The APOC Framework for the Comparison of Agent Architectures

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

Abstract

In this paper, we present APOC, an agent architecture framework intended for the analysis, comparison, and evaluation of agent architectures. We demonstrate how four main architectures, GRL, ICARUS, PRODIGY, and SOAR can be translated into APOC and briefly discuss how these translations could be used to compare architectures and possibly integrate features from these different architectures within one common architecture.

Introduction

Agent architectures are blueprints of control systems of agents, depicting the arrangement of basic control components and, hence, the functional organization of the overall agent control system. In addition to the functionality of its control components, an agent architecture determines the representational repertoire available for data structures and information processing in the system as well as the structural modifications that can be made (if at all) to the components and their connections. Typically, agent architectures use general purpose programming languages to allow for the definition of data structures, processing components, and their connections.

Various different architectures and architecture design methodologies have been proposed for intelligent agents in the history of AI, ranging from cognitive architectures for complex (possibly human-like) agents (e.g., SOAR (Laird, Rosenbloom, & Newell 1986; Laird, Newell, & Rosenbloom 1987), ACT-R (Anderson *et al.*), PRODIGY (Veloso *et al.* 1995), and others), to layered architectures for simulated and robotic agents (e.g., 3T (Bonasso *et al.* 1997), ICARUS (Langley *et al.* 2003; Langley, Cummings, & Shapiro 2004) , AuRA (Arkin & Balch 1997), subsumption (Brooks 1986), motor schemas (Arkin 1989), GRL (Horswill 2000)). While all of these architectures have introduced new architectural concepts (such as new types of components or new structural features specifying their interconnections, etc.), it is difficult to compare these concepts across architectures in the absence of a general language or formalism, in which all the different architectural features can be expressed or formalized. We believe that such a comparison, however, could be useful for several reasons. For one, it might contribute to an understanding of why some architectures are better suited for a particular class of tasks than others (e.g., where processing bottlenecks are, why one functional organization is more robust and less susceptible to system overload than another, etc.). It might also help to integrate proven features from different architectures in an effective, efficient way (e.g., a hierarchical control system like RCS (Albus 1992) might be integrated with a schema-based reactive control and a higher-level planner). Finally, it might lead to a development process for agent architectures that allows for the reuse and integration of functional components akin to what is typically done in software engineering with shared libraries, APIs, and other software abstractions (such as packages, modules, etc.).

What is needed is an *architecture framework*, which is general enough to allow researchers to evaluate and compare different kinds of architectures, but is at the same time conceptually parsimonious enough to employ only a few intuitive, basic concepts by virtue of which architectural features and mechanisms can be expressed and defined.[1] To our knowledge, no satisfactory framework is currently available that is conceptually "simple" and "small" (in the number of basic concepts), while achieving high expressiveness at different levels of abstraction (although promising efforts are under way, see the other contributions to this workshop). As a first step towards the development of such a general framework, we propose the APOC agent architecture framework under development in our lab. We will first provide a brief overview of APOC and then demonstrate its utility as a framework for comparing agent architectures by showing how four main architectures, SOAR, PRODIGY, ICARUS, and GRL, can be translated into

[1] Being conceptually parsimonious is critical, for otherwise the framework may end up being as complex as any of the higher, universal programming languages in which agent architectures are defined (obviously, such a framework would defeat its purpose).

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

it. The subsequent discussion will briefly sketch how such translations can be used to compare architectural mechanisms and possibly integrate different architectural concepts within one combined architecture.

A Brief Overview of the APOC Framework

APOC is an acronym for "Activating-Processing-Observing-Components", which summarizes the functionality on which the APOC agent architecture framework is built: heterogeneous computational units called "components" that can be connected via four types of communication links.

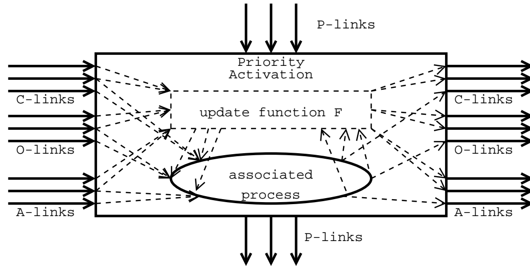

Figure 1: The basic structure of an APOC component showing bundles of incoming and outgoing links of each of the four types as well as the priority and activation levels together with the update function F and the associated process.

The four link types defined in APOC are intended to cover important interaction types among components in an agent architecture: the *activation link* (A-link) allows components to send data structures to and receive data structures from other components; the "observation link" (O-link) allows components to observe the state of other components; the "process control link" (P-link) enables components to control and influence the computation taking place in other components, and finally the "component link" (C-link) allows a component to instantiate other components and links among them.

Each *component* in APOC has an activation and a priority level and can receive inputs from and send outputs to other components via any of its links. Inputs (from incoming links) are processed and outputs (to outgoing links) are produced according to an *update function* F, which determines the functionality that the component implements, i.e., the mapping from inputs and internal component states (e.g., the current activation and priority values) to outputs and updated internal component states (e.g., new activation and priority values). The update function thus provides the specification for a computational process that, in an instantiated component, continuously updates the component's overall state. The particular algorithm for implementing the update function F has to be defined separately for each component type employed in an architecture. Figure 1 summarizes the basic structure of an APOC component.

Different from components in other formalisms such as *schemas* in *RS* (Lyons & Arbib 1989) or the *augmented finite state machines* (AFSMs) in the subsumption architecture (Brooks 1986), an APOC component can also have an "associated process" (in addition to the computational process updating the state of the component), which it can *start*, *interrupt*, *resume*, or *terminate*. The associated process can either be a physical process external to the architecture (e.g., a process controlling the motors in a wheeled robot), or a self-contained computational process that takes inputs from the component and delivers outputs to it (e.g., a process implementing an image analysis algorithm that takes an image and returns a high-level representation of all three dimensional objects found in the image).

APOC serves at least three distinct roles: (1) as an analysis tool for the evaluation of architectures, (2) as a design tool for developing architectural components, and (3) as a platform for the definition of agent architectures.

APOC is a useful tool for the analysis and comparison of agent architectures because it can express any agent architectures in a unified way (e.g., cognitive architectures such as SOAR, ACT-R, and others, as well as behavior-based architectures such as subsumption, motor schemas, situated automata, etc.). Furthermore, APOC has a notion of *cost* defined for components and links that allows for the systematic assessment of "structural cost" and "processing cost" of the whole instantiated architecture at runtime. Consequently, it is possible to analyze properties of architectures and their subarchitectures (e.g., the action selection mechanism in Maes' ANA architecture (Maes 1989) requires global control despite some claims that it uses only local mechanisms) and compare their trade-offs with respect to some particular function (e.g., two different architectures implementing an "target-finding task" can be compared with respect to their performance-cost ratio).

As a design tool, APOC allows for the definition of a large variety of different mechanisms within the same architecture, hence concepts from one formalism can often be transferred to another by virtue of a unified representation in APOC (e.g., semantic nets, neural nets, conditions-action rules, or conceptual hierarchies can all be defined in a similar way). It is thus possible to study different designs of mechanisms (e.g., how to do behavior arbitration or how to implement Sloman's global alarms, or how to actively manage finite resources at the architecture level). Since algorithms are in general implemented in APOC components, APOC automatically yields a way of distributing computations in terms of asynchronous computational units and communication links among them. Furthermore, the resource requirements and computational cost of the architecture can be determined and compared to other architectures implementing different algorithms for the same task.

Finally, instead of transcribing and modeling other

architectures, APOC can be also used to define new concepts and implement new architectures directly (e.g., we coined the term "dynamic architectures" (Scheutz & Andronache 2003b) for architectures that modify themselves over time; for this an architecture has to be capable of modifying its own description, e.g., as part of a learning process, which is possible in APOC). For more details on APOC as well as the APOC development environment ADE, which is based on the APOC framework and allows for the direct implementation of APOC specifications in JAVA, see (Scheutz & Andronache 2003a; 2003b; Andronache & Scheutz 2004b; 2004a).

Translation other Architectures into the APOC Framework

Components in APOC can vary with respect to their complexity and the level of abstraction at which they are defined. They could be as simple as a connectionist unit (e.g., a perceptron) or as complex as a full-fledged condition-action rule interpreter (e.g., SOAR or PRODIGY). Hence, there are typically many ways in which any given mechanism or architecture can be expressed in APOC.

One interesting way of utilizing the APOC component model, especially in the context of translating other architectures, is to view a component (i.e., the process updating the state of a component based on the definition in the update function F) as a *process manager* of the component's associated process. This construal makes it possible to separate information concerned with architecture-internal processing, i.e., *control information*, from other information (e.g., sensory information that is processed in various stages).

In the following, we will sketch generic translations of four main architectures, GRL, ICARUS, PRODIGY, and SOAR, into the APOC framework.

GRL

GRL is a functional language for behavior-based systems (based on the programming language SCHEME), which makes the generalization of treating arbitration mechanisms as higher-level procedures. The GRL translation into APOC can be seen below:

- Each procedure maps onto an APOC component.
- A-links are used for data transfer from the environment to behaviors and among behaviors.
- O-links are used for data transfer from behaviors to arbitration components.
- Arbitration schemes are defined as APOC components. These components receive inputs from the arbitrated components, process them according to internal rules and produce overall outputs for the system. The internal rules can implement any arbitration mechanism: competitive, cooperative, or a combination.
- Sequencing can be obtained through the use of "flag" variables within components. Other components, which depend on prior computation, use O-links to observe the flag variables and only start their computation once the flag variable has changed to a predetermined value.

The GRL definition of a behavior is as follows:

```
(define-group-type behavior
  (behavior act-level motor-vector)
  (activation-level act-level)
  (motor-vector motor-vector))
```

The weighted sum operator which would be required for a motor-schema implementation is:

```
(define-signal (weighted-sum . behavs)
  (apply + (weighted-motor-vec behavs)))

(define-signal (weighted-motor-vec beh)
  (* (activation-level beh)
     (motor-vector beh)))
```

In APOC, with each behavior and the arbitration algorithm being embedded in separate components, this systems creates the structure (left in Figure 2), where b_1 to b_n are the behaviors used in the system.

ICARUS

ICARUS is a cognitive architecture with the capability of learning hierarchical skills. The ICARUS representation of skills is related to both production rules and STRIPS operators. ICARUS divides memory space into conceptual memory and skill memory. Conceptual memory is the residence of states, such as a description of a desk, and relations, such as a description of "on top of." Skill memory contains the system's knowledge about actions, such as "put object A on object B." ICARUS also divides memory into long-term and short-term memories. Each element in long term memory is a symbolic description with an associated numeric function which computers the value of that description in terms of the current sensory value reading. Each element in short term memory is an instance of a long term memory element.

Long-term Conceptual Memory Long-term conceptual memory contains definitions of concepts, such as car, and of relations, such as "in left lane." To translate ICARUS long term memory into APOC we use the following rules:

- Each concept is represented as an APOC component. The characteristics of each object are embedded within its equivalent component. For example, a component which represent a numeric concept, such as speed or distance, embeds the arithmetic function which computes the quantity associated with that concept within its update function F.
- Higher-level concepts, which have other concepts as positive or negative preconditions check for the achievement of those preconditions in F.

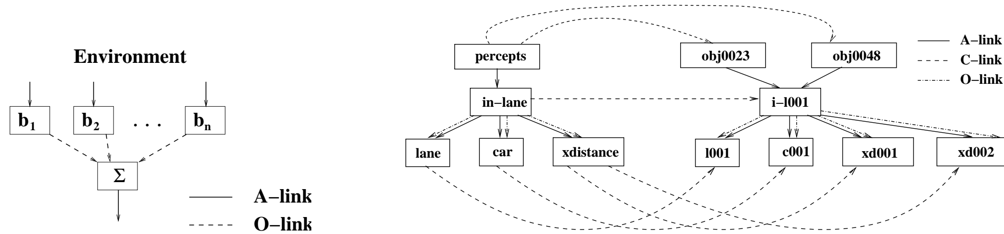

Figure 2: An example of an APOC translation of a GRL structure (left) and an ICARUS structure (right).

- Higher-level concepts whose preconditions need to be checked against specific objects connect via A-links to the lower-level concepts representing those preconditions. The links are used to send across the parameters to which the tests of lower level concepts are applied.
- Higher level concepts connect via O-links to lower level concepts to ascertain whether their positives and negatives are satisfied.
- Higher level concepts can create "instances" of their knowledge: the Lane-To-Right component can instantiate a Lane-To-Right-Instance, identifying a particular line found to the right. Therefore, concepts are connected to each of their instances through a C-link.

Long-term Skill Memory Long-term skill memory contains knowledge about ways to act and achieve goals, such as how to overtake a car moving slowly ahead. To map long-term skill memory to APOC we use the following rules:

- Each skill is represented as an APOC component.
- Each skill connects through O-links to sub-skills/rules in order to verify their completion.
- Each skill component connects through O-links to concepts in order to verify that the pre-requirements (start:) and the continuing requirements (required:) are met (if necessary).
- Distinctions between ordered: and unordered: are implemented in the update function, F, or in the associated process.
- The evaluation function for a skill decomposition is defined in the update function, F.

Short-term Conceptual Memory Each instance of a long term concept which can be created based on current sensory information is represented as an APOC component.

Short-term Skill Memory This memory contains the skills the agent intends to execute. Each element represents an instance of a long-term memory skill and has concrete arguments. Each element is represented as an APOC component and is linked through A-links and O-links to the percepts and short-term memory concepts which form its arguments. In primitive skills, actions are mapped onto effectors.

Perceptual Buffer Percepts are represented as APOC components. For example, the literal (#speed car-007 20.3) is represented as a component. A concept connects via O-links to percepts in order to read their values.

To learn hierarchical skills in APOC, an analysis component is connected to all skills and checks preconditions. This component constructs the precondition hierarchy, as described in (Langley, Cummings, & Shapiro 2004). In APOC terms, a new component is created for each common precondition of two or more skills and it connects to the skills which have the common precondition. The link structure thus created determines the memory hierarchy and therefore the skill hierarchy.

A simple example of a translation is presented below. Consider the function

```
(in-lane (?car ?lane)
    (lane ?lane ?left-line ?right-line)
    (car car?)
    (\#xdistance ?car ?left-line ?dleft)
    (\#xdistance ?car ?right-line ?dright)
    (< ?dleft 0) (> ?dright 0))
```

An APOC translation following the above rules can be seen on the right in Figure 2. It should be noted that there are two instances of the #xdistance concept in this description. We chose this implementation in order to illustrate the flexibility of APOC and to show how an APOC-based architecture could make use of the facilities available in the system, in this case assuming there are enough resources to duplicate a functional unit, in order to maximize system performance. The instance of in-lane, i-l001, sends object data to the lane instance component, l001 through the A-link. Then it observes through the O-link to see whether the object sent is a lane. Similar processes take place with the other instantiated components. For this example, we left the magnitude comparisons, $<$ and $>$, in the update function, F, of i-l001, due to the simplicity of the functionality represented.

PRODIGY

PRODIGY is a mixture of planning and learning, consisting of a general purpose planner and several learning systems (see the left of Figure 3).

Knowledge in this system is represented in terms of operators. A central module decomposes a given problem into subproblems. In PRODIGY, a planning domain is specified as a set of objects, operators, and inference rules which act on those objects. To translate a PRODIGY system to APOC, we use the following rules:

1. Each operator type maps onto an APOC type component.
2. Each bindings component (instantiated operator) maps onto an APOC component.
3. Each object in the knowledge base maps onto an APOC component.
4. Each control rule maps onto an APOC component.
5. Each goal maps onto an APOC component.
6. Goal and bindings components have a cost field and a computation of cost implemented in the update function, F. Operator types do not have costs; in APOC types are not part of the traversed graph, resulting in a slightly different, though functionally equivalent, structure from the graph described with PRODIGY (shown on the right in Figure 3).
7. Wherever applicable, each component computes its own cost in the update function, F. The cost of the top component then represents the cost of the plan.
8. Operators have O-links to object representations or other operators, which determine the meaning of the operator.
9. The central module is implemented in its own APOC component, with connections to all other components in the system. This component has P-links to the Back-Chainer and Operator-Application to determine which executes at each step. Recursion is obtained by repeated application, e.g., by cycling through the architecture.
10. The Back-Chainer is an APOC component.
11. The Back-Chainer can create bindings components and pass as arguments to the new components the ids of objects/operators to which the new instantiated operators should connect.
12. The Operator-Application is implemented in its own APOC component, which has O-links to all objects and operators. This component is connected through an O-link to the Back-Chainer in order to observe the state of the tail plan.
13. Each of the other elements of the system, such as EBL and Hamlet, are separate components, which connect to PRODIGY through both O-links and A-links. These components connect only to those components required for their functionality. For example, QUALITY receives the current plan from PRODIGY and attempts to refine the plan and generate new control rules which will allow PRODIGY to generate better plans. QUALITY may need to connect to all components which are part of the plan, or it may simply operate on an abstract representation of the plan and system state, created by PRODIGY.
14. An external user can create a plan using APOC components. The components which need to use information from this plan can connect to the user-created plan through O-links.

The initial state of a problem is the initial state of the system.

SOAR

The structure of the SOAR architecture, as described in (Laird, Newell, & Rosenbloom 1987), can be seen in Figure 4. Five main components are present in SOAR:

1. A *Working Memory*, which is a container with information about *Objects* (goals and states of the system), *Preferences* (structures indicating the acceptability and desirability of objects in a particular circumstance), and a *Context Stack*, which specifies the hierarchy of active goals, problem spaces, states and operators.
2. A *Decision Procedure*, which is a function that examines the context and preferences, determines which slot in the context stack requires an action (replacing an object in that context), modifying the context stack as required.
3. A *Working Memory Manager*, which determines which elements of the working memory (contexts and objects) are irrelevant to the system and deletes them
4. A *Production Memory*, which is a set of productions that can examine any part of working memory, add new objects and preferences to it, and add new information to existing objects.
5. A *Chunking Mechanism*, which is a learning mechanism for new productions

Since descriptions of architectures in APOC can be done at various levels of detail, there are several possible translations of SOAR to APOC, with varying levels of detail hidden in the process associated with each APOC component. However, to better make use of the intrinsic power of the framework, we describe SOAR at a fairly detailed level. Two types of links can be distinguished in Figure 4. Some links have associated operations, which denote the fact that through those links elements can be either created (+) or deleted (-). The other links are simply data transfer links. The latter link type translates directly onto the APOC O-link. Thus, the following O-link connections occur in an APOC implementation of SOAR.

- The *Chunking Mechanism* is connected to all *Preferences*, *Objects*, and the *Context Stack*. The

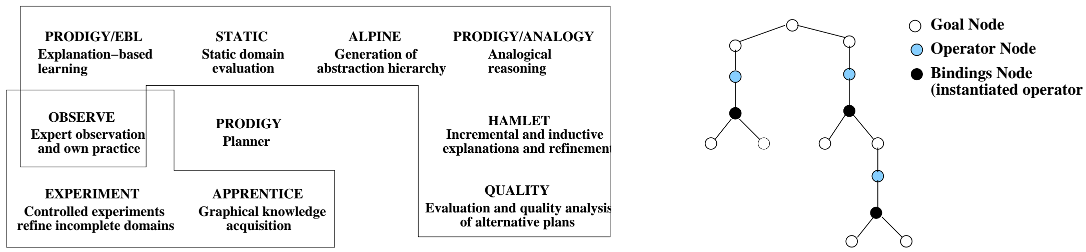

Figure 3: The PRODIGY system with its component parts (left) and the structure of a PRODIGY plan (right).

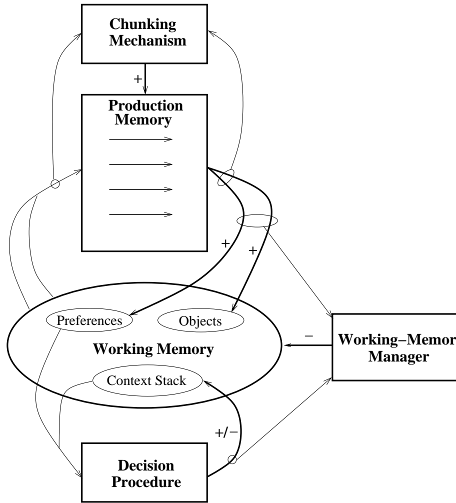

Figure 4: The SOAR architectural structure

items observed are the working memory elements created in the subgoal being processed. Thus, instead of choosing to describe the *Working Memory* as an APOC component, we describe *Preferences*, *Objects* and *Goals* as the basic components of an APOC-based implementation of SOAR, imposing the structure of the SOAR architecture through APOC links. Similarly, the production memory is mapped at the level of each production as an APOC component.

- The *Chunking Mechanism* is connected to all *Productions* in order to trace the productions fired during the subgoal being processed. Thus, the item observed is a boolean variable indicating the status of a production.
- The *Working Memory Manager* is connect to all *Productions*. Observed items are preferences and objects produced, whose information can be gathered from the production such that a direct O-link to those objects can be created.
- The *Working Memory Manager* is connected to the *DecisionProcedure*. Observed items are the contexts produced, whose information can be gathered such that a direct O-link to those objects can be created.
- The *Working Memory Manager* is connected to all *Preferences*, *Objects*, and the *Context Stack*. The contents of each context of the *Context Stack* are compared against the identifiers of elements of the *Preferences* and *Objects* sets. Thus, the items observed are the elements of contexts and the identifiers of objects.
- The *Decision Procedure* is connected to all *Preferences* and the *Context Stack*. The contents of each context of the *Context Stack* are observed and processed. Preferences are observed for content and checked for matches against the context currently being processed.

The creation/deletion functionality of SOAR maps directly onto the APOC C-link. The creation process may require additional information to be passed to the newly created node (e.g., the conditions in which a new production fires need to be sent to the production when a generic production is created and objects need to be given identifiers). An A-link is then created through the C-link and used for information passing.

The deletion process requires that information is known about the situation state of working memory (e.g., determining if an object is used in any context on the context stack). This information is retrieved in APOC through the O-link mechanism. Thus, a C-link from the "Decision Procedure" or "Working Memory Manager" to a preference, object, or a goal create an O-link upon their creation and this link is thereafter used to observe that entity as described above.

Discussion

In the previous section we have indicated how four major architectures could be expressed in the APOC framework. Here we will briefly discuss how these translations could be used for a comparison of different architectural features and possibly their integration within one architecture.

Comparisons of Agent Architectures

First and foremost, APOC translations of architectures allow for a direct comparison of their structural architectural mechanisms as they have to be formulated within the same APOC link model. Depending on

the level of detail of the translation, this might only allow for the comparison of the very general control flow within any two given architectures, or it might reveal a very detailed control and information flow among many components that might be useful to analyze and compare performance bottlenecks and other limitations across architectures. In particular, for the performance of an architecture on a given task, APOC translations can be used to assess the *cost induced by an architecture*, which can be defined in terms of the cost associated with its *structures*, its *processes*, and the *actions* that can be performed on it (e.g., modifications of the layout, instantiation of new data structures, etc.):

- *Structural costs* are incurred as a result of merely having a certain component or link instantiated. They can be thought of as maintenance costs that are associated with any work that needs to be done to keep the object up to date.
- *Process costs* are those associated with running processes. They include computational costs, and possibly the costs of I/O and other such operations. Typically process costs will be proportional to the complexity of the computation performed by the process.
- *Action costs* are those associated with primitive operations on the architecture (such as instantiating a new component or link, or interrupting a process). Each action has a fixed cost, making the computation of action costs a simple matter of assessing the associated cost whenever the action is executed.

The notion of cost induced by an architecture is then inductively defined in terms of these three basic cost types. For rule-based systems, this cost can be determined at a very fine-grained level if each goal, each rule, and each concept (where appropriate) is directly modeled by a separate APOC component (e.g., as in the above translation for ICARUS).

Assuming that the respective costs of two architectures can be assessed for the same task, their *performance-cost tradeoff* can then be determined if their respective performances on the task (e.g., based on a task-dependent *performance measure*) are known. In the ideal case, it will be possible to abstract over the individual task and make some general statements about the mechanisms, either in absolute terms, or relative to mechanisms in other architectures (e.g., the arbitration mechanism in architecture A, all other things being equal, makes better decisions about what action to execute that arbitration mechanism B).

We believe that the *performance-cost tradeoff* is an important measure for the evaluation of architectures and/or architectural mechanisms, as it is one way to assess their utility based on the involved processing, structural, and action costs. While *absolute performance* (i.e., performance regardless of cost) is a good way to evaluate systems in principle or in cases where computational resource requirements are of no or only minor concern, relative performance is what matters in practice, especially under severe resource constraints. For an embedded, autonomous system, for example, the absolute performance (e.g., how close it is to a perfect solution) will often matter much less than the actual energy and resource expenditure (e.g., in the case of a rover roaming the surface of a distant planet). Hence, specific mechanisms that are very "efficient" with respect to some notion of cost while still giving rise to acceptable performance might by preferable over much more costly universal mechanisms (e.g., universal rule-based systems).

Integration of Architectural Mechanisms

Translations of architectures in APOC can be useful beyond the comparison of agent architectures, as they might suggest ways of combining and integrating architectural mechanisms from different architectures within one common architecture. A first simple example would be the combination of GRL behavior-based procedures with any of the other three architectures described above. Such a combination would allow for a direct way of connecting symbolic reasoning engines to embodied, robotic agents. In the simplest case, GRL could be used to implement the mechanism by which a goal of the system, for example, a "move left goal" (possibly a subgoal of the "pass the car in front" goal) could be achieved in the real world (e.g., a schema-based car navigation system).

More generally, GRL could be used to implement "alarm mechanisms" in any of the three other architectures, which achieve the high reactivity necessary for the survivability of autonomous, embodied systems like robots. This could be done in at least two ways: (1) the other architectures are implemented in the same way as "behaviors" in GRL; that way behavior arbitration in GRL would allow them to become active and remain in control of the agents' effectors as long as no emergency requires special emergency behaviors that would override them. Alternatively, (2) right-hand sides of rules that contain effector commands could be directly connected to arbitration schemes via the GRL signaling mechanisms, thus allowing for a tight integration of behavior arbitration and the rule-based system (this integration, moreover, allows for the dynamic modification of arbitration schemes through the rule interpreter, a possibility we are currently investigating further).

Another example of a whole category of potential combinations would be the transfer of learning mechanisms from one cognitive architecture to another:

- The chunking mechanism in SOAR could be used as a learning mechanism in the context of PRODIGY.
- The reverse can also be employed: the goal stack structure in a SOAR-based system could be passed to one of the analysis tools in PRODIGY.
- The hierarchical structure learned by ICARUS can be adapted and used in a SOAR system within the context of APOC, exploiting similarities in the hierarchical organization of the goal structure.

In general, any *architectural mechanisms* from different architectures that are by definition not knowledge-based can be integrated *as such* within one architecture (i.e., at the level of the architecture–for an example, see (Scheutz & Andronache 2003b)). Among the advantages of such an integration are:

- a richer architecture with more built-in capabilities, which allows agent designers to use these features and algorithms directly without having to re-implement them in terms of other architectural mechanisms (e.g., in terms of condition-action rules)
- a processing speedup (as architectural mechanisms can be directly implemented in the underlying virtual machine without the need for intermediate interpretation)
- the possibility of integrating existing systems and thus reusing functional components that have proven successful (e.g., a system A utilizing architectural mechanisms M_A and a system B utilizing architectural mechanism M_B could be run as part of one architecture)

Conclusion

The APOC framework is still in its infancy. Yet, we believe that it already demonstrates the potential utility of agent architecture frameworks for the design and use of future agent architectures.

We are currently working on a formal specification of APOC that will eventually integrate methods from model checking, process algebra, Petri nets, and hybrid systems to provide a formalism for APOC that has provable properties (e.g., the correctness of interactions among components, the timely routing of data through a network of components, etc.) and allows for the application of the many formal tools used by software engineers to verify the functionality of their code. We believe that at least the possibility of formal verification of properties will be crucial for the development of future agent architectures, especially in the context of complex, autonomous, intelligent agents.

References

Albus, J. S. 1992. A reference model architecture for intelligent systems design. In Antsaklis, P. J., and Passino, K. M., eds., *An Introduction to Intelligent and Autonomous Control*, 57–64. Boston, MA: Kluwer Academic Publishers.

Anderson, J. R.; Bothell, D.; D., B. M.; and Lebiere, C. An integrated theory of the mind. To appear in Psychological Review.

Andronache, V., and Scheutz, M. 2004a. Ade - a tool for the development of distributed architectures for virtual and robotic agents. In *Proceedings of the Fourth International Symposium "From Agent Theory to Agent Implementation"*.

Andronache, V., and Scheutz, M. 2004b. Integrating theory and practice: The agent architecture framework apoc and its development environment ade. In *Proceedings of AAMAS 2004*.

Arkin, R. C., and Balch, T. R. 1997. Aura: principles and practice in review. *JETAI* 9(2-3):175–189.

Arkin, R. C. 1989. Motor schema-based mobile robot navigation. *International Journal of Robotic Research* 8(4):92–112.

Bonasso, R. P.; Firby, R.; Gat, E.; Kortenkamp, D.; Miller, D.; and Slack, M. 1997. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence* 9(1).

Brooks, R. A. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation* 2(1):14–23.

Horswill, I. 2000. Functional programming of behavior-based systems. *Autonomous Robots* (9):83–93.

Laird, J. E.; Newell, A.; and Rosenbloom, P. S. 1987. SOAR: An architecture for general intelligence. *Artificial Intelligence* 33:1–64.

Laird, J.; Rosenbloom, P.; and Newell, A. 1986. Chunking in soar: The anatomy of a general learning mechanism. *Machine Learning* 1:11–46.

Langley, P.; Shapiro, D.; Aycinena, M.; and Siliski, M. 2003. A value-driven architecture for intelligent behavior. In *Proceedings of the IJCAI-2003 Workshop on Cognitive Modeling of Agents and Multi-Agent Interactions*.

Langley, P.; Cummings, K.; and Shapiro, D. 2004. Hierarchical skills and cognitive architectures. In *Proceedings of the Twenty-Sixth Annual Conference of the Cognitive Science Society*.

Lyons, D. M., and Arbib, M. A. 1989. A formal model of computation for sensory-based robotics. *IEEE Transactions on Robotics and Automation* 5(3):280–293.

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

Scheutz, M., and Andronache, V. 2003a. APOC - a framework for complex agents. In *Proceedings of the AAAI Spring Symposium*. AAAI Press.

Scheutz, M., and Andronache, V. 2003b. Growing agents - an investigation of architectural mechanisms for the specification of "developing" agent architectures. In Weber, R., ed., *Proceedings of the 16th International FLAIRS Conference*. AAAI Press.

Veloso, M.; Carbonell, J.; Pérez, A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating planning and learning. *Journal of Experimental and Theoretical Artificial Intelligence* 7(1).