent

It should be noted that the APOC links depicted are, with two exceptions, generic links, the actual type depending on the specific system depicted. It should be noted that the conversion has been done in a manner which illustrates the correspondence that can be drawn between UCM and APOC component, and not in the most concise manner allowable by APOC (e.g. the *b* nodes could be combined with C-links into a single B(2,3) type node).

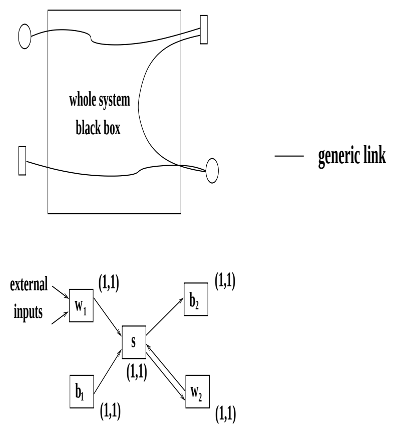

Figure 9: Black box UCM example

Figure 9 illustrates a high-level UCM description and requires an abstract APOC node to represent the entire system. Figure 10 illustrates how a more detailed map can be ported to APOC.

Cellular automata simulation

For cellular automata, we start be defining the basic entities in the APOC framework. A cellular automaton is an array of identically programmed automata which interact with one another [Wolfram1984]. Each cell is defined by

- state - a variable that takes a different separate for each cell. Cell state can be easily implemented as part of the *maintenance* section of an APOC node.
- neighborhood - the set of cells that it interacts with. The neighborhood of a cell can be described in terms of those cells to which it is connected via C-links.
- program - the set of rules that defined how its state changes in response to its current state, and that of its neighbours. This is easily implementable in the *action* section of an APOC node.

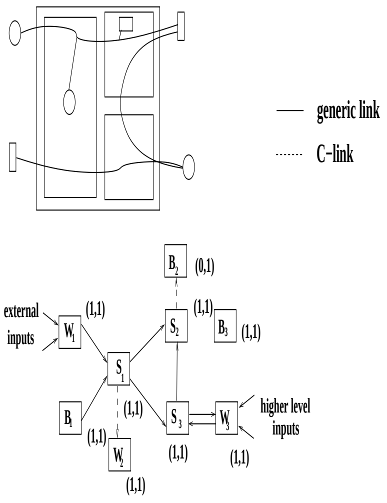

Figure 10: White box UCM example

Consider the simple example of the game of life as illustrated in Figure 11, with the following rules:

1. A living cell with only 0 or 1 living neighbours dies from isolation.
2. A living cell with 4 or more living neighbours dies from overcrowding.
3. A dead cell with exactly 3 living neighbours becomes alive.
4. All other cells remain unchanged.

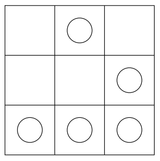

Figure 11: Game of life example

The APOC implementation of cellular automata requires each cell to keep track of its overall position in the system in the form of an integer coordinate pair. With that provision, the implementation is made as follows:

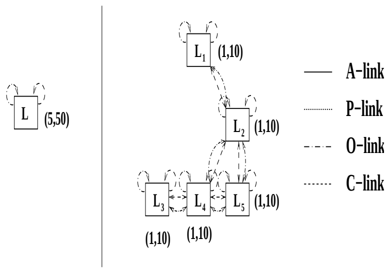

Figure 12: Game of life - sample APOC initial configuration

- Specify an initial configuration through the type configuration, as seen in Figure fig:life-init. The left side indicates the most concise method of specifying the game. For that specification to be feasible, type L has to contain the initial configuration of the system. The node arrangement on the right is specified according to the life example above for ease of understanding. All nodes in the type specification are identical with the exception of the location, which is specified for each instance that is produced during the development of the system. It should be noted that the specification on the left is rather more general, as the resource restriction is a general one. This implies that the 50 units could be instantiated by a single cell and its descendants if permitted by the configuration, as opposed to the 10 instances per "type" indicated in the right side configuration.
- At instantiation, each node attempts to instantiate a "dead" cell at the eight positions around it. This simply means that a cell of coordinates (x, y), attempts to instantiate cells which are given coordinates $(x-i, y-j)$, with $i \in [-1, 0, 1]$, $j \in [-1, 0, 1]$ and i and j not simultaneously 0. If a cell with those coordinates already exists, the node simply connects to it through a C-link, as illustrated in 13.[2].

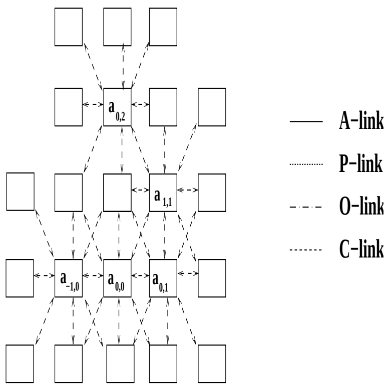

Figure 13: Game of life - APOC configuration

- Each cell checks the number of incoming C-links and changes its incoming C-links and changes its status according to the rules described above. A 'dying' cell deletes all its outgoing links.
- Finally, a dead cell with no incoming C-links deletes itself.

With the set-up described above, a dynamic game-of-life system whose size need not be limited at design time is easily implemented *via* APOC constructs. The system described above will have the live-cell configuration shown in Figure 14 after one iteration.

The basic concepts described in this section can be extended to other cellular automata problems.

[2]For the sake of readibility, only C-links are shown in Figures 13 and 14. In the actual system, each link is accompanied by an O-link

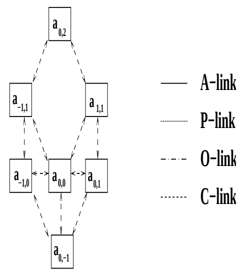

Figure 14: Game of life - Live-cell configuration configuration

Architecture analysis

As mentioned in previous sections, APOC architectures can be specified at high levels of abstraction. This allows the designer to provide only "limits" for the development of the architecture (e.g. resource limitations, or limiting a particular type to using A-links), but not the particular architectural layout. The possibility thus arises that useful functional capacities which were not present in the original architecture develop through interaction with the environment.

For many agents, prolonged environmental interaction can lead to the development of complex systems, performing specialized functions geared towards the survival of the agent. In such circumstances, gaining an understanding of the role various components play in the architecture becomes an extremely difficult task in the absence of specialized architecture analysis tools. It is in this context that we provide in this section a first look at the process of analyzing these self-developing architectures. The following section introduces an algorithm for basic architecture analysis.

Hierarchies in non-hierarchical architectures

Architectures defined in APOC often do not have explicit hierarchies defined at design time. However, C-links impose a degree of structure on these architectures. It is this structure that can be exploited in order to analyze the architectures with a view towards the complexity of the computation performed in the *action* section of each node.

Hierarchy Identification Even though APOC architectures are specified at the level of types of instances, thus hiding to a certain degree the underlying structure, certain relations can be distinguished among type-level components connected *via* C-links. These relations can be broken down to two different classes:

- C-link connection from a complex type to a simpler type (e.g., a type that achieves part of the goal of the more complex type)
- C-link connection between two related types of similar complexity (e.g., as seen in the planner example above)

The above breakdown can be used to identify substructures as defined by the network of C-links within an architecture by following outgoing C-links from the type-level description of the architecture. Two possibilities arise: the chain (graph) of links eventually ends in a type with no outgoing C-links, or the original type is encountered a second time. With this in mind, we introduce the following definition:

Definition 1. *The* ***abstraction level*** *of a node, n, in the runtime architecture is defined as the maximum number of consecutive C-links that can be followed out of n until either:*

- *A node with no outgoing C-links is encountered*
- *Node n is reached again without repeating any C-links along the way*

Following this definition, the algorithm for determining the abstraction level of a node, n, takes two arguments: the directed graph $G = (V, E)$, where V is the set of type nodes and E is the set of C-links, and node n and returns the level of abstraction of node n.

```
FIND-ABSTRACTION-LEVEL(G,n)
$path\_length \leftarrow 0$
for each vertex u $\in$ V[G]
  if u has no outgoing C-links
    do $path\_length_u \leftarrow$ FLSP(n,u,G) [3]
    if $path\_length_u > path\_length$
      $path\_length \leftarrow path\_length_u$
$path\_length_n \leftarrow$ FLSP(n,n,G)
if $path\_length_n > path\_length$
  $path\_length \leftarrow path\_length_n$
return $path\_length$
```

In the algorithm above, FIND-LONGEST-SIMPLE-PATH is a modified shortest path algorithm [Cormen, Leiserson, & Rivest1990] and returns only the length, in number of links, of the longest simple path between n and u. In the next section we look at how abstraction level information applies to some the examples presented in this paper and what that information could tell us about the system if no prior knowledge of the system existed.

Uses of hierarchical information

In this section we consider how the abstraction level concept applies to some of the examples presented earlier in the paper and we investigate the type of information that abstraction levels can provide to an observer without prior knowledge of the systems.

Neural networks In Figure 3, there are only three nodes arranged in a strict hierarchy with respect to C-links. Applying the abstraction level algorithm gives node $T1$ an abstraction level of 1, node $T2$ an abstraction level of 1, and node $T3$ an abstraction level of 0. In this case the implications of the respective abstraction level numbers are clear: node $T1$ represents either the most "abstract" type in the system, or the level at which external inputs enter the system; node $T2$ represents an intermediate layer; node $T3$ represents either the basic action or the output layer of the system.

[3]FLSP denotes FIND-LONGEST-SIMPLE-PATH

Planning In Figure 7, there are four nodes, each of which has an abstraction level of 2. The implications here are that:

- There is an interdependence between node P on one hand and the $L, R,$ and S nodes on the other.
- All four nodes have some capability of processing environmental/internal information. This processing may lead to the instantiation of a new node
- All four nodes have the ability to terminate the action of the system

Cellular automata simulation The cellular automata illustrates one possible problem with the abstraction level analysis. If the analysis is performed on the left side of Figure 12, it seems that there is one type, which is self sufficient. It receives information from the environment, processes it, and makes a decision based on that processing. However, if the analysis is performed on the more detailed original specification, different levels of abstraction are obtained for each node. This is an example of the type of situation for which a more refined notion of "level of abstraction" than the one provided above is required to be able to extract automatically the functional organization of an APOC architecture. Nevertheless, the previous examples show that with a notion as simple as the one defined valuable information can be already extracted from the C-link architecture.

Discussion

The APOC architecture framework discussed in this paper is still very much in the development stages. However, as the above examples with the current version of APOC indicate, there can be great utility to an architecture framework that allows one to cast at different levels of abstraction several *prima facie* unrelated formalisms (such as neural networks, cellular automata, and planning algorithms). Not only is it possible to compare the functionality of these different formalisms in a unified framework, but it is is also possible to assess the resource requirements of these formalisms at the architecture level. Furthermore, a framework like APOC can provide the basis for the definition of algorithms that can automatically extract information about substructures (such as modules) and their functional organization from architecture descriptions. This is particularly interesting for virtual machines that have undergone a learning or adaptation process, which modified their original architecture in such a way that functional components of their current architecture are not easily identifiable (e.g., neural networks after learning and/or growing processes).

It is also worth mentioning that APOC is not limited to architectures of single agents or to agent architectures, for that matter. Rather, it is possible to define multi-agent systems at the level of indiviual perceptions and actions in terms of the APOC framework: each individual agent is modelled by an APOC node, which in turn has O-links (modelling the perceptions of the agent) and A-links (modelling the actions of the agent). To model procreation in biological systems, C-links can be used to allow agents to instantiate copies of themselves. In general, APOC could be used to model both centralized and distributed control systems.

To support the implementation of agent architectures in the APOC framework, we are currently developing a software tool that will allow users to develop APOC architectures in JAVA. Invidiual APOC nodes can be defined depending on the task at hand, which can then be linked together with a graphical tool using any of the four link types. The APOC development environment, furthermore, allows users to instantiate architectures and monitor their states over time as the virtual machine is running, i.e., to run simulations of control systems defined in APOC. Examples of such simulations range from neural network simulations to real-time robot control systems.[4]

References

Andronache, V., and Scheutz, M. 2002. Contention scheduling: A viable action-selection mechanism for robotics? In Conlon, S., ed., *Proceedings of the Thirteenth Midwest Artificial Intelligence and Cognitive Science Conference, MAICS 2002*, 122–129. Chicago, Illinois: AAAI Press.

Buhr, R. J. A.; Amyot, D.; Elammari, M.; Quesnel, D.; Gray, T.; and Mankovski, S. 1998. 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)*, 255–276.

Carpenter, G., and Grossberg, S. 1988. The art of adaptive pattern recognition by a self-organizing neural network. *IEEE Computer* 77–88.

Cooper, R., and Shallice, T. 2000. Contention scheduling and the control of routine activities. *Cognitive Neuropsychology* 17(4):297–338.

Cormen, T. T.; Leiserson, C. E.; and Rivest, R. L. 1990. *Introduction to Algorithms*. Cambridge, MA: MIT Press.

Maes, P. 1989. How to do the right thing. *Connection Science Journal* 1:291–323.

Scheutz, M. 2001. Ethology and functionalism: Behavioral descriptions as the link between physical and functional descriptions. *Evolution and Cognition* 7(2):164–171.

Sloman, A., and Logan, B. S. 1998. Architectures and tools for human-like agents. In Ritter, F., and Young, R., eds., *Proceedings of the 2nd European Conference on Cognitive Modelling*, 58–65. Nottingham, UK: Nottingham University Press.

Wolfram, S. 1984. Cellular automata as models of complexity. *Nature* 311:419–424.

[4]A beta-version of the software can be downloaded from HTTP://WWW.CSE.ND.EDU/~AIROLAB/APOC/.