1994-17

Proceedings of the

Twenty-Seventh Hawaii International Conference on System Sciences

Volume III:

Information Systems: Decision Support and Knowledge-Based Systems

Edited by Jay F. Nunamaker, Jr. and Ralph H. Sprague, Jr.

Sponsored by:
The University of Hawaii
The University of Hawaii College of Business Administration

In cooperation with:
The IEEE Computer Society
Association for Computing Machinery

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

346-0801

Towards a Unified AI Formalism

Deepak Kumar
Dept. of Mathematics
Bryn Mawr College
Bryn Mawr, PA 19010
dkumar@cc.brynmawr.edu

Susan Haller
Dept. of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260-2000
haller@cs.buffalo.edu

Syed S. Ali
Dept. of Computer Science
Southwestern Missouri State University
Springfield, MO 65804
ssa231f@csm560.smsu.edu

Abstract

This paper presents a unified approach to building Intelligent architectures. Our approach relies on making some semantic, ontological, as well as architectural commitments. Semantically, we commit ourselves to principles governing the nature of the entities represented by the knowledge representation formalism, and the relationships between the faculties of reasoning, acting, and natural language understanding. More specifically, we concentrate on the nature and representation of entities, beliefs, actions, plans, and reasoning and acting rules. Based on these principles we present the design of an integrated AI architecture that has a unified knowledge representational formalism as well as a unified reasoning and acting component. The design of the architecture is based on object-oriented principles. We also show that such a design is amenable to implementation in a concurrent object-oriented programming paradigm. We demonstrate the advantages of our approach using several examples from our work.

1 Introduction

In this paper, we take a different approach towards the integration process. We start with the premise that regardless of the individual AI subfield involved, all intellectual activity involves representation of knowledge and reasoning. Thus, all knowledge required to fulfill various tasks (NLU, planning and acting, reasoning, etc.) is to be represented in a common KR formalism. We also take a unified view of reasoning and acting and incorporate that into a single acting and inference engine called a *rational engine*. In what follows, we will identify the basic KR principles, the relationship between acting and inference, and use that to present the design of a new unified AI formalism called the Object-based (or OK) Architecture. The design of the OK architecture employs an object-oriented methodology that is also amenable to concurrent implementations. We also present several examples of the kinds of behavior exhibited by computational agents modeled using this paradigm.

2 The OK BDI Architecture

We begin by pointing out that we are interested in modeling computational rational agents. The behavior of these agents must be driven by their beliefs, desires, and intentions. AI architectures that enable modeling of such agents have come to be called BDI architectures [8]. Most work on BDI architectures has as its underlying motivation the need to examine the feasibility of modeling practical reasoning in resource bounded agents [18, 6]. To that effect these architectures attempt to integrate reasoning, planning, and acting capabilities. Our work encompasses the practical reasoning issues as well as additional issues involved in NLU and KRR. Additionally, the architecture we are about to describe is designed using the principles of object-oriented programming. Such a design naturally exploits all the benefits of object-oriented design, namely, extendibility, and amenability to concurrency. We will return to these in a later section. First we will present our underlying commitments.

3 Commitments

In the design of the OK architecture we have made several commitments. Most of them deal with representational issues. The representational formalism is designed to facilitate the representation of *intensional concepts* [22]. All entities represented in the KR formalism are intensional in that it will be possible to have two or more instances that denote as many entities and yet may correspond to exactly one extensional object (this is the "Principle of Fine-Grained Representation" [22]). Thus, the identity conditions for two entities will depend upon their manner of representation. Additionally, there is no requirement that the entity actually exists in the world (e.g, unicorns) or that it is possible (e.g., square circles) (this is the "Principle of Displacement" [22]). In general, we would like the entities represented to be extensional as well as intensional. For example, the name of the entity may be one way of linking up an entity to the external world. In the OK Formalism, we also enforce the Uniqueness Principle of representing intensional concepts—there is a one-to-one correspondence between terms representing entities and intensional concepts [22]. The Uniqueness Principle will be applied to all entities in the formalism.

In previous work we have argued for a unified view of acting and inference [13]. We have argued that inference be treated as a kind of acting (mental acting). Reasoning itself can be viewed as a sequence of *actions* performed in applying inference rules to derive beliefs from other beliefs. Thus, an inference rule can be treated as a rule specifying an *act*—that of believing some previously non-believed proposition (i.e., the believe act is implicitly included in the semantics of the propositional connective). This leads us to make two commitments in the OK architecture—that there be a single operating module that is responsible for reasoning and acting behavior; and that there be a proper semantic distinction between entities that represent beliefs,

1060-3425/94 $3.00 © 1994 IEEE

acts, rules for reasoning, and rules for acting (plans). The first, a unified module for acting and inference, is called a *rational engine* (as opposed to an *inference engine*). The second leads to an ontological commitment on the part of the KR formalism. In what follows, we first describe the OK Formalism, that is an object-oriented, intensional, propositional knowledge representation formalism. Then we will present the design of the OK Rational Engine that is responsible for implementing the unified view of acting and inference. Together, they constitute the OK BDI architecture (see [13] for a detailed description).

4 The OK Formalism

The basic class hierarchy of the OK Formalism is depicted in Figure 1. A conceptual entity is anything a cognitive agent can think about. In intensional representational formalisms, conceptual entities include individuals, variables, acts, and beliefs.

4.1 Conceptual Entities

The class `Conceptual Entity Term` is the topmost class in the OK Formalism. All instances of this class as well as all instances of its subclasses are terms of the OK formalism that denote intensional entities in the domain of discourse of the modeled agent. All instances of this class are uniquely identified by a label which is denoted by an uppercase letter followed by a number. The letters help identify the kind of entity represented. For example, `E1, E2, ...` denote basic (un-named) entities, `I1, I2, ...` denote named individuals, `B1, B2, ...` denote propositions, `A1, A2, ...` denote acts, and `V1, V2, ...` denote variables.

4.2 Named Individuals

The class `Individual Term` is comprised of instances that represent named entities in the domain of discourse. For example, a term that denotes the named individual [JOHN] is written as `JOHN`.

4.3 Variables

Instances of the class `Variable Term` denote arbitrary conceptual entities—individuals, beliefs, acts, etc. Variables are useful for representing generic individuals (e.g., 'a thing'); general propositions (e.g., 'a thing that is a block'); generic propositions (e.g., 'something that John believes'); and generic acts (e.g., 'pick up a thing'). We have given special attention to variables that represent indefinite noun phrases and anaphora that models their use in natural language understanding. We have arrived at a non-atomic representation of variables (also called *structured variables*) [3, 4, 5]. The structure of the variable includes complex internalized constraints (that includes, but is not limited to, the type of the entity a variable may be bound to) and internalized quantifier structures. The semantics of structured variables is an augmented (by the addition of arbitrary individuals) semantic theory based on [7, 22]. The use of such object-oriented variables leads to much simpler representations for tasks as disparate as representing natural language sentences and operators in a planning formalism. Additionally, structured variables allow a form of subsumption (a more general, or less restricted, variable subsumes another more restricted variable of the same sort) that corresponds directly to similar natural language reasoning based on description subsumption. We will illustrate the utility of structured variables with a natural language processing example later.

4.4 Propositions

As mentioned above, propositions are also treated as conceptual entities. Specific instances of the class `Proposition Term` represent propositions. Asserted propositions form agent's beliefs. It is possible for the modeled agent to have representations of propositions that it may not necessarily believe. Since all instances of the class `Conceptual Entity Term` are also terms, this enables the agent to have beliefs about its own beliefs as well those of others. The agent's "mind" is modeled as a set of *belief spaces*. Each belief space consists of the propositions believed by the agent either explicitly (hypotheses) or via derivation. The instances representing propositions also contain information typically employed by a truth-maintenance system (TMS). In the OK architecture, we necessitate the presence of a TMS because it facilitates several advantages in the representation of actions and plans [15]. Also, it enables the agent to store results of its inferences in order to make future recalls more efficient. For example, an instance of an object that represents a proposition "A is a block" would be written as

`B1! Isa(A, BLOCK)` $\langle$`HYP`$, \{$`B1!`$\}, \{\}\rangle$

where `Isa` is a subclass of the class `Proposition Term`, `A` and `BLOCK` are instances of the class `Individual Term`, `B1` is the label of the instance, and the proposition as shown is believed by the agent (denoted by writing the exclamation following the label) as a hypothesis (`HYP`). See [15, 13] for more details. The syntax of writing propositions is not unique and is defined by the "print-methods" of the associated class. We are claiming that such object-centered representations are inherently canonical in that they can be translated into another KR formalism simply by supplying a different print-method for each class. In fact, in [13] we have shown that the OK Formalism is isomorphic to a semantic network formalism, SNePS [22].

4.5 Acts

An *act* is a mental concept of something that can be performed by various actors at various times. This is also important for plan recognition (facilitating a plausible conclusion that an agent performing an act could be acting to fulfill some plan). By the Uniqueness Principle, a single act must be represented by a single object even if there are several different objects representing beliefs that several different agents performed that act at different times. Acts are represented by objects that are instances of the subclasses of the class `Act Term`. Acts can be *primitive* or *complex* (not shown in the figure). A primitive act has an effectory procedural component which is executed when the act is performed. Complex acts have to be decomposed into plans.

Our present model of acting is based upon a state-change model (see [14]). We identify three types of states—

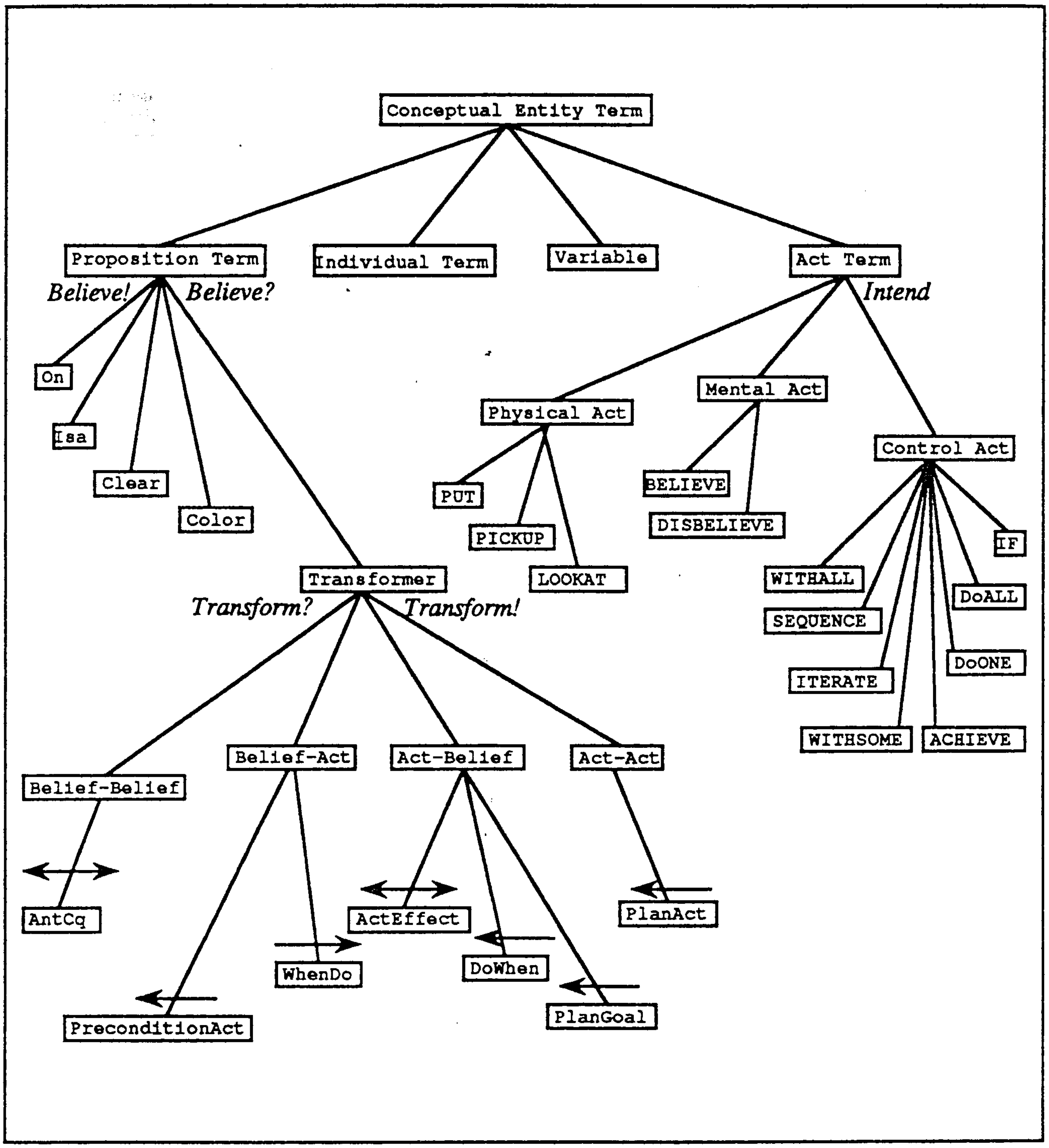

Figure 1: The class hierarchy of the OK Formalism.

external world states, mental states (belief space), and intentional states (agent's current intentions). Accordingly, we identify three classes of actions—*physical actions, mental actions*, and *control actions* that bring about changes in their respective states.

4.5.1 Physical Acts

Physical acts are domain-specific acts that affect the outside world. For example, if the agent has an arm and is asked to pick up an object, the arm actually moves to the object, grasps it, and then lifts it up. Depending on the set of interfaces provided to the agent we need coresponding actuators to enable the carrying out the action in the external world. This is done by specifying the effectory component of the action by writing procedures that access the actuator interface. PICKUP is a physical action in a blocksworld. For example, the act of picking up a block named A is represented as an object instance that is written as

$$\mathtt{A1 : PICKUP(A)}$$

4.5.2 Mental Acts

Beliefs of the agent may change when actions are performed or an inference is carried out. We have BELIEVE and DISBELIEVE as mental actions whose objects are beliefs. Simply adding and removing propositions from the belief-space poses the danger of leaving the belief space inconsistent. This is especially true when all derived beliefs are also added to the belief space. The presence of a TMS solves this problem. Typically, TMSs provide explicit operations for adding and deleting new beliefs to and from a belief space. These operations, in addition to the asserting/deleting propositions, perform consistency checks in order to guarantee a consistent resulting belief space. Thus, in the presence of a TMS, the effectory components of the two mental actions should be implemented using the appropriate TMS operations. This strategy implements the *extended STRIPS assumption* [8].

4.5.3 Control Acts (Plans)

Plans, in our ontology, are also conceptual entities. However, we will not define a separate class for them as they are also acts—albeit control acts. Control acts, when performed, change the agent's intentions about carrying out acts. Our repertoire of control actions includes *sequencing* (for representing linear plans), *conditional, iterative, disjunctive* (equivalent to the OR-splits of the Procedural Net formalism [19, 23]), *conjunctive* (AND-splits), *selective*, and *achieve* acts (for goal-based plan invocation). These are summarized in Table 1. These control acts are capable of representing most of the existing plan structures found in traditional planning systems (and more). We should emphasize, once again, that since plans are also conceptual entities (and represented in the same formalism) they can be represented, reasoned about, discussed, as well as followed by an agent modeled in this architecture.

4.6 Transformers

In addition to standard beliefs that an agent is able to represent, we also define a special class of beliefs called *transformers*. A *transformer* is a propositional representation that subsumes various notions of inference and acting. Being propositions, transformers can be asserted in the agent's belief space; they are also beliefs. In general, a transformer is a pair of entities—$(\langle a\rangle, \langle b\rangle)$, where both $\langle a\rangle$ and $\langle b\rangle$ can specify beliefs or acts. Thus, when both parts of a transformer specify beliefs, it represents a reasoning rule. When one of its parts specifies beliefs and the other acts, it can represent either an act's preconditions, or its effects, or a reaction to some beliefs, and so on. What a transformer represents is made explicit by specifying its parts. When believed, transformers can be used during the acting/inference process, which is where they derive their name: they transform acts or beliefs into other beliefs or acts and vice versa. Transformations can be applied in forward and/or backward chaining fashion. Using a transformer in forward chaining is equivalent to the interpretation "after the agent believes (or intends to perform) $\langle a\rangle$, it believes (or intends to perform) $\langle b\rangle$." The backward chaining interpretation of a transformer is, "if the agent wants to believe (or know if it believes) or perform $\langle b\rangle$, it must first believe (or see if it believes) or perform $\langle a\rangle$." There are some transformers that can be used in forward as well as backward chaining, while others may be used only in one of those directions. This depends upon the specific proposition represented by the transformer and whether it has any meaning when used in the chaining process. Since both $\langle a\rangle$ and $\langle b\rangle$ can be sets of beliefs or an act, we have four types of transformers— *belief-belief, belief-act, act-belief*, and *act-act*.

4.6.1 Belief-Belief Transformers

These are standard reasoning rules (where $\langle a\rangle$ is a set of antecedent belief(s) and $\langle b\rangle$ is a set of consequent belief(s)). Such rules can be used in forward, backward, as well as bidirectional inference to derive new beliefs. For example, a class of transformers that represent antecedent-consequent rules is called AntCq transformers. We use the notation

$$\langle a\rangle \rightarrow \langle b\rangle$$

to write them. For example "All blocks are supports" is represented as

$$\mathtt{B1!} : \forall x[\mathtt{Isa}(x,\mathtt{BLOCK}) \rightarrow \mathtt{Isa}(x,\mathtt{SUPPORT})]$$

In addition to the connective above (which is also called an or-entailment), our current vocabulary of connectives includes and-entailment, numerical-entailment, and-or, thresh, and non-derivable. Other quantifiers include the existential, and the numerical quantifiers (see [21]). Given the object-oriented design of the architecture one can define any additional classes of connectives depending on their own logical commitments.

4.6.2 Belief-Act Transformers

These are transformers where $\langle a\rangle$ is a set of belief(s) and $\langle b\rangle$ is a set of acts. Used during backward chaining, these can be propositions specifying preconditions of actions, i.e.

Control Action	Description
SEQUENCE(a_1, a_2)	The acts a_1 and a_2 are performed in sequence. Example: SEQUENCE(PICKUP(A),PUT(A,TABLE)) is the act of first picking up A and then putting it on the table.
DoONE(a_1,...,a_n)	One of the acts a_1,...,a_n is performed. Example: DoONE(PICKUP(A),PICKUP(B)) is the act of picking up A or picking up B.
DoALL(a_1,...,a_n)	All of the acts a_1,...,a_n are performed in some order. Example: DoALL(PICKUP(A),PICKUP(B)) is the act of picking up A and picking up B.
IF((p_1,a_1),...,(p_n,a_n))	Some act a_i whose p_i is believed is performed. Example: IF((Clear(A),PICKUP(A)),(Clear(B),PICKUP(B))) is the act of either picking up A (if A is clear) or picking up B (if B is clear).
ITERATE((p_1,a_1),...,(p_n,a_n))	Some act in a_i whose corresponding p_i is believed is performed and the act is repeated. Example: ITERATE((Clear(A),PICKUP(A)),(Clear(B),PICKUP(B))) the act of picking up A (if A is clear) and picking up B (if B is clear).
ACHIEVE(p)	The act of achieving the proposition p. Example: ACHIEVE(Clear(A)) is the act of achieving that A is clear.
WITHSOME(x,y,...)(p(x,y,...),a(x,y,...))	Find some x, y, etc that satisfy p and perform the act a on them. Example: WITHSOME(x)(Held(x), PUT(x, TABLE)) is the act of putting on the table something that is being held.
WITHALL(x,y,...)(p(x,y,...),a(x,y,...))	Find all x, y, etc that satisfy p and perform the act a on them. Example: WITHALL(x)(Held(x), PUT(x, TABLE)) is the act of putting on the table everything that is being held

Table 1: Summary of control actions

⟨a⟩ is a precondition of some act ⟨b⟩. We will call it a PreconditionAct transformer and write it as a predicate

PreconditionAct(⟨a⟩,⟨b⟩)

For example, the sentence "Before picking up A it must be clear" may be represented as

B26! : PreconditionAct(Clear(A),PICKUP(A))

Used during forward chaining, these transformers can be propositions specifying the agent's desires to react to certain situations, i.e. the agent, upon coming to believe ⟨a⟩ will form an intention to perform ⟨b⟩. We will call these WhenDo transformers and denote them as

WhenDo(⟨a⟩,⟨b⟩)

For example, a general desire like "Whenever something is broken, fix it" can be represented as

B100! : ∀x[WhenDo(Broken(x),FIX(x)]

4.6.3 Act-Belief Transformers

These are the propositions specifying effects of actions as well as those specifying plans for achieving goals. They will be denoted ActEffect and PlanGoal transformers respectively. The ActEffect transformer will be used in forward chaining to accomplish believing the effects of act ⟨a⟩. For example, the sentence, "After picking up A it is no longer clear" is represented as

B30! : ActEffect(PICKUP(A),¬Clear(A))

It can also be used in backward chaining during the plan generation process (classical planning). The PlanGoal transformer is used during backward chaining to decompose the achieving of a goal ⟨b⟩ into a plan ⟨a⟩. For example, "A plan to achieve that A is held is to pick it up" is represented as

B56! : PlanGoal(PICKUP(A),Held(A))

Another backward chaining interpretation that can be derived from this transformer is, "if the agent wants to know if it believes ⟨b⟩, it must perform ⟨a⟩," which is represented as a DoWhen transformer. For example, "Look at A to find out its color" can be represented as

DoWhen(LOOKAT(A),Color(A,?color))

4.6.4 Act-Act Transformers

These are propositions specifying plan decompositions for complex actions (called PlanAct transformers), where ⟨b⟩ is a complex act and ⟨a⟩ is a plan that decomposes it into simpler acts. For example, in the sentence, "To pile A on B first put B on the table and then put A on B" (where piling involves creating a pile of two blocks on a table), piling is a complex act and the plan that decomposes it is expressed in the proposition

B71! : PlanAct(SEQUENCE(PUT(B,TABLE),PUT(A,B)),
PILE(A,B))

5 The OK Rational Engine

The OK Rational Engine is an interpreter that operates on specific instances of objects representing beliefs, transformers and acts. In other words, it is the operational component of the architecture that is responsible for producing the modeled agent's reasoning and acting behavior. It is specified by three types of methods (or messages)—

Believe– A method that can be applied to beliefs for assertional or querying purposes. Consequently there are two versions—

  Believe(*p*)!– where *p* is a belief, the method denotes the process of asserting the belief, *p*, in the agent's belief space. It returns all the beliefs that can be derived via forward chaining inference/acting.

  Believe(*p*)?– where *p* is a belief, it denotes the process of querying the assertional status of *p*. It returns all the beliefs that unify with *p* and are believed by the modeled agent either explicitly or via backward chaining inference/acting.

Intend– that takes an act as its argument (*Intend*(*a*)) and denotes the modeled agent's intention to perform the act, *a*.

Transform– These methods enable various transformations when applied to transformers. Corresponding to backward and forward chaining interpretations there are two versions— *Transform*? and *Transform*!, respectively.

Notice that the first two also correspond to the propositional attitudes of belief and intention. The methods *Believe* and *Intend* can be invoked by a user interacting with the agent. New beliefs about the external world can be added to the agent's belief space by using *Believe*! and queries regarding agent's beliefs are generated using *Believe*?. These methods, when used in conjunction with transformers lead to chaining via the semantics of the transformers defined above. Figure 2 shows the general algorithm for the *Believe*? method. Notice how backchaining through the `DoWhen` and `AntCq` transformers is accomplished. It is possible for a query to result in the agent forming an intention to do some act in order to answer the query. [13] contains detailed descriptions of all the methods. We will present several examples below illustrating various features of the architecture.

The architecture also inherently provides capabilities for consistency maintenance. Each specific object that is a belief can have slots for its underlying support. The support is updated and maintained by the *Believe* methods as well as the mental actions `BELIEVE` and `DISBELIEVE` (together they form the TMS). The effectory procedures for `BELIEVE` and `DISBELIEVE` are implemented as belief revision procedures. We have found that such an integrated TMS facility simplifies several action and plan representations (see [15] for details). The *Intend* method is used to specify the fulfillment of agent's intentions by performing acts. All these methods can be specified (and specialized) for the hierarchy as well as inherited. Thus, domain specific acts (physical acts) will inherit the standard method for the agent to accomplish its intentions (i.e. the specific theory of intentionality employed), where as specializations of the *Intend* method can be defined for mental and control acts (to implement the semantics of respective acts).

6 Towards Concurrency

In this section, we will present issues relating to a concurrent implementation of the OK BDI architecture. The central idea in any object-oriented system is that objects represent logical or physical entities that are self-contained and are provided with a uniform communication protocol. These two properties facilitate orderly interactions, which tend to be a perfect ground for concurrent programming. Nelson [17] goes so far as to proclaim that "every object-oriented programming language should be concurrent in nature." In a concurrent implementation of the OK rational engine, acting and inference will be carried out by object instances sending and receiving messages to and from each other. The methods of the rational engine directly correspond to messages. For each specific message, the sender and receiver are explicitly identified. For example, the invocation

```
Believe?(PreconditionAct(?x,PICKUP(A)))
```

can be viewed as the message

Send message `Believe?` to the object
`B143: PreconditionAct(?x, PICKUP(A))`

Thus, `B143` is the receiver and the object sending the message in the example above could be the act `PICKUP(A)` itself. This is very similar to the Actor model of concurrent object-based computation [1, 2]. Like Actor systems, message passing is employed as a basis of computation. Object instances denote individual actors. Labels of objects denote their *mail addresses*. The only difference is that the behavior of the objects is determined by inherited methods, something that is missing in Actor systems. Nevertheless, this seems to indicate that the OK architecture can be implemented using Actor systems. Building AI architectures has been a long-term goal of Actor-based systems. In fact, the original motivations for Actor systems came out of Carl Hewitt's work on PLANNER [12]. To this date, no attempts have been made to implement AI systems using Actor languages. This is probably because we don't, as yet, have adequate primitives and environments to build large software using Actor languages. The Actor view of the OK rational engine arises out of considerations in making some representational as well as some behavioral commitments. Only the latter are similar (or conform) to the Actor view of computation. Transformers of the OK formalism help capture the overall embedded nature of the architecture. It is our hope that we would be able to explore this in the future. In the meanwhile, we have implemented the rudimentary components of the OK architecture in a conventional, quasi-concurrent paradigm.

7 Examples

We started this paper by indicating that we were interested in building integrated rational cognitive agents. These agents are capable of natural language interaction, reasoning, acting, reacting, and knowledge acquisition behavior.

7.1 The Blocksworld Domain

For instance, in the blocksworld domain, the agent is capable of understanding the following paragraph:

```
Method
    BELIEVE?($\phi$ : Proposition Term; $\sigma$ : Substitution := NIL)
is
        let $o \leftarrow \phi\sigma$

        if ASSERTED?(o) then
            return the set $\{o\}$
        elseif PATTERN?(o) then
            RESULT $\leftarrow$ a set containing all asserted instances of o
        endif

        find the set T of applicable AntCq transformers, i.e.
        let $T \leftarrow \{t \mid \phi_t \neq NIL \wedge BELIEVE?(t)$, where $\phi_t \leftarrow UNIFY(CQ(t),o)\}$
        also find all the applicable DoWhen transformers, i.e.,
        let $T \leftarrow T \cup \{t \mid \phi_t \neq NIL \wedge BELIEVE?(t)$, where $\phi_t \leftarrow UNIFY(WHEN(t),o)\}$
        for each $t \in T$ loop
            RESULT $\leftarrow$ RESULT $\cup$ TRANSFORM?($t,\phi_t$)
        endloop

        find the set of matching propositions, i.e.,
        let $B \leftarrow \{b \mid \phi_b \neq NIL$, where $\phi_b \leftarrow UNIFY(b,o)\}$
        for each $b \in B$ loop
            RESULT $\leftarrow$ RESULT $\cup$ BELIEVE?($b, \phi_b$)
        endloop

        return RESULT
end BELIEVE?
```

Figure 2: The *Believe*? Method. See [13] for details of other methods.

> There is a table. The table is a support. Blocks are supports. A is a block. B is a block. C is a block. C is clear and on the table. A is clear and on the table. B is clear and on the table.
>
> Picking up is a primitive action. Putting is a primitive action. Before picking up a block the block must be clear. If a block is on a support then after picking up the block the block is not on the support. If a block is on a support then after picking up a block the support is clear. After picking up a block the block is held. Before putting a block on a support the block must be held. Before putting a block on a support the support must be clear. After putting a block on a support the block is clear. After putting a block on a support the block is on the support. After putting a block on another block the latter block is not clear.
>
> A plan to achieve that a block is held is to pick up the block. A plan to achieve that a block is on a support is to put the block on the support. If a block is on a support then a plan to achieve that the support is clear is to pick up the block and then put the block on the table.

The agent parses the above sentences and uses the formalism described to represent them. It can then perform actions in the blocksworld using the information provided in these sentences. It is also capable of carrying out requests like

```
Pick up a clear block.
```

Notice that the sentence describes a variable that is a block and it is clear. The request is represented using the WITHSOME act as

```
WITHSOME(x)((Isa(x,BLOCK) ∧ Clear(x)), PICKUP(x))
```

Structured variables can also be used to represent such actions. The agent in carrying out the intention of picking up a clear block determines all the blocks that are clear and picks up one of them. In a situation where, say, the blocks A, B, C are clear, the agent will respond

```
The designator Isa(x,BLOCK) ∧ Clear(x)
is effective on the following
Isa(A,BLOCK) ∧ Clear(A)
Isa(B,BLOCK) ∧ Clear(B)
Isa(C,BLOCK) ∧ Clear(C)
for the act
WITHSOME(x)((Isa(x,BLOCK) ∧ Clear(x)), PICKUP(x))

I intend to do
DoONE(PICKUP(A), PICKUP(B), PICKUP(C))

Chose to do the act
PICKUP(C)

Now doing: PICKUP(C)
BELIEVE(Held(C))
DISBELIEVE(Clear(C))
DISBELIEVE(On(C, TABLE))
```

Next, let us assume that in addition to the knowledge described in the paragraphs above, the agent also believes

```
1. All red colored blocks are wooden.
```

```
2. Look is a primitive action.
3. If you want to know the color of a block look
at it.
```

If the agent is then asked the query:

```
Is A wooden?
```

At this point, the agent knows that A is a block but has no beliefs about its colors. The query will backchain as follows:

```
I wonder if Isa(A, WOODEN)
I wonder if Isa(A, BLOCK)
I know Isa(A, BLOCK)
I wonder if Color(A, RED)
```

The query has backchained through (1) above. next it backchains through (3). First, it derives the specific transformer

```
DoWhen(LOOKAT(A), Color(A, ?color))
```

which is then applied and the act LOOKAT(A) is performed. As a result (assuming that A is colored red), the belief

```
Color(A, RED)
```

is added to the agent's belief space which completes the earlier chain of inference and returns the answer (that A is wooden). Thus we see that it is also possible for inference to lead to actions. Similarly, one can forward chain through WhenDo transformers in order to react to situations. The above example has illustrated an interesting artifact of our architecture—that an agent is capable of using acting in service of inference as well as inference in service of acting (only the latter being the typical case in most planning/acting systems).

7.2 Planning Discourse to Discuss Plans

We are also modeling an agent that is capable of describing and justifying domain plans in an interactive natural language setting [10, 11]. The Interactive Discourse Planner (IDP) relates two areas of research. The first area, plan recognition, focuses on analyzing natural language that is *about* plans to recognize the speaker's intent and provide helpful responses. The second area, discourse planning, is concerned with *using* plans for selecting and structuring text that achieves communicative goals. IDP operates in a collaborative mode in which the system is the primary speaker and the user is the primary listener. IDP is responsible for planning text to describe and/or justify a domain plan, and the user is responsible for providing feedback that lets IDP know how to continue the discussion in a way that is helpful.

IDP exploits the OK formalism by using a uniform representation for the text plan that it formulates and executes incrementally, and the domain plans that are under discussion. In this way, the text plan and the domain plans are both accessible for analyzing the user's feedback. IDP can interpret vaguely articulated feedback, generate concise replies and metacomments, and detect user feedback that initiates a digression. As a testbed for our model, IDP discusses driving routes as the domain plans.

```
User:  Should I take Maple or Sheridan
       to go to the Eastern Hills Mall?
IDP:   you should take Maple.
User:  Why?
IDP:   taking Maple avoids heavy traffic.
User:  Why?
IDP:   since taking Maple there are fewer
       businesses than taking Sheridan
```

By modeling the discourse with its evolving text plan, IDP can produce concise text that does not repeat old information along with the new. These types of interactions are coherent and collaborative because each participant knows how his contribution fits in. Levelt refers to this discourse property as *discourse deixis* [16]. Consistent with the Gricean Maxims [9], IDP processes feedback with the idea that the more the user says, the more he feels he needs to say for the system to identify the text plan expansion that is sought. Therefore, IDP uses any additional information that the user provides to try to recognize a text plan. This can lead to text plan expansions that are not immediate continuations of what the system said last. The following example demonstrates this type of processing:

```
User:  Should I take Maple or Sheridan
       to go to the Eastern Hills Mall?
IDP:   take Maple.
User:  Why not go on Sheridan?
IDP:   you could take Sheridan however,
       taking Maple avoids heavy traffic.
```

In this interaction, the user's feedback indicates that he would like the system's response to include information about the feasibility of an alternative route. IDP uses the mentioned act, *go*-ing *on Sheridan*, to identify a discourse entity and a text plan expansion that uses it. This is how IDP continues to pursue its own intentions while providing the implicitly requested information. IDP can also detect user-imposed digressions:

```
User:  Should I take Maple or Sheridan
       to go to the Eastern Hills Mall?
IDP:   take Maple.
User:  Why should I take Maple?
IDP:   taking Maple avoids heavy traffic.
User:  Why?
IDP:   taking Maple there are fewer
       businesses than taking Sheridan.
User:  Why is there heavy traffic now?  *
IDP:   since now is rush hour,
       as I was saying, taking Maple
       avoids heavy traffic.
```

The user's third question (marked with an asterisk *) relies on an inference that he made from the proposition that IDP conveyed in its second response. IDP uses its text plan to detect that this question, unlike the user's first two questions, initiates a digression. The system makes this determination when it cannot find a way of expanding the focussed portion of its text plan to answer it. When IDP detects a user-imposed digression, it answers the question, and then it immediately goes back to pursuing to its own intention by expanding is text plan further. In these examples that intention is to have the user adopt the plan of taking the Maple Road route. As demonstrated by the last two lines of IDP's final response, the uniform representation that is used for all information allows IDP to use its own text plan as content to do this.

7.3 Structured Variables

Our system is capable of natural language processing using structured variables. It includes a generalized augmented transition network (GATN) natural language parser and generation component linked up to the knowledge base (based on [20]). A GATN grammar specifies the translation/generation of sentences involving complex noun phrases into/from structured variable representations.

An advantage of the use of structured variables lies in the representation and generation of complex noun phrases that involve restrictive relative clause complements. The restriction set of a structured variable typically consists of a type constraint along with property constraints (adjectives) and other more complex constraints (restrictive relative clause complements). So, when parsing a noun phrase, all processing is localized and associated with building its structured variable representation. When generating a surface noun phrase corresponding to the structured variable, all constraints associated with the variable are part of its structure and can be collected and processed easily. This is in contrast to non-structured variable representations (such as FOPL) where the restrictions on variables are disassociated from the variables themselves, in the antecedents of rules. The interaction below shows sentences with progressively more complex noun phrases being used. These noun phrases are uniformly represented using structured variables. Parsing and generation of these noun phrases is simplified because structured variables collect all relevant restrictions on a variable into one unit, a structured variable (user input is shown italicized).

: *Every man owns a car*
```
I understand that every man owns some car.
```
: *Every young man owns a car*
```
I understand that every young man owns some car.
```
: *Every young man that loves a girl owns a car that is sporty*
```
I understand that every young man that loves any
girl owns some sporty car.
```
: *Every young man that loves a girl that owns a dog owns a red car that is sporty*
```
I understand that every young man that loves any
girl that owns any dog owns some red sporty car.
```
: *Every young man that loves a girl and that is happy owns a red sporty car that wastes gas*
```
I understand that every young happy man that
loves any girl owns some
sporty red car that wastes gas.
```

The parser parses the user's sentence and builds a representation of the user input. The resulting representation is then passed to the generation component, which generates the output response (sometimes prefixed by the canned phrase `I understand that`). If constraints on variables corresponding to the complex noun phrases were represented using first-order logic-based representations, then it would be difficult to generate natural language noun phrases corresponding to these variables. This is because the constraints on variables would, likely, be well separated from the variables in the antecedents of rules involving these variables. This is not the case in a structured variable representation.

Because the structure of representations of rules using structured variables is "flat", that is, there is not the artificial antecedent-consequent structure associated with first-order logic-based representations, it is possible to frame questions whose answers are rules and not just ground formulas. Because of the subsumption relation between structured variables (a more general, or less restricted, variable subsumes another more restricted variable), useful inferences are possible, directly. Since the structure of the question will mirror the structure of the rule, any rule that is subsumed by a question is an answer to that question. What follows is a sample dialog involving questions whose answers are ground propositions (e. g., *Is John mortal*) as well as questions whose answers are rules (e. g., *Who is mortal*):

: *Every man is mortal*
```
I understand that every man is mortal.
```
: *Who is mortal*
```
Every man is mortal.
```
: *Is any rich man mortal*
```
Yes, every rich man is mortal.
```
: *John is a man*
```
I understand that John is a man.
```
: *Is John mortal*
```
Yes, John is mortal.
```
: *Who is mortal*
```
John is mortal and every rich man is mortal and
every man is mortal.
```
: *Are all rich young men that own some car mortal*
```
Yes, every young rich man that owns some car is
mortal.
```
: *Any rich young man that owns any car is happy*
```
I understand that every young rich man that owns
any car is happy.
```
: *Is John happy*
```
I don't know.
```
: *Young rich John owns a car*
```
I understand that mortal rich young John owns
some car.
```
: *Who owns a car*
```
Mortal rich young John owns some car.
```
: *Is John happy*
```
Yes, mortal rich young John is happy.
```

This dialog also illustrates the uses of subsumption. Since we told the system *Every man is mortal*, it follows that any more specifically constrained man (e. g., *Every rich young man that owns some car*) must also be mortal. Note that this answer (a rule) follows directly by subsumption from a rule previously told to the system. This is another way in which rules may be answers to questions, in a representation using structured variables. The utility of structured variables is a pressing argument for the use of object-oriented design at all levels of an AI formalism.

8 Remarks

We have presented a unified formalism for modeling computational rational agents. In doing so, we have made some ontological as well as semantic commitments. The object-oriented design of the formalism enables a canonical representation of entities. The architecture has been specifically designed to be extendable—in its ontology, as well as the underlying logic and action theory. This is a direct benefit from the object-oriented design. One can easily extend the formalism by defining additional classes and/or subclasses. The rational engine can be modified by changing (or specializing) its methods. We have also briefly addressed the possibility of implementing a concurrent rational engine. Finally, we presented several examples from our work illustrating various AI faculties that can be modeled.

Acknowledgement

All the work discussed in this paper was carried under the direction of Dr. Stuart C. Shapiro. Our representational commitments are a direct result of over two decades of work of Dr. Shapiro and his associates on the SNePS semantic network processing system.

References

[1] Gul Agha. Concurrent Object-Oriented Programming. *Communications of ACM*, 33(9):125–141, 1990.

[2] Gul Agha and Carl Hewitt. Concurrent Programming Using Actors. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 37–53. MIT Press, Cambridge, MA, 1987.

[3] Syed S. Ali. *A "Natural Logic" for Natural Language Processing and Knowledge Representation.* PhD thesis, State University of New York at Buffalo, Computer Science, 1993. Forthcoming.

[4] Syed S. Ali. A Structured Representation for Noun Phrases and Anaphora. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, pages 197-202, Hillsdale, NJ, June 1993. Lawrence Erlbaum.

[5] Syed S. Ali and Stuart C. Shapiro. Natural Language Processing Using a Propositional Semantic Network with Structured Variables. *Minds and Machines*, 3(4), November 1993. Special Issue on Knowledge Representation for Natural Language Processing.

[6] Michael E. Bratman, David J. Israel, and Martha E. Pollack. Plans and Resource-Bounded Practical Reasoning. *Computational Intelligence*, 4(4), 1988.

[7] Kit Fine. *Reasoning with Arbitrary Objects.* Basil Blackwell, Oxford, 1985.

[8] Michael P. Georgeff. Planning. In *Annual Reviews of Computer Science Volume 2*, pages 359–400. Annual Reviews Inc., Palo Alto, CA, 1987.

[9] H. P. Grice. Logic and conversation. In P. Cole and J. L. Morgan, editors, *Syntax and Semantics 3: Speech Acts.* Academic Press, New York, 1975.

[10] S. M. Haller. Interactive generation of plan justifications. In *Proceedings of the Fourth European Workshop on Natural Language Generation*, pages 79–90, 1993.

[11] S. M. Haller. An interactive model for plan explanation. In *Proceedings of the Australian Joint Conference on Artificial Intelligence*, Melbourne, Australia, November 1993. to appear.

[12] C. Hewitt. Procedural embedding of knowledge in planner. In *Proceedings 2nd IJCAI*, pages 167–182, 1971.

[13] Deepak Kumar. *From Beliefs and Goals to Intentions and Actions— An Amalgamated Model of Acting and Inference.* PhD thesis, State University of New York at Buffalo, 1993.

[14] Deepak Kumar and Stuart C. Shapiro. Architecture of an intelligent agent in SNePS. *SIGART Bulletin*, 2(4):89–92, August 1991.

[15] Deepak Kumar and Stuart C. Shapiro. Deductive efficiency, belief revision and acting. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, 5(2), 1993. Forthcoming.

[16] W. J. M. Levelt. *Speaking: From Intention to Articulation.* MIT Press, Cambridge, 1989.

[17] Michael L. Nelson. Concurrency & Object-Oriented Programming. *ACM SIGPLAN Notices*, 26(10):63–72, October 1991.

[18] Anand S. Rao and Michael P. Georgeff. An Abstract Architecture for Rational Agents. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proceedings of the 2nd Conference on Principles of Knowledge Representation and Reasoning*, pages 439–449, San Mateo, CA, 1992. Morgan Kaufmann Publishers.

[19] Earl D. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier North Holland, New York, NY, 1977.

[20] S. C. Shapiro. Generalized augmented transition network grammars for generation from semantic networks. *The American Journal of Computational Linguistics*, 8(1):12–25, 1982.

[21] S. C. Shapiro and The SNePS Implementation Group. *SNePS-2 User's Manual.* Department of Computer Science, SUNY at Buffalo, 1989.

[22] S. C. Shapiro and W. J. Rapaport. SNePS considered as a fully intensional propositional semantic network. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier*, pages 263–315. Springer–Verlag, New York, 1987.

[23] David E. Wilkins. *Practical Planning–Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, Palo Alto, CA, 1988.