Contents

List of Figures

List of Tables

From Beliefs and Goals to Intentions and Actions: An Amalgamated Model of Inference and Acting

by

Deepak Kumar

A dissertation submitted to
Faculty of the Department of Computer Science
and Graduate School of
State University of New York at Buffalo
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Dissertation Committee:
Dr. Stuart C. Shapiro, Chairman
Dr. Bharadwaj Jayaraman
Dr. William J. Rapaport

1993

Abstract

The behavior of computational rational agents should be driven by their beliefs, desires, and intentions. A survey of AI systems reveals that it is somewhat awkward to do planning and acting in systems designed for reasoning about beliefs and that it is similarly awkward to study representational and reasoning issues in planning/acting systems. This is an investigation of the relationships among beliefs, plans, effective acts, and between reasoning and acting, in the context of a rational cognitive agent. It turns out that one can establish a closer relationship between inference and acting if one views inference as a kind of mental acting; i.e., a deduction rule is a propositional connective that, when applied, constitutes the mental act of believing the rule's consequents. This suggests that one only needs a single module for reasoning and acting. Once integrated in such a way, it is also possible to visualize connectives that transform beliefs into an intention to do something in reaction to some new beliefs or in the quest to answer some query. These ideas have been formalized by defining the notion of a *transformer* and results in a model of a cognitive agent that is able to achieve the posted goals in a unified fashion. The investigation leads to the design of an architecture called the Object-based Knowledge Belief-Desire-Intention (OK BDI) architecture. It is comprised of a representational formalism called the OK Formalism and a module called the OK Rational Engine. The OK Formalism is designed using object-oriented principles. As an implementation, we have extended the SNePS semantic network processing system into what we call the SNePS BDI architecture, which includes representations for various transformers and a rational engine, called SNePS Rational Engine (SNeRE), that is responsible for integrated reasoning and acting.

Acknowledgements

Several people supported me in one way or the other over the duration of this work. I would like to especially thank Stu Shapiro, Bill Rapaport, and Bharat Jayaraman, for direct involvement and advising on my work. Stu has been a major influence and source of inspiration in not only this work but also on defining my career as a computer scientist. Drew McDermott was my external reader. His comments never made it to me.

Tony Ralston, Pat Eberlein, Helene Kershner, Michael Leyton, Harry Delano, Ken Smith, and Devon Bowen of the Department of Computer Science at SUNY/Buffalo. Also, Russ Miller, for providing 'the rare professional opportunity'.

Hans Chalupsky, Richard Wyatt, Syed Ali, Nuno Mamede, Carlos Pinto-Ferreira, and other members of SNeRG, the SNePS Research Group.

Aditya Mathur, Sudeb Basu, Humayun Hayat Khan, Bazmi Rizwan Hussain, Uday Reddy, Kulbir Arora, Kamal Lodaya, Arvino, and the immortal IBM 1130 at BITS, Pilani.

Vic, Kristi, George, Marc, PB, Bruce, Nyle (aka Orlando), Sheri, Kim, Phil, Jack, Bobby, Marshal, and other 'mutants'.

Ellie, Gloria, and Sally, my foster moms.

My parents, and Debika, my wife.

And Gaurav, this is dedicated to you.

Chapter 1

Introduction

Over the years, the field of Artificial Intelligence (AI) has been subdivided into various subfields that explore computational models of various individual intellectual faculties such as knowledge representation, natural language, learning, planning, acting, vision, etc. Although this division has allowed the field to make progress in understanding each of these components, issues involved in integrating them into a single, complete computational model have largely been ignored. Attempts to build systems that integrate more than one field tend to take a modular, interchangeable-parts approach; for example, the attempt to put together, say, natural language understanding, generation, and learning capabilities in one system, using this approach, may end up trying to plug together, in a modular fashion, three existing systems that demonstrate those capabilities. All such systems have to build explicit interfaces between the modules that have to interact with each other. Thus, in the example above, one may end up with as many as six interfaces in the case where each module has to communicate (or transfer knowledge) to all the others. In general, if N such modules using N different technologies are to be integrated and all modules need to communicate with every other module, one will end up building N! different interfaces. Moreover, if any one of the N modules has to be replaced $2(N - 1)$ more interfaces will have to be developed.

The research presented here takes a different approach to building integrated systems. Its main premise lies in the observation that regardless of the subfield involved, all intellectual activity involves representation of knowledge and reasoning. Using a uniform representation and reasoning formalism for all subfields eliminates the need for building interfaces. However, this is easier said than done. In this dissertation, I present an investigation of the relationships among beliefs, plans, effective acts, and sensory acts, and between reasoning and acting, in the context of a *computational rational agent.* A computational rational agent is a computer model of an agent that can reason about its beliefs, goals, and actions, and is capable of affecting and reacting to the external world. I will demonstrate that such an approach results in a representational formalism, semantics, and inference/acting mechanism that unifies these subjects while clearly maintaining the proper distinctions among them.

The behavior of computational rational agents should be driven by their beliefs, goals (also called desires), and intentions. They should be capable of reasoning as well as acting in the real world. This research addresses the problem of designing and implementing representations and reasoning mechanisms required to model rational cognitive agents. I design and explore appropriate mechanisms that facilitate representation of an agent's beliefs, actions, plans, and intentions. Using these representations, I have developed appropriate reasoning and acting techniques that would enable the modeled agent to reason about and act based on its representations. In addition, these representations and reasoning and acting mechanisms provide reactive capabilities for the agent to respond to changes in its environment while it is in the process of carrying out its tasks. As mentioned above, this work is a result of investigating the relationship between beliefs, inference, intentions, and acting. The underlying thesis of this research is that an approach based on the relationship between inference and acting provides a unified framework for modeling rational agents because of the following:

- The modeled agent is able to represent and reason about its beliefs and those of others.
- The agent is able to represent and reason about actions and plans the same way as it does its beliefs.
- Using these representations, the agent is able to plan its future actions.
- The agent is able to perform the intended actions in a real world.
- The agent is able to use its representations and beliefs to react to events happening around it.
- The agent is also able to discuss its actions, plans, and beliefs with other agents in the world.

This research is an attempt to synthesize an integrated representation, reasoning, and acting system that provides appropriate mechanisms for dealing with sensory acts, external events, and how the behavior of a rational agent can be affected by them. It builds upon and, where necessary, modifies and extends the existing representational framework of the SNePS semantic network processing system [Shapiro 1979, Shapiro and Rapaport 1987, Shapiro and Rapaport 1992].

1.1 Motivation

Research in planning and acting has progressed independently of research in knowledge representation and reasoning. While it is good to maintain this dichotomy to focus on specific issues, it turns out that current planning/acting systems provide very limited capabilities for knowledge representation and reasoning. Facts stored in a planning agent's world model correspond to the agent's beliefs. The kind of reasoning done on these beliefs is limited to basic retrieval; sometimes, using simple inference rules, simple consequences of current beliefs can be derived. The state of the art in knowledge representation and reasoning is much more advanced than that. Current knowledge representation and reasoning systems are capable of dealing with issues in natural language understanding, representing beliefs of the agent as well as others, belief revision using truth maintenance procedures, and other subtle issues. Some of these representations also deal with beliefs about agents performing actions and events taking place. A survey of various planning/acting systems and various knowledge representation systems reveals that in most systems it is awkward to do acting in logic-based systems (but it is convenient to talk about knowledge representation issues), and it is awkward to research representation and reasoning issues in systems designed for planning/acting. I.e., putting it more bluntly, good planning systems are bad knowledge representation and reasoning systems and *vice versa*. This is also reflected in the following quote from [Georgeff 1987] (p. 394):

> Another promising approach to providing the kind of high-level goal-directed reasoning capabilities, together with the reactivity required for survival in the real world, is to consider planning systems as rational agents that are endowed with the psychological attitudes of belief, desire, and intention. The problem that then arises is specifying the properties we expect of these attitudes, the way they interrelate, and the ways they determine rational behavior in a situated agent.

In the past, most efforts of researchers of the SNePS research group have centered around representation and reasoning about beliefs derived from natural language interaction with the user. This work extends the SNePS approach to modeling cognitive agents by integrating the notions of acting and planning.

1.2 Desiderata for Rational Agents

Any model of a rational cognitive agent should be able to do the following:

- Reason about its beliefs and those of others. For example, it should be able to understand sentences like "If a block is on a support then the block is not on another support." "All men are mortal." "Socrates is a man." "Block A is on top of block B.", etc., and use its understanding of these to answer queries like "Who is mortal?", "Is block A on top of block C?", "Is Socrates mortal?", etc.

- Accept instructions on how to behave under various circumstances and on how to decompose complex actions into plans. For example, it should be able to use the information contained in these sentences: "A plan to achieve that a block is held is to pick up the block." "A plan to pile a block on another block on a third block is to put the third block on the table and then put the second block on the third block and then put the first block on the second block." "A plan to sound the alarm is to go to the alarm and then break the glass and then pull the lever."

- Actually perform actions in the external world upon request. For example, "Pile A on B on C." " Pile A on B on C and then pick up the red block."

- React to events happening around it. For example, "If you drop something, pick it up." "In case of a fire, sound the alarm and leave the building."

- React to actions performed by other agents. For example, the response to "John picked up block A" should at least prompt it to update its model of the world.

- Understand its own acts and those of others, as in "Before picking up a block, make sure that it is clear." "Before crossing the street, make sure the walk signal is lit." "After putting a block on another block the latter block is not clear." "John put block A on block B."

- Reason about its actions and plans.
- Discuss its actions and plans with other agents around it. For example, "What would you do in the case of a fire?" "How would you sound the alarm?" "What happens when someone picks up a block?"

The ultimate goal of a system (modeled agent) that can plan and reason about actions is to survive in the real world (or the world it is currently situated in). Classical AI planning systems [Fikes and Nilsson 1971, Sacerdoti 1977, Tate 1977, Wilkins 1984, Tate 1987] treat the process of plan generation and plan execution as two separate phases. The plan executor forms the acting component of such systems. Some form of *execution monitoring* is required to ensure that all plans are successfully executed. Failure to do an action (or achieve a certain state) may result in *replanning*. However, in real world situations, events that are outside of the control of the agent are constantly changing the agent's surroundings. This makes it hard to construct plans that would succeed in all situations. To avoid this problem, some approaches try to interleave plan construction and execution [Georgeff *et al.* 1987]. The basic premise underlying this approach is that new information will influence future actions; hence, decisions about future actions should be delayed as much as possible to take into account all the new information. Although the need to deal with new information in action-decision making has been recognized, a serious attempt to provide a model for acquisition of new information in the course of acting has yet to be made.

A desirable feature of systems that act in the real world is reactivity to situations. Reactivity is different from the process of replanning (for example, using an umbrella when it begins to rain on the way to work (i.e., reacting) as opposed to doing something else when a game is cancelled due to rain (i.e., replanning)). Replanning is done to ensure overall success in achieving some goal. However, an agent may be required to react to certain situations in the course of its regular activity. Most classical planners [Fikes and Nilsson 1971, Sacerdoti 1977, Tate 1977, Wilkins 1984, Tate 1987] are totally committed to a constructed plan. Reactivity involves modifying intentions (and maybe goals). Once again, there is a need to acquire new information/beliefs and form appropriate intentions to react to them (if needed).

It is clear that for an agent to be able to plan, act, react, and reason about its actions, it needs to be endowed with a set of beliefs, desires, and intentions. Beliefs can change based on sensory input from the dynamically changing environment, and they can lead to new intentions. Reports of events happening in the world can lead to new (or revised) beliefs, which in turn can lead to the formation of new intentions. This suggests that a closer relationship needs to be established between reasoning about beliefs and actions, and the process of planning and acting. The motivations underlying this work are derived from observations made on existing models of inference (i.e., the representations of rules of inference, and the mechanisms of forward and backward inference) as well as current models of planning and acting (i.e., the representations of operators and plans, and plan execution and monitoring). This study yields a better model of knowledge representation, reasoning, and acting and solves most of the problems mentioned above.

1.3 Organization of the Dissertation

In the next chapter, some of my observations regarding traditional as well as current approaches are supported by a survey of existing systems. In Chapter 3, I highlight the relationship between acting and inference. The unified view of acting and inference presented in Chapter 3 is the underlying premise for the rest of the dissertation. In Chapters 4–8, I present the specification of a new BDI (for *B*eliefs, *D*esires, and *I*ntentions) architecture that utilizes the unified approach to acting and inference. The architecture is being called an OK (for *O*bject-based *K*nowledge) architecture. It constitutes a unified representational formalism and an operating module called a *rational engine* that implements the unified view of acting and inference. Chapter 9 describes, in detail, the SNePS BDI formalism, which is isomorphic to the OK formalism (in fact the design of the OK formalism is based on the same principles as SNePS). Chapter 11 describes the SNePS rational engine—SNeRE. Chapters 10 and 12 describe a domain and examples of reasoning and acting in that domain by a computational rational agent implemented using the SNePS BDI architecture. Chapter 13 summarizes the main contributions of this work.

Chapter 2

Related Work

Early research on planning in STRIPS [Fikes and Nilsson 1971], NOAH [Sacerdoti 1977], NONLIN [Tate 1977], SIPE [Wilkins 1984], and even O-PLAN [Tate 1987] was centered around the idea of representing actions as operators. Initially, a plan was represented simply as a list of instantiated operators (STRIPS). This evolved into hierarchical representations of actions and non-linear and conditional plan representations (NOAH, NONLIN, and SIPE). Some mechanisms for reasoning about plans were also developed (NOAH, NONLIN, and O-PLAN). Research in belief representation and reasoning proceeded in isolation from most of the planning work. This research on planning has come to be called *classical planning*.

Some of the more recent approaches to planning and acting have used alternative control paradigms to attempt to solve the problems concerned with reactivity and sensory inputs. Some go so far as to banish the idea of symbolic representations. These include the work on subsumption architectures [Brooks 1986, Agre and Chapman 1987], and situated automata [Rosenschein and Kaelbling 1985, Kaelbling 1986]. McDermott [McDermott 1992] has called these *minimalist* approaches. This chapter is explicitly concerned with symbolic representations of plans and acts and hence will not survey the minimalist approaches. Among the recent symbolic approaches, some researchers have tried to use more advanced notions of belief, knowledge, and reason maintenance in their modeling strategies. I will survey some classical approaches, followed by an overview of some current techniques. This survey illustrates salient features of various representations. To keep the discussion brief, I have omitted explaining in detail all the terms used. The reader is encouraged to refer to appropriate sources.

2.1 Classical Planning

I will begin by examining the so called classical AI planning paradigm [Wilkins 1988]. Under this paradigm, a planning system is modeled using a world model, a set of acts, and a *truth criterion procedure*. Classical planning systems traditionally use three different levels of representations:

- representations of world models
- representations of actions
- representations of plans

As a consequence, the modeled agent has to perform reasoning at three different levels:

- reasoning about the world model
- reasoning about actions
- reasoning about plans

Reasoning about a world model corresponds to reasoning about the agent's beliefs. The planner (or the planning module) reasons about actions when formulating plans. Reasoning about plans is normally done by *critics*. This involves reasoning about the structure of plans (or about an agent's intentions). What follows is a survey of some planning systems to illustrate this point.

STRIPS [Fikes and Nilsson 1971] uses a set of well-formed formulas to represent a world model. The set of actions is represented using an *operator schema*. For example, the action of putting a block on another block is represented by the operator schema shown in Figure 2.1. Plans produced by STRIPS are linear, represented by a list of instantiated operators. For example,

$$\{puton(C, TABLE), puton(B, C), puton(A, B)\}$$

STRIPS employs resolution theorem proving to answer questions about particular world models and uses means-ends analysis to reason about actions during plan generation. STRIPS has no capabilities for reasoning about plans.

puton(m,n):	*Put block m on top of n*
preconditions:	CLEAR(m) $\wedge$ CLEAR(n)
delete:	CLEAR(n)
add:	ON(m,n)

Figure 2.1: The STRIPS representation for *puton*

NOAH [Sacerdoti 1977] uses *expressions* to represent the state of the world; actions are coded as operators that are expressed using the language SOUP (which is similar to, and built on top of, the QLISP system). Figure 2.2 shows NOAH's representation for the *Puton* action. Plans are represented as *procedural networks*, which are generated when the SOUP code for the actions is interpreted. For example, when applied, the *Puton* operator will generate the procedural network (plan) shown in Figure 2.3. Reasoning about plans in NOAH is done using various *procedural critics*. For example, the "resolve conflicts" critic identifies endangered subgoals in parallel conjuncts and reorders actions to resolve conflicts. It uses a mechanism called a TOME (Table Of Multiple Effects) for identifying conflicts.

NONLIN [Tate 1977] uses a declarative *task formalism* to represent actions. It is based upon the operator schemas of STRIPS. Figure 2.4 shows the *Puton* operator expressed in NONLIN's task formalism. The structure of plans is similar to that of NOAH. Reasoning about plans is done using a TOME and a "goal structure" (GOST) to detect and resolve conflicts. As an improvement over NOAH, it has a declarative representation of operators, it allows typed preconditions, it allows alternatives to be explored, and it introduces a

```
(PUTON
  (QLAMBDA
     (ON <-X <-Y)
     (PAND
         (PGOAL (Clear $X)
                (CLEARTOP $X)
                APPLY
                (CLEAR))
         (PGOAL (Clear $Y)
                (CLEARTOP $Y)
                APPLY
                (CLEAR)))
     (PGOAL (Put $X on top of $Y)
            (ON $X $Y)
            APPLY NIL)
     (PDENY (CLEARTOP $Y))))
```

Figure 2.2: NOAH's representation for *Puton* (from [Sacerdoti, 1977], p. 14)

better question-answering procedure that deals with queries in a partially ordered network of nodes. O-PLAN [Tate 1987, Currie and Tate 1991] uses similar but more generalized representations than the ones used by NOAH and NONLIN. The control structure of O-PLAN uses a blackboard scheme.

SIPE's [Wilkins 1983, Wilkins 1984, Wilkins 1988] representations are richer and more flexible than NOAH's. Actions are described in an *Operator Descriptor Language (ODL)* (see Figure 2.5 for a description of *Puton* in ODL). Procedural networks similar to NOAH's are used for representing plans. SIPE can represent some world knowledge using operators in the form of *domain rules.* Domain rules (see Figure 2.6) can be *state rules* that are used to represent domain constraints within a world state, or they can be *causal rules* that permit

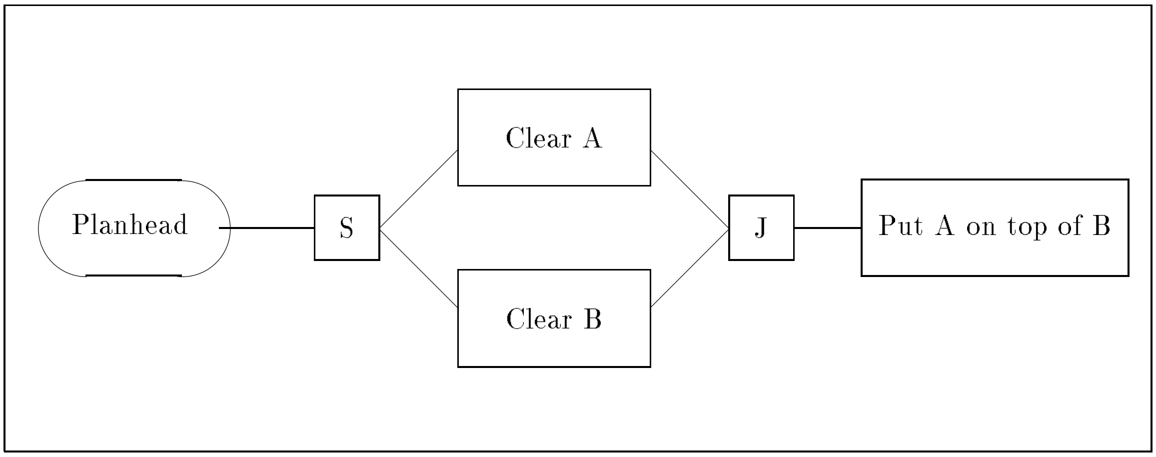

Figure 2.3: An example of NOAH's procedural networks

```
ACTSCHEMA PUTON
   PATTERN <<PUT $*X ON TOP OF $*Y>>
   CONDITIONS USEWHEN <<CLEARTOP $*X>> AT SELF
              USEWHEN <<CLEARTOP $*Y>> AT SELF
              USEWHEN <<ON $*X $*Z>> AT SELF
   EFFECTS + <<ON $*X $*Y>>
           - <<CLEARTOP $*Y>>
           - <<ON $*X $*Z>>
           + <<CLEARTOP $*Z>>
   VARS X UNDEF Y UNDEF Z UNDEF;
END;
```

Figure 2.4: *Puton* expressed in NONLIN's task formalism (from [Tate, 1977] p. 890)

access to different world states. However, these rules only aid in making the operators more general and are used (by the question-answering procedure called *the truth criterion procedure*) only in deducing context-dependent effects of actions. Reasoning about plans is done using an improved version of TOME-based procedural critics.

2.2 Observations

In all these systems, actions are represented by *operators*. The only purpose of an operator is to specify changes from one state to another when the corresponding action is performed. Such a representation is useful only to a planner. It is totally divorced from the set of beliefs of the modeled agent. Since the formalism used to represent operators is different from that used to represent beliefs, the agent cannot have any explicit beliefs about operators. Consequently, it does not *understand* actions as it does its beliefs. It can use and perform actions using the planning and execution modules. If an agent is to be able to reason about another agent's actions, it should be able to represent them and even discuss them. Having different representations for beliefs and actions leads to different reasoning modules. The same argument goes against having a different representation for plans.

In order to have one reasoning module that is able to reason about beliefs, actions, and plans, one has to represent all of them in a uniform manner while maintaining appropriate semantic distinctions. Such a system should be capable of having beliefs about actions and at the same time able to use them to plan and act. This is the first step towards an integrated model of acting and inference.

2.3 Current Approaches

An attempt to represent world state, plans, and reasoning rules in a uniform way was first made by McDermott [McDermott 1978]. He developed a theory of planning and acting, which is implemented in the NASL problem solver. The state of the world, tasks and plans, the state of the problem solver, and the rules governing them are represented in

Operator: Puton
Arguments: block1, object1 Is Not block1;
Purpose: (On block1 object1);
Plot:
Branch1:
Goals: (Clear object1);
Branch2:
Goals: (Clear block1);
End Parallel

Process
Action: Puton.Primitive;
Arguments: block1, object1;
Resources: block1;
Effects: (On block1 object1);

End Plot Operator

Figure 2.5: The representation of *Puton* in ODL (from [Wilkins, 1988], p. 38)

a database of predicate-calculus statements. Plans are represented in a declarative task network schema. The interpreter works on assigned tasks and uses a theorem prover for deductive retrieval of facts as well as instantiated plan schemas (decompositions). An example of database formulas for achieving (ON x y) is shown in Figure 2.7. Notice that in the plan schema (task network) defined by the formula "on-plan", the relationships between subtasks are defined by the predicates "subtask", "successor", and "main". The NASL interpreter is capable of reasoning about these relationships so as to execute subtasks in the specified order. This work has evolved into RPL (Reactive Plan Language) [McDermott 1991, Beetz and McDermott 1992] which is an elaborate LISP like language for writing reactive plans, which was work done on the RAP system by [Firby 1987, Firby 1989]. RPL is being used for high-level reactive planning.

The interaction between knowledge and action was first explored by Moore, who uses modal logic with a possible-worlds semantics to develop a formal integrated theory of knowledge and action [Moore 1980, Moore 1985]. He argues that it is necessary to explicitly reason about what knowledge is needed to carry out a plan and how that knowledge can be obtained. He explains that one needs to consider knowledge prerequisites in addition to physical prerequisites for reasoning about actions. Using his logic, one is able to reason about the knowledge prerequisites (for example, "knowing the combination for a safe in order to open it") by knowing what action to take (e.g., "dial the combination of the safe") and deciding whether one knows how to perform an action ("open a safe at any given time"). The theory is also able to handle how agents infer new knowledge as a result of an action (like being able to infer the acidity of a solution by observing the results of a litmus paper test).

Allen [Allen and Koomen 1983, Allen 1984, Allen and Kautz 1985] uses a temporal reasoning framework to reason about actions and plans. In this framework, a world model

Causal-rule: Not-on
Arguments: object1, object2, object3;
Trigger: (On object1 object2);
Precondition: (On object1 object3);
Effects: ¬(On object1 object3);

State-rule: Deduce-clear
Arguments: object5, object6, object4 Class Existential;
Trigger: ¬(On object5 object6);
Condition: ¬(On object4 object6);
Effects: (Clear object6);

Figure 2.6: SIPE's domain rules (from [Wilkins, 1988], p. 88)

consists of all of the agent's knowledge of the past and present, and predictions about the future expressed in an interval-based temporal logic. Time intervals are related to each other by a set of seven primitive relations and their converses. Actions are represented in a STRIPS-like fashion along with temporal augmentations (see Figure 2.8). A plan is a collection of assertions viewed as an abstract simulation of some future world. In a complete plan ("useful solution"), the initial and final states have causal connections. The task of planning constitutes decomposing actions and doing constraint propagation over temporal relationships to determine causal connections between initial and the final states. Thus, reasoning about actions is done in the classical sense (à la STRIPS) and reasoning about plans is accomplished via constraint-based temporal reasoning. Reasoning about temporal constraints allows the problem-solver to determine constraints on action ordering (as opposed to using procedural critics à la NOAH). Domain rules for the world model are also specified using temporal domain constraints (see Figure 2.9). Allen's formalism allows for representation of overlapping and simultaneously occurring actions, and actions involving non-activity.

Drummond has used concepts from net theory to develop the notion of *plan nets* [Drummond 1985, Drummond 1987]. The plan net formalism was designed to allow for descriptions of an action's sensory results (e.g., a weight sensor in the robot arm would record the weight of an object when the robot performs a *pickup* action on that object). In addition, the representation of action and belief is able to describe iterative and conditional behaviors. Operators and plan nets are defined in terms of sets of formulae of a *belief language (L_B),* an *event language (L_E),* a *flow relation,* and a *before relation (execution advice).* Both L_B and L_E are defined using a *base language (L)* whose terms denote objects (constants and variables) of the application domain. The flow relation is a four-tuple, the first two of which *(enable-out (In0)* and *enable-in (In1))* are used to describe sets of formulas that must not *(In0)* or must *(In1)* be present in the agent's beliefs for an event to be enabled (i.e., preconditions); the latter two *(external results (Out0)* and *internal results (Out1))* are used to describe the physical and the mental results of the denoted action, respectively. Figure 2.10 shows a representation of the *Puton* operator in this formalism, which is graphically represented as a net in Figure 2.11. Note that the representation of *Puton* does not specify any beliefs to be deleted. Drummond proposes to compute *reconciliation sets* via a

```
(defmla on-plan
      (→ a (plan-instance ?plan (ach-on-schema ?x ?y) ?sup)
              (and
                 (task (clear-step ?plan ⟨⟩
                       (lambda () (achieve '(clear ?x))) ⟨⟩)
                 (task (find-space-step ?plan) ⟨⟩
                       (lambda () (achieve '(space-for ?x ?y))) ⟨⟩ )
                 (task (move-step ?plan) ⟨⟩
                       (lambda () (puton ?x ?y)) ⟨⟩)

                 (subtask (move-step ?plan) ?sup)
                 (subtask (clear-step ?plan) ?sup)
                 (subtask (find-space ?plan) ?sup)
                 (successor (clear-step ?plan) (move-step ?plan))
                 (successor (find-space-step ?plan) (move-step ?plan))
                 (main (move-step ?plan) ?sup))))

(defmla puton-defn-1
      (→ c (on ?a ?x)
              (mod-manip ?task (puton ?a ?b) ⟨ '(on ?a ?x)⟩ ⟨⟩)))

(defmla puton-defn-2
      (mod-manip ?task (puton ?a ?b) ⟨⟩⟨ '(on ?a ?b) ⟩))

(defmla to-do-ach-on
      (to-do ?task (achieve '(on ?x ?y)) ⟨⟩
              (do-subnet (ach-on-schema ?x ?y) ⟨⟩)))

(defmla on-plan-prereqs
      (→ a (plan-instance ?plan (ach-on-schema ?x ?y) ?sup)
              (and
                 (prereq ?sup (clear-step ?plan) (move-step ?plan)
                       ⟨ '(clear ?x) ⟩)
                 (prereq ?sup (find-space-step ?plan) (move-step ?plan)
                       ⟨ '(space-for ?x ?y) ⟩))))
```

Figure 2.7: NASL rules for achieving (ON ?a ?b) (from [McDermott, 1978], p. 91)

occurs(PUTON(x,y), t)
  only if, holds(CLEAR(x), t1) where t *finishes* t1
      holds(CLEAR(y), t2) where t *finishes* t2
  and the effect is holds(ON(x,y), t3) where t *meets* t3.

Figure 2.8: Allen's representation of *Puton* (from [Allen and Kautz, 1985] p. 263)

If holds(CLEAR(x), TCx) and holds(ON(y,x), TOyx),
then TCx (m mi ><) TOyx
i.e. TCx and TOyx cannot overlap in any manner.

Figure 2.9: A domain rule expressed as a temporal constraint (from [Allen and Koomen, 1983], p. 743)

consistency maintenance function. Adding of new beliefs as a result of performing an action may yield an inconsistent belief set. In conjunction with a set of user specified *domain constraints* (for example, "it is not possible for an object to be simultaneously on top of two objects"), the consistency maintenance function computes possible deletion sets (which are subsets of old beliefs) such that deletion of any one of the deletion sets and addition of an action's results to the set of old beliefs will result in a consistent belief set. Drummond assumes that a *reconciliation selection* function exists (it is mentioned that it will be based on heuristics and will be extremely domain-dependent). Drummond also describes a mechanism for analyzing plan behaviors in his later work [Drummond and Bresina 1990].

Plan Net = ((Belief Set, Event Set; Flow Relation), Execution Advice)

Execution Advice = ϕ

Belief Set = {block(x),clear(x),clear(y),on(x,y),weight(x,w)}
Event Set = {puton(x,y)}

Flow Relation = (In0,In1,Out0,Out1)

In0 = ϕ
In1 = { (block(x), puton(x,y)),
    (clear(x), puton(x,y)),
    (clear(y), puton(x,y)) }
Out0 = { (puton(x,y), on(x,y)) }
Out1 = { (puton(x,y), weight(x,w)) }

Figure 2.10: *Puton* described in the plan net formalism

Nilsson [Nilsson 1988] is developing an architecture called *action networks* for modeling agents whose behavior is guided by their goals and changing environmental conditions. He calls such agents *teleo-reactive*. The modeled agent is supposed to have real-time control of some environment with which it is connected via sensors and effectors. The control

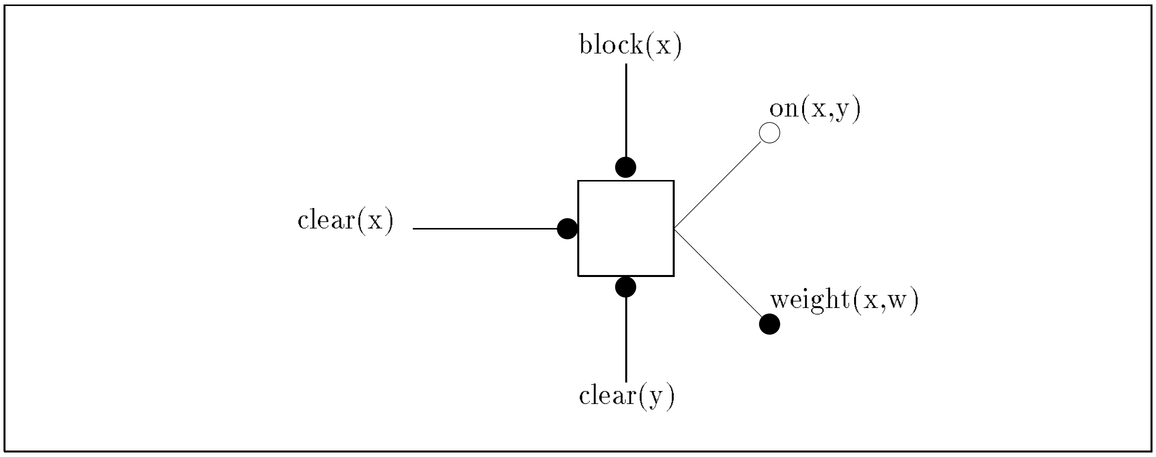

Figure 2.11: The *Puton* plan net, graphically represented

program is an action network modeled using combinational logic circuits. The agent's beliefs are represented by a *belief structure*, which is a collection of binary-valued beliefs. Similarly, there is a set of propositions that forms the *goal structure* of the agent. Actions are represented by a special kind of logical gate called an *action unit*. Each action unit (see Figure 2.12) has three kinds of inputs: *precondition inputs* (B_i), which originate in the agent's belief structure; a *purpose input* (P), which also originates in the belief structure; and a *goal* or *calling input* (G), which originates in the agent's goal structure. The action unit has one output, which may represent a calling input for some other action unit or a call to execute some primitive action. Since these are logical values, a call to a primitive action denotes the triggering of some effector in the external environment. The output of an action unit is active (i.e., the action is performed) if and only if the purpose input is *off*, the goal input is *on*, and some function of precondition inputs (usually an AND) is *on*. The action unit representing *Puton* is shown in Figure 2.13. An action network is created for each task. Nilsson is developing a programming language called *ASTRAL* that can be used to describe actions. Action networks can then be created from ASTRAL programs just as procedural nets are created from SOUP code in NOAH. This architecture is being developed to solve the problems of "real-time AI". Although this architecture falls under the class of *minimalist* approaches we have discussed it here because of some of the problems Nilsson explicitly identifies. One of the major concerns that needs to be solved, Nilsson mentions, is how the connection between a general purpose reasoning system and such an acting system is to be achieved (see [Nilsson 1988], p. 49).

Vere [Vere and Bickmore 1990] has constructed a "basic agent" (HOMER) that integrates limited natural language understanding and generation, temporal planning and reasoning, plan execution, simulated symbolic perception, episodic memory, and some general world knowledge. HOMER is a simulated autonomous underwater cognitive agent that behaves as a submarine in a simplified simulated Seaworld. The temporal task planner and reasoner are based on DEVISOR V [Vere 1985]. The actions of HOMER are represented using *state transition semantics*, which is basically a frame-like description of verbs augmented by a description of the effect of the action on the world state. Thus, the agent maintains a dual representation for executable actions—a linguistic model and a planner's model (see Figure 2.14).

The PRS system [Georgeff *et al.* 1987, Georgeff *et al.* 1985, Georgeff and Lansky 1987,

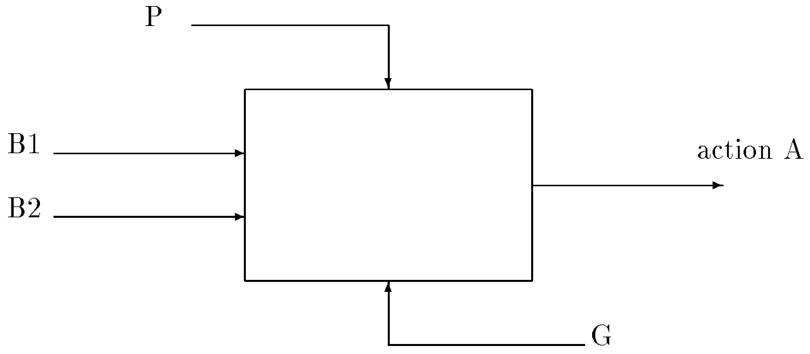

Figure 2.12: An action unit (from [Nilsson, 1988], p. 24)

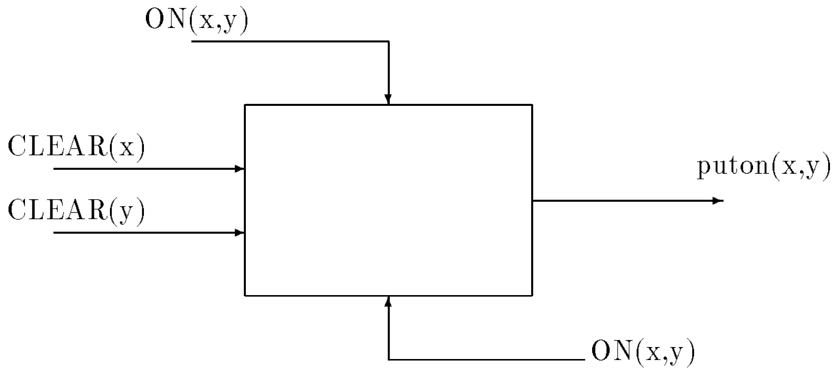

Figure 2.13: A *Puton* action unit

Georgeff 1988] employs the notion of a *process* (also called *Knowledge Areas*) to represent plans (or plan schemas). A process is a declarative procedure specification that can be used to generate a sequence of world states (called *behaviors*). A process is modeled by a transition network whose nodes represent control points and whose arcs are labeled by *action descriptions*. A complete *process description* consists of a set of control points, a set of arcs, a transition function called the *process control function*, a start node, a set of final nodes, and a set of action descriptions (also called *goal descriptions*). The three types of action descriptions used are: achieve a given state condition p (represented as `(! p)`), test for p (represented as `(? p)`), and preserve p (represented as `(# p)`). In addition, a metalevel action description believe p (represented as (=> `p`)) is used to add facts to the database. For example, a process for *Puton* is represented by the process description shown in Figure 2.15. Since a process is described declaratively, one can have *process assertions* of the form $c\langle P\rangle g$, where P is a process description, c is a invocation precondition for P, and g is an effect. A pictorial representation of the process assertion for *Puton* $(TRUE\langle Puton\rangle(on\ x\ y))$ is shown in Figure 2.16. A process assertion expresses a declarative fact about the effects of performing a certain sequence of actions under certain conditions. Some of the preconditions of process assertions can be used to specify *reactive invoking conditions*; i.e., the associated

```
(WORD put
  linguistic definition
  (VERB
    ATTRIBUTES
      ((TRANSITIVE YES)
       (AGENT.POSSIBLE? T))
  action schema
  (ACTION (.x .y)
  ! (SCHEMA STATE.CHANGE
  ! ! (CONSTRAINTS
  ! !           (ISA .x BLOCK)
  ! !           (ISA .y BLOCK))
  ! ! (ANTECEDENT
  ! !           (CLEAR .x)
  ! !           (CLEAR .y))
  ! ! (CONSEQUENT
  ! !           (NOT (CLEAR .y))
  ! !           (ON .x .y))
  ! )
  )
)
```

Figure 2.14: The dual representation of *Puton* in HOMER

process will be invoked as soon as the system believes the invoking condition. This is responsible for PRS's reactive behavior. An *interpreter* manipulates these representations by maintaining a stack of invoked processes that represents the system's intentions.

Another approach, discussed in [Lifschitz 1987], suggests that we should view STRIPS as a form of logic, and STRIPS operators as rules of inference in this logic. Here, the notion of planning is governed by the operators representing the rules of inference and that of acting is defined by associating action-functions with each operator. These functions, when interpreted, will model the appropriate state transitions for each action. While it is good to take this approach to prove formal properties about the model, it is not clear as to how these ideas relate to the physical realization of such logics, in particular, how the rules of inference are used by the inference mechanisms to exhibit planning (and decision making). Moreover, there is no mention of how they relate to the inference rules of the world model.

2.4 BDI Proposals

Recently there have been attempts at developing models of rational agents to which are attributed the psychological attitudes of belief, desire, and intention (BDI). Much of this work is being done as a follow up to Bratman's analysis of the important role of intentions (besides beliefs and desires) in modeling rational behavior [Bratman 1987, Bratman 1990].

```
Puton = < {S,N1,F},
          {arc1,arc2},
          <<S,arc1,N1>, <N1,arc2,F>>,
          S,
          {F},
          <<arc1,(! ((clear x) ∧ (clear y)))>, <arc2,(=> (on x y)))>>>
```

Figure 2.15: A PRS process description for *Puton*

Cohen and Levesque, based on Bratman's ideas, present a formal analysis of basic principles governing the rational balance among an agent's beliefs, actions, and intentions [Cohen and Levesque 1987, Cohen and Levesque 1991]. They use a possible-worlds model for beliefs and goals, and propose to model intentions as a kind of persistent goal and to explore various properties of intention (such as revising intentions, joint intentions, joint commitments).

Bratman observed that agents in dynamic environments benefit from having plans because their plans can constrain the amount of subsequent reasoning they need to perform [Bratman 1987]. Bratman *et al.* [Bratman *et al.* 1988, Pollack 1991] describe an abstract belief/desire/intention (BDI)-architecture that accounts for practical reasoning of resource-bounded agents. Their work concentrates mainly on the control paradigm of an architecture of a practical reasoning agent that has to perform means-ends analysis, weigh competing alternatives, and address the problem of resource boundedness during deliberation.

Rao and Georgeff, also exploring Bratman's ideas, are using another possible-worlds formalism, with a branching time logic, to capture the relationships between beliefs, desires, and intentions [Rao and Georgeff 1991b, Rao and Georgeff 1991a]. They have shown that intentions have to be modeled as independent entities so as to overcome some of the problems that arise in Cohen and Levesque's model [Rao and Georgeff 1991a]. In [Rao and Georgeff 1992], they have provided an abstract architecture for rational agents based on their theory. The architecture is supposed to provide a link between the theoretical foundations and the PRS system. However, the implementation of the architecture is too simplistic and far from the functionality of PRS.

2.5 Observations

In the research surveyed above, there are clear distinctions in the approaches used and the problems attacked. Moore and Lifschitz are concerned about formal aspects of reasoning about knowledge and action. Their models stress the semantics of the representations used. Allen, Drummond, Nilsson, and Georgeff are exploring different architectures or models to approach the problem of reasoning about actions and beliefs. The choice of models used is motivated by different requirements demanded of the modeled agents. Allen developed a temporal logic to be able to reason about actions and events in a temporal framework; Drummond uses net theory to account for acquisition of sensory information and also to model conditional and iterative behaviors; Nilsson is using action nets to model

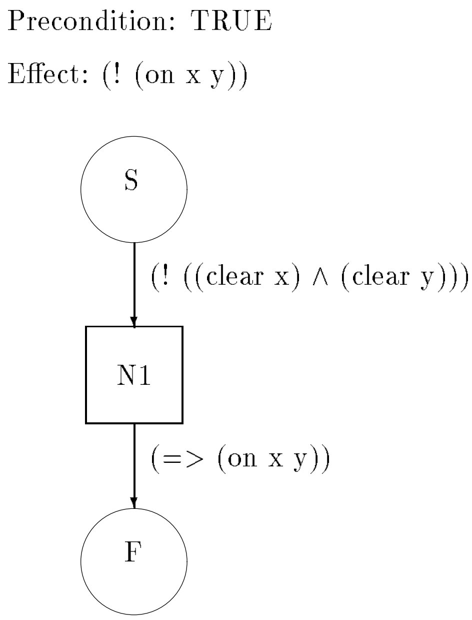

Figure 2.16: A pictorial representation of the *Puton* process assertion

teleo-reactive agents; and Georgeff uses procedural networks for modeling real-time process control agents. McDermott developed an informal theory of planning and acting and described it using the NASL interpreter. Vere is trying to build autonomous modeled agents that integrate various state-of-the-art AI faculties.

While all the approaches represent major advances in planning, acting, and reasoning about actions and beliefs as described, the need for an integrated reasoning and acting system as pointed out in Section 3.2 still remains unfulfilled. Moore's theory provides a model for reasoning about knowledge and action, but that is only part of the picture. Such a reasoning agent should ultimately perform and discuss the actions it reasons about. There is no attempt to formalize reactivity in the model, which is a major concern for "effective agents." In the work presented here, we will develop a model of acting where actions of the agent can lead to other actions, and where acquisition of new beliefs may also lead to actions.

Chapter 3

Acting and Inference

In the past decade, research in AI planning has proceeded independently of research in knowledge representation. A survey of AI systems reveals that it is somewhat awkward to do acting in reasoning (or logic-based) systems (but it is convenient to talk about representational and reasoning issues using them), and it is awkward to study reasoning and representational issues in systems designed for acting/planning. Thus, most "good" planning/acting systems are "bad" knowledge representation and reasoning (KRR) systems and vice versa. For example, in a recent symposium on "Implemented KRR Systems" [Rich 1991b], out of a total of 22 KRR systems presented, only 4 systems had capabilities for representation and reasoning about actions/plans (RHET [Allen 1991], CYC [Lenat and Guha 1991], CAKE [Rich 1991a], and SNePS [Shapiro 1991b]). My work has emerged out of studying reasoning and acting issues in the context of SNePS. In this chapter, we will examine the issues that arise when one tries to incorporate planning/acting capabilities in a system primarily designed for representation and reasoning (and vice versa). We will use the SNePS acting system [Kumar *et al.* 1988, Shapiro *et al.* 1989b, Shapiro *et al.* 1989a, Kumar *et al.* 1990, Kumar and Shapiro 1991a] as a case study. The SNePS acting system successfully integrates representations and reasoning about beliefs, acts, and plans. However, as we will point out, it still suffers from several drawbacks. The observations we will make will lead us to reexamine the relationship between acting and inference. We will conclude the chapter by stating the underlying premise of this work. The rest of this thesis will be devoted to a concrete development of the ideas presented in this chapter.

3.1 Acting Is Different from Inference

Systems that do both reasoning and acting either use different modules—an inference engine for reasoning and an acting executive (also called a plan/act interpreter) for acting; or they tend to confuse the distinction between acting and inference and claim that 'rules' in these systems play the role of 'plans' as well as 'deduction rules'. As discussed in the previous chapter, systems of the former kind typically employ different languages for representing beliefs and actions and consequently different modules for reasoning and acting. Even when one uses the same language for representing beliefs, acts, and plans, we still need separate modules for reasoning and acting. This is discussed in detail later. Systems of the latter kind are discussed first.

3.1.1 Rules as Plans for Acting

It is recommended in Prolog (and production systems) that rules be used to represent plans as well as reasoning rules.[1] For example, consider standard (non-concurrent) Prolog or some arbitrary production-rule system. Such a system relies on a semantic ambiguity between the logical *AND* and the procedural **and then** (or *sequencing*). For example, in Prolog, the rule

$$p(X) \ :- \ q(X),\ r(X). \tag{3.1}$$

either means "For any X, $p(X)$ is true if $q(X)$ and $r(X)$ are true" or it means "For any X, to do p on X, first do q on X and then do r on X." Guaranteeing the proper ordering of behavior in the procedural interpretation is only possible by giving up the freedom to reorder, for efficiency, the derivations of $q(X)$ and $r(X)$ in the logical interpretation. The example is made more striking by appending

$$q(Y) \ :- \ s(Y),\ t(Y). \tag{3.2}$$

$$r(Z) \ :- \ s(Z),\ u(Z). \tag{3.3}$$

and considering the query

$$?-\ p(a). \tag{3.4}$$

Under the logical interpretation, it would be efficient for the system to try finding if $s(a)$ holds only once, instead of once when rule 3.2 is being used and once when rule 3.3 is being used. However, as a plan for acting, it is reasonable to do $s(a)$ twice—there are many plans that require performing the same act more than once. The strict left-right, depth-first control structure of most implementations of Prolog guarantees that all the subgoals are activated and evaluated as many times as they are encountered regardless of whether the rule is supposed to be a reasoning rule or a plan for acting.

We can see that the use of rules as plans for acting in Prolog depends on a pun: ',' is simultaneously interpreted as the logical operator *AND* and as the programming language *sequencing* operator. This pun also leads to ontological problems—the subgoals $q(X)$, $r(X)$, etc, represent beliefs as well as actions, depending upon the interpretation applied to them. Moreover, the propositional attitudes of *belief* and *intention* cannot be modeled correctly. While *belief* is an attitude towards a proposition, *intention* is an attitude only towards acts.[2] Since there is no ontological distinction between beliefs and acts in Prolog or other production systems, there is no way to explicitly model the two attitudes.

3.1.2 Employing an Acting Executive

It is desirable to have a reasoning engine that optimizes its deductions by "remembering" past inferences (as long as they are valid). This is the way SNIP, the SNePS Inference

[1] The example as well as the discussion presented in this section (except the last paragraph) are taken directly from [Shapiro *et al.* 1989a]. This discussion, in a slightly different form, also appears in [Shapiro and Rapaport 1991]. A presentation of this by Stuart Shapiro at a research meeting in 1986 is what motivated me towards this research.

[2] A more detailed analysis of intentions was presented by Castañeda who introduced *practitions* to refer to intentions in a more general sense than the one we are using. Our usage of the term *intention* is not inconsistent since practitions include an action constituent as an attribute. See [Castañeda 1975] for more details.

Package, has been implemented [McKay and Shapiro 1981, Choi 1993]. In the example above, SNIP would not attempt to work on $s(a)$ the second time. Therefore, SNIP rules cannot be used for plans of acting. However, under the procedural interpretation, it may be perfectly reasonable to perform $s(a)$ twice. So, the behavior that optimizes logical reasoning destroys procedural rule following. The following is from [Shapiro and Rapaport 1991] (p. 241):

> It should be pointed out that the above design of SNIP is not defective for two reasons. First, SNIP's refusal to work on the same subgoal more than once is not only efficient, it is one of the mechanisms that allow SNIP to work with unrestricted recursive rules [Shapiro and McKay 1980, McKay and Shapiro 1981], a facility Prolog does not have. Second, as already elaborated above, the Prolog pun amounts to a semantic confusion between the logical notion of truth (or belief) and the practical notion of action.

It is possible to extend the ontology of a representational formalism to represent acts and plans. This facilitates the same reasoning engine to reason about beliefs, acts, and plans. However, when it comes to acting, one still needs to have an acting executive (act/plan interpreter) that carries out the modeled agent's intentions. This is the approach taken in work on the SNePS Acting System (SNACTor) [Kumar *et al.* 1988, Shapiro *et al.* 1989b, Kumar *et al.* 1990, Kumar and Shapiro 1991a, Kumar and Shapiro 1991b]. We will briefly summarize it below.

3.1.3 The SNePS Acting System

The representations of plans and acts described in [Kumar *et al.* 1988, Shapiro *et al.* 1989b, Kumar *et al.* 1990, Kumar and Shapiro 1991a, Kumar and Shapiro 1991b] are such that an agent can be made to reason about beliefs as well as actions and plans in a uniform manner. The architecture of the SNePS Acting System is shown in Figure 3.1. The SNePS actor operates in a world inhabited by itself (i.e., a single-agent world). The agent has beliefs that are stored as SNePS propositions in the agent's belief space (called a *context*, see [Martins and Shapiro 1988]). For example, the proposition "A is on top of B" is represented by a node (say $\mathsf{M23}$) whose structure is:

$$\mathsf{M23} \equiv \{\langle \mathsf{REL}, \{\langle \mathsf{LEX}, \mathsf{ON}\rangle\}\rangle, \langle \mathsf{ARG1}, \{\langle \mathsf{LEX}, \mathsf{A}\rangle\}\rangle, \langle \mathsf{ARG2}, \{\langle \mathsf{LEX}, \mathsf{B}\rangle\}\rangle\}$$

A pictorial representation of $\mathsf{M23}$ is shown in Figure 3.2. The node is stored in the agent's belief space as a *supported wff* containing a set of *hypotheses*, an *origin set*, and a *restriction set* (see [Shapiro and Martins 1990]). Thus, $\langle \mathsf{M23}, \mathsf{HYP}, \{\mathsf{M23}\}, \{\}\rangle$ represents the supported wff $\mathsf{M23}$ in the agent's belief space, as a hypothesis whose origin set contains the node itself and whose restriction set is empty. This enables SNeBR—the SNePS system for Belief Revision, an assumption-based truth maintenance system [Martins and Shapiro 1986a, Martins and Shapiro 1986b, Martins and Shapiro 1988], to ensure that the agent's belief space is always consistent.[3]

[3] During the course of acting, beliefs are removed and added. This is done using SNeBR operations. For example, one of the things SNeBR takes care of is that when a belief is removed as a consequence of performing an action, all propositions derived using that belief are also removed. See [Kumar and Shapiro 1993].

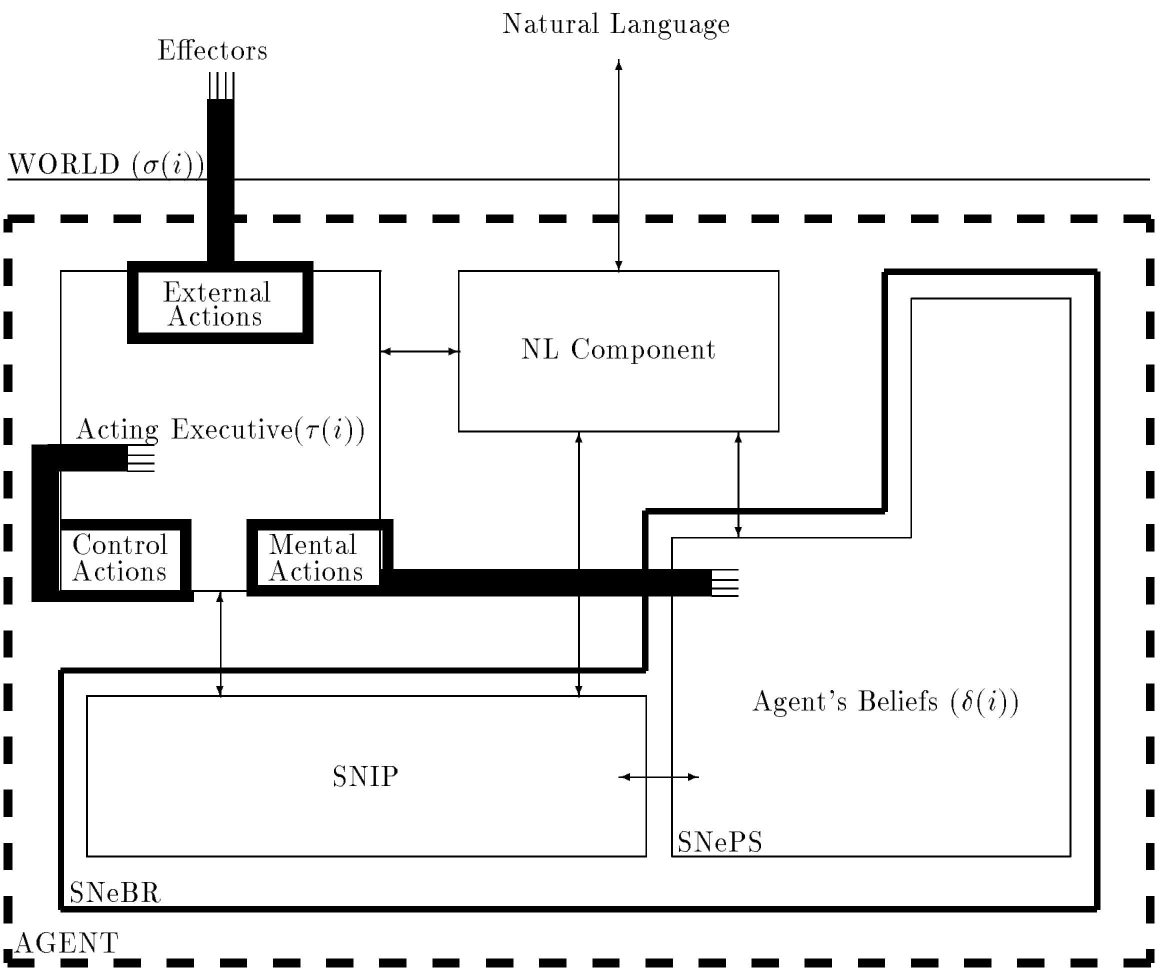

Figure 3.1: Architecture of The SNePS acting system.

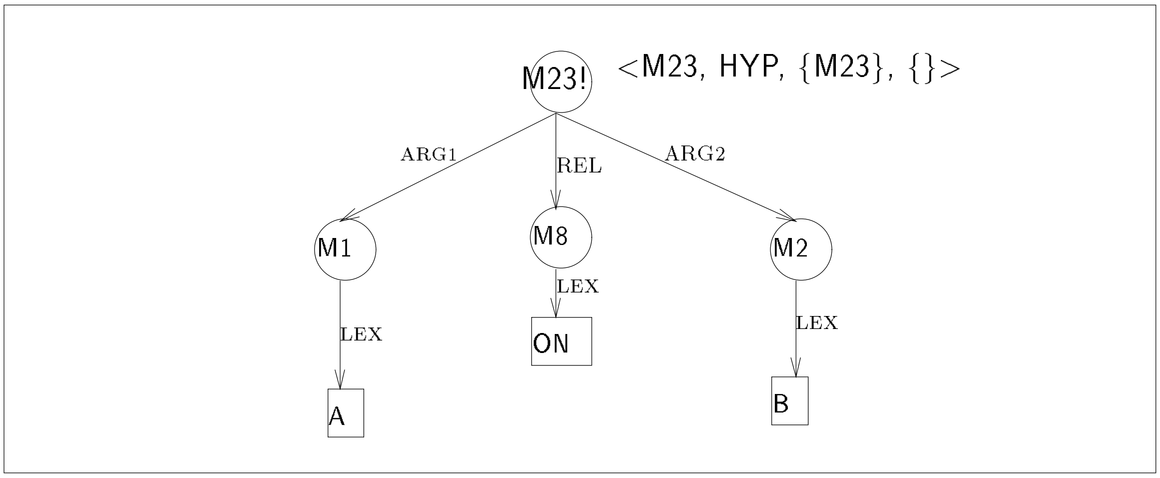

Figure 3.2: A Pictorial representation of the supported wff M23.

An important feature of SNePS representations is that M23 is interpreted as a "mental entity" (or an "intensional object"). The agent can have beliefs only about intensional objects (see [Maida and Shapiro 1982, Rapaport 1986, Shapiro and Rapaport 1987, Shapiro 1989]). All interaction with the agent is done using the natural language component (see Figure 3.1). Sentences are parsed by a grammar (written as an ATN), translated into SNePS networks via SNePSUL (the SNePS User Language) commands, which form beliefs in the agent's belief space. World-model rules for reasoning in the agent's belief space are also translated and represented as the agent's beliefs. An inference rule in SNePS is a structured proposition node that is a specification of antecedent and consequent beliefs using appropriate quantifiers and connectives. SNIP, the SNePS Inference Package, can do forward, backward, or bidirectional inference using the same set of rules. Acts and plans are treated as mental objects. This enables the agent to discuss, formulate, use, recognize, and reason about acts and plans [Shapiro *et al.* 1989b]. The representations for acts, goals, and plans build upon and add to the intensional propositional representations of SNePS. This framework is able to tackle various tasks in a uniform and coherent fashion.

Actions are classified as being *physical*—those that affect the outside world, *control*—those that affect the acting executive, and *mental*—those that affect the set of beliefs. Plans (or complex acts) comprising a set of physical and control actions are represented as structured nodes. Decompositions of plans and goals as well as preconditions and effects of actions are also specified using propositions.

Requests to perform an action are serviced by the acting executive (see Figure 3.1). The request (which is represented as an act node) gets scheduled on an acting queue maintained by the executive. This represents the agent's intentions. Plans are structured using *control actions* that, when interpreted, affect the queue of intentions. The repertoire of control actions includes sequencing (snsequence), conditional (snif), iteration (sniterate), and a few others (see [Kumar *et al.* 1988]). *Physical actions* affect the external world via their associated procedures. The acting executive uses SNIP to derive plans, plan decompositions, and the effects and preconditions of actions. It schedules *mental actions* to believe effects of actions. It also schedules acts to achieve preconditions of actions in case they are not satisfied.

SNIP, the SNePS Inference Package [Hull 1986, Ferreira *et al.* 1989, Choi 1993], is implemented on top of MULTI [McKay and Shapiro 1980a] which is a simulated, LISP-based multiprocessing system that maintains a queue of parallel processes, executing processes from the process queue until the queue is empty. An inference is carried out by SNIP by activating a node[4] as an appropriate process. Executing such processes may result in requests or reports to other node processes in the system. When a requesting process receives a report it had asked for, a conclusion is drawn and added as a derived belief to the belief space. This completes an inference.

3.1.4 A Critique of the SNePS Acting System

An agent modeled using the SNePS acting system satisfies some of the desiderata listed in Chapter 1. Let us evaluate each of these.

- *The modeled agent is able to represent and reason about its beliefs and those of others.* SNePS already had existing facilities for this.

- *The agent is able to represent and reason about actions and plans the same way as it does its beliefs.*
  Acts, actions, and plans are terms in the language of SNePS just like entities, individuals, and propositions. This enables the agent to have beliefs about acts and plans. Some of these beliefs can be the preconditions and effects of actions. Decomposition of complex acts and goals into plans are also represented as propositions. Thus, using a single reasoning system (SNIP), the agent is able to reason about acts and plans.

- *The agent is able to decompose complex actions into plans.*
  While the agent as described does not have a plan-generation capability in the traditional sense, it is able to deduce plans to achieve specific goals as well as derive plan decompositions for abstract (complex) actions (hierarchical planning).

- *The agent is able to perform the intended actions in a real world.*
  The modeled agent is able to perform intended actions in a simulated world where it is the only actor. Acting in a multi-agent environment is more complicated. The modeled agent is missing some of the basic capabilities, like sensing and reacting, to achieve that. While it is not our immediate intention to build full-fledged, real-world acting capabilities, we have developed some basic primitives that enable the agent to sense, act, and react in such a domain. This is discussed below.

- *The agent is able to discuss its actions, plans, and beliefs with other agents in the world.*
  The representations for acts, actions, and plans were designed so that the same representations could be used by the agent to discuss, reason, and act. The agent is capable of discussing its beliefs, acts, actions, and plans with the user.

[4] In the case of forward inference, the node is the belief to be added to the belief space of the agent. In the case of backward inference, it is the node representing the query.

The resulting system has a uniform representation of beliefs, plans, and acts. The AI techniques employed by the implemented architecture include path-based inference, node-based inference, and belief revision. From the perspective of an AI architecture, we have gained very little. Even though several facilities are integrated, the overall organization of the architecture (reproduced here in Figure 3.1) is quite traditional in that beliefs are represented using SNePS, an inference engine can operate on these beliefs, and an acting executive interprets actions and plans and governs the behavior of the inference engine when the agent is deliberating about actions and plans during the course of acting.

The SNePS actor is a model of an agent that is a "pure effector" living in a world populated only by itself. However, a rational agent has to be able to act in a real world inhabited by other agents (effector systems). Also, the world is being affected by the constant occurrence of natural phenomena. To account for these, rational agents should be endowed with sensory capabilities so they can update their beliefs about the world. Regardless of the sensory interface between the agent and the world, sensory information results in the formation of new beliefs. These beliefs are acquired on a continual basis. There can also be sensory activity intended by the agent in order to acquire some information. A model of a rational agent should be able to account for both types of sensory activity.

In trying to incorporate sensory capabilities into a system, it is a natural tendency to think of creating a new module for the architecture that would provide a sensory interface with the world. However, how is the acquired sensory information treated by the agent? We need a model of attention to filter beliefs acquired by sensory activity. As a consequence, some beliefs will simply be added to the agent's belief space. Also, adding some beliefs may require the agent to perform some reactive actions. An agent's reactions should be governed by its "desires".

In the SNePS acting system (and the framework of classical planning and acting), the modeled agent is always under the control of an acting executive. Once the agent is asked to do something, it would do it; only after it is done and the agent has nothing left to do in its intention queue would it wait for another command. Also, if, in the middle of carrying out its intentions, the world changes, it would have no way of detecting that change, however crucial it may be to the completion of its current plans. (The STRIPS triangle tables can be used to successfully complete a plan, but the agent will never know that some of its actions are correcting earlier failures.) The agent may intend to do something and try to do it, but a failure of that act would result in undefined consequences. To deal with such issues, a mechanism is needed that would enable an agent to reason about or respond to a new belief in the middle of doing whatever its current intentions are. Reacting to a new belief may result in a new immediate intention for performing some action (which may or may not be related to its current plan). This requires a mechanism that would allow us to model the forming of an intention as a result of some new belief entering the agent's current set of beliefs. Currently in SNePS, adding a new belief can lead to forward inference by chaining through the antecedents of the rules that get triggered (or are already active because of some earlier bidirectional inference). However, this forward inference will only lead to the formation of new beliefs, since reasoning rules in SNePS only pass a belief status from the antecedents to the consequents (or the other way round in the case of backward inference).

The above discussion clearly identifies the need for a model that can use the beliefs and desires of an agent to create new intentions to perform some actions, i.e., a model where an agent can form an intention during reasoning in order to react to something or to acquire new beliefs. The SNePS acting system only does inference in the service of acting. We also need to be able to act in the service of inference. We have outlined a scenario where there needs to be a closer relationship between an agent's beliefs, actions, and intentions. This leads us to re-examine the relationship between acting and inference.

3.2 The Relationship between Acting and Inference

First, let us look closely at the mechanism of inference. Reasoning is the process of forming new beliefs from other beliefs using inference rules. The connectives and quantifiers of the inference rules govern the derivation of new beliefs. Reasoning can be looked at as a sequence of *actions* performed in applying inference rules to derive beliefs from other beliefs (this was first explored in [Morgado and Shapiro 1983]). Thus, an inference rule can be viewed as a rule specifying an *act*—that of believing some previously non-believed proposition—but the "believe" action is already included in the semantics of the propositional connective. Thus, another way of characterizing an inference engine is as a *mental actor* or a *mental acting executive*. During backward chaining, the mental acting executive forms the intention of believing the consequents of a rule if its antecedents are satisfied (i.e., preconditions are fulfilled). Similarly for forward chaining.

Alternatively, plans can be viewed as rules for acting. Reasoning rules pass a *truth* or a *belief* status from antecedent to consequent, whereas acting rules pass an *intention* status from earlier acts to later acts. McCarthy has also suggested that inference can be treated as a mental action [McCarthy 1986]. This suggests that we may be able to integrate our models of inference and acting by eliminating the acting executive. While it may sound appealing to redefine all the inference mechanisms as a bunch of explicit plans (under the new interpretation, this is theoretically possible), we will refrain from doing so. The trade-off here is that of the long-standing tradition of inference being a basic primitive in an AI system as well as the optimized implementation of inference (where previous deductions are not repeated, if valid), which is a necessity. It should be pointed out at this stage that this is not the same as a Prolog inference engine. The unified acting and reasoning engine operates on representations that make a distinction between beliefs and acts. Accordingly, it applies the attitudes of belief and intention on appropriate entities.

Based on the above ideas, we will develop a Belief-Desire-Intention (BDI) architecture that can be used to model agents that are able to represent beliefs and desires. We will describe an ontology for the representation of beliefs, goals, acts, plans, and rules. We will also develop a computational model, called a *Rational Engine*, that implements the unified model of acting and inference. The Rational Engine implements the underlying logic as well as notions of intentionality.

3.3 Discussion

We started this chapter by pointing out that most "good" planning/acting systems are "bad" knowledge representation systems and vice versa. We showed that Prolog-like systems do not make an ontological distinction between beliefs and acts and also rely on a confusion between inference and acting. We also described an approach that bridges the representational gap by providing representations for beliefs, plans, and acts in a single formalism. While this enables reasoning about beliefs, acts, and plans, it still requires different "interpreters" for acting and reasoning. Although this approach significantly advances the capabilities of traditional KRR systems, it requires separate reasoning and acting modules. In the last section, we presented an outlook that proposes to treat an inference engine as a mental acting system. Given proper ontological distinctions between beliefs, acts, and plans, we have proposed that it may be feasible to build a unified reasoning/acting module that reasons as well as acts. I.e., it does both mental as well as physical acting, the former including inference.

In what follows, we first present a general BDI architecture based upon the above ideas. We define a uniform ontology of beliefs, acts, rules, and plans. We also define a uniform reasoning/acting module, called a *Rational Engine*, that operates on the unified representations. We then outline a parallel implementation of such a rational engine. The resulting model is implemented by extending the SNePS inference engine into a system called SNeRE (for SNePS Rational Engine). This is what we call the SNePS BDI architecture, presented in Chapters 9 and 11.

Chapter 4

Towards a BDI Architecture

In the previous chapter, we discussed the relationship between acting and inference. In this chapter, we begin the development of an AI architecture that is defined using a unified view of acting and inference. We have already pointed out that having a unified ontology for representing beliefs, goals, acts, and plans is not sufficient. A unified methodology for reasoning and acting is also required. This also provides an opportunity to further extend the unified ontology—by introducing representations that can transform beliefs/acts into other beliefs/acts. Rather than extending SNePS into a BDI architecture, we will develop a general BDI architecture, called the *Object-based Knowledge* (*OK*) *BDI architecture*. The new architecture has a unified representational formalism (called the *OK formalism*) as well as a unified reasoning and acting methodology (called the *OK Rational Engine*). After we have defined the formalism, we will show that the SNePS representational formalism is an instance of the OK formalism in that both formalisms follow the same representational principles and are isomorphic. In addition to the goals listed in Chapter 1, the general architecture attempts to satisfy several other objectives at the same time (see Section 4.1 below)—it will be developed using object-oriented design principles. This has several advantages. Besides serving as an object-oriented definition of SNePS itself, it exploits all the advantages of object-oriented design, namely, extendibility of architecture and amenability towards concurrency.

4.1 An Object-Oriented Approach

The architecture we are about to develop will fulfill all the goals stated in Chapter 1. In addition, an object-oriented design offers the following:

- The vocabulary of the agent's beliefs as well as acts will be user-definable and extendible.
- The logic underlying the reasoning/acting capabilities of the modeled agent will be user-definable and extendible.
- The reasoning/acting capabilities will implicitly account for a closed-loop interaction between the agent and its environment.
- The agent's action ontology will be powerful enough to express the above considerations.

We should point out here that this approach is different from the design of frame-based systems that also exploit object-oriented principles. In frame-based systems, class hierarchies are defined directly corresponding to what is being represented. For example, since all birds are animals, the class of birds is defined as a subclass of animals, so that properties/slots can be inherited. Thus, in frame-based systems, the principles of object-oriented design apply directly to the domain being modeled. In the approach taken here, we are going to apply object-oriented principles to the design of KRR systems themselves. In this approach, the class hierarchy is defined based on entities and components of the KRR system rather than any specific domain. Inheritance, as employed by frame-based systems, will be modeled using the underlying logic (as is traditional for most KRR formalisms) rather than a terminological approach. This work is unique in this respect. After we have presented a design of the architecture, we will present an analysis of this approach. We begin by presenting the basic class hierarchy.

4.2 Conceptual Entities

A conceptual entity is anything a cognitive agent can think about. In intensional representational formalisms (like SNePS), conceptual entities include individuals, variables, and acts, as well as beliefs. Thus, in our conceptual class hierarchy, the topmost class is called Conceptual Entity Term. All instances of this class as well as all instances of its subclasses denote intensional entities in the domain of discourse of the modeled agent.

Definition 1:

Class: Conceptual Entity Term

Superclass(es): None.

Slots: All instances of this class have at least the following slots:

> LABEL: Every conceptual entity represented is identified by a unique label. A label is denoted by an uppercase letter followed by a number. For example, E1, E2, B1, B2, I1, I2, A1, A2, V1, V2,..., etc.

Methods: If o is an instance of this class (or its subclasses) then

> LABEL(o): Returns the contents of the LABEL slot.

Syntax: Objects that are direct instances of this class are expressed using the LABEL slot. Labels are assigned depending on the type of entity being represented. The following convention will be used:
E1, E2, ...: denote basic (un-named) entities
I1, I2, ...: denote named individuals (see below)
B1, B2, ...: denote propositions (see below)
A1, A2, ...: denote acts (see below)
V1, V2, ...: denote variables (see below)
All instances of this class are terms in the underlying logic.

Description: A conceptual entity is anything a cognitive agent can think about. If E1 is the label of an instance of this class, then the intensional entity denoted by it will be written as $[\![E1]\!]$.

Since all instances of all classes will have a unique label, we will use the LABEL slot as the standard reference. The entities represented by instances of the class (or subclasses) are intensional in the the sense 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" [Shapiro and Rapaport 1987]). 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" [Shapiro and Rapaport 1987]). In general, we would like to say that entities represented by instances of the class of Conceptual Entity Term are extensional as well as intensional entities. One way to "link" up entities, if desired, to the external world is to give them a linguistic attachment—a name. These form a subclass of Conceptual Entity Term and are described next.

4.3 Named Individuals

Named individuals are entities the agent can express in English. In other words, when the user as well as the agent refer to an entity, they use its lexical name.

Definition 2:

Class: Individual Term

Superclass(es): Conceptual Entity Term

Slots: In addition to the inherited slot, LABEL,[1] all instances of this class have (at least) the following slot:

NAME: The lexical name of the individual (object). For example, A, B, TABLE, CHAIR, DOG, JOHN, MARY, ..., etc.

Methods: If o is an instance of this class then

NAME(o): Returns the contents of the NAME slot.

Syntax: Objects that are direct instances of this class are expressed using the NAME slot. All entities that are instances of this class are terms in the underlying logic.

Description: Instances of this class denote named entities in the domain of discourse. For example, an instance whose NAME slot has the value JOHN denotes the individual ⟦JOHN⟧.

Although the subclass Individual Term inherits the LABEL slot from the class Conceptual Entity Term, the agent always uses the name of the individual rather than its label when referring to it. Thus, the value of the NAME slot is the agent's "link" to the external world.

4.4 Variables

Variables denote arbitrary conceptual entities—individuals, beliefs, acts, etc. Variables are useful in representing generic individuals (e.g. 'a thing'); general propositions (e.g., 'a thing that is a block'); also called *patterns* (see Definition 4 below); generic propositions (e.g., 'something that John believes'); and generic acts (e.g., 'picking up a thing'). Syntactically, variables can be bound (by a quantifier) or free (i.e., unbound).

[1]Hereafter, we will refrain from mentioning inherited slots.

Definition 3:

Class: Variable Term

Superclass(es): Conceptual Entity Term

Slots: All instances of this class have at least the following slots:

QUANTIFIER: Can be a quantifier in the underlying logic. For example, a value ALL (written as $\forall$) will represent a universally quantified variable. A value of NIL will represent an unbound (free) variable.

NAME: The name of the variable. For example, x, y, z, ...

Methods: If o is an instance of this class then

QUANTIFIER(o): Returns the value of the QUANTIFIER slot.

NAME(o): Returns the contents of the NAME slot.

Syntax: Objects that are direct instances of this class are expressed using the NAME slot if they appear in argument position as a term. The quantification and the name will appear in the beginning of the formula that has the variable as an argument (as in first-order logic). Free variables (whose QUANTIFIER field is NIL) will be written preceded with a '?'. All entities that are instances of this class are terms in the underlying logic.

Description: Instances of this class represent arbitrary entities.

It is not possible for the same variable to have a free as well as a bound occurrence. These are two distinct instances and, therefore, two different variables. For example, a variable written as $\forall$x and one written as ?x are two distinct instances. One way, which we will use, to avoid this confusion is to assign unique names for all variables. Variables, as defined above, are neither typed nor have any restrictions in their bindings. Since all instances of the class Conceptual Entity Term can be terms in the underlying logic, that is a reasonable approach. Still, there are reasons for addressing such concerns [Israel 1983]. One approach taken by Ali [Ali 1993a, Ali 1993b, Ali 1993c] proposes to use *structured* variables for the representation of generically described entities (like "a thing that is a block"). The resulting knowledge representation system, called ANALOG, provides a stronger conceptual basis for variables and leads to some interesting forms of inference. In the work described here, we will continue to use variables as defined above. Ali's work on variables nicely complements this work and can be incorporated as a part of the OK formalism. This, as we pointed out earlier, is one of the advantages of using an object-oriented approach. The architecture is inherently extendible in such directions.

4.5 Propositions

In our formalism, propositions are also treated as intensional entities. Asserted propositions form the agent's beliefs. An agent may have representations of propositions even though it may not believe them. Since all instances of Conceptual Entity Term are terms in the underlying logic, this enables the agent to have beliefs about its own beliefs as well as those of others. The subclass Proposition Term is a metaclass[2]—its instances are classes whose

[2] We are using the term *metaclass* to refer to a class all of whose instances are also classes. This is consistent with the standard usage of the term where the class of the class of an object is called a metaclass [Steele 1990].

instances are objects that represent propositions. 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. Recall that in Chapter 3 we mentioned that it is desirable for inference to be optimized in such a way that all derived beliefs are added to the agent's belief space so as to facilitate the answering of same queries in the future by recall rather than rederivation. A truth maintenance system (TMS) becomes a necessity for such systems in order to guarantee consistency of the belief space. The objects representing beliefs also contain TMS related information (i.e., whether they are hypotheses or derived, as well as their justifications/support in case they are derived).

Definition 4:

Class: Proposition Term

Superclass(es): Conceptual Entity Term

Slots: All instances of this class have at least the following slots:

- BELIEF-STATUS: Can be T (indicating that the proposition represented by the instance is asserted) or NIL (indicating that the proposition represented by the instance is not asserted).
- SUPPORT: The assumptions/justifications underlying the belief (TMS related information).

Methods: If o is an instance of one of the subclasses of this class, then

- **PROP(o):** Returns the name of the proposition (the name of class of which o is an instance).
- **ASSERTED?(o):** Returns T/NIL (i.e., the value of the BELIEF-STATUS slot).
- **ASSERT(o):** Asserts the proposition o as a hypothesis.
- **ARGS(o):** Returns a list containing the defined arguments of the proposition.
- **SUPPORT(o):** Returns the contents of the SUPPORT slot.
- **PATTERN?(o):** Returns T if o is a pattern. An object is a pattern if it dominates a variable.
- **BELIEVE?(o,σ):** Returns a list of propositions that unify with o and are believed by the agent either explicitly or via backward chaining inference. The second argument, σ. is optional; if present, σ is a list of known applicable variable bindings. See Chapter 6.
- **BELIEVE!(o,σ):** Asserts the proposition o and returns a list of all propositions that can be derived via forward chaining inference. See Chapter 6.

Syntax: Only the instances of subclasses of this class (i.e., actual propositions) can be printed. The printed representation is the value of the LABEL slot, followed by an '!' if the object is asserted, followed by ':', followed by the name of the subclass written beginning with an uppercase followed by all lower-case letters, followed by argument parentheses containing terms in the argument slots. E.g., if o is an object, and ARG1, ARG2, ... are names of slots, then

$$\langle \mathsf{LABEL(o)} \rangle \{!\} : \langle \mathsf{PROP(o)} \rangle (\langle \mathsf{ARG1(o)} \rangle, \langle \mathsf{ARG2(o)} \rangle, ...)$$

When instances of subclasses appear in argument positions, we ignore the "$\langle \mathsf{LABEL(o1)} \rangle \{!\}$:" part. See examples below.

Description: This is the generic class of all propositions the agent can represent. Individual propositions are instances of subclasses. See examples below.

Example 1:

Class: Isa

Superclass(es): Proposition Term, Conceptual Entity Term

Slots: All instances of this class contain the following ordered argument slots:[3]

MEMBER: Can be any conceptual entity.

CLASS: Can be any conceptual entity.

Methods: If o is an instance of this class, then

MEMBER(o): Returns the conceptual entity in the MEMBER slot.

CLASS(o) Returns the conceptual entity in the CLASS slot.

Description: The entity represented by MEMBER(o) is a member of the class represented by CLASS(o).

Example Instances: The propositions "A is a block" and "B is a block" will be represented as:

$$\mathsf{B1!} : \mathsf{Isa}(\mathsf{A}, \mathsf{BLOCK})$$

$$\mathsf{B2!} : \mathsf{Isa}(\mathsf{B}, \mathsf{BLOCK})$$

where A, B, and BLOCK are instances of the class Individual Term; B1 and B2 are labels; and the propositions are asserted in the agent's belief space.

Thus, an object that represents a proposition "A is a block" would be an instance of the class Isa, which is a subclass of the metaclass Proposition Term and written as B1!:Isa(A, BLOCK).

Example 2:

Class: On

Superclass(es): Proposition Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

REL: The named individual ON.

ARG1: Can be any conceptual entity.

ARG2: Can be any conceptual entity.

Methods: If o is an instance of this class, then

REL(o): Returns the individual ON.

ARG1(o): Returns the conceptual entity in the ARG1 slot.

ARG2(o): Returns the conceptual entity in the ARG2 slot.

PROP(o): This method is specialized to return the value of the REL slot.

Description: The entity represented by ARG1(o) is on top of the entity represented by ARG2(o).

Example Instances: The proposition "A is on B" can be represented by an object written as:

$$\mathsf{B3!} : \mathsf{On}(\mathsf{A}, \mathsf{B})$$

where A and B are instances of the class Individual Term; B3 is the label; and the proposition is asserted in the agent's belief space.

---

[3]Argument slots are the slots that form the syntactic arguments of the proposition. We will list them in the same order as they would appear in a term. The ARGS method returns the slots in the proper order.

In the example above, notice that the name of the subclass is the same as the name of the REL slot. While this may seem redundant at first, it is not. The reasons are ontological as well as pragmatic. Conceptually, if we didn't have the REL slot, our ontology is incomplete. This is because even though we have an entity that represents an "object on top of another object" that is an instance of the class On, we do not have a representation of the concept "on". One may then argue that the subclass On should not be defined as such. Instead, one should have a subclass of all (2-ary?) relations, called RELATION. This is perfectly reasonable, and, if the knowledge designer so desires, it can be done. On the pragmatic side, having a specific subclass hierarchy may enable speed-up in the inference process (since entities are unifiable only if they are instances of the same subclass). Having defined the class On and keeping the REL slot solves both concerns. Additionally, as we will see in Chapter 9, this leads to a canonical representation—instances of these classes can conform to several different representational formalisms. In all the examples that follow, we will continue to use this technique.

Example 3:

Class: Clear

Superclass(es): Proposition, Conceptual Entity

Slots: The following slots are defined for all instances of this class:

> OBJECT: This is the defined argument of the class. Its value can be any conceptual entity.
>
> PROPERTY: The named entity CLEAR.

Methods: If o is an instance of this class, then

> OBJECT(o): Returns the conceptual entity in the ARG1 slot.
>
> PROPERTY(o): Returns the named individual CLEAR.

Description: The entity represented by OBJECT(o) is clear.

Example Instances: The proposition "A is clear" can be represented by an object written as:

$$\text{B4!} : \text{Clear(A)}$$

> where A is an instance of the class Individual Term; B4 is the label; and the proposition is asserted in the agent's belief space.

The class hierarchy is shown in Figure 4.1. Instances of all subclasses of Proposition Term respond to two methods—BELIEVE? and BELIEVE!. BELIEVE?(o), where o is some proposition, denotes a query to the agent whether it believes o either explicitly or via inference/acting (backward chaining). Similarly, BELIEVE!(o) denotes a request to the agent to start believing o and all its consequences in the current belief space (forward chaining). Representations that partake in forward/backward chaining are classified as *transformers*, which will be discussed later. For the time being, it is sufficient to know that modeling of the propositional attitude of belief is carried out by the two methods operating on anything that is an instance of the subclass of Proposition Term. In the absence of any transformers, there would be no inference. In that case, the methods BELIEVE? and BELIEVE! may simply be defined as follows (the second argument, σ, is not used):

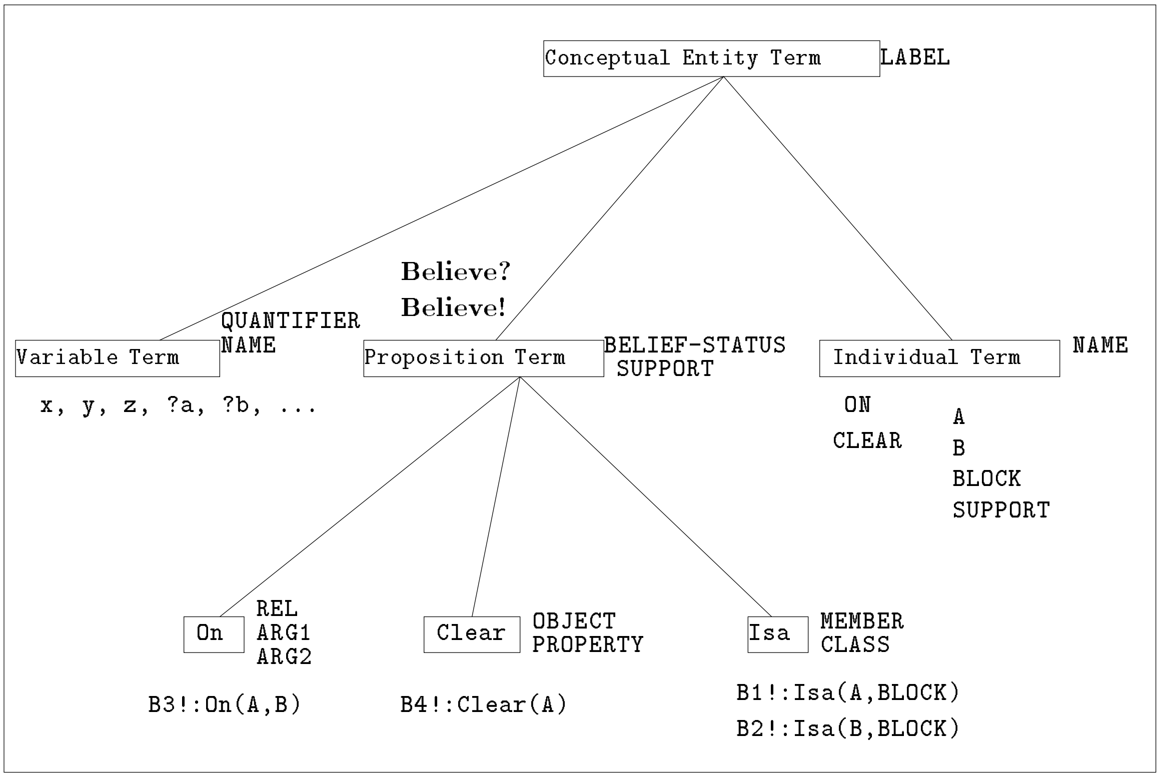

Figure 4.1: An example of a belief hierarchy. Slots defined for each class are written near to right of the box containing class names. Methods defined for each class are written in bold-face. Example instances are shown below each class.

```
Method
        BELIEVE?(φ : Proposition Term; σ : Substitution := NIL)
is
                if ASSERTED?(φ) then
                        return { φ }
                elseif PATTERN?(φ) then
                        return a set containing all asserted instances of φ
                endif
end BELIEVE?
```

```
Method
        BELIEVE!(φ : Proposition Term; σ : Substitution := NIL)
is              ASSERT(φ)
                return { φ }
end BELIEVE!
```

We will use the above format for specifying algorithms. Formal parameters will be followed by their class names indicating the parameter specializers for the purpose of defining applicable methods for instances of each class. Optional parameters will specify a default value using the := notation. The above method specifications will be modified later when we have introduced transformers. In an object-oriented class hierarchy, all subclasses of a specific class inherit all the methods defined for that class. One can, of course, shadow inherited methods by defining specific method specializations of a generic function for any subclass in the hierarchy. Figure 4.1 shows the class hierarchy discussed so far.

One more thing to be pointed out at this stage is that in the representational formalism being described, we observe the Uniqueness Principle [Maida and Shapiro 1982, Shapiro and Rapaport 1987] of representing intensional concepts—there is a one-to-one correspondence between instances of Conceptual Entity Term and intensional concepts. Thus, in the examples above, all references to the named individual A, for instance, refer to the same instantiated object. I.e., there is exactly one object corresponding to the named individual A. The Uniqueness Principle applies to all instances of all subclasses of Conceptual Entity Term.

4.6 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. Some of these concerns were also explored in the work of Almeida [Almeida 1987]. Acts are represented by objects that are instances of the class Act Term which is defined as follows:

Definition 5:

Class: Act Term

Superclass(es): Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

ACTION: This is the named individual that represents the action concept.

Methods: If o is an instance of this class (or any of its subclasses), then

INTEND(o): The generic function corresponding to the agent's carrying out an intention to perform the act o. See below.

Syntax: Only the instances of subclasses of this class can be printed. The printed representation is the value of the LABEL slot, followed by the name of the subclass (or name of the object in the ACTION) slot written in uppercase followed by argument parentheses containing entities in the object slots. E.g., if o1 is the object, and OBJECT1, OBJECT2, ... are names of slots, then

$$\langle \mathsf{LABEL(o1)} \rangle : \langle \mathsf{NAME(ACTION(o1))} \rangle (\langle \mathsf{OBJECT1(o1)} \rangle, \langle \mathsf{OBJECT2(o1)} \rangle, \ldots)$$

When instances of subclasses are printed in argument positions, we ignore the "⟨LABEL(o1)⟩" part. See examples below.

Description: Acts are also conceptual entities. Like Proposition Term, this is the generic class of all acts.

Example 4:

Class: PICKUP

Superclass(es): Act Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

OBJECT1: can be any conceptual entity.

Methods: If o is an instance of this class, then

ACTION(o): Returns the named individual of the action concept (PICKUP).

OBJECT1(o): Returns the conceptual entity in the OBJECT1 slot.

Description: The act of picking up the entity represented by OBJECT1(o).

Example Instances: The act "Pick up A" can be represented by an object written as:

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

where A is an instance of the class Individual Term and A1 is the label.

Example 5:

Class: PUT

Superclass(es): Act Term, Conceptual Entity Term

Slots: All instances of this class have the following slots:

OBJECT1: Can be any conceptual entity.

OBJECT2: Can be any conceptual entity.

Methods: If o is an instance of this class, then

ACTION(o): Returns the named individual of the action concept (PUT).

OBJECT1(o): Returns the conceptual entity in the OBJECT1 slot.

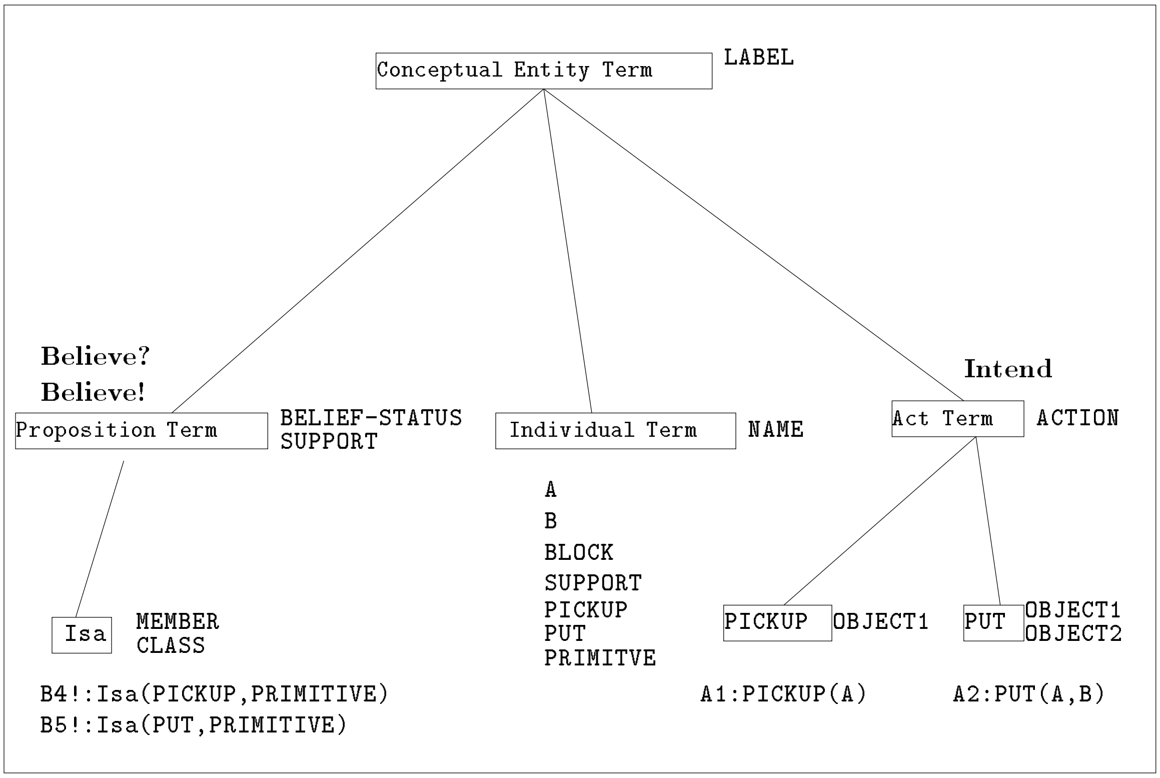

Figure 4.2: The class hierarchy with the class Act included.

OBJECT2(o): Returns the conceptual entity in the OBJECT1 slot.

Description: The act of putting the entity represented by OBJECT1(o) on top of the entity represented by OBJECT2(o).

Example Instances: The act "Put A on B" can be represented by an object written as:

$$\mathsf{A2 : PUT(A, B)}$$

where A and B are instances of the class Individual Term and A2 is the label.

An *action* is that component of an act that is what is done to the object or objects. One can only *perform* something that is an act (an action on an appropriate object). In what follows, instead of saying "performing an act whose action is x," we will say "performing the action x," and hope the reader will note the distinction between acts and actions. Figure 4.2 pictorially shows the class hierarchy of acts as defined above. This approach for representing acts is taken from our earlier work on the SNePS acting system (see Section 3.1.3). In later chapters, we will enrich this ontology by introducing separate subclasses for different types of acts (physical, mental, intentional). For the time being, it is sufficient to assume the class hierarchy depicted in Figure 4.2.

Instances of the subclasses of Act Term respond to a method INTEND. Thus INTEND(a), where a is some act, represents the agent's intention to perform the act a. Performing an act involves ensuring that the preconditions of the act are believed by the agent. Also, after the act is performed, its effects have to be believed. The next chapter will describe representations of preconditions and effects of actions. We will defer our presentation of the INTEND method until Chapter 7.

Any behaving entity has a repertoire of *primitive actions* that it is capable of performing. Every primitive act has an *effectory* component—one that actually brings about the changes that represent the effects of the action. The effectory component (also called an *actuator*) is a procedure that when executed upon the act's objects constitutes the execution of the act. We will say that an act whose action is primitive is a *primitive act.* That an action is primitive is a belief held by the agent after it is told so. This can be represented by an Isa proposition. For example, the proposition "Pickup is a primitive action" is represented as

$$\mathsf{B6!} : \mathsf{Isa}(\mathsf{PICKUP}, \mathsf{PRIMITIVE})$$

Notice the use of the concept of PICKUP to make assertions about the act. This is the reason for including the ACTION slot for all instances of the class Act Term.

Non-primitive acts, which we will term *complex*, can only be performed by decomposing them into a structure of primitive acts, called a *plan.* Like acts, we also represent plans as conceptual entities. This enables the agent to discuss, formulate, use, recognize, and reason about acts as well as plans. A plan is basically a structure of acts. The structuring syntax for plans is described in terms of control actions, which are described in Chapter 7. As we will see in Chapter 7, in our ontology, plans are also acts. The structure of a plan can determine how the agent performs the component acts, because the structure, itself, is an act. We will detail the class hierarchy of acts after we discuss transformers.

4.7 Summary

In this chapter, we have laid the foundations of a unified representational formalism for BDI architectures—one that allows for representation of individuals, beliefs, and acts in a single formalism. In the next chapter, we extend the conceptual class hierarchy to include representations for transformers. We will identify different types of transformers and the role they play in acting and inference. The following chapters also describe the unified inference and acting mechanism as well as the structure of plans.

Chapter 5

Transformers

In Chapter 3, we introduced the idea of a *transformer* as a proposition 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—$(\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)$, where both $\langle \mathsf{a} \rangle$ and $\langle \mathsf{b} \rangle$ 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 $\langle \mathsf{a} \rangle$, it believes or intends to perform $\langle \mathsf{b} \rangle$." The backward chaining interpretation of a transformer is, "if the agent wants to believe (or know if it believes) or perform $\langle \mathsf{b} \rangle$, it must first believe (or see if it believes) or perform $\langle \mathsf{a} \rangle$." 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. In this chapter, we extend the class hierarchy of the previous chapter to include various transformers. We identify different propositions represented by different transformers and their role in the acting/inference process. Details of the transformation process itself will be deferred until the next chapter. Here, then, is a definition of the metaclass of all transformers:

Definition 6: ___

Class: Transformer

Superclass(es): Proposition Term, Conceptual Entity Term

Slots: none.

Methods: If o is an instance of some subclass of this class and σ is a set of known variable bindings, then

TRANSFORM?(o, σ)**:** This is the generic backward chaining procedure. See below.

TRANSFORM!(o, σ)**:** This is the generic forward chaining procedure. See below.

Syntax: The syntax of transformers varies. It will be defined for each subclass below.

Description: This is the generic class of all transformers. TRANSFORM? and TRANSFORM! are generic procedures which will be specified by individual methods with each subclass below.

Since both ⟨a⟩ and ⟨b⟩ can be propositions (beliefs) or an act, we have four types of transformers—*belief–belief, belief–act, act–belief,* and *act–act*, each of which will be defined as a subclass of Transformer. Thus, the class Transformer is a metaclass.

Instances of all subclasses of Transformer are also propositions (i.e., they may be asserted as beliefs of the modeled agent). Hence, they also respond to the BELIEVE? and BELIEVE! methods. Only when an object is asserted does it partake in the transformation process. The transformation process is defined by two methods, TRANSFORM? and TRANSFORM!, corresponding to backward and forward chaining transformations respectively. The transformations are typically invoked by parts of the transformers when they receive forward/backward chaining messages (method invocations). We will define various specializations of transformation methods once we have defined a specific set of transformers.

5.1 Belief–Belief Transformers

Belief–belief transformers are propositions that represent standard reasoning rules, where ⟨a⟩ is an antecedent belief and ⟨b⟩ is a consequent belief. Such rules can be used in forward, backward, as well as bidirectional inference to derive new beliefs. The forward chaining interpretation applied to these transformers is, "after the agent believes ⟨a⟩, it believes ⟨b⟩". The backward chaining interpretation is, "if the agent wants to know if it believes ⟨b⟩, it must first see if it believes ⟨a⟩".

Definition: 7 ---

Class: Belief–Belief

Superclass(es): Transformer, Proposition Term, Conceptual Entity Term

Slots: none.

Syntax: none.

Methods: none.

Description: This is the generic class of all belief–belief transformers.

The specific subclasses of belief–belief transformers can be defined by the user depending on the underlying logical commitments. Each logical connective capable of deriving new beliefs (via chaining) will have to be defined as a distinct subclass of belief–belief transformers. The rules of inference that form the logical basis are implemented in the respective transform methods. An example of a specific belief–belief transformer that represents antecedent–consequent rules is presented below:

5.2 Antecedent–Consequent Transformers

Both parts of an antecedent–consequent transformer denote propositions. Variables occurring in these transformers can be quantified.

Definition 8:

Class: AntCq

Superclass(es): Belief-Belief, Transformer, Proposition Term, Conceptual Entity Term

Slots: All instances of this class have the following slots defined:

VARS: A list of instances of the class Variable Term.

ANT: Any object that is a Proposition Term (i.e., the $\langle a\rangle$ part of the transformer $(\langle a\rangle, \langle b\rangle)$).

CQ: Any object that is a Proposition Term (i.e., the $\langle b\rangle$ part of the transformer $(\langle a\rangle, \langle b\rangle)$).

Methods: If o is an instance of this class then

VARS(o): Returns the contents of the VARS slot.

ANT(o): Returns the antecedent object.

CQ(o): Returns the consequent object.

Syntax: Instances of this class will be written starting with the label, assertional status, a ':', followed by variables preceded by their quantifiers (if any), followed by a '[', followed by the antecedent, followed by a '$\rightarrow$', followed by the consequent proposition, followed by ']'. I.e.,

$$\langle \mathsf{LABEL(o)}\rangle\{!\} : \langle \mathsf{VARS(o)}\rangle[\langle \mathsf{ANT(o)}\rangle \rightarrow \langle \mathsf{CQ(o)}\rangle]$$

Description: This is the class of transformers that represent logical implication rules.

For example, the rule "All blocks are supports" can be represented as an antecedent-consequent transformer,

$$\mathsf{B5!} : \forall \mathsf{x}[\mathsf{Isa(x, BLOCK)} \rightarrow \mathsf{Isa(x, SUPPORT)}]$$

As mentioned above, for a transformer to be used in reasoning, it must itself be believed. The above transformer will be used in backward chaining whenever a BELIEVE?(o) is invoked for some object o that unifies with the consequent of the transformer. Similarly, a BELIEVE!(o) will forward chain through it if o unifies with its antecedent. The transformation, in this case, represents the firing of the rule represented by the transformer. The firing criteria will be implemented by each of the specific transformation methods for this class. These will be defined in the next chapter.

We have maintained that all transformers are also propositions, which in turn are conceptual entities. This means that all instances of AntCq (and other such transformers) are also terms in the underlying logic. This enables the agent to reason about rules, i.e., to do meta-reasoning. It also facilitates representations of beliefs about reasoning rules of other agents (as in "John believes that all blocks are supports").

Notice that, as defined so far, the above specification implies that there can be only one object in the antecedent and the consequent positions. This, however, is not a restriction. If we were to commit ourselves to a logic where formulas could be formed using the logical connectives $\wedge$, $\vee$, and $\neg$, for conjunction, disjunction, and negation, respectively, we can define subclasses of propositions for each of these connectives.

Example 1:

Class: And

Superclass(es): Proposition Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

ARG1: Can be any object that is a Proposition Term.

ARG2: Can be any object that is a Proposition Term.

Methods: If o is an instance of this class, then

ARG1(o): Returns the conceptual entity in the ARG1 slot.

ARG2(o): Returns the conceptual entity in the ARG2 slot.

Syntax: Instances of this class are expressed by writing the two arguments, separated by a '$\wedge$', enclosed in parentheses. The label, assertional status, and colon (:) are ignored if o appears as an argument.

Description: The entity represented by o is a conjunction of two propositions represented by ARG1(o) and ARG2(o).

Example Instances: The proposition "A is clear and B is clear" can be represented by an object written as:

$$\mathsf{B14!} : (\mathsf{Clear(A)} \wedge \mathsf{Clear(B)})$$

Example 2:

Class: Or

Superclass(es): Proposition Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

ARG1: Can be any object that is a Proposition Term.

ARG2: Can be any object that is a Proposition Term.

Methods: If o is an instance of this class, then

ARG1(o): Returns the conceptual entity in the ARG1 slot.

ARG2(o): Returns the conceptual entity in the ARG2 slot.

Syntax: Instances of this class are expressed by writing the two arguments, separated by a $\vee$, in parentheses. The label, assertional status, and colon (:) are ignored if o appears as an argument.

Description: The entity represented by o is a disjunction of two propositions represented by ARG1(o) and ARG2(o).

Example Instances: The proposition "A is clear or B is clear" can be represented by an object written as:

$$\mathsf{B15!} : (\mathsf{Clear(A)} \vee \mathsf{Clear(B)})$$

Example 3:

Class: Neg

Superclass(es): Proposition Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

ARG: Can be any object that is a Proposition Term.

Methods: If o is an instance of this class, then

ARG(o): Returns the conceptual entity in the ARG slot.

Syntax: Instances of this class are expressed by writing the argument preceded by a $\neg$. The label, assertional status, and colon (:) are ignored if o appears as an argument.

Description: The entity represented by o is a negation of the proposition represented by ARG(o).

Example Instances: The proposition "A is not clear" can be represented by an object written as:

$$\mathsf{B16!} : \neg\mathsf{Clear(A)}$$

Thus, given the class descriptions above, one can represent rules for and-entailment[1] as well as or-entailment[2] [Shapiro and Group 1989]. For example, "If a block is on anther block then it is not on any other block" can be represented as

$$\mathsf{B17!} : \forall \mathsf{x} \forall \mathsf{y} \forall \mathsf{z}[(((\mathsf{Isa(x, BLOCK)} \wedge \mathsf{Isa(y, BLOCK)}) \wedge \mathsf{Isa(z, BLOCK)}) \wedge \mathsf{On(x, y)}) \rightarrow \neg\mathsf{On(x, z)}]$$

The variables in an instantiation of the rule above will never all be bound to the same entity. This is because we will enforce the Unique Variable Binding Rule (UVBR)—"no instance of a rule is allowed in which two distinct variables of the rule are replaced by the same term, nor in which any variable is replaced by a term already occurring in the rule" ([Shapiro 1986], p. 1358).

One may also need to define further specializations of the BELIEVE? and BELIEVE! methods for the classes And, Or, and Neg. For example, to answer the query

$$\mathsf{BELIEVE?((Clear(A)} \wedge \mathsf{Clear(B)))}$$

one has to issue the two queries

$$\mathsf{BELIEVE?(Clear(A))}$$

$$\mathsf{BELIEVE?(Clear(B))}$$

(this is also called the rule of $\wedge$-introduction). Similarly, the invocation

$$\mathsf{BELIEVE!((Clear(A)} \wedge \mathsf{Clear(B)))}$$

should lead to the invocations

$$\mathsf{BELIEVE!(Clear(A))}$$

$$\mathsf{BELIEVE!(Clear(B))}$$

for full forward chaining (this is the rule of $\wedge$-elimination). Once such classes of connectives have been defined, one can specialize the appropriate methods. We will present details of these methods in the next chapter.

[1]The conjunction of the antecedents implies the conjunction of the consequents.

[2]The disjunction of the antecedents implies the conjunction of the consequents.

5.3 Belief–Act Transformers

Belief–act transformers are transformers where $\langle \mathsf{a} \rangle$ is a belief and $\langle \mathsf{b} \rangle$ is an act. The definition of the class Belief–Act is similar to that of the class Belief–Belief (Definition 7). These transformers enable the representation of preconditions of actions as well as basic rules for reactivity. We will define these as two separate subclasses and specify when they can be used. Used during backward chaining, these transformers can be propositions specifying preconditions of actions, i.e., $\langle \mathsf{a} \rangle$ is a precondition of some act $\langle \mathsf{b} \rangle$ (under the interpretation, "if the agent wants to perform the act $\langle \mathsf{b} \rangle$, it must first believe $\langle \mathsf{a} \rangle$). The precondition $\langle \mathsf{a} \rangle$ must be believed by the agent before it performs the act $\langle \mathsf{b} \rangle$.

Definition 9:

Class: PreconditionAct

Superclass(es): Belief-Act, Transformer, Proposition Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

PRECONDITION: Can be any object that is a Proposition Term.

ACT: Can be any object that is an Act Term.

Methods: If o is an instance of this class, then

PRECONDITION(o): Returns the object in the PRECONDITION slot.

ACT(o): Returns the object in the ACT slot.

Syntax: These transformers are written just like beliefs:

$$\langle \mathsf{LABEL(o)} \rangle \{!\} : \mathsf{PreconditionAct}(\langle \mathsf{PRECONDITION(o)} \rangle, \langle \mathsf{ACT(o)} \rangle)$$

Description: The propositions represented by PRECONDITION(o) are the preconditions of the act represented by ACT(o).

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

$$\mathsf{B6!} : \mathsf{PreconditionAct}(\mathsf{Clear(A)}, \mathsf{PICKUP(A)}).$$

Thus, whenever the agent intends to perform the act PICKUP(A), it must first ensure that it believes Clear(A). This is an example of a backward chaining belief–act transformation. Like general AntCq rules, one can define general preconditions by including a VARS slot for the class PreconditionAct. The VARS slot has not been specified in Definition 9 because of some stylistic concerns—since variables can represent arbitrary entities, it is usually necessary to provide additional restrictions on their bindings. This is typically the case for objects of an action. In other words, specification of preconditions on objects of an action applies to only those entities that are eligible to be an action's objects. It is possible to represent general preconditions of actions using AntCq transformers. For example, "Before picking up any block, make sure that it is clear" can be represented as

$$\mathsf{B11!} : \forall \mathsf{x}[\mathsf{Isa(x, BLOCK)} \rightarrow \mathsf{PreconditionAct}(\mathsf{Clear(x)}, \mathsf{PICKUP(x)})]$$

from which B6 above can be derived via backward chaining. The restriction that the objects of the PICKUP action can only be blocks is specified using the antecedent of the rule. If the formalism employs structured variables [Ali 1993a, Ali 1993b, Ali 1993c], these restrictions can be specified on the variables themselves and one doesn't need the AntCq transformer. The VARS slot can be used for the PreconditionAct transformer if actions can take arbitrary objects. The propositional approach of specifying preconditions of actions is quite different from traditional operator-based action representations. We will return to this in Chapter 7.

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 $\langle \mathsf{a} \rangle$, will form an intention to perform $\langle \mathsf{b} \rangle$. We will call these WhenDo transformers.

Definition 10:

Class: WhenDo

Superclass(es): Belief-Act, Transformer, Proposition Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

VARS: (optional slot) Can be a list of variables.

WHEN: Can be any object that is a Proposition Term.

DO: Can be any object that is an Act Term.

Methods: If o is an instance of this class then:

VARS(o): Returns the list of variables in the VARS slot.

WHEN(o): Returns the object in the WHEN slot.

DO(o): Returns the act object in the DO slot.

Syntax: These transformers are written as

$$\langle \mathsf{LABEL(o)} \rangle \{!\} : \langle \mathsf{VARS(o)} \rangle [\mathsf{WhenDo}(\langle \mathsf{WHEN(o)} \rangle, \langle \mathsf{DO(o)} \rangle)]$$

Description: This transformer represents the agent's desire to react to certain beliefs.

For example, the sentence representing the desire "When A is clear, pick it up" can be represented as

$$\mathsf{B7! : WhenDo(Clear(A), PICKUP(A))}$$

Thus, whenever the agent executes a BELIEVE!(Clear(A)), it forms the intention to perform the act PICKUP(A). A general desire like "Whenever something is broken, fix it" can be represented as

$$\mathsf{B100!} : \forall \mathsf{x}[\mathsf{WhenDo(Broken(x), FIX(x))}]$$

where Broken is a subclass of Proposition Term and FIX is a subclass of Act Term. Where an action's objects are restricted to specific kinds of entities, one can use the AntCq transformer to specify the restrictions on variable bindings. For example,

$$\mathsf{B101!} : \forall \mathsf{x}[\mathsf{Isa(x, BLOCK)} \rightarrow \mathsf{When(Clear(x), PICKUP(x))}]$$

represents the general case from which B7 can be derived. The details of the transformation process are presented in the next chapter.

5.4 Act–Belief Transformers

The metaclass Act–Belief has a definition similar to the class Belief–Belief (Definition 7). Under the forward chaining interpretation, these transformers ($\langle a \rangle$ is an act, and $\langle b \rangle$ is a set of beliefs) specify effects of actions.

Definition 11:

Class: ActEffect

Superclass(es): Act-Belief, Transformer, Proposition Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

ACT: Can be any object that is an Act Term.

EFFECT: Can be any object that is a Proposition Term.

Methods: If o is an instance of this class then

ACT(o): Returns the act object in the ACT slot.

EFFECT(o): Returns the proposition object in the EFFECT slot.

Syntax: These transformers are written just like beliefs:

$$\langle \mathsf{LABEL(o)} \rangle \{!\} : \mathsf{ActEffect}(\langle \mathsf{ACT(o)} \rangle, \langle \mathsf{EFFECT(o)} \rangle)$$

Description: Specifies the beliefs about an act's effects.

The ActEffect transformer will be used in forward chaining to accomplish believing the effects of act $\langle a \rangle$. For example, the sentence "After picking up A, it is held" is represented as

$$\mathsf{B8! : ActEffect(PICKUP(A), Held(A))}$$

For the same reasons mentioned for class PreconditionAct we have refrained from using the VARS slot. The general effect can be represented using AntCq transformers. Thus,

$$\mathsf{B115!} : \forall \mathsf{x}[\mathsf{Isa(x, BLOCK)} \rightarrow \mathsf{ActEffect(PICKUP(x), Held(x))}]$$

represents, "After picking up a block it is held." These transformers can also be used in backward chaining during the plan generation process (à la classical planning)—"Whenever the agent wants to believe $\langle b \rangle$, it *may* perform $\langle a \rangle$". Notice the use of the word 'may' in the previous sentence, as opposed to 'must'. The latter interpretation is represented using another transformer, which is defined below.

Used during backward chaining, these transformers can also represent the decomposition of the achieving of a goal $\langle b \rangle$ into a plan $\langle a \rangle$ (i.e., "if the agent wants to believe $\langle b \rangle$, it must perform $\langle a \rangle$"). Thus, we are representing goals as propositions. A proposition becomes a goal when an agent decides to achieve a state in which the proposition is believed. Recall that in the last chapter we mentioned that plans in our ontology are also acts (detailed definitions of plans are presented in Chapter 7). The transformer below facilitates the explicit treatment of propositions as goals and acts as plans.

Definition 12:

Class: PlanGoal

Superclass(es): Act-Belief, Transformer, Proposition Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

PLAN: Can be any object that is an Act Term.

GOAL: Can be any object that is a Proposition Term.

Methods: If o is an instance of this class then

PLAN(o): Returns the contents of the PLAN slot.

GOAL(o): Returns the contents of the GOAL slot.

Syntax: These transformers are written just like beliefs:

$$\langle \mathsf{LABEL(o)} \rangle \{!\} : \mathsf{PlanGoal}(\langle \mathsf{PLAN(o)} \rangle, \langle \mathsf{GOAL(o)} \rangle)$$

Description: That something is a plan for achieving is a goal can also be a belief held by the agent.

For example, "A plan to achieve that A is held is to pick it up" can be represented as

$$\mathsf{B9!} : \mathsf{PlanGoal}(\mathsf{PICKUP(A)}, \mathsf{Held(A)}).$$

Just like preconditions and effects, we can represent general PlanGoal propositions. Thus,

$$\mathsf{B112!} : \forall \mathsf{x}[\mathsf{Isa(x, BLOCK)} \rightarrow \mathsf{PlanGoal}(\mathsf{PICKUP(x)}, \mathsf{Held(x)})]$$

is the general plan for holding a block. This transformer can also serve a useful purpose during plan recognition—if the agent believes that $\langle \mathsf{a} \rangle$ is a plan for the goal $\langle \mathsf{b} \rangle$, then, if some act a1 is part of the plan a, and the agent believes that someone performed the act a1, it might be that they were engaged carrying out the plan a in order to achieve the goal b.

Another backward chaining interpretation that can be derived from this transformer is "if the agent wants to know if it believes $\langle \mathsf{b} \rangle$, it must perform $\langle \mathsf{a} \rangle$," which is represented as a DoWhen transformer.

Definition 13:

Class: DoWhen

Superclass(es): Act-Belief, Transformer, Proposition Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

VARS: (optional slot) Can be a list of variables.

DO: Can be any object that is an Act Term.

WHEN: Can be any object that is a Proposition Term.

Methods: If o is an instance of this class then

VARS(o): Returns the contents of the VARS slot.

DO(o): Returns the contents of the DO slot.

WHEN(o): Returns the contents of the WHEN slot.

Syntax: These transformers are written as

$$\langle \mathsf{LABEL(o)} \rangle \{!\} : \langle \mathsf{VARS(o)} \rangle [\mathsf{DoWhen}(\langle \mathsf{DO(o)} \rangle, \langle \mathsf{WHEN(o)} \rangle)]$$

Description: This transformers links up a backward chaining inference with acting.

For example, "Look at A to find what its color is" can be represented as

$$\mathsf{B10!} : \mathsf{DoWhen}(\mathsf{LOOKAT}(\mathsf{A}), \mathsf{Color}(\mathsf{A}, ?\mathsf{color}))$$

or for the general form, "Look at an object if you want to know what its color is" we will have the belief

$$\mathsf{B120!} : \forall \mathsf{x}[\mathsf{DoWhen}(\mathsf{LOOKAT}(\mathsf{x}), \mathsf{Color}(\mathsf{x}, ?\mathsf{color}))]$$

where LOOKAT is a class of acts, Color is a subclass of Proposition, and ?color is an unbound variable (recall Definition 3, Chapter 4). Thus, during backward chaining, this will be a belief–act transformation.

5.5 Act–Act Transformers

Act–act transformers are propositions specifying plan decompositions for complex actions (called PlanAct transformers), where ⟨b⟩ is a complex act and ⟨a⟩ is a plan that possibly decomposes it into simpler acts.

Definition 14:

Class: PlanAct

Superclass(es): Act-Act, Transformer, Proposition Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

PLAN: Can be any object that is an Act Term.

ACT: Can be any object that is an Act Term.

Methods: If o is an instance of this class then

PLAN(o): Returns the contents of the PLAN slot.

ACT(o): Returns the contents of the ACT slot.

Syntax: These transformers are written just like beliefs:

$$\langle \mathsf{LABEL(o)} \rangle \{!\} : \mathsf{PlanAct}(\langle \mathsf{PLAN(o)} \rangle, \langle \mathsf{ACT(o)} \rangle)$$

Description: That some plan is a plan that decomposes a complex act is also a proposition.

Used during backward chaining, these transformers specify plan decompositions of complex acts. For example, the sentence, "To stack A on B, put A on B," where stacking is a complex act (which happens to have a simple plan decomposition!), can be represented as

$$\mathsf{M71!} : \mathsf{PlanAct}(\mathsf{PUT}(\mathsf{A}, \mathsf{B}), \mathsf{STACK}(\mathsf{A}, \mathsf{B}))$$

and the general version can be represented as

$$\mathsf{B122!} : \forall \mathsf{x} \forall \mathsf{y}[(\mathsf{Isa}(\mathsf{x}, \mathsf{BLOCK}) \wedge \mathsf{Isa}(\mathsf{y}, \mathsf{BLOCK})) \rightarrow \mathsf{PlanAct}(\mathsf{PUT}(\mathsf{x}, \mathsf{y}), \mathsf{STACK}(\mathsf{x}, \mathsf{y}))]$$

We have purposely omitted descriptions of complex plan structures here, because we have not yet defined them. In Chapter 7, when we define the structure of plans, we will present more examples involving complex plan structures. Like the PlanGoal transformer, the PlanAct transformer may also be used in plan recognition.

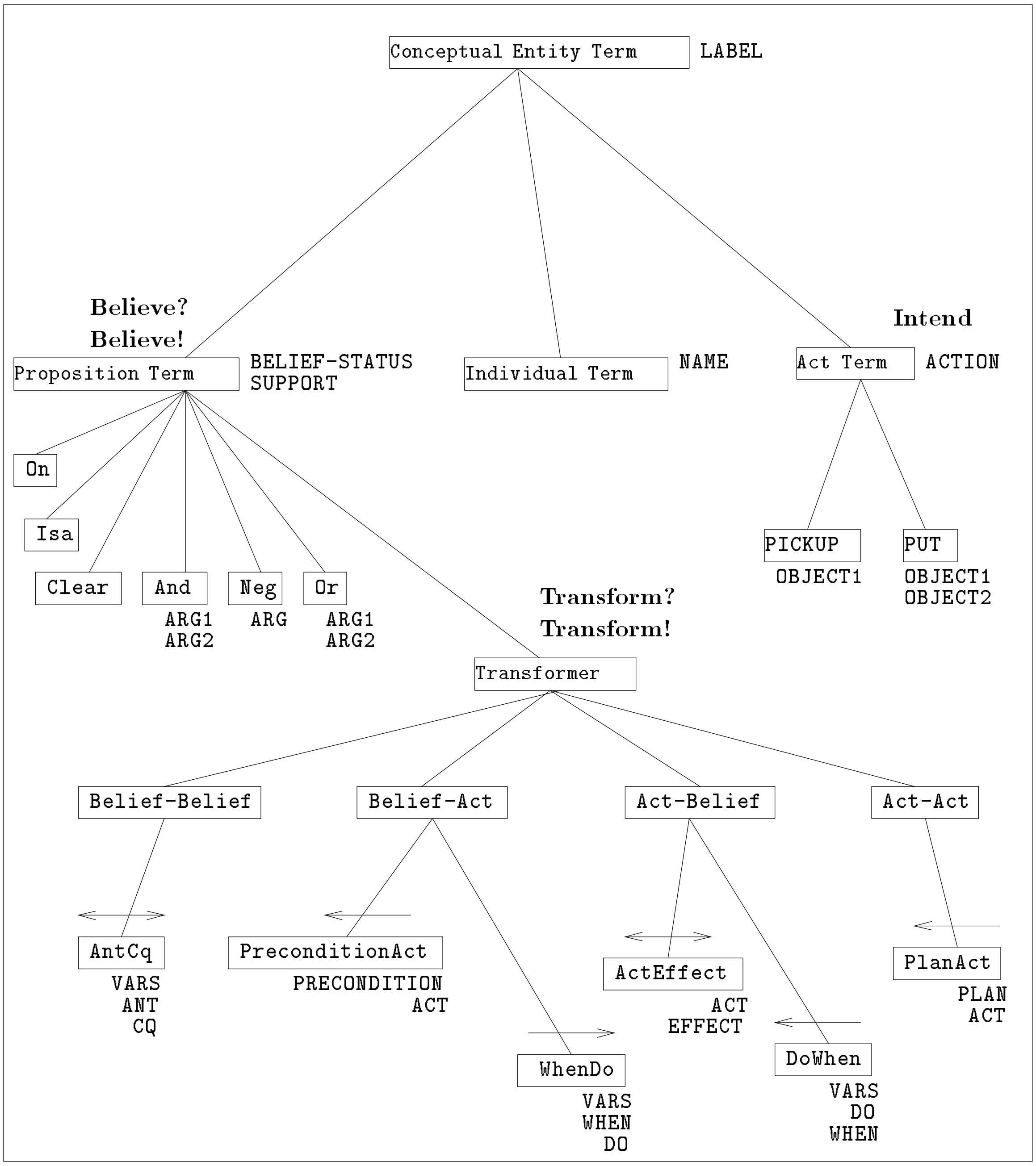

Figure 5.1: The extended class hierarchy that shows various transformers. The directions of arrows indicate when the transformer can be used. The left to right arrow indicates forward chaining; right to left indicates backward chaining.

5.6 Discussion

Figure 5.1 shows the transformer class hierarchy as defined in this chapter. So far, we have accomplished two things—we have provided representations for reasoning rules (AntCq), reaction rules (WhenDo), knowledge acquisition rules (DoWhen), preconditions and effects of actions (PreconditionAct and ActEffect), as well as plan specifications for goals and acts (PlanGoal and PlanAct); we have also informally indicated how they can be used during inference and acting. This unified approach to inference and acting explicitly identifies the relationship between inference and acting. The representations are defined in a unified formalism. Moreover, the operations on these representations (as will be seen in the next chapter) clearly identify situations when beliefs and intentions get transformed into other beliefs and intentions and culminate in different reasoning/acting behaviors on the part of the modeled agent.

Chapter 6

A Rational Engine

In previous chapters, we have described the basic class hierarchy of the OK formalism. In this chapter, we will outline core details of the integrated reasoning and acting module—the OK *Rational Engine* (as opposed to an inference engine that only performs inference). The OK rational engine is an interpreter that operates on specific instances of objects representing beliefs, transformers, and acts. It is comprised of the methods defined on the classes of the OK formalism.

We have already identified five generic procedures that comprise the rational engine. If ϕ is an instance of any subclass of Proposition Term and α an instance of any subclass of Act Term, then

BELIEVE?(ϕ, σ): Returns a list of propositions that unify with ϕ (after applying any known substitutions from σ) and are believed by the agent either explicitly or via backward chaining inference/acting. The second argument, σ, is optional and, if present, is a list of known applicable variable bindings that may have been acquired during the chaining process.

BELIEVE!(ϕ, σ): Asserts the proposition $\phi\sigma$ and returns a list of all propositions that can be derived via forward chaining inference/acting.

INTEND(α): The generic function corresponding to the agent's carrying out an intention to perform the act α.

TRANSFORM?(ϕ, σ): The method that performs transformations during backward chaining. The optional argument σ may be used to convey variable bindings during the chaining process.

TRANSFORM!(ϕ, σ): The method that performs transformations during forward chaining. The optional argument σ may be used to convey variable bindings during the chaining process.

In the previous chapter, we described four different types of transformations and identified specific transformers and the direction in which they are to be used. In this chapter, we furnish details of all those transformations by defining specific transformation procedures for each class of transformers.

6.1 Inference—Backward Chaining

Querying the agent about any of its beliefs is carried out by invoking the BELIEVE? method. If a queried proposition is explicitly asserted, a list containing the proposition instance is returned as an answer. If it is not explicitly believed, it could be derived via backward chaining through belief-belief transformers or DoWhen act–belief transformers. Since, some of the transformers represent patterns of beliefs, we could also determine the answer to the queried belief by matching it against such patterns and then recursively invoking the BELIEVE? procedure on the instantiated patterns. A basic outline of the BELIEVE? method is presented below. One should read the algorithm from the point of view of understanding the mechanism of backward chaining inference using transformations. Several implementational details (as well as efficiency considerations) have been purposely omitted to focus on main aspects of the method. Also, the version presented below appears to be using a serial execution model. However, the method (and the ones that follow) is amenable to concurrent execution. For instance, all loops that operate non-deterministically on elements of a set can be executed in parallel. We will present a concurrent model of execution in Chapter 8.

```
Method
    BELIEVE?(φ : Proposition Term; σ : Substitution := NIL)
is
        let o ← φσ

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

        find the set T of applicable AntCq transformers, i.e.
        let T ← {t | φ_t ≠ NIL ∧ BELIEVE?(t), where φ_t ← UNIFY(CQ(t),o)}
        also find all the applicable DoWhen transformers, i.e.,
        let T ← T ∪ {t | φ_t ≠ NIL ∧ BELIEVE?(t), where φ_t ← UNIFY(WHEN(t),o) }
        for each t ∈ T loop
            RESULT ← RESULT ∪ TRANSFORM?(t,φ_t)
        endloop

        find the set of matching propositions, i.e.,
        let B ← {b | φ_b ≠ NIL, where φ_b ← UNIFY(b,o)}
        for each b ∈ B loop
            RESULT ← RESULT ∪ BELIEVE?(b, φ_b)
        endloop

        return RESULT
end BELIEVE?
```

The operation $\phi\sigma$ denotes the substitution instance obtained by applying the substitution σ to the proposition ϕ. For example, if

$$\phi = \mathsf{Isa}(\mathsf{x}, \mathsf{SUPPORT})$$

and

$$\sigma = \{\mathsf{A}/\mathsf{x}\}$$

then $\phi\sigma$ is the object instance

$$\mathsf{Isa(A, SUPPORT)}$$

The operation *UNIFY* is the unification procedure defined as follows:

UNIFY(ϕ_1, ϕ_2): Where ϕ_1 and ϕ_2 are instances of some subclass of Conceptual Entity Term. The function returns *NIL*, the empty set, or a set of variable bindings if ϕ_1 and ϕ_2 do not unify, unify without any substitutions, or unify under a set of substitutions, respectively.

Notice the use of the TRANSFORM? method above. The invocation tries to apply all possible transformations (belief–belief through AntCq, and belief–act through DoWhen transformers). The loop that applies the TRANSFORM? method operates non-deterministically on the transformers in the set T. In the case of AntCq transformers, the transformations amount to serially back chaining through all the applicable transformers. However, in the presence of a parallel implementation (or interleaved execution), all possible transformations could be applied concurrently (this is discussed later in Chapter 8). Once a suitable answer is found, the remaining transformations can be abandoned. In the case of DoWhen transformations, if several transformers are applicable, the rational engine will carry out all of them non-deterministically. Depending upon the domain of application, this may (or may not) be a desirable behavior. For example, consider the following two beliefs:

$$\mathsf{B64!: \forall x[DoWhen(LOOKAT(x), Color(x, ?color)]}$$

$$\mathsf{B65!: \forall x[DoWhen(ASK(CYC, Color(x)), Color(x, ?color))]}$$

where B64 represents the desire, "If you want to know the color of something, look at it" (LOOKAT is the physical act of looking at an object) and B65 represents, "if you want to know the color of something, ask CYC" (ASK is the physical act of asking an agent if it believes a proposition). Both of these transformers will be applicable if one is answering the query BELIEVE?(Color(A)) ultimately (using the TRANSFORM? method defined in Section 6.6) resulting in the invocations INTEND(LOOKAT(A)) and INTEND(ASK(CYC, Color(A, ?color))). Ideally, the agent should fulfill these intentions serially, invoking others only if the previous one failed to yield an answer. The agent should also be capable of making a choice (that looking at an object is more desirable than asking CYC). However, the choice among these alternatives depends upon the domain being modeled. Thus, the method, as described, should only be modified to try all the possibilities until a suitable answer is found. This does not introduce any restrictions in the cases above. For example, if one wants to be able to choose among the possible alternatives, they can employ complex plans. For example, here are some possibilities:

$$\mathsf{B66!: \forall x[DoWhen(DoONE((LOOKAT(x)\ ASK(CYC, Color(x, ?color)))), Color(x, ?color))]}$$

or

$$\mathsf{B67!: \forall x[DoWhen(DoALL((LOOKAT(x)\ ASK(CYC, Color(x, ?color)))), Color(x, ?color))]}$$

B66 represents the desire to non-deterministically do one of the two acts. B67 represents the desire to non-deterministically do both the acts (the acts DoONE and DoALL are discussed in Chapter 7).

The set T, in the method above, contains only those transformers that are themselves asserted. Notice the use of BELIEVE?(t) in the computation of the set T. Since transformers themselves are terms in the underlying logic, the system should be able to reason about them (i.e., meta-reasoning). Consider the situation when some query, say, BELIEVE?(ϕ), is issued. Also, assume that there is an AntCq transformer of the form:

$$\alpha \rightarrow (\beta \rightarrow \gamma)$$

asserted in the current belief space such that γ, the consequent of the transformer $(\beta \rightarrow \gamma)$ (which is not asserted) unifies with ϕ. The computation of the set T in the method would try to first answer the query BELIEVE?($(\beta \rightarrow \gamma)$) which will back chain through the transformer $\alpha \rightarrow (\beta \rightarrow \gamma)$. If α is satisfied, the transformer $(\beta \rightarrow \gamma)$ will first be asserted and only then will it be included in the set T and used to answer the query regarding ϕ. Similarly, if $\alpha \rightarrow (\beta \rightarrow \gamma)$ is not asserted, its derivation will first be attempted, and so on. It may also be the case that there is a DoWhen transformer in the consequent position of an AntCq transformer, i.e.

$$\alpha \rightarrow \mathsf{DoWhen}(\beta, \gamma)$$

such that ϕ unifies with γ. In this case, too, the query BELIEVE?(DoWhen(β, γ)) would first be generated. Subsequently, the transformer DoWhen(β, γ) may be asserted and used. For example, the sentence, "ask an agent if you want to know the color of something", can be represented as

$$\mathsf{B64!} : \forall x \forall y [\mathsf{Agent}(x) \rightarrow \mathsf{DoWhen}(\mathsf{ASK}(x, \mathsf{Color}(y, ?color)), \mathsf{Color}(y, ?color))]$$

Thus, if the following is also asserted

$$\mathsf{B60!} : \mathsf{Agent}(\mathsf{CYC})$$

the transformer

$$\mathsf{B65!} : \mathsf{DoWhen}(\mathsf{ASK}(\mathsf{CYC}, \mathsf{Color}(\mathsf{A}, ?color)), \mathsf{Color}(\mathsf{A}, ?color))$$

may be asserted in the course of answering the query BELIEVE?(Color(A, ?z)).

The TRANSFORM? methods are defined in a later section. Beliefs can be queried by the user in a direct invocation, or also by acts in the process of fulfilling some intention (as part of the INTEND method).

6.2 Inference—Forward Chaining

One can initiate forward chaining via explicitly invoking the BELIEVE! method. This typically happens in situations where changes in the environment are either brought about by the agent (while performing some act) or by exogenous events that get reported by the agent's perceptory mechanism. The forward chaining method BELIEVE! is defined as follows:

Method
    BELIEVE!(ϕ : Proposition Term; σ : Substitution := NIL)
is
        let $o \leftarrow \phi\sigma$

```
        if PATTERN?(o) then
            Error: Trying to assert a pattern
            return without doing anything
        endif

        ASSERT(o)
        RESULT ← { o }

        find the set T of applicable AntCq transformers, i.e.
        let T ← {t | φ_t ≠ NIL ∧ BELIEVE?(t), where φ_t ← UNIFY(ANT(t),o)}
        also find the applicable WhenDo transformers, i.e.
        let T ← T ∪ {t | φ_t ≠ NIL ∧ BELIEVE?(t), where φ_t ← UNIFY(WHEN(t),o) }
        for each t ∈ T loop
            RESULT ← RESULT ∪ TRANSFORM!(t, φ_t)
        endloop

        find the set of matching propositions, i.e.,
        let B ← {b | φ_b ≠ NIL, where φ_b ← UNIFY(b,o)}
        for each b ∈ B loop
            RESULT ← RESULT ∪ BELIEVE!(b,φ_b)
        endloop

        return RESULT
end BELIEVE!
```

As we can see, only the non-pattern beliefs can be asserted. After being asserted, the procedure tries to apply all possible forward chaining transformations via belief–belief (AntCq) transformers or belief–act (WhenDo) transformers. The transformers may have to be derived via backward chaining before being used. The same issue arises here—that of deciding the order of applying the transformations. We will maintain that all transformations are applied in non-deterministic fashion (or in parallel in the presence of a concurrent model). The procedure also tries to forward chain through all pattern instances that unify with the newly asserted proposition.

The above method also allows for the possibility of deriving new transformers in order to apply some transformations. For example, consider the invocation BELIEVE!(ϕ). If there is an AntCq transformer of the form

$$\alpha \rightarrow (\beta \rightarrow \gamma)$$

or one that has a WhenDo transformer in its consequent

$$\alpha \rightarrow \mathsf{WhenDo}(\beta, \gamma)$$

then, if ϕ unifies with β, in the former case the transformer $(\beta \rightarrow \gamma)$ would first be derived and then used; in the latter case, the transformer WhenDo(β, γ) would be derived and then used.

6.3 Acting

Acting, as mentioned in earlier chapters, is the process of performing an act. This is done by the INTEND procedure. Earlier, we said that acts can be primitive or complex. The former have an effectory component, which is a procedure that drives the agent's actuators. In order to perform an act, the agent has to ensure that the act's preconditions are satisfied, then it executes the effectory procedure, followed by believing the act's effects. The believing of the act's effects could be done à la STRIPS (i.e., they are simply added (or deleted) to (or from) the belief space), or some other belief-updating mechanism may be employed. In case an act is a complex act, it has to be decomposed into a plan (which is a structure of acts), and the intention to perform the act gets replaced by the intention to carry out a plan that decomposes the act. In the absence of any beliefs about existing plans, the agent can indulge in a plan derivation phase. All these steps form a part of the INTEND method that is responsible for performing actions. Below is a specification of the INTEND method that ensures that the act's preconditions are satisfied, determines the act's believed effects, executes the act's actuator (if the act is primitive), finds plan decompositions for complex acts, and believes the act's effects:

```
Method
    INTEND(α : Act Term)
is
        PRECONDS ← BELIEVE?(PreconditionAct(?x,α))
        for each p ∈ PRECONDS loop
            TRANSFORM?(p)
        endloop

        EFFECTS ← BELIEVE?(ActEffect(α,?x))

        if BELIEVE?(Isa(α,PRIMITIVE)) then
            execute the actuator for α
            for each e ∈ EFFECTS loop
                TRANSFORM!(e)
            endloop
        else
            PLANS ← BELIEVE?(PlanAct(?x,α))
            if PLANS then
                PLANS ← { PLAN(p_i) | ∀p_i ∈ PLANS}
                INTEND(DoONE(PLANS))
                for each e ∈ EFFECTS loop
                    TRANSFORM!(e)
                endloop
            else
                use a planning algorithm to derive a plan if possible
            endif
        endif
end INTEND
```

Notice that the act's effects are derived before the act's actuator is executed (or decomposed into a plan). This is important. An act's effects may depend upon the situation in which the act is performed. For example, the sentence, "If a block is on a support, then an effect of picking up the block is that the support is clear and the block is not on the support," gets represented as two beliefs:

$$\mathsf{B31!} : \forall x \forall y[((\mathsf{Isa}(x, \mathsf{BLOCK}) \wedge \mathsf{Isa}(y, \mathsf{SUPPORT})) \wedge \mathsf{On}(x, y)) \rightarrow \mathsf{ActEffect}(\mathsf{PICKUP}(x), \mathsf{Clear}(y))]$$

$$\mathsf{B32!} : \forall x \forall y[((\mathsf{Isa}(x, \mathsf{BLOCK}) \wedge \mathsf{Isa}(y, \mathsf{SUPPORT})) \wedge \mathsf{On}(x, y))$$

$$\rightarrow \mathsf{ActEffect}(\mathsf{PICKUP}(x), \neg \mathsf{On}(x, y))]$$

After performing the act of picking up a block, the effect will be applicable to the specific block under the one that is picked up. This will vary from situation to situation. Thus, the specific effects are derived each time the act is performed *in* the situation it is performed. This is just one of the advantages of having a propositional representation of acts and effects. In operator-based representational formalisms (STRIPS, for example), acts that have context-sensitive effects have to be defined as separate operators. The STRIPS model of the act of picking up a block, for instance, consists of two operators—PICKUP, for picking up a block from a table, and UNSTACK, for the act of picking up a block from the top of another block.

Using the PreconditionAct transformations as specified in the method above is also subject to some discussion. The method assumes that once the transformation is applied, the 'achieved' precondition persists. If an act has several preconditions, it may be possible that one transformation may undo something that was also a precondition of the act about to be performed. Sometimes this situation can be avoided by simply reordering the achieving of the act's preconditions. We have already pointed out earlier that all loops that iterate over a set may be thought of as concurrently executable entities. In that situation, one may lose control over the achieving of preconditions. We will not commit ourselves to these details, since they are dependent upon the specific instance of an action theory employed. We will, however, point out that one can incorporate specific changes to these methods. For example, that section of the method could be replaced by the following one:

loop
    exit loop when $\bigwedge_{p \in PRECONDS}$ *BELIEVE?(PRECONDITION(p))*
    INTEND(DoALL((ACHIEVE(p_1) ... ACHIEVE(p_n)))), where $n = |PRECONDS|$
endloop

DoALL((ACHIEVE(p_1)...ACHIEVE(p_n))) is the act of achieving all of the act's preconditions in some non-deterministic fashion (see Chapter 7 for details of these acts). What we have done above is implicitly performed the PreconditionAct transformation (i.e., bypassed the TRANSFORM? method). Doing this, however, enables us to model an agent that is somewhat skeptical about achieving its preconditions. It will perform the intended act *only* if all of the act's preconditions are believed in the same state. In case achieving of an act's precondition undoes another one, the agent will remain in the loop and try to achieve them all over again.

Similarly, the loops that invoke the TRANSFORM! procedure on the act's effects can be replaced by the following call to INTEND:

INTEND(DoALL((BELIEVE(e_1) ... BELIEVE(e_n)))), where $n = |EFFECTS|$

Regardless of how these parts of the method are implemented, the transformations can be applied explicitly by calling the appropriate transform method or implicitly as shown above. The point of this discussion is to illustrate the precise nature of the transformations rather than presenting details of how they are implemented. It should, however, be noted that regardless of the nature of the implementation, the close relationship between acting and inference is being exploited.

Plans that decompose complex acts are also derived from the agent's belief space (PlanAct propositions). It is possible that an act may have several plans that decompose it (i.e., several PlanAct propositions can be deduced). In the method shown above, we translate the intention to do a complex act into an intention to perform an act called DoONE, whose arguments are all the plans found. This is a control act that will be defined in the next chapter. Basically, one of the plans is chosen and intended.

Typically, complex acts represent abstract domain-specific acts that have plan decompositions comprised of primitive acts. Depending upon the domain, an agent will have to be provided with general plan decompositions for complex acts. In the case where no plan decompositions are derivable, one may be able to use a planning algorithm to decompose the complex act. A classical planner would be usable if at least the complex act's effects are known. In such a case, effects of the complex act can be treated as new goals of the agent, and the agent may be able to derive plans to achieve those goals (via PlanGoal propositions). Typically, it is also the case that effects of complex acts are simply the effects of all the acts that constitute a plan that decomposes that act. Thus, no specific effects may be attributable to the complex act. In such cases, the complex act is merely an abstraction of the plan itself and it may not be possible to use a general planning algorithm. Given a domain, such considerations can be specifically addressed and an appropriate planning algorithm employed.

Next, we will furnish details of various transformations. For each specific class of transformers identified in the previous chapter, we will describe the applicable transformation processes.

6.4 From Beliefs to Beliefs

Instances of the subclasses of Belief–Belief transformers perform belief–belief transformations. We mentioned that these transformers represent reasoning rules. In the context of the underlying logic, rules can be applied to derive new beliefs from existing ones. This can be done using forward as well as backward chaining. We specified that all belief–belief transformers can be used in forward as well as backward chaining. Thus belief–belief transformations are always invoked by either the BELIEVE? or the BELIEVE! procedures. This is where various inference rules for the underlying logic are implemented. In the previous chapter, we defined reasoning rules that were represented using AntCq transformers. AntCq transformers, when used, implement the *modus ponens* rule of inference (also called $\supset$-elimination). The backward chaining transformation procedure is defined as follows:

Method
    TRANSFORM?(t : AntCq; σ : Substitution := NIL)
is
        RESPONSES $\leftarrow$ BELIEVE?(ANT(t)σ)

        if RESPONSES tally with firing criteria for t then
            RESULT $\leftarrow$ { CQ(t)σ}
            foreach r $\in$ RESULT loop
                ASSERT(r)
            endloop
        endif
        return RESULT
end TRANSFORM?

In order to apply a transformation during backward chaining (i.e., $\supset$-elimination), first we determine if the antecedent (or instances of it) are believed. Tallying responses with the firing criteria involves determining instances of antecedents that enable the application of the rule of inference under appropriate substitutions. For each such antecedent, the substitutions are applied to the consequents, and the resulting propositions are asserted and returned as the result of the transformation. If a TMS is employed, this is the place where justifications and support also get computed. Similarly, the forward chaining transformation procedure is as follows:

Method
  TRANSFORM!(t : AntCq; σ: Substitution := NIL)
is
    RESPONSES $\leftarrow$ BELIEVE?(ANT(t)σ)

    if RESPONSES tally with firing criteria for t then
      RESULT $\leftarrow$ $\{CQ(t)\sigma\}$
      foreach $r \in$ RESULT loop
        RESULT $\leftarrow$ RESULT $\cup$ BELIEVE!(r)
      endloop
    endif
    return RESULT
end TRANSFORM!

The above procedure is invoked when the antecedent or a part of it (in case the antecedent is a connective proposition, like And, for example) is BELIEVE!-ed. Thus, the first thing to do is to confirm that the antecedent is believed and then apply the firing criteria. For each proposition that is the consequence of the transformer, a forward chaining BELIEVE! method is applied.

The TRANSFORM? and TRANSFORM! methods are always invoked on transformers that are already asserted. These methods are always invoked by the BELIEVE?, BELIEVE!, and INTEND methods that ensure that the transformers being used are asserted. This applies to all the transformers discussed in this chapter.

6.5 Belief–Act Transformations

While determining the preconditions of an act, an intention to perform the act is carried out only if all the preconditions of the act are satisfied. This represents a backward-chaining belief–act transformation—an intention to carry out an act gets transformed into a BELIEVE? procedure to determine the belief status of the precondition. This may lead to further intentions to perform other acts in order to achieve the preconditions that are not satisfied. The TRANSFORM? method described below returns only after the said precondition is satisfied.

Method
  TRANSFORM?(t : PreconditionAct; σ : Substitution := NIL)
is
    let p $\leftarrow$ PRECONDITION(t)σ
    if BELIEVE?(p) then
      return

```
        else
            INTEND(ACHIEVE(p))
        endif
end TRANSFORM?
```

In the above, ACHIEVE is an act whose object is a proposition that becomes the goal to be achieved. The ACHIEVE act is defined in the next chapter. Achieving a goal involves deriving a plan, via either a PlanGoal proposition or a classical planning procedure. This is defined as a part of the INTEND method for the ACHIEVE action in the following chapter. The TRANSFORM? method above is always invoked by the INTEND method (Section 6.3). As mentioned earlier, if the INTEND method explicitly performs this transformation, the above method is unnecessary.

A belief–act transformation also takes place using the WhenDo transformer that transforms a BELIEVE! into an intention to do some act in reaction to a belief.

```
Method
    TRANSFORM!(t : WhenDo; σ : Substitution := NIL)
is
        RESPONSES ← BELIEVE?(WHEN(t)σ)

        if RESPONSES tally with t then
            REACTION ← ACT(t) σ
            INTEND(REACTION)
        endif
end TRANSFORM!
```

Once again, as in the case of TRANSFORM! for AntCq above, we first determine that the firing criterion is satisfied—the proposition specified in the When slot of the transformer is believed. Since this transformer is a proposition that represents the agent's desire to react to certain situations (beliefs), firing the transformer (or applying the transformation) involves determining the act (the Do part) and intending to perform it as a reaction.

This is a unique approach to modeling reactivity for several reasons. First, the WhenDo transformer enables the representation of an agent's desire to react to certain situations. For example,

$$\mathsf{B69!} : \forall \mathsf{x}[\mathsf{WhenDo}(\mathsf{Broken}(\mathsf{x}), \mathsf{FIX}(\mathsf{x})]$$

represents the general desire, "whenever something is broken, fix it." Second, the desire itself is represented as a proposition. Thus, when asserted, it is believed by the agent, and so the agent can reason about it (see section 6.2 above), discuss it (as it does its other beliefs), as well as *use* it. Usage of the desire is modeled by the transformation shown above. Third, it is tightly coupled with the architecture's reasoning and acting mechanism. This is a direct consequence of the underlying relationship between acting and inference. Lastly, it provides a natural, simple, yet powerful and expressive mechanism that leads an agent to perform actions in response to newly formed beliefs. Several models have been proposed for reacting to situations; however, none of them integrate reactivity via a model of reactivity as an agent's desire to react.

6.6 Act–Belief Transformations

After an action has been executed, its effects are to be believed. This is done by forward chaining through the ActEffect transformers for that act. The INTEND method described above already shows invocation of the TRANSFORM! procedure. In the simplest model, the agent has to believe the specified effect of the act. This is shown below:

Method
  TRANSFORM!(t : ActEffect; σ : Substitution := NIL)
is
    let EFFECT $\leftarrow$ EFFECT(t)σ
    INTEND(BELIEVE(EFFECT))
end TRANSFORM!

The act BELIEVE is a mental act that, when performed, results in the EFFECT being believed. The BELIEVE act will be defined in the following chapter.

Transformations carried out by the DoWhen transformer in backward chaining are also act-belief transformations. The transformer is called with a TRANSFORM? method during backward chaining, resulting in a query BELIEVE?(b), where b = WHEN(t) for the DoWhen transformer. The transformation involves an invocation of an intention to perform the transformer's act component.

Method
  TRANSFORM?(t : DoWhen; σ : Substitution := NIL)
is
    let ACT $\leftarrow$ DO(t) σ
    INTEND(ACT)
    return BELIEVE?(When(t)σ)
end TRANSFORM?

The specification above assumes that the transformation will end up changing the belief space of the agent in such a way that when the BELIEVE? method is called again, it will respond with an answer. In adverse circumstances, when the act may not be able to produce the desired effect, this method will lead to an infinite loop. If this seems unreasonable, the method above can be changed. Remember that this is a very specific type of transformer that is to be used as a part of an agent model only in situations where such a transformation is feasible.

This kind of a representation is unique in many respects. First, the DoWhen transformer enables the representation of an agent's desire to acquire new information. For example,

$$\mathsf{B65!:DoWhen(ASK(CYC, Color(A, ?color)), Color(A, ?color))}$$

represents the desire, "if you want to know the color of A, ask CYC." Second, the desire itself is represented as a proposition. Thus, when asserted, it is believed by the agent, and so the agent can reason about it (see Section 6.2), discuss it (as it does its other beliefs), as well as *use* it. Usage of the desire is modeled by the transformation shown above. Third, it is tightly coupled with the architecture's reasoning and acting mechanism. This is a direct consequence of the underlying relationship between acting and inference. Lastly, it provides a natural, simple, yet powerful and expressive mechanism that leads an agent to perform actions during backward chaining inference in order to acquire some information.

6.7 Other Belief–Belief Transformations

Depending on the underlying logical commitments, it may also be possible to define further transformers. For instance, in default logic [Reiter 1980], it is possible to make (defeasible) deductions using default rules. Consider the standard form of a default rule:[1]

$$\frac{\alpha(\mathsf{x}) : \beta(\mathsf{x})}{\gamma(\mathsf{x})}$$

This can be used to represent "typically, birds fly" as

$$\frac{\mathsf{Bird}(\mathsf{x}) : \mathsf{Fly}(\mathsf{x})}{\mathsf{Fly}(\mathsf{x})}$$

In SNePSwD [Cravo and Martins 1993], the above rule is represented as the default rule:

$$\bigtriangledown(\mathsf{x})[\mathsf{Bird}(\mathsf{x}) \rightarrow \mathsf{Fly}(\mathsf{x})]$$

where $\bigtriangledown$ is called a *default quantifier*, the presence of which makes it a default rule. The SNePSwD rule is a belief–belief transformer. However, it is unlike the ones we have seen so far. The above transformer, which implements the semantics of a default rule (default-$\supset$–elimination), can be interpreted as, "when something is a bird then, *it is typically the case* that it flies". For a specific constant, say TWEETY, about which the proposition Bird(TWEETY) is asserted, the transformation is applicable only if it can be determined that TWEETY is a typical bird (or in terms of the language used by Cravo and Martins, the instance of the default rule with substitution {TWEETY/x} is applicable). This involves confirming that the resulting theory with the conclusion added is consistent. We can express the TRANSFORM! method for such transformers (we will call them DefaultAntCq) as follows:[2]

Method
    TRANSFORM!(t : DefaultAntCq; σ: Substitution := NIL)
is
        RESPONSES ← BELIEVE?(ANT(t)σ)

        if RESPONSES tally with firing criteria for t
            and BELIEVE?(¬CQ(t)σ) returns NIL then
            RESULT ← {CQ(t)σ}
            foreach r ∈ RESULT loop
                RESULT ← RESULT ∪ BELIEVE!(r)
            endloop
        endif
        return RESULT
end TRANSFORM!

There is an extra condition added to the firing of the rule—that of ensuring that the negation of the consequent does not follow from the current context (theory). The TRANSFORM? method can be similarly described. A formalism with default rules leads to other

[1] The default rule has the following interpretation: if, in a given belief space, there is an instance of x for which $\alpha(\mathsf{x})$ follows, and if it is consistent to believe $\beta(\mathsf{x})$ for the same instance, then $\gamma(\mathsf{x})$ can also be believed for that instance.

[2] The procedure is simplistic in that it does not say anything about maintaining *extensions*. It is not our intention here to indulge in that level of detail.

concerns about the nature of truth-maintenance and defeasibiltiy (in light of belief-revision). See [Cravo and Martins 1993] for a specification of such a logic as well as a theory of belief revision. We have inserted this discussion here to highlight two things—that the formalism can be extended to non-monotonic logics and that the rational engine is capable of providing a correct implementation of the proof theory.

6.8 Discussion

In this chapter, we have described the chief constituents of a rational engine that operates on the representations defined in the last two chapters. Notice that, due to the design of the BDI architecture as an object-oriented system, the specification of each component of the rational engine is localized to individual classes of beliefs and acts. This makes the architecture naturally extendible. For instance, each specific acting or reasoning strategy (and hence the underlying logic) can be easily specified. One can introduce additional classes of beliefs/transformers, especially for experimenting with different logics, and then define explicit transformation procedures without having to modify the existing components themselves. Also, as pointed out earlier, various transformations correspond to the notions of acting, inference, reacting, plan decomposition, etc., in a uniform and intuitive manner.

In Figure 5.1 the arrows (and their directions) indicate the existence of specific transformation procedure for each class of transformers. In the next two chapters, we define a specific ontology of acts and show how the rational engine above can be extended to incorporate the semantics of the defined acts. That will complete the specification of the core constituents of the OK BDI architecture along with its rational engine. That will be followed by an exploration of a parallel implementation model for such architectures. A specific instantiation of the OK architecture will then be presented.

Chapter 7

Acts and the Structure of Plans

One of the claims we are making about the OK BDI architecture defined in the last three chapters is that it is extremely extendible. We have made a representational commitment in terms of defining a class hierarchy of 'knowledge objects' that correspond to individuals, beliefs, transformers, and acts. The architecture can be used for modeling different ontologies of beliefs as well as acts. Moreover, the design of the inference and acting capabilities using a rational engine facilitates user-definable deliberation strategies (or logics) as well as action theories. In this chapter, we extend our ontological commitments by defining a subclass hierarchy of acts. Recall that acts are also conceptual entities and all objects that represent specific acts respond to INTEND methods. In this chapter, we outline an ontology of acts whose design is based upon a classification of acts into physical, mental, and intentional categories. Every act that can be carried out by the modeled agent affects different "states", and, is classified accordingly. For all classes of acts, we will also outline how each responds to the INTEND methods.

7.1 Types of Acts

Earlier, we mentioned that every primitive act has an effectory component—one that actually brings about the changes that represent the effects of the action. Depending upon the kind of state changed by an action, we identify different types of actions. We explicitly identify three states in which an agent can explicate its behavior (or bring about changes):

external world states
: These are the states of the physical world in which the agent is situated.

mental states
: A belief space (i.e., the set of beliefs of the agent) is the agent's mental state. A belief space includes all the asserted propositions (hypotheses as well as derived).

intentional states
: The set of the agent's current intentions form an intentional state.

Accordingly, we identify three classes of acts:

physical acts (also called *external acts*)
: These are the acts whose actions affect the world.

mental acts

These are the acts whose actions affect the agent's beliefs.

control acts (also called *intentional acts*)

These are the acts whose actions affect the agent's intentions.

The above classification is taken directly from the design of the SNePS acting system [Kumar and Shapiro 1991b]. Let us now discuss each of these acts and how they are modeled in the OK BDI formalism.

7.2 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 a block, the arm actually moves to the block, grasps it, and then lifts it up. Depending on the set of interfaces provided to the agent (like an arm, a speech synthesizer, etc.), we need corresponding actuators to be able to carry 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. The class of physical acts is a meta-class defined as follows:

Definition 15:

Class: Physical Act

Superclass(es): Act Term, Conceptual Entity Term

Slots: none.

Methods: none.

Syntax: Same as defined for the class Act Term.

Description: This is the class of acts whose instances are domain specific classes denoting physical acts.

For example, in the blocksworld domain, the acts of picking up and putting down a block are defined as follows:

Example 1:

Class: PICKUP

Superclass(es): Physical Act, Act Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

OBJECT1: can be any conceptual entity that represents a block.

Methods: If o is an instance of this class, then

ACTION(o): Returns the named individual of the action concept (PICKUP).

OBJECT1(o): Returns the value of the OBJECT1 slot.

Description: The act of picking up the entity represented by OBJECT1(o1).

Example Instances: The act "Pick up A" can be represented by an object written as:

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

where A is an instance of the class Individual Term, A1 is the label.

Example 2: ______

Class: PUT

Superclass(es): Physical Act, Act Term, Conceptual Entity Term

Slots: All instances of this class have the following slots:

OBJECT1: Can be any conceptual entity that represents a block.

OBJECT2: Can be any conceptual entity that represents a support.

Methods: If o is an instance of this class, then

ACTION(o): Returns the named individual of the action concept (PUT).

OBJECT1(o): Returns the value of the OBJECT1 slot.

OBJECT2(o): Returns the value of the OBJECT1 slot.

Description: The act of putting the entity represented by OBJECT1(o) on top of the entity represented by OBJECT2(o).

Example Instances: The act "Put A on B" can be represented by an object written as:

$$\mathsf{A2 : PUT(A, B)}$$

where A and B are instances of the class Individual, and A2 is the label.

Other examples of physical acts that we have used in discussions in the previous chapters are LOOKAT, ASK, etc. All these acts respond to the INTEND method as defined previously in Section 6.3.

7.3 Acts vs. Operators

At this point, it is appropriate to discuss an important consideration regarding the representations of acts. In Chapter 2, we reviewed the formalisms of several other systems that provide action representations for the purposes of reasoning about actions and plans. Actions were represented using operator schemas that specified the action's objects, preconditions, and effects. Unlike our representations, these schemas were different from the formalism used for representing beliefs (or facts). Consequently, specialized procedures had to be defined in order to *use* the operators (for example, a planner) so as to model reasoning about actions. Since the formalism used to represent operators was different from that used to represent beliefs, the agent could not have any explicit beliefs about operators. Consequently, it did not *understand* its actions as it did its beliefs. It could only use and perform actions using the planning and execution modules. For example, the representation of the *Puton* operator in SIPE [Wilkins 1988] is reproduced below:

Operator: Puton
Arguments: block1, object1 Is Not block1;
Purpose: (On block1 object1);
Plot:
Branch1:
Goals: (Clear object1);
Branch2:
Goals: (Clear block1);
End Parallel

Process
Action: Puton.Primitive;
Arguments: block1, object1;
Resources: block1;
Effects: (On block1 object1);

End Plot Operator

In the OK formalism, we have achieved a truly integrated approach towards representation and reasoning about beliefs and acts. Both beliefs as well as acts are conceptual entities. Since all conceptual entities are terms in the underlying logic, the agent is able to represent beliefs about beliefs as well as acts. The latter can be propositions that represent preconditions and effects of actions (we have defined the PreconditionAct and ActEffect transformers). The agent can have other beliefs about the act as well, for example, a belief that an act is primitive, or that it was performed by another agent, etc. The set of propositions below, for example, represent beliefs about the act PUT:

$\mathsf{B701!}: \mathsf{Isa(PUT, PRIMITIVE)}$
$\mathsf{B702!}: \forall x \forall y[(\mathsf{Isa}(x, \mathsf{BLOCK}) \wedge \mathsf{Isa}(y, \mathsf{SUPPORT})) \rightarrow \mathsf{PreconditionAct}(\mathsf{Held}(x), \mathsf{PUT}(x, y))]$
$\mathsf{B703!}: \forall x \forall y[(\mathsf{Isa}(x, \mathsf{BLOCK}) \wedge \mathsf{Isa}(y, \mathsf{SUPPORT})) \rightarrow \mathsf{PreconditionAct}(\mathsf{Clear}(y), \mathsf{PUT}(x, y))]$
$\mathsf{B704!}: \forall x \forall y[(\mathsf{Isa}(x, \mathsf{BLOCK}) \wedge \mathsf{Isa}(y, \mathsf{SUPPORT})) \rightarrow \mathsf{ActEffect}(\mathsf{PUT}(x, y), \neg \mathsf{Held}(x))]$
$\mathsf{B705!}: \forall x \forall y[(\mathsf{Isa}(x, \mathsf{BLOCK}) \wedge \mathsf{Isa}(y, \mathsf{SUPPORT})) \rightarrow \mathsf{ActEffect}(\mathsf{PUT}(x, y), \mathsf{Clear}(x))]$
$\mathsf{B706!}: \forall x \forall y[(\mathsf{Isa}(x, \mathsf{BLOCK}) \wedge \mathsf{Isa}(y, \mathsf{SUPPORT})) \rightarrow \mathsf{ActEffect}(\mathsf{PUT}(x, y), \mathsf{On}(x, y))]$
$\mathsf{B707!}: \forall x \forall y[(\mathsf{Isa}(x, \mathsf{BLOCK}) \wedge \mathsf{Isa}(y, \mathsf{BLOCK})) \rightarrow \mathsf{ActEffect}(\mathsf{PUT}(x, y), \neg \mathsf{Clear}(y))]$

The above models of SIPE and OK are different in the way they represent the act of putting a block on another block. The fundamental difference in the representations is the formalism used to represent the action. As we have defined it, the OK rational engine is able to reason about beliefs as well as acts in a uniform manner. If you look at the definition of the INTEND method (Section 6.3) closely, you will notice that the above beliefs (which all happen to be generic rules!) will be used in order to fulfill any intention to carry out the act of putting a block on something. We claim that our representations are also more general. For example, look at the proposition B707—it will be used only in the situation when the second object of the action is another block (assuming that UVBR, as described in Section 5.1, is enforced). The SIPE operator for *Puton* specifically excludes this situation. It uses yet another operator for this situation. Thus, in operator-based action formalisms, a single act is actually represented by several operators (we have already pointed out that STRIPS uses two operators—STACK and PUT—to model the act of putting a block on another object). In the OK formalism, we do not have to worry about this. A specific act's preconditions and effects are derived from the agent's beliefs about the act at the time the act itself is performed.

7.4 Mental Acts

Beliefs of the agent may change when actions are performed or an inference is carried out. We have already shown how the latter is accomplished (using the methods BELIEVE? and BELIEVE!). When an inference leads to derivation of new beliefs, the beliefs are asserted in the agent's belief space by the methods themselves. This represents an implicit implementation of the mental act of believing, which is an optimization we have decided to incorporate in the OK architecture (this was discussed in Chapter 3). Effects of acts also lead to adding or deleting of new beliefs. In the OK architecture, we define two mental actions—BELIEVE and DISBELIEVE—that are used to update the beliefs of the agent after an action is performed. The following is a definition of the class of mental actions:

Definition 16: ______

Class: Mental Act

Superclass(es): Act Term, Conceptual Entity Term

Slots: none.

Syntax: Same as acts.

Methods: none.

Description: This is the metaclass of mental acts. Instances of subclasses of this class are acts that bring about changes in the agent's belief space.

Definition 17: ______

Class: BELIEVE

Superclass(es): Mental Act, Act Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

PROPOSITION: Can be an instance of any subclass of Proposition Term.

Syntax: Same as acts.

Methods: If o is an instance of this class then

ACTION(o): Returns the named individual BELIEF.

PROPOSITION(o): Returns the object proposition.

Description: If the proposition ¬o is already asserted, it is unasserted. Then the proposition o is asserted in the belief space.

Definition 18: ______

Class: DISBELIEVE

Superclass(es): Mental Act, Act Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

PROPOSITION: Can be an instance of any subclass of Proposition Term.

Syntax: Same as acts.

Methods: If o is an instance of this class then

ACTION(o): Returns the named individual BELIEF.

PROPOSITION(o): Returns the object proposition.

Description: If the proposition o is already asserted, it is unasserted. Then the proposition ¬o is asserted in the belief space.

Simply adding and removing propositions from a 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 (this is something we have called *deductive efficiency*; see [Kumar and Shapiro 1993]). The presence of a truth-maintenance system (TMS) can easily help solve this problem. Typically, TMSs provide explicit operations for adding and deleting new beliefs to and from a belief space. These operations, in addition to 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 above two actions should be implemented using the appropriate TMS operations. This strategy implements the *extended STRIPS assumption* [Georgeff 1987]. Having a TMS also simplifies several plan representations as well as enabling the uniform representation of acts with context-sensitive effects. These are important results from this work.

7.5 Control Acts—The Structure of Plans

Plans, in the OK formalism, are also mental entities. However, we will not define a separate class for them, since they are also acts—albeit control acts. Control acts, when performed, change the agent's intentions about carrying out acts. Thus, when an agent intends to do a control act (as part of intending to do some plan), it changes its intentions as specified by the control act. All control acts are instances of the metaclass Control Act, which is defined below:

Definition 19:

Class: Control Act

Superclass(es): Act Term, Conceptual Entity Term

Slots: none.

Syntax: Same as defined for the class Act Term.

Methods: none.

Description: This is a metaclass of control acts. Specific control acts are classes that are instances of this class. All control acts, when performed, bring about changes in the intentional states of the modeled agent.

We have several different types of control acts that enable us to express and formulate various plan structures. Like specific classes of propositions defined in Chapter 4, we will define subclasses for each type of control act.

7.5.1 Sequencing Acts

The simplest plan structure is a linear plan (the kind generated by the STRIPS planner [Fikes and Nilsson 1971]), which contains a sequence of acts to be executed in the order given. In our ontology, we define a control action called Sequence that takes two acts as its objects and executes one after the other. The class Sequence is defined below:

Definition 20:

Class: Sequence

Superclass(es): Control Act, Act Term, Conceptual Entity Term

Slots: All instances of this class contain the following slots:

OBJECT1: Can be any object that is an Act Term.

OBJECT2: Can be any object that is an Act Term.

Methods: If o is an instance of this class then

ACTION(o): is defined to return nothing.

OBJECT1(o): Returns the act object in the OBJECT1 slot.

OBJECT2(o): Returns the act object in the OBJECT2 slot.

Syntax: The syntax of these acts is the name of the class SEQUENCE followed by its two objects in argument position. I.e.,

$$\mathsf{SEQUENCE}(\langle \mathsf{OBJECT1(o)} \rangle, \langle \mathsf{OBJECT2(o)} \rangle)$$

Description: This act enables the representation of sequential (linear) plan structures. This representation is based on the design of the sequential control act of the SNePS acting system [Kumar and Shapiro 1991b].

The ACTION method for instances of this class is shadowed by a NULL method.[1] This is because in our ontology we do not, as yet, recognize the concept of sequencing (i.e., the named individual SEQUENCE) as an intensional entity. Sequencing is an act the agent can perform by applying the INTEND method (shown below). However, it does not have any beliefs about the individual named SEQUENCE as such. This can also be illustrated via examples in natural language. While talking about sequential activity, we normally do not explicitly use the term SEQUENCE; rather, we use indicators like "and then", in which the notion of a sequence is implicit. This is true for all control acts. While the acts themselves are intensional entities, the names of the actions are not. Of course, in a situation where one explicitly needs to model these as intensional concepts, one is free to use the ACTION slot as for physical and mental acts. The INTEND method for instances of this class is specialized using the definition below (where the act α is of the form $\mathsf{SEQUENCE}(\alpha_1, \alpha_2)$):

Method
    INTEND (α : Sequence)
is
        INTEND(OBJECT1(α))
        INTEND(OBJECT2(α))
end INTEND

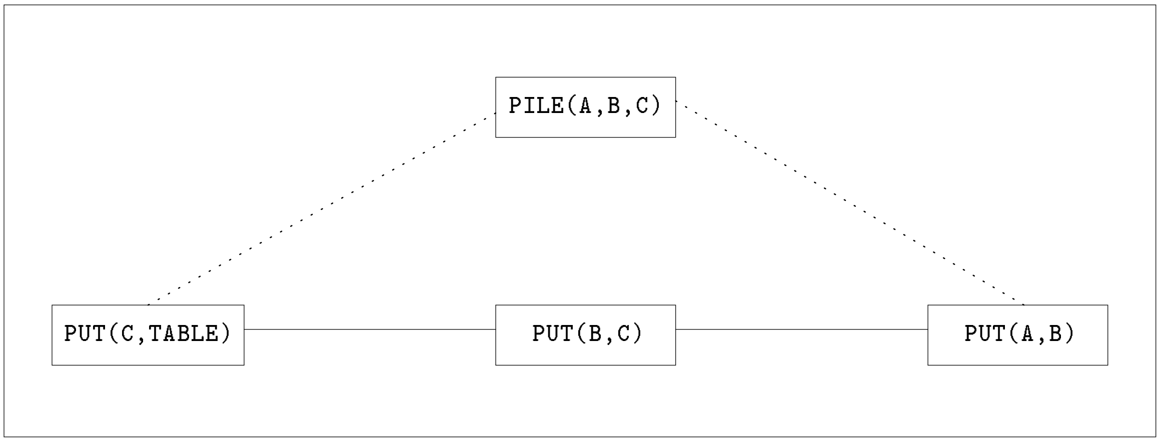

Figure 7.1: An example of sequencing. We are using the pictorial representation of plans as procedural nets as defined in NOAH. Thus PILE(A,B,C) is a complex act which gets elaborated into a sequence of three PUT acts as shown (the dotted line represents expansion). The pictorial representation being used serves to relate the ontology to existing plan representations.

Since both OBJECT1 and OBJECT2 slots of a sequencing act can themselves be sequencing acts, we have a general structure for plans of sequential actions. For example, "A plan for piling A on B on C is to first put C on the table and then put B on C and then put A on B" is represented as

$$\begin{aligned}\mathsf{B203!}: \mathsf{PlanAct}(&\mathsf{SEQUENCE}(\mathsf{PUT}(\mathsf{C},\mathsf{TABLE}),\mathsf{SEQUENCE}(\mathsf{PUT}(\mathsf{B},\mathsf{C}),\mathsf{PUT}(\mathsf{A},\mathsf{B}))),\\ &\mathsf{PILE}(\mathsf{A},\mathsf{B},\mathsf{C}))\end{aligned}$$

which is pictorially depicted in Figure 7.1. Notice the use of the PlanAct proposition to specify the expansion of the complex act PILE(A,B,C). The agent may believe this proposition either via a derivation from a reasoning rule like

$$\begin{aligned}&\mathsf{B210!}: \forall \mathsf{x}\forall \mathsf{y}\forall \mathsf{z}[((\mathsf{Isa}(\mathsf{x},\mathsf{BLOCK})\wedge \mathsf{Isa}(\mathsf{y},\mathsf{BLOCK}))\wedge \mathsf{Isa}(\mathsf{z},\mathsf{BLOCK}))\rightarrow\\ &\quad \mathsf{PlanAct}(\mathsf{SEQUENCE}(\mathsf{PUT}(\mathsf{z},\mathsf{TABLE}),\mathsf{SEQUENCE}(\mathsf{PUT}(\mathsf{y},\mathsf{z}),\mathsf{PUT}(\mathsf{x},\mathsf{y}))),\mathsf{PILE}(\mathsf{x},\mathsf{y},\mathsf{z}))]\end{aligned}$$

as discussed in Section 5.4, where the rule may have been explicitly told to the agent as part of agent's domain knowledge, or the agent may indulge in a planning activity (like traditional classical planners) and come up with the expansion. In our discussion, we will continue to use the former technique of plan derivation. Where applicable, we will indicate the type of planning algorithm/strategy that may be employed (if it exists).

7.5.2 Nondeterministic Choice

Whenever there are many plans that can achieve a goal (or decompose a complex act), the agent has to make a choice. If there is no way to choose (or all plans seem equally

[1]Since Sequence is a subclass of Act Term, it inherits the ACTION slot accessor method. A NULL method shadows an inherited method to return nothing.

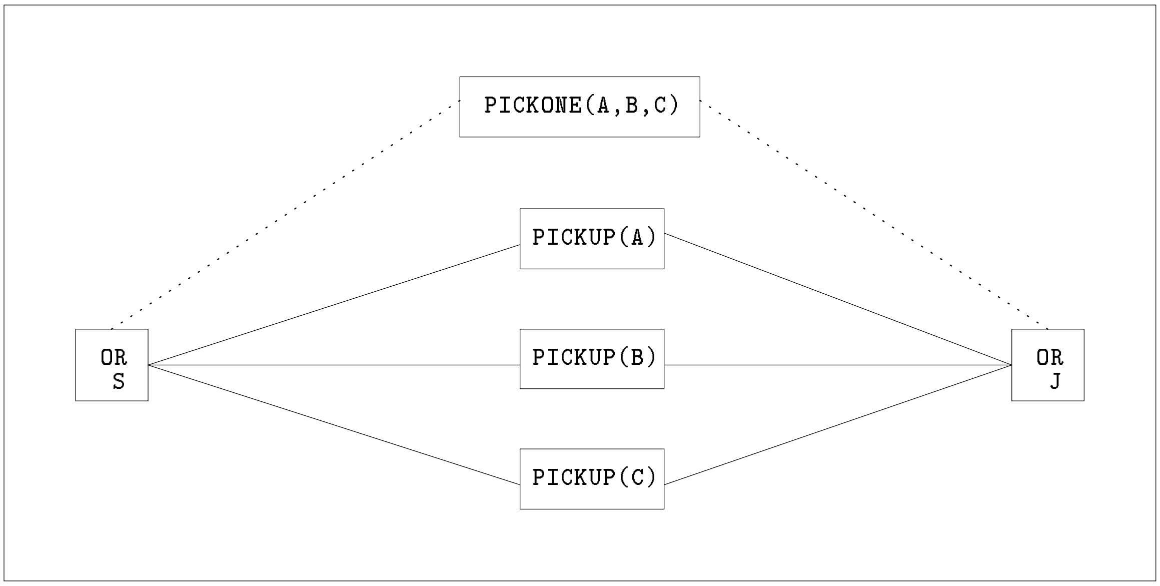

Figure 7.2: An example of nondeterministic choice

viable), the agent can make a nondeterministic commitment to any one of the options. The plan structure (control act) representing a nondeterministic choice of acts/plans is called a DoONE.

Definition 21:

Class: DoONE

Superclass(es): Control Act, Act Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

OBJECT1: Contains a set of acts.

Methods: If o is an instance of this class then

ACTION(o): Shadowed to a NULL method.

OBJECT1(o): Returns the contents of the OBJECT1 slot.

Syntax: If $(\alpha_1 \alpha_2 \ldots \alpha_n)$ is the value of OBJECT1(o), then we write o as

$$\mathsf{DoONE}((\alpha_1 \alpha_2 \ldots \alpha_n))$$

Description: The act of randomly choosing an act to perform from a set of acts. The representation is based on the design of the DoONE control act of the SNePS acting system [Kumar and Shapiro 1991b].

The INTEND method for instances of this class is specified below (where the act α is of the form $\mathsf{DoONE}((\alpha_1\ \alpha_2 \ldots \alpha_n)))$:

Method
    INTEND(α : DoONE)
is
        let $i \leftarrow$ *a random number between* $1 \ldots n$
        INTEND(α_i)
end INTEND

This is similar to the OR Splits of the procedural net formalism [Sacerdoti 1977]. For example, "A plan to pick up one of the blocks A, B, and C is to either pick up A, or pick up B, or pick up C" will be represented as

B715! : PlanAct(DoONE((PICKUP(A) PICKUP(B) PICKUP(C)), PICKONE(A, B, C))

The equivalent procedural net is shown in Figure 7.2.

7.5.3 Nondeterministic Ordering

Nondeterministic ordering is similarly an act that enables the agent to perform all of its object acts in some nondeterministic order. The class of acts, called DoALL, is defined as follows:

Definition 22:

Class: DoALL

Superclass(es): Control Act, Act Term, Conceptual Entity Term

Slots: All instances of this class have the following slots defined:

OBJECT1: Contains a set of acts.

Methods: If o is an instance of this class then

ACTION(o): is defined to return nothing.

OBJECT1(o): Returns the contents of the OBJECT1 slot.

Syntax: If $(\alpha_1 \alpha_2 \ldots \alpha_n)$ is the value of OBJECT1(o), then we write o as

$$\mathsf{DoALL}((\alpha_1\ \alpha_2\ \ldots \alpha_n))$$

Description: The act of doing all the object acts in some unspecified order. This is based on the design of the DoALL control act of the SNePS acting system [Kumar and Shapiro 1991b].

The INTEND method for instances of this class is specified below (where the act α is of the form DoALL$((\alpha_1\ \alpha_2\ \ldots \alpha_n))$):

Method
    INTEND(α : DoALL)
is
        let ACTS $\leftarrow$ *OBJECT1(α)*
        foreach $\alpha_i \in$ *ACTS in random fashion loop*
            INTEND(α_i)
        endloop
end INTEND

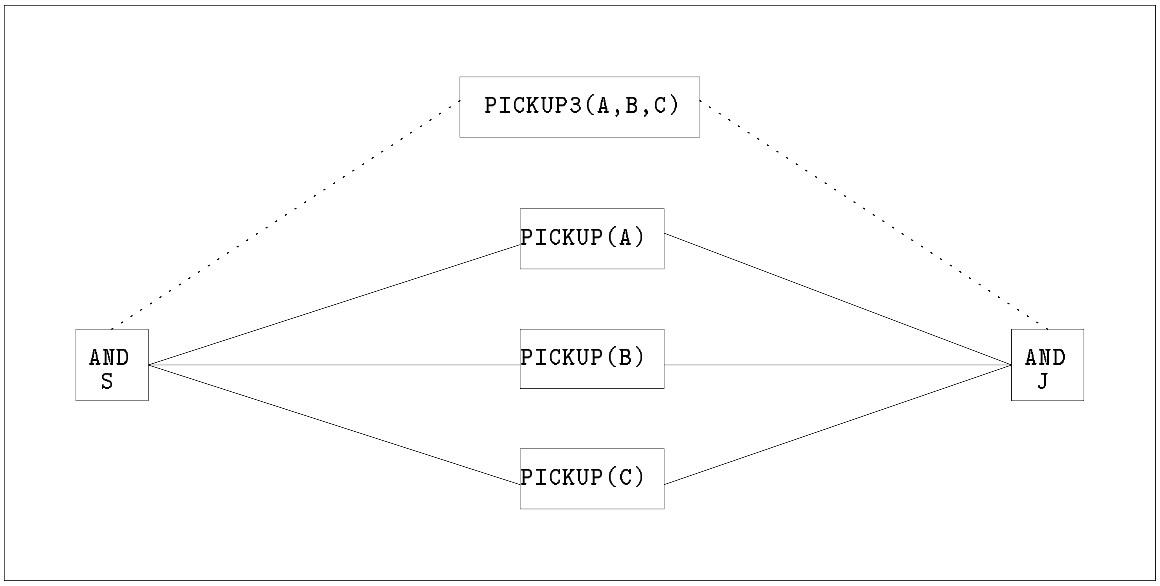

Figure 7.3: An example of nondeterministic ordering

This is similar to the AND Splits of the procedural net formalism. For example, "A plan to pick up three blocks A, B, and C is to pick up A, pick up B and pickup C" will be represented as

$$711! : \mathsf{PlanAct}(\mathsf{DoALL}((\mathsf{PICKUP}(A)\ \mathsf{PICKUP}(B)\ \mathsf{PICKUP}(C))),\ \mathsf{PICKUP3}(A, B, C))$$

This is depicted in Figure 7.3. One typical situation where a DoALL act is employed is during the planning process for achieving conjunctive goals where each act or plan in the list of acts of the DoALL achieves (at least) one conjunct.

7.5.4 Conditional Acts

Acts and plans sometimes are carried out only under certain conditions. Say we have a choice of one or more plans or acts, where each can achieve (or decompose) some goal (or complex act). However, at any given instant one (or more) of them may be eligible (and only one of them needs to be performed anyway). Conditional acts facilitate the specification of eligibility conditions. Our representation of a conditional act, called IF, is based upon Dijkstra's guarded-IF command [Dijkstra 1976] and is defined as follows:

Definition 23:

Class: IF

Superclass(es): Control Act, Act Term, Conceptual Entity Term

Slots: All instances of this class have the following slots defined:

OBJECT1: Contains a set of guarded acts.

Methods: If o is an instance of this class then

ACTION(o): Shadowed to a NULL method.

OBJECT1(o): Returns the contents of the OBJECT1 slot.

Syntax: If $(\gamma_1\gamma_2\ldots\gamma_n)$ is the value of OBJECT1(o) then we write o as

$$\mathsf{IF}((\gamma_1\gamma_2\cdots\gamma_n))$$

where the guarded acts $\gamma_1, \gamma_2, \ldots, \gamma_n$ are written as specified below.

Description: This is the class of conditional acts. The representation is based on the design of the conditional control act of the SNePS acting system [Kumar and Shapiro 1991b].

A guarded act is a structured control act defined as follows:

Definition 24:

Class: Guarded Act

Superclass(es): Control Act, Act Term, Conceptual Entity Term

Slots: All instances of this class have the following slots defined:

CONDITION: Can be any object that is a Proposition Term.

THEN: Can be any object that is an Act Term.

Methods: If o is an instance of this class then

ACTION(o): Shadowed to a NULL method.

CONDITION(o): Returns the contents of the CONDITION slot.

THEN(o): Returns the contents of the THEN slot.

Syntax: If ϕ = CONDITION(o) and α = THEN(o) then, o is written as

$$[\phi \rightsquigarrow \alpha]$$

Description: The class of guarded acts. Guarded acts only appear as objects of conditional and iterative acts (see below).

The guarded act as defined above may seem very similar to a Belief–Act transformer defined in Chapter 5, but it isn't. For one, a guarded act is not a proposition (belief), so it cannot be believed. Second, it does not partake in the transformation process, like transformers. As we will see below, when a condition of a guarded act is believed, it is not necessary that the corresponding act will be performed. The condition part of a guarded act is the way of specifying eligibility conditions for acts in conditional acts. The INTEND method for instances of the class IF is specified below (where the act α is of the form $\mathsf{IF}((\gamma_1\ \gamma_2\ \ldots\gamma_n))$):

Method
    INTEND(α : IF)
is
    let POSSIBLE-ACTS $\leftarrow$ { }
    let ACTS $\leftarrow$ *OBJECT1(α)*

    foreach $\gamma_i \in$ *ACTS loop*

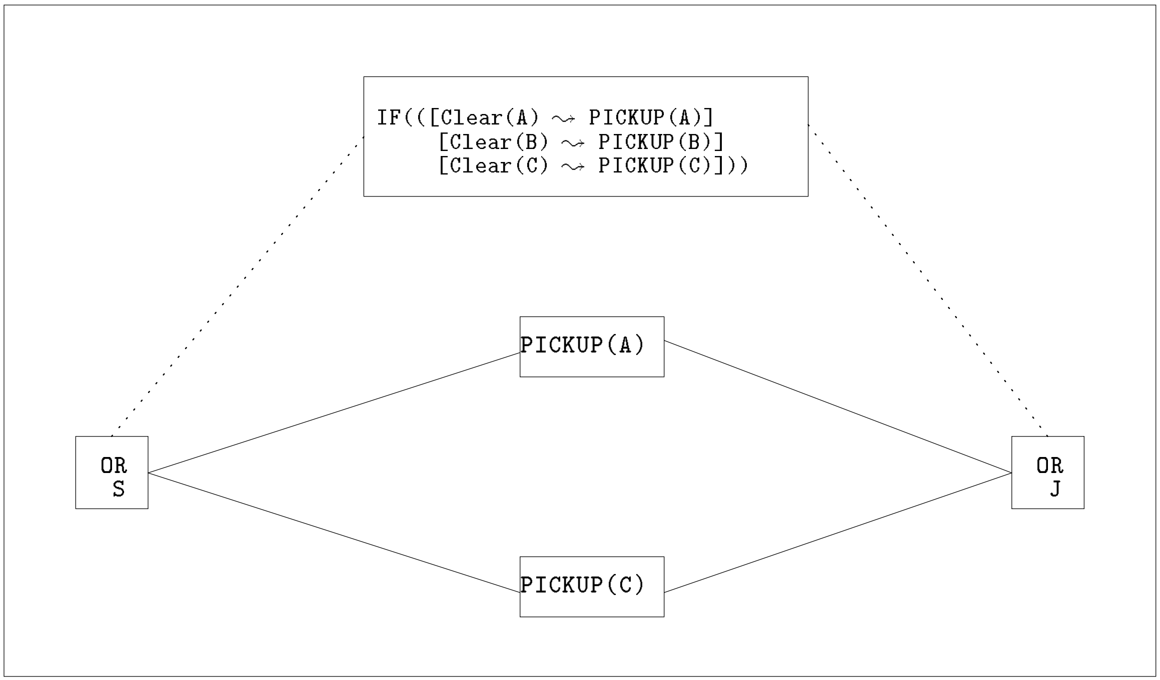

Figure 7.4: An example of a conditional act

if BELIEVE?(CONDITION(γ_i)) then
POSSIBLE-ACTS ← POSSIBLE-ACTS ∪ THEN(γ_i)
endif
endloop

if POSSIBLE-ACTS then
INTEND(DoONE(POSSIBLE-ACTS))
else
Report Error
endif
end INTEND

The method above specifies a "plan decomposition" of the conditional act. Every time such an act is decomposed, we may get different expansions depending upon the conditions that hold in the belief space at the time the IF act is executed. For example, if the agent believes "A is clear" and "C is clear", the intention to perform the act

$$\mathsf{IF}(([\mathsf{Clear(A)} \rightsquigarrow \mathsf{PICKUP(A)}]\ [\mathsf{Clear(B)} \rightsquigarrow \mathsf{PICKUP(B)}]\ [\mathsf{Clear(C)} \rightsquigarrow \mathsf{PICKUP(C)}]))$$

will result in the plan

$$\mathsf{DoONE((PICKUP(A)\ PICKUP(C)))}$$

This is depicted in Figure 7.4. If, in another situation, the agent believes that "B is clear" and "C is clear", then the IF act will expand into the act

$$\mathsf{DoONE((PICKUP(B)\ PICKUP(C)))}$$

Of course, if none of the guards is satisfied, it results in an error. This implies that not all the situations were accounted for and that the agent is not able to carry on. Different approaches are possible here, but they are not relevant to our present discussion. Conditional acts may be generated by a planning algorithm like CNLP[2] [Peot and Smith 1992] or specified explicitly to the agent as plan decompositions (using PlanAct or PlanGoal propositions).

7.5.5 Iterative Acts

In some situations, a plan or act may have to be repeated several times in order to accomplish a goal (or a complex act). Iterative acts facilitate the representation of such plans. Like conditional acts, iterative acts (ITERATE) are formulated in the spirit of Dijkstra's guarded-DO.

Definition 25:

Class: Iterate

Superclass(es): Control Act, Act Term, Conceptual Entity Term

Slots: All instances of this class have the following slots defined:

OBJECT1: Contains a set of guarded-acts.

Methods: If o is an instance of this class then

ACTION(o): Shadowed to a NULL method.

OBJECT1(o): Returns the contents of the OBJECT1 slot.

Syntax: If $(\gamma_1\gamma_2 \ldots \gamma_n)$ is the value of OBJECT1(o), then we write o as

$$\mathsf{ITERATE}((\gamma_1\gamma_2 \ldots \gamma_n))$$

where the guarded acts $\gamma_1, \gamma_2, \ldots, \gamma_n$ are written as specified above.

Description: The class of iterative acts. This representation is based on the design of the iterative control act of the SNePS acting system [Kumar and Shapiro 1991b].

The INTEND method for instances of this class is specified below (where the act α is of the form $\mathsf{ITERATE}((\gamma_1\ \gamma_2\ \ldots\gamma_n)))$:

Method
    INTEND(α : Iterate)
is
    let POSSIBLE-ACTS $\leftarrow$ NIL
    let ACTS $\leftarrow$ OBJECT1(α)

    foreach $\gamma_i \in$ ACTS loop
        if BELIEVE?(CONDITION(γ_i)) then
            POSSIBLE-ACTS $\leftarrow$ POSSIBLE-ACTS $\cup$ THEN(g_i)
        endif
    endloop

    if POSSIBLE-ACTS then

[2] Conditional Nonlinear Planning

INTEND(SEQUENCE(DoONE(POSSIBLE-ACTS), α))
endif
end INTEND

As we can see from above, the decomposition of an iterative act is a plan with two acts—first, the nondeterministic choice among the acts whose guards are believed and, second, the iterative act itself. When none of the guards are believed, the iteration terminates. For example, given that "A, B, and C are clear", the agent's intention to do the act

A710 :
ITERATE(([Clear(A) $\rightsquigarrow$ PICKUP(A)][CLEAR(B) $\rightsquigarrow$ PICKUP(B)][Clear(C) $\rightsquigarrow$ PICKUP(C)]))

will get decomposed into the act

SEQUENCE(DoONE((PICKUP(A) PICKUP(B) PICKUP(C))), A710)

(where we have chosen to use the label A710 to refer to the iterative act). If the agent chooses to pick up B, the next expansion of the iteration will be

SEQUENCE(DoONE((PICKUP(A) PICKUP(C))), A710)

since, after picking up B, B is no longer clear (the guard of the act [Clear(B) $\rightsquigarrow$ PICKUP(B)] no longer holds). Next, if the agent chooses to do the act PICKUP(C), the expansion for the subsequent iteration will be

SEQUENCE(DoONE((PICKUP(A))), A710)

which will result in performing act PICKUP(A) followed by A710 again. However, this time, none of the guards holds, so the iteration terminates. Figure 7.5 shows the expansions.

7.5.6 Achieve Acts

Performing an act (primitive or complex) is indicated by an INTEND method applied to the instance representing the act. It is also possible to request the agent to achieve a goal (as is done in classical planning systems). A goal is a proposition. The intention to bring about a state where a proposition is believed (or accomplishing a goal) is modeled using the control act ACHIEVE, which is defined below:

Definition 26:

Class: Achieve

Superclass(es): Control Act, Act Term, Conceptual Entity Term

Slots: All instances of this class have the following slots defined:

OBJECT: Can be any object that is a Proposition Term.

Methods: If o is an instance of this class then

ACTION(o): Shadowed to a NULL method.

OBJECT(o): Returns the contents of the OBJECT slot.

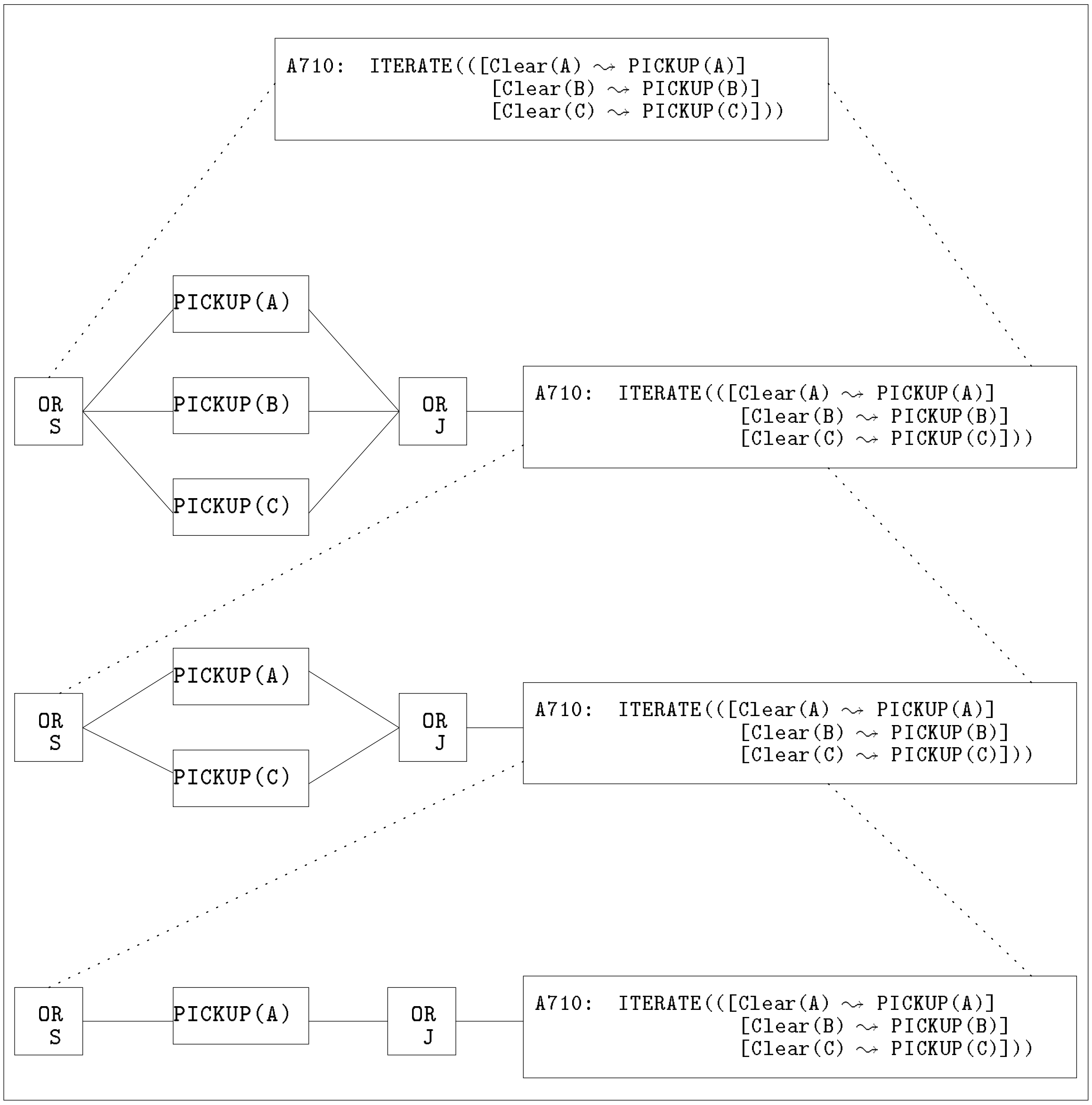

Figure 7.5: An example of an iterative act

Syntax: If p is the value of OBJECT(o), then we write o as

$$\mathsf{ACHIEVE(p)}$$

Description: This is the class of achieve acts. This representation is based on the design of the achieve control act of the SNePS acting system [Kumar and Shapiro 1991b].

Achieving a goal involves deriving a plan that, after it is executed, will result in a situation where the proposition that is the goal is believed. If the goal proposition is already believed in the situation where the achieve act is intended, there is no need to do anything. The INTEND method for instances of this class is specified below (where the act α is of the form ACHIEVE(ϕ)):

Method
    INTEND(α : Achieve)
is
    let GOAL ← OBJECT(α)

    if NOT BELIEVE?(GOAL) then
        DECOMPOSITIONS ← BELIEVE?(PlanGoal(?x,GOAL))
        if DECOMPOSITIONS then
            let PLANS ← NIL
            foreach $\delta \in$ DECOMPOSITIONS loop
                PLANS ← PLANS ∪ PLAN(δ)
            endloop
            INTEND(DoONE(PLANS))
        else
            Use a classical planning algorithm to generate plan(s)
                and intend to do one of those plans as above
        endif
    endif
end INTEND

Thus, one can write plans for achieving goals as well as plans for decomposing a complex act. The domain normally determines the kinds of plans required (i.e., goal-based, act-decomposition based, or both). However, there are domains (even in the blocksworld, for instance), where it may be hard to distinguish between something that characterizes a state and something that expresses an act. For example, "Clear A" could be interpreted as a command to perform the act of clearing A or a goal to achieve a state in which A is clear. This is a modeling issue and needs further exploration. In any case, if required, both the above interpretations can be modeled in our paradigm. As earlier, we are refraining from committing ourselves to any specific planning algorithm but provide an appropriate place where it can be placed and incorporated into the architecture.

7.5.7 Qualifier Acts

We have defined acts as actions performed on specific objects. At the time an act is intended, all its objects are already specified. Using reasoning rules, it is possible to represent general plan derivations. For example

B722! : $\forall$x$\forall$y$\forall$z[((Isa(x, BLOCK) $\wedge$ Isa(y, BLOCK)) $\wedge$ Isa(z, BLOCK))

$\rightarrow$ Plan(SEQUENCE(PUT(z, TABLE), SEQUENCE(PUT(y, z), PUT(x, y))), PILE(x, y, z))]

is a rule for representing a general plan for piling any three blocks. At the time a specific act, say PILE(A,B,C), is intended, since it is a complex act, it is decomposed into the specific plan

SEQUENCE(PUT(C, TABLE), SEQUENCE(PUT(B, C), PUT(A, B)))

which is then intended. Thus, given a specific act (i.e., an act with all its objects specified) one can derive a specific plan via inference. There are, however, situations where plans may contain acts whose specific objects will not be known at the time the plan is intended. It is also possible to ask the agent to perform acts whose objects are not completely specified. The agent should be able to represent and carry out such acts. For example, the requests, 'Pick up a block', 'Pick up a clear block', 'Pick up all blocks that are clear', 'Pick up all red blocks that are clear', etc., are perfectly reasonable and the agent should be able to carry them out. Notice that none of these sentences have a straightforward representation in the ontology defined so far. In this section, we will define act representations that are capable of representing such sentences as well as define INTEND methods that will enable the agent to perform such acts.

Notice that, in all the sentences above, the agent has to "decide" what the objects of each act will be. There are two choices, given the current model—the agent can "choose" objects from the set of objects that it has beliefs about, or it may perform some perceptual task to identify specific objects. Normally, the former is the first viable option; if it fails to identify objects, we use the latter. Acts whose objects are not specified, but are described, are called *qualifier acts*. We will define two types of qualifier acts—WITHSOME and WITHALL—adequate representational as well as behavioral properties described above.

7.5.8 Single-Object Qualifiers

Sentences that qualify a single entity to be the object of an act are represented using WITHSOME acts and are defined as follows:

Definition 27:

Class: WithSome

Superclass(es): Control Act, Act Term, Conceptual Entity Term

Slots: All instances of this class have the following slots defined:

VARS: Can be a list of free variables.

SUCHTHAT: Can be any object that is a Proposition Term.

DO: Can be any Act Term.

Methods: If o is an instance of this class then

VARS(o): Returns the contents of the VARS slot.

SUCHTHAT(o): Returns the contents of the SUCHTHAT slot.

DO(o): Returns the contents of the DO slot.

ACTION(o): Shadowed to a NULL method.

Syntax: Instances of this class are written as follows:

WITHSOME⟨VARS(o)⟩[⟨SUCHTHAT(o)⟩]⟨DO(o)⟩

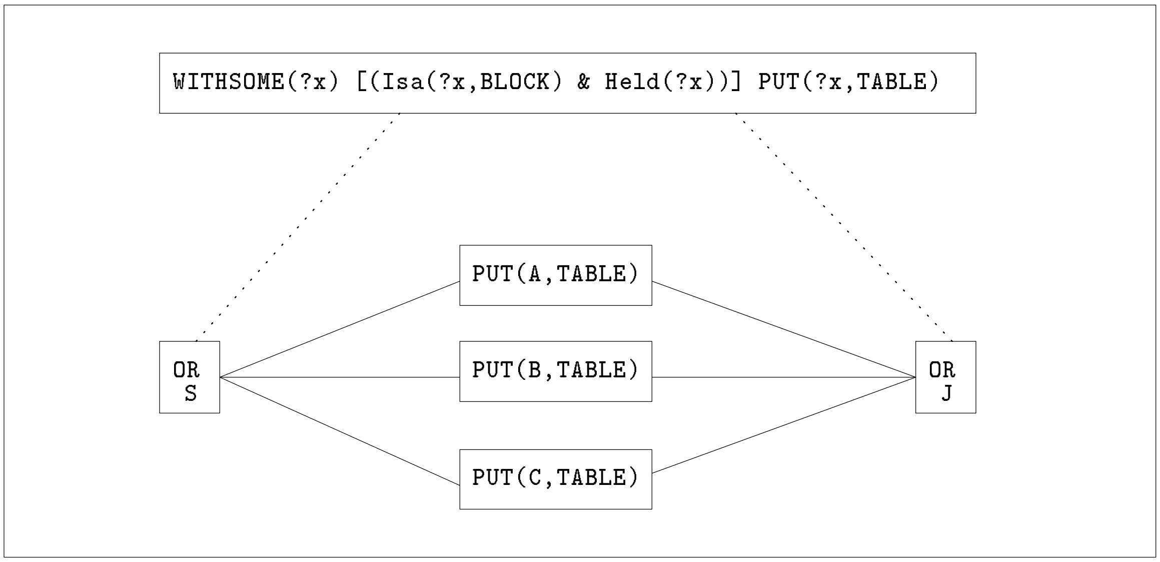

Figure 7.6: Expansion of a WITHSOME act representing the sentence, 'Put on the table some block that is being held' in a situation where it is believed that A, B, and C are blocks and they are all held.

Description: The class of single object qualifier acts.

Acts that are instances of this class can have any number of variables in the VARS slot as long as all the variables are used somewhere in the proposition specified in the SUCHTHAT slot. For example, the sentence 'Pick up a block' can be analyzed as an act, PICKUP, whose object is any entity that is a block, and represented as

$$\mathsf{WITHSOME(?x)\ [Isa(?x, BLOCK)]\ PICKUP(?x)}$$

Similarly, the sentence 'Pick up a clear block' can be represented as the act

$$\mathsf{WITHSOME(?x)[(Isa(?x, BLOCK)} \wedge \mathsf{Clear(?x))]\ PICKUP(?x)}$$

Also, consider the sentence, 'Pick up a clear block that is on another block', which requires two variables (one for the object of the pick up action and the other for identifying the block under the object). Thus,

$$\mathsf{WITHSOME(?x\ ?y)}\\ \mathsf{[(((Isa(?x, BLOCK)} \wedge \mathsf{Isa(?y, BLOCK))} \wedge \mathsf{On(?x, ?y))} \wedge \mathsf{Clear(?x))]\ PICKUP(?x)}$$

would be the resulting representation. McDermott [McDermott 1992], in RPL (Robot Plan Language), calls the qualifiers *effective designators*. Identifying a specific object that fulfills the description can be done by finding out if it is possible to retrieve or derive the beliefs from the belief space and then picking up any one of the possible responses to identify the specific object. This is specified in the INTEND method below (where α is of the form $\mathsf{WITHSOME}(\nu)[\phi]\beta$):

Method
    INTEND(α : WithSome)
is
        let DESIGNATOR $\leftarrow$ *SUCHTHAT(α)*
        let ACT $\leftarrow$ *DO(α)*
        let POSSIBLE-ACTS $\leftarrow \{\ \}$

        let ANSWERS $\leftarrow$ *BELIEVE?(DESIGNATOR)*
        foreach $\phi \in$ *ANSWERS loop*
            let $\sigma \leftarrow$ *UNIFY(ϕ, DESIGNATOR)*
            POSSIBLE-ACTS $\leftarrow$ *POSSIBLE-ACTS* $\cup$ *ACT* σ
        endloop
        INTEND(DoONE(POSSIBLE-ACTS))
end INTEND

It is possible for the *BELIEVE?(DESIGNATOR)* query to chain through a DoWhen transformer whose act component is a perceptual act that may identify a specific object. Rather than chosing a specific object as part of the INTEND method, we transform the intention to perform the WITHSOME act into an intention to perform the act, DoONE, of doing one of the possible acts. Figure 7.6 pictorially shows the expansion of a WITHSOME act into a DoONE act. Thus, carrying out any one of the objects of the DoONE act will fulfill the intention to perform the WITHSOME act.

7.5.9 Multiple-Object Qualifiers

Similarly, sentences 'Pick up all blocks that are clear', 'Pick up all red blocks that are clear', etc., that specify an act to be performed on all its known designators can be represented by a WithAll act, which is defined below:

Definition 28:

Class: WithAll

Superclass(es): Control Act, Act Term, Conceptual Entity Term

Slots: All instances of this class have the following slots defined:

VARS: Can be a list of free variables.

SUCHTHAT: Can be any object that is a belief.

DO: Can be any act.

Methods: If o is an instance of this class then

VARS(o): Returns the contents of the VARS slot.

SUCHTHAT(o): Returns the contents of the SUCHTHAT slot.

DO(o): Returns the contents of the DO slot.

ACTION(o): Shadowed to a NULL method.

Syntax: Instances of this class are written as follows:

$$\mathsf{WITHALL}\langle\mathsf{VARS(o)}\rangle[\langle\mathsf{SUCHTHAT(o)}\rangle]\langle\mathsf{DO(o)}\rangle$$

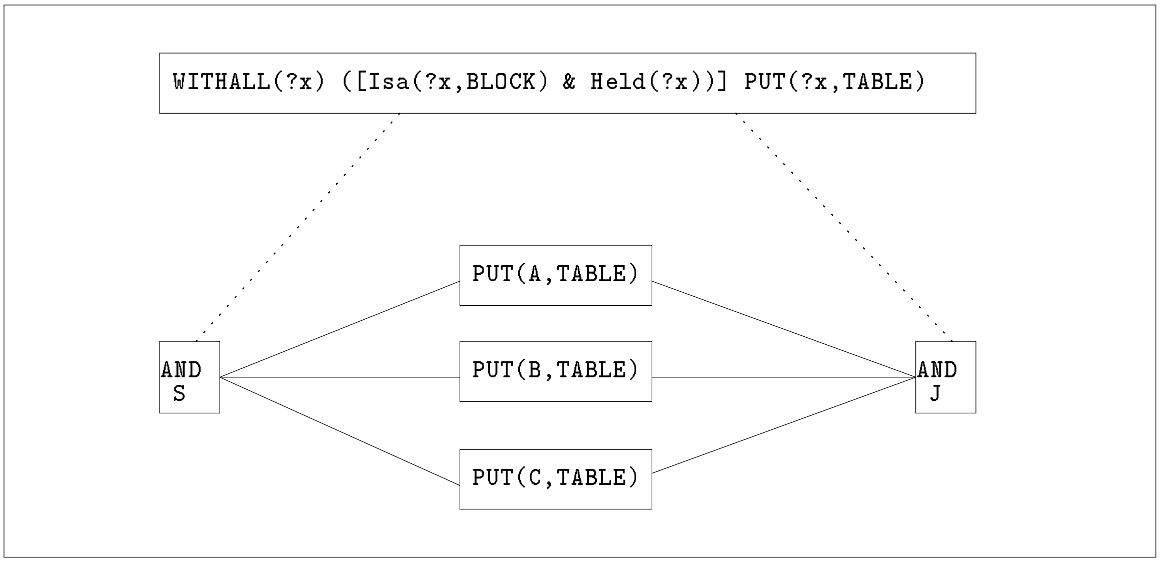

Figure 7.7: Expansion of a WITHALL act representing the sentence, “Put on the table all blocks that are being held” in a situation where it is believed that A, B, and C are blocks and they are the only ones that are being held.

Description: This is the class of multiple-object qualifier acts.

For example, the following acts

$$\mathsf{WITHALL}(?x)[(\mathsf{Isa}(?x, \mathsf{BLOCK})\ \wedge\ \mathsf{Clear}(?x))]\ \mathsf{PICKUP}(?x)$$

$$\mathsf{WITHALL}(?x)[((\mathsf{Isa}(?x, \mathsf{BLOCK})\ \wedge\ \mathsf{Clear}(?x))\ \wedge\ \mathsf{Color}(?x, \mathsf{RED}))]\ \mathsf{PICKUP}(?x)$$

represent the two sentences “Pick up all blocks that are clear” and “Pick up all red blocks that are clear”, respectively. The INTEND method for objects of this class is very similar to the one for WITH-ONE above, except that the resulting act is a DoALL act rather than a DoONE, as specified below (where α is of the form $\mathsf{WITHALL}(\nu)[\phi]\beta$):

```
Method
    INTEND(α : WithAll)
is
        let DESIGNATOR ← SUCHTHAT(α)
        let ACT ← DO(α)
        let POSSIBLE-ACTS ← NIL

        let ANSWERS ← BELIEVE?(DESIGNATOR)
        foreach φ ∈ ANSWERS loop
            let σ ← UNIFY(φ, DESIGNATOR)
            POSSIBLE-ACTS ← POSSIBLE-ACTS ∪ ACT σ
        endloop   INTEND(DoALL(POSSIBLE-ACTS))
end INTEND
```

Figure 7.7 shows the expansion of a WITHALL act into a DoALL act. Thus, carrying out the DoALL act will fulfill the intention to perform the WITHALL act.

One of the important features RPL[3] [McDermott 1992, McDermott 1991] introduces is the treatment of *bound variables*. These are plan variables that at best can be bound to descriptions of objects. The descriptions should be sufficient to manipulate and identify specific objects to be operated upon by a plan's actions. The idea of an effective designator for a plan variable is introduced to refer to objects that can only be bound by performing some sensory activity. Our representations of qualifier acts provide a more general representation of effective designators. In addition to solving the concerns addressed by RPL, the qualifier acts enable representations of natural language utterances that describe generic acts. The presence of a unified underlying theory of acting and inference enables a representation (and an implementation) that tries to identify described instances by inference (or introspection) as well as sensory activity. No special mechanism needs to be provided for modeling sensory activity, since backward chaining of inference can lead to sensory acts via the DoWhen transformers.

This is also in response to the criticisms offered with regards to the nature of symbolic representations in an acting system [Agre and Chapman 1987, Agre and Chapman 1990]. Agre and Chapman argue in favor of *indexical-functional* (or *diectic*) representations for individuating objects of activity—they are indexically qualified relative to the agent and they are functionally qualified by the agent's purposes. Qualifier acts, in conjunction with DoWhen transformers, provide an adequate representational formalism as well as a mechanism for individuating objects by means of the agent's beliefs as well as its environment.

7.6 Discussion

This chapter has fleshed out the subclass hierarchy of acts in the OK formalism. We have identified different types of acts—physical, mental, and control. Control acts provide a formalism for representing plans in the OK formalism. The INTEND methods for control acts were also specified. The control acts provide a rich vocabulary for structuring plans. One can represent, reason about, as well as carry out sequential, conditional, as well as iterative plans in the OK formalism. Additionally, qualifier acts were introduced so as to represent natural language utterances about generic acts as well as acts whose objects could only be described, rather than specified. The OK rational engine, by way of its methods (INTEND, BELIEVE?, and BELIEVE!) uniformly accomplishes the tasks of inference and acting (see Figure 7.8).

The last four chapters have described the various constituents of the OK BDI architecture. In the next chapter, we will examine a concurrent design of the OK rational engine. Subsequently, we will present the SNePS instantiation of the OK architecture. Several examples of various features of the OK architecture will also be presented.

[3] Reactive Plan Language

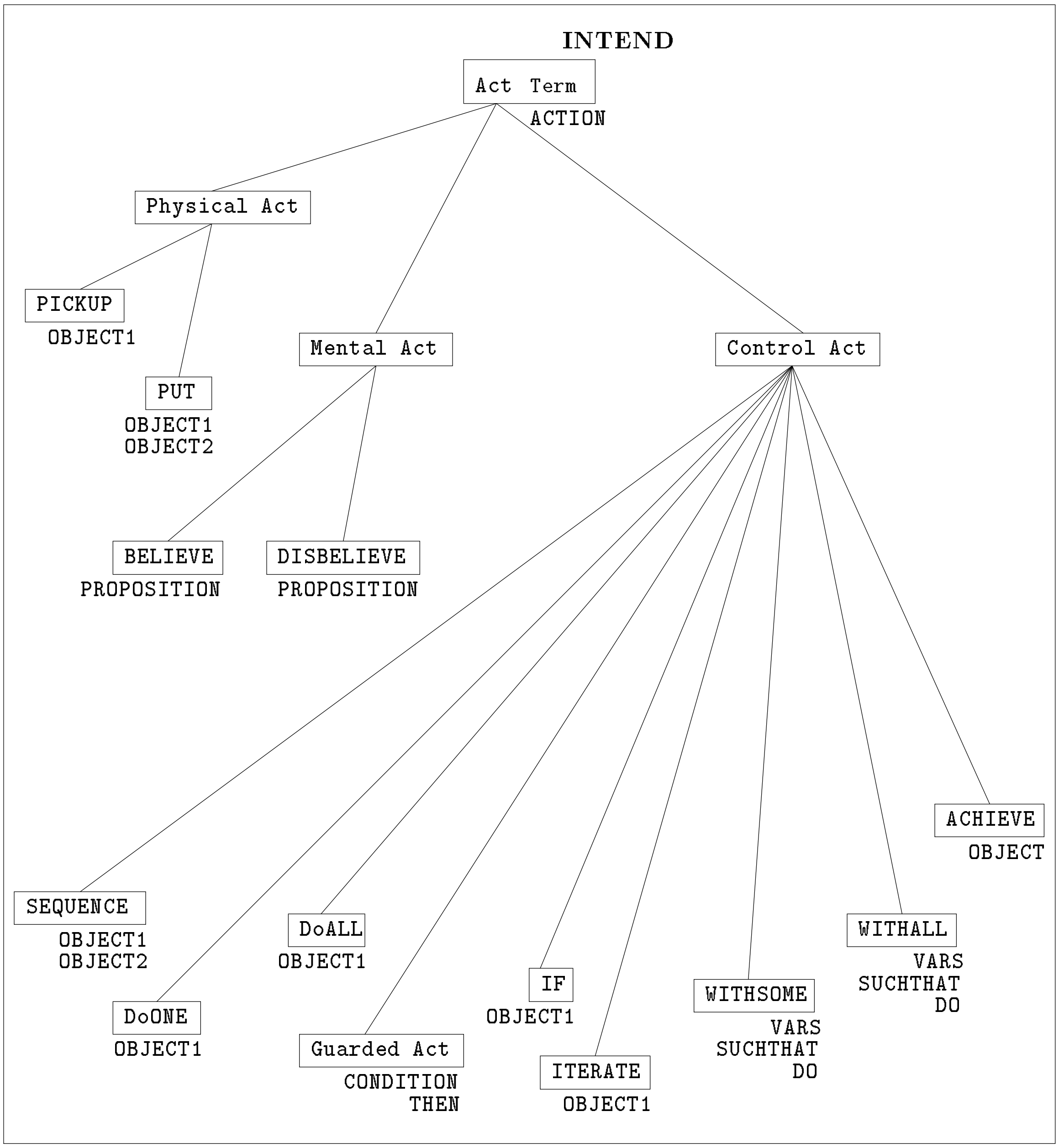

Figure 7.8: The subclass hierarchy of acts

Chapter 8

Towards Concurrency

This chapter concerns itself with some implementational issues in the light of advances in the field of programming paradigms. We will explore the feasibility of implementing the OK architecture using an emerging programming paradigm—concurrent object-oriented programming. Since the paradigm itself is not as yet suitable for large-scale software development, we will also present a simplified, quasi-concurrent computational model that can be employed. The following chapters will present an actual implementation using the quasi-concurrent model.

8.1 Concurrent Object-Oriented Programming

Concurrent object-oriented programming (COOP) is a relatively new and rapidly growing paradigm. Since the field is still in its infancy, there are several approaches being pursued in order to attain concurrency in object-oriented programs [Yonezawa and Tokoro 1987a]. [Yonezawa and Tokoro 1987b] identify four aspects of COOP that present issues and challenges that need to be addressed—computation models, languages and systems, architectures, and applications. The last one, applications, is interesting since it indicates that it has been established that applications themselves have interesting properties that need to be examined in order to make them concurrent. Thus, it is already established that some of the issues in the development of the COOP paradigm are posed by applications themselves. In this chapter, we will concentrate on identifying issues related to a concurrent implementation of the OK BDI architecture as a viable application for the COOP paradigm.

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 1991] goes so far as to proclaim that "every object-oriented programming language should be concurrent in nature." Among the existing paradigms of COOP, researchers differentiate between two approaches—object-*based* concurrent programming and object-*oriented* concurrent programming. The former applies to systems (languages) that support objects as a language feature. However, they lack the property of inheritance (or delegation, which is a generalization of inheritance [Wegner 1987]), the presence of which makes them object-oriented [Chin and Chanson 1991]. Any object-oriented (or object-based) system can be made concurrent by providing three additional features [Nelson 1991]:

- processes corresponding to objects,
- synchronizing facilities, and
- message passing.

Among the existing object languages, Ada [ANSI, Inc. 1983], Modula2 [Wirth 1988], Actors [Agha 1986a, Agha 1986b, Agha and Hewitt 1987, Agha 1990], CLU [Liskov *et al.* 1977] are considered object-based, while C++ [Stroustrup 1987], CLOS [Steele 1990], Smalltalk [Goldberg and Robson 1983] are object-oriented. Of these, Ada, Smalltalk, and Actor languages can be considered concurrent. However, as explained in [Wegner 1987], the "kind" of concurrency is determined by the nature of processes, which can be

- sequential (as in Ada),
- quasi-concurrent (as in Smalltalk), or
- concurrent (as in Actor-based languages).

There are also issues relating to inter- as well as intra-level concurrency of processes. These issues pose design-level *vs.* implementation-level trade-offs. Decisions at these levels can affect each other both ways. In our work, we identify some of these issues (both at the design and implementation levels). While we have already committed ourselves to an object-oriented design (i.e., an object-based system that also utilizes inheritance), we may employ any of the possible approaches to concurrency at the implementation level.

8.2 Methods as Messages

As indicated above, methods of the rational engine correspond to messages in the concurrent model. For example, the intention to perform the act of putting A on B would be translated into the following method invocation:

$$\mathsf{INTEND}(\mathsf{PUT}(\mathsf{A},\mathsf{B}))$$

Instead, we would like to view it as a message sent to the object corresponding to the act, PUT(A, B), i.e.,

Send message INTEND to the object A11:PUT(A, B)

In order to process the message, we will assume that objects receiving messages have corresponding *processes* that possess the computational power and the communication abilities required. Specific details of the nature of processes are the same as the definitions of the respective methods. However, in pursuing the "method as a message" philosophy, we have to relinquish the "method as a function" model and hence address the issue of synchronization. For instance, consider the definition of INTEND in Section 6.3, which is reproduced below:

Method
    INTEND(α : Act Term)
is

```
        PRECONDS ← BELIEVE?(PreconditionAct(?x,α))
        for each p ∈ PRECONDS loop
            TRANSFORM?(p)
        endloop

        EFFECTS ← BELIEVE?(ActEffect(α,?x))

        if BELIEVE?(Isa(α,PRIMITIVE)) then
            execute the actuator for α
            for each e ∈ EFFECTS loop
                TRANSFORM!(e)
            endloop
        else
            PLANS ← BELIEVE?(PlanAct(?x,α))
            if PLANS then
                PLANS ← { PLAN(p_i) | ∀p_i ∈ PLANS}
                INTEND(DoONE(PLANS))
                for each e ∈ EFFECTS loop
                    TRANSFORM!(e)
                endloop
            else
                use a planning algorithm to derive a plan if possible
            endif
        endif
end INTEND
```

For the invocation of the INTEND method shown above, the first statement of the INTEND method,

PRECONDS ← *BELIEVE?(PreconditionAct(?x,*α*))*

translates to:

PRECONDS ← *BELIEVE?(PreconditionAct(?x,PUT(A, B)))*

which, in the concurrent model, would appear as

Send message BELIEVE? to the object B143:PreconditionAct(?x,PUT(A, B)))

so that *all* the preconditions of the act PUT(A, B) can be deduced in parallel. The collection of the results of this inference so that the next step (confirming that all the preconditions are believed) can proceed requires a synchronization mechanism. One might think it desirable to confirm all preconditions asynchronously (i.e., as they are derived). However, remember that confirming preconditions may lead to further intentions by way of precondition-act transformations. Confirming preconditions synchronously gives a handle on ordering the achieving of preconditions in case they are mutually conflicting. While it is desirable to proceed synchronously in order to confirm preconditions of actions, confirming the antecedents of a rule can be carried out asynchronously. This "eager-beaver" mode of confirming antecedents is actually beneficial [Shubin 1981] and will be adopted in our implementation model. In what follows, we first describe the kinds of messages, the nature of the processes, and the synchronization mechanism required in order to achieve concurrency.

8.3 Concurrency in the OK Rational Engine

In a concurrent implementation of the OK rational engine, acting and inference is carried out by object instances sending and receiving messages to and from each other. As has already been shown in the example above, the messages that objects send or receive correspond directly to the methods of the rational engine. For each specific message, we explicitly identify the sender and the receiver of the message. For example, when the following message

Send message BELIEVE? to the object B143:PreconditionAct(?x,PUT(A, B)))

is sent in the process of carrying out the intention to perform the act denoted by A11, then A11 is the sender and B143 is the receiver of the message BELIEVE?. The name of the sender is included as a part of the message itself, so that the receiver, if required, can appropriately respond to the sender with an answer (this makes it equivalent to a functional invocation). In order to achieve concurrency, we will specify that at any instant there could be several messages being sent or received. For instance, if PUT(A, B) has several preconditions, all of them could be derived in parallel by sending BELIEVE? messages to all the objects that unify with the object PreconditionAct(?x, PUT(A, B)). If one takes a close look at the specification of the various methods, all loops could then be executed concurrently, thus generating several messages at the same time.

Figure 8.1 shows an event diagram depicting concurrent derivation of the preconditions of the act PUT(A, B). Event diagrams show an abstract view of computation in a concurrent system [Agha 1990]. A vertical line under an object instance represents a thread of computation. An arrow going out, in the left-to-right direction, of a vertical line indicates a message sent by the object. A left-to-right arrow indicates an arrival of a message. A right-to-left arrow coming into a vertical line indicates a response from an earlier message. Placement of the arrows on a vertical line indicates the linear ordering of communications to or from that object. In Figure 8.1, all left-to-right arrows not labeled represent BELIEVE? messages. At instant t0, an INTEND message is sent to the object PUT(A, B). At t1, PUT(A, B) sends a BELIEVE? message to the object PreconditionAct(?x, PUT(A, B)) (see the description of the INTEND method above). At instant t2, the two objects that successfully unify with PreconditionAct(?x, PUT(A, B)) receive a BELIEVE? message from it. The two objects (see the definition of the act PUT in Section 7.2.1) happen to be consequents of the AntCq transformers, B703 and B702. Thus, they receive TRANSFORM? messages (at instances t3 and t4). At instances t5 and t6, the antecedents receive a BELIEVE? message from their respective transformers. If the propositions Isa(A, BLOCK) and Isa(B, SUPPORT) are asserted, then, at instances t9 and t10, the messages sent out at t5 and t6 receive responses, leading to the firing of the two transformers, resulting in responses to PreconditionAct(?x, PUT(A, B)), which get accumulated (i.e., the list (PreconditionAct(Held(A), PUT(A, B)) PreconditionAct(Clear(B), PUT(A, B)))) and sent to PUT(A, B) as a response to the message that was sent at t1. Thus, at instant t15, the object PUT(A, B) is ready to proceed with its fulfillment of the INTEND method.

From the description above, one can see that the model of computation depicted is very similar to the Actor model of concurrent object-based computation. Like Actor systems, message passing is employed as a basis of concurrent computation. Object instances denote individual actors. Labels of objects denote their *mail addresses*. The only difference is that the behavior of objects is determined by inherited methods, something that is missing

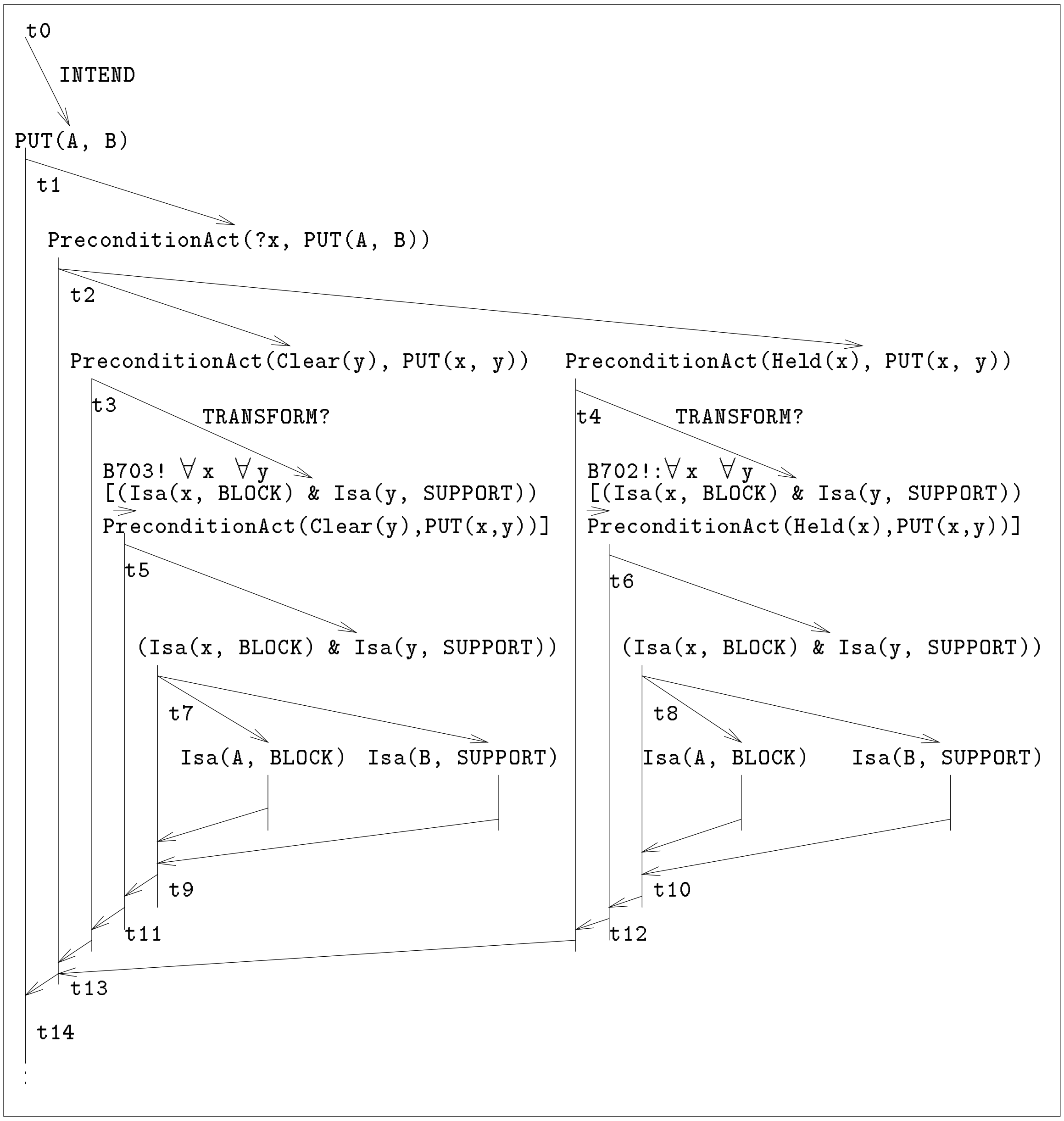

Figure 8.1: An event diagram showing parallel threads of computation, message invocations, and object instances involved in the derivation of the preconditions of the act PUT(A, B).

in Actor systems. This seems to indicate the possibility that the OK architecture could be implemented using an actor language. 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 [Hewitt 1971, Hewitt 1972]. 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 powerful enough primitives and environments to build large software using Actor languages—this area of research is still in its infancy. Also, the focus of researchers working on Actor languages has shifted over the years. They mostly concentrate on (and appropriately so) the linguistic and computational properties of Actor languages. Also, the Actor-view of the OK rational engine presented above arises out of considerations involved in making some representational as well as some behavioral commitments. Only the latter are similar (or conform) to the Actor view of computation. Transformers, an integral part of the OK architecture, help capture the the overall embedded nature of the architecture. For instance, the WhenDo belief-act transformers and the backward chaining DoWhen transformers model reactivity as well as the agent's desire to perform actions in order to acquire beliefs. These, too, form desirable traits of actor systems (in the spirit of open systems). It is the central premise underlying this thesis—the unified view of acting and inference—that makes Actor systems an attractive option for implementing OK architectures. It is our hope that we would be able to explore this avenue in the future. In the meanwhile, we will present a simplified model of concurrency that can be implemented on any existing platform (including a sequential architecture). The following chapters describe an actual implementation.

8.4 A Quasi-Concurrent Model

[Wegner 1987] describes a quasi-concurrent process as a process that has at most one active thread of control. Quasi-concurrent processes allow threads of control to be suspended while waiting for a condition to be fulfilled. The suspended thread can be resumed when the condition is satisfied. This solves the synchronization problem. We have already shown a concurrent model of computation that can be used to implement the OK rational engine. In a concurrent model, there can be several active threads. In the example shown in Figure 8.1, the processing of the BELIEVE? message by the transformers B703 and B703 can be carried out concurrently. Thus, at any instant, in a quasi-concurrent model, there can be contending active processes. We will resolve this situation by employing an interleaved execution model, i.e., treating active processes like coroutines.

In AI systems, especially like the OK BDI architecture defined in this work, objects can be viewed as knowledge sources with computing power and communication capability. The amount of computing power and communication abilities attributed to objects is defined by the "role" of the object in the reasoning/acting process. Since the methods that comprise the rational engine mainly deal with objects that are beliefs and acts, we will focus the design of the concurrent rational engine on these methods only (i.e., BELIEVE?, BELIEVE!, INTEND, TRANSFORM?, and TRANSFORM!). Thus, objects that are instances of the classes Individual Term and Variable Term do not send or receive messages. When variables are bound to propositions or acts their bindings may get involved in message passing. We will also make another simplifying assumption—our concurrent model only strives for inter-level concurrency. I.e., scripts of processing individual messages themselves will be sequential. This gives us a handle on treating some methods (mainly slot accessors) as function calls while treating the others as messages. We should point out that we are making these methodological simplifications at this stage mainly because of the absence of an existing programming-language implementation that provides inter- as well as intra-level concurrency. We will return to this discussion later.

8.4.1 MULTI Revisited

MULTI [McKay and Shapiro 1980a, McKay and Shapiro 1980b] is a multiprocessing package that enables a LISP program to define and use processes. It implements a quasi-concurrent, interleaved model of computation as described above. This is especially desirable for inference. For example, one can carry out an interleaved deduction of the formula $(\phi_1 \vee \phi_2 \vee \ldots \vee \phi_n)$, where ϕ_i are propositions, by trying to show each of the ϕ_i's in parallel and terminating the proofs of all the other disjoints as soon as any one is proved. MULTI was designed specifically as a computational model for carrying out inference in SNIP—the SNePS Inference Package [Hull 1986, Ferreira *et al.* 1989]. In what follows, we will first give an abstract overview of the MULTI paradigm. Then we will describe how the OK rational engine can be implemented in it.

In MULTI, a *process* is an executable entity consisting of an *action* (representing the method specification) and a set of *registers* and *bindings* (representing the state of the computation). A process is created whenever an object instance receives a message (i.e., the process becomes active). All active processes are scheduled on *process queues*. As we will see in a little bit, there can be several process queues organized by a priority scheme. The main control structure of MULTI acts as a dispatcher of active processes to processors. In the case of a single-processor architecture, this amounts to dispatching an active process to the processor. The following describes the control loop of MULTI:

while there is a non-empty queue loop
    choose the highest priority non-empty queue
    remove a process from it
    dispatch the process to a processor
endloop

It has been shown [McKay and Shapiro 1980a, McKay and Shapiro 1980b] that the simple control paradigm of MULTI shown above can effectively be used to model several control structures—recursion, backtracking, coroutines, generators, etc. It has also been shown that, using a supplier-producer-consumer model, this control strategy is amenable to the implementation of forward, backward, as well as bi-directional inference [Shubin 1981]. Let us identify specific components (i.e., messages, and queues) of the OK rational engine that will enable us to use MULTI as a computational model.

8.4.2 Messages and Processes

The following is a description of the methods of the OK rational engine viewed as messages:

BELIEVE?(ϕ, σ): where ϕ is a proposition. This message, also called a *request*, denotes that the sender is asking (requesting) the object instance ϕ about its assertional status, given the substitution σ.

BELIEVE!(ϕ, σ): where ϕ is a proposition. This message, also called a *report*, denotes that the sender is confirming to ϕ that ϕ is asserted, given the substitution σ.

TRANSFORM?(ϕ, σ): where ϕ is a transformer. This message denotes that the sender is asking the transformer to attempt a backward chaining transformation, given the substitution σ.

TRANSFORM!(ϕ, σ): where ϕ is a transformer. This message denotes that the sender is asking the transformer to attempt a forward chaining transformation, given the substitution σ.

INTEND(α): where α is an act. This message denotes that the sender has asked α to be performed.

A process is created for each message sent to an object. The method descriptions are the same as given in Chapter 6, except that all loops in the scripts correspond to a concurrent broadcast of the message to all entities of the iteration (as shown in Figure 8.1). In the OK architecture, all instances of the classes Proposition and Act have associated processes.

8.4.3 Queues

Sending a message to an object in the MULTI paradigm corresponds to creating an active process, assigning it a priority, and scheduling it on an appropriate MULTI queue. If one employs an elaborate priority scheme, only one process queue will suffice. Alternatively, if the range of priorities is not very high, one can employ as many queues as there are priority levels. In the latter case, the scheduler has a simple job to do—find out the priority of the process and schedule it accordingly on the appropriate queue. Having different queues for each priority level also enables a different scheduling and dispatching strategy (i.e., FIFO, LIFO, etc) for each queue. In the rational engine, we recognize that all inferences are to be treated equally; i.e., there is no way for the control strategy to decide if certain chains of inference are more important than others. In order to utilize the concurrent, interleaved strategy described above effectively, we recognize that responses from processes should be given a higher priority. Thus, a process receiving a response should be preferred over one that is simply trying out a viable chain. This is very similar to the idea of *futures* in the Actor model. Acts receiving INTEND messages also have to be treated a little differently. We have already seen that an intention to carry out an action may lead to other INTEND messages to acts representing the achieving of preconditions which have to be fulfilled before the act itself can be performed. Thus, a LIFO scheduling strategy should be employed for act processes. The act processes should be treated at the lowest priority. This is because the INTEND method itself employs inferences that have to be carried out as a part of performing the action. It implicitly models the "think before you act" strategy. Thus, in the OK rational engine, we recognize three process queues:

- a low priority, FIFO, inference queue for scheduling all processes activated by sending requests,
- a high priority, FIFO, inference queue for scheduling processes that receive responses, and

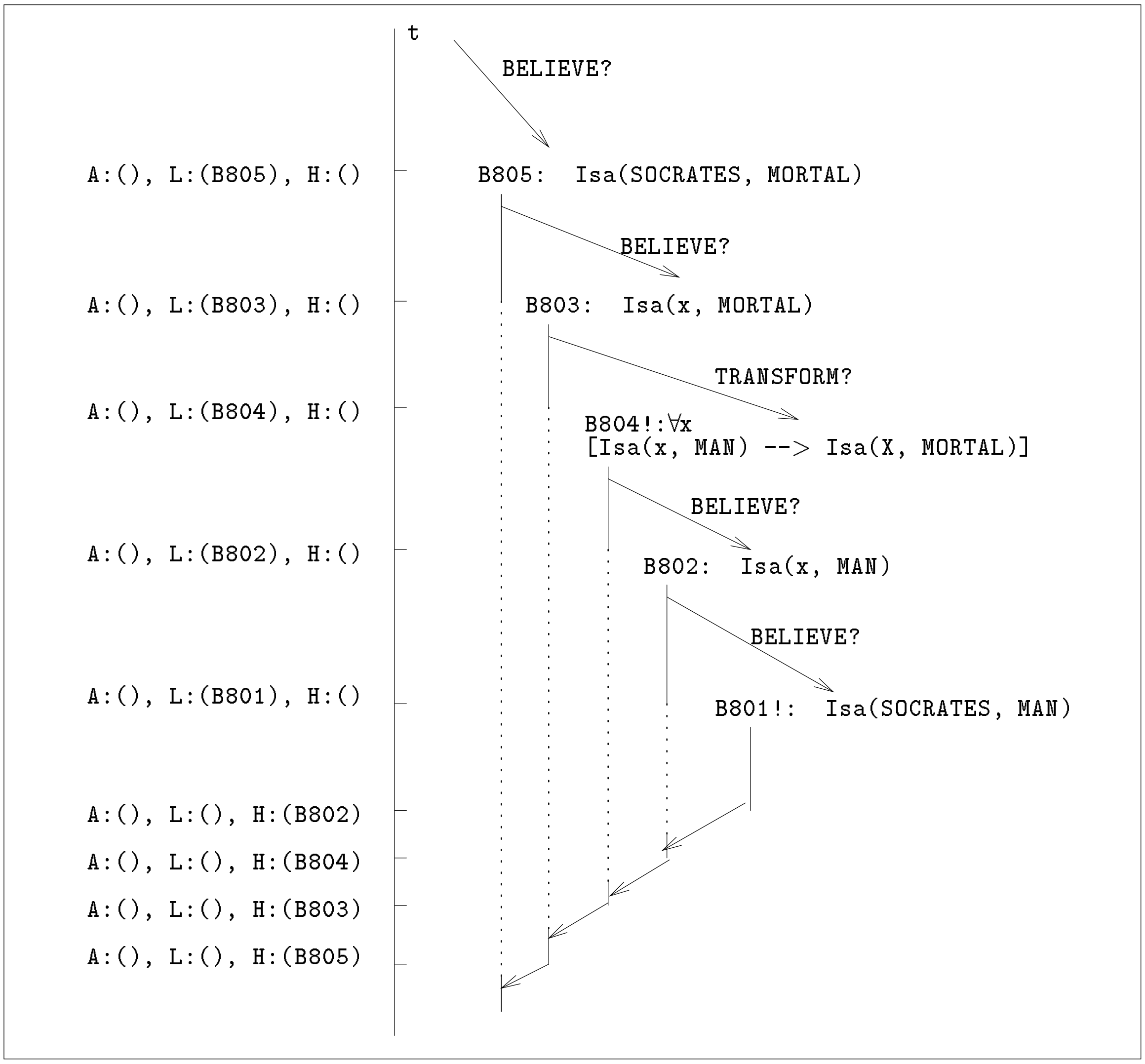

Figure 8.2: An event diagram showing a quasi-concurrent backward chaining inference.

- a lowest priority, LIFO, act queue for scheduling act processes receiving INTEND messages.

We will refer to these as L (low-priority), H (high-priority), and A (act) queues, respectively.

Let us present an example to illustrate a backward chaining inference using the above scheme. Assume that the agent's belief space consists of the following asserted propositions:

$$\mathsf{B801!} : \mathsf{Isa}(\mathsf{SOCRATES}, \mathsf{MAN})$$

$$\mathsf{B804!} : \forall \mathsf{x}[\mathsf{Isa}(\mathsf{x}, \mathsf{MAN}) \rightarrow \mathsf{Isa}(\mathsf{x}, \mathsf{MORTAL})]$$

representing the propositions, "Socrates is a man" and "all men are mortal". A backward chaining inference will be initiated by the following message:

Send message BELIEVE? to B805: Isa(SOCRATES, MORTAL)

This will result in a process being scheduled on the low-priority queue. The following shows the status of the MULTI queues at this instant:

$$\mathsf{A} : (), \mathsf{L} : (\mathsf{B805}), \mathsf{H} : ()$$

where B805 is actually a process instance of the object. Since that is the only active process, it is dispatched, resulting in the executing of the BELIEVE? script—a BELIEVE? message is sent to the object Isa(x, MORTAL) with the substitution set {SOCRATES/x}. The process B805 becomes suspended as a result, and MULTI is ready to dispatch another process. The queues, at this instant, have the following processes:

$$\mathsf{A} : (), \mathsf{L} : (\mathsf{B803}), \mathsf{H} : ()$$

where B803 is the consequent of B804. B803 then ends up sending a TRANSFORM? message to B804 and becoming inactive, resulting in the queues:

$$\mathsf{A} : (), \mathsf{L} : (\mathsf{B804}), \mathsf{H} : ()$$

B804 then sends a BELIEVE? message to B802, the antecedent of B804, thus resulting in the queues:

$$\mathsf{A} : (), \mathsf{L} : (\mathsf{B802}), \mathsf{H} : ()$$

Next, B802 sends a BELIEVE? message to B801:

$$\mathsf{A} : (), \mathsf{L} : (\mathsf{B801}), \mathsf{H} : ()$$

The process for B801 can terminate by responding to B802:

$$\mathsf{A} : (), \mathsf{L} : (), \mathsf{H} : (\mathsf{B802})$$

I.e., the responses get scheduled on the high-priority queue. B802 is then executed from the point where it had left off, i.e., collecting the response, and sending a response to B804:

$$\mathsf{A} : (), \mathsf{L} : (), \mathsf{H} : (\mathsf{B804})$$

B804 decides that the transformer can be fired; it asserts its instantiated consequents (i.e., B805) and sends a response to B803:

$$\mathsf{A} : (), \mathsf{L} : (), \mathsf{H} : (\mathsf{B803})$$

which in turn sends a response to B805:

$$\mathsf{A} : (), \mathsf{L} : (), \mathsf{H} : (\mathsf{B805})$$

thus completing a backward-chaining inference. The process is depicted pictorially in Figure 8.2 (the dotted lines represent suspended states for processes). Figure 8.3 shows the event diagram for the derivation of preconditions of the act PUT(A, B) shown earlier in Figure 8.1.

8.5 Discussion

In this chapter, we have presented an overview of the concurrent object-oriented programming paradigm. We have shown that the OK architecture is amenable to a concurrent computational model. We also presented a quasi-concurrent computational model of implementation using the MULTI paradigm. This has served two purposes—to show that a quasi-concurrent model is as viable an option as a concurrent model and that MULTI can be considered a simplified model of the Actor paradigm. We presented some examples of inference and acting using the MULTI model. This concludes the definition and an abstract overview of the OK architecture. In the next few chapters, we will present a specific instance of the OK architecture—the SNePS BDI architecture. We will present SNePS representations for beliefs, acts, transformers, etc., and show that they are isomorphic to the OK formalism. We will also show an implementation of SNeRE—the SNePS Rational Engine that uses MULTI. We will present several actual examples and conclude this dissertation by summarizing the main results.

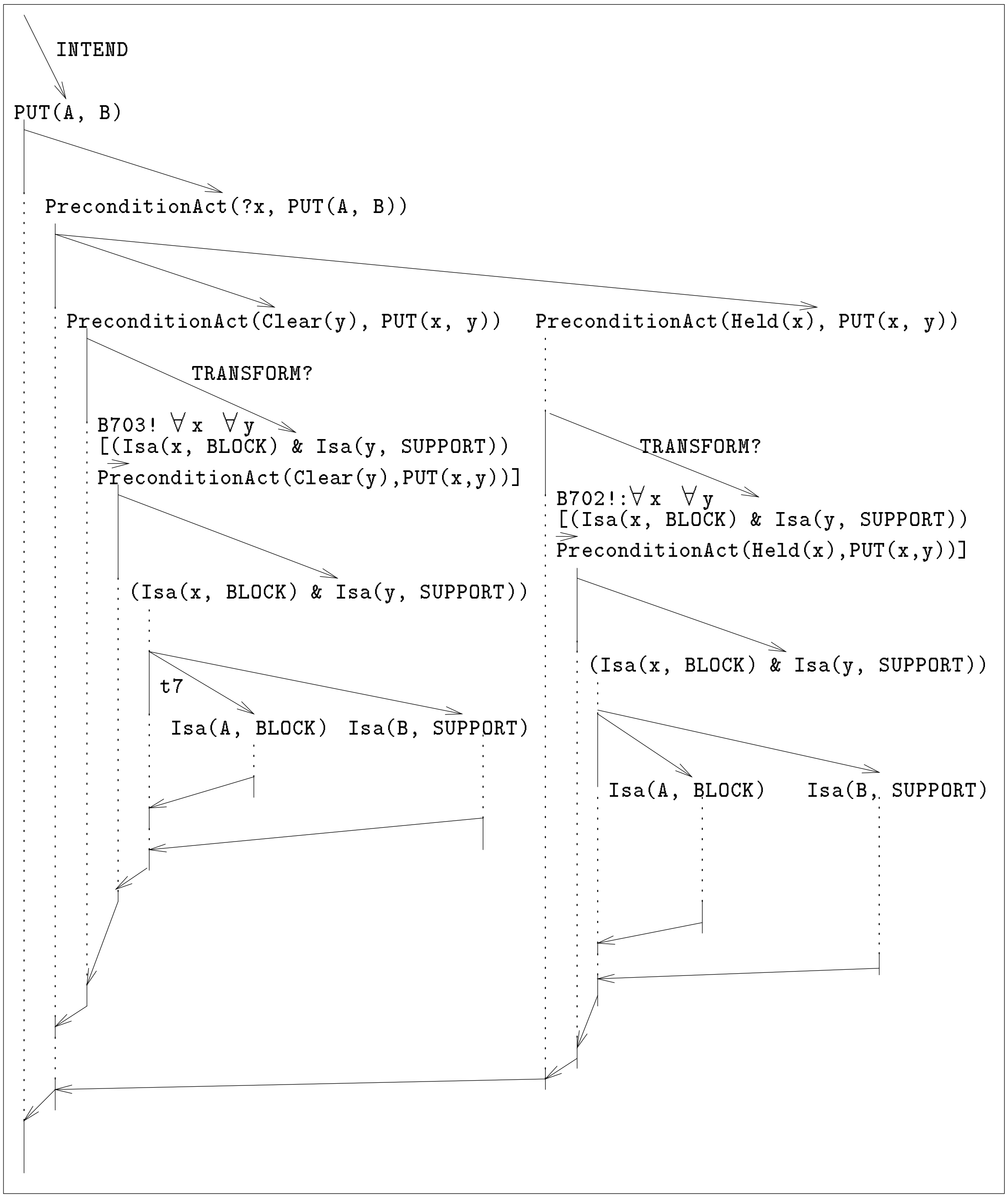

Figure 8.3: An event diagram showing a quasi-concurrent derivation of the preconditions of the act PUT(A, B)

Chapter 9

The SNePS BDI Formalism

We started this work by presenting a critique of the SNePS acting system. That led us to the motivations underlying this work—the relationship between acting and inference. Based on these ideas, we developed a new, object-oriented AI architecture, called the OK BDI architecture. In the specification of the OK representational formalism, we have followed the same representational principles as those underlying SNePS. There were several reasons for this: we strongly felt that the SNePS approach to knowledge representation and reasoning provides a good approach to the modeling of rational cognitive agents; the principles underlying SNePS have been tried and tested over the last twenty years of research [Shapiro and Rapaport 1992]; and we have extended the SNePS formalism and its implementation to demonstrate our results. Thus, the enhanced SNePS architecture is an instantiation of the OK architecture. In this and the following chapters, we will describe the SNePS BDI architecture as an instantiation of the OK BDI architecture. This involves two things—showing that the SNePS formalism is isomorphic to the OK formalism and that the SNePS Rational Engine (SNeRE) is an instantiation of the OK rational engine. In the design of the OK formalism, we have constantly maintained that we have preserved all the underlying principles adopted by SNePS. For that reason, one will notice that it is fairly easy to show the isomorphism between the two formalisms. In fact, given the object-oriented design of the OK formalism, any implementation of the OK architecture can be trivially made to look like a SNePS semantic network.

9.1 The SNePS Formalism

SNePS is a semantic network processing system. All conceptual entities are represented in the semantic network by *nodes*. The SNePS formalism identifies four types of nodes:

Base nodes: These are nodes that do not have any arcs emanating from them. A base node represents a basic conceptual entity.

Variable Nodes: These also do not have any arcs emanating from them. They represent arbitrary individuals.

Molecular Nodes: These do have labeled-arcs (also called *relations*) emanating from them. They may represent propositions (including rules) or *structured individuals*. A molecular node that represents a proposition may be asserted or unasserted.

Pattern Nodes: These also have arcs emanating from them. They represent arbitrary propositions or arbitrary structured individuals and are similar to open sentences in predicate logic.

Each node in the semantic network is identified by a unique label (written inside the node). Labels for base nodes begin with the letter B followed by a number, e.g., B1, B2,..., etc; variable nodes are labeled similarly beginning with the letter V; molecular-node labels begin with the letter M; and pattern nodes are labeled beginning with a P. The structure of molecular nodes is also formally defined using *wires*, *cables*, and *cablesets* [Morgado 1986]. The following definitions of wires, cables, and cablesets are taken from [Shapiro 1991a]:

- A *wire* is an ordered pair $\langle r, n\rangle$, where r is a SNePS relation, and n is a SNePS node.
- A *cable* is an ordered pair $\langle r, ns\rangle$, where r is a SNePS relation, and ns is a non-empty set of SNePS nodes.
- A *cableset* is a non-empty set of cables, $\{\langle r_1, ns_1\rangle, \ldots, \langle r_n, ns_n\rangle\}$, such that $r_i = r_j \Leftrightarrow i = j$.
- Every cableset is a SNePS *node*.
- A *molecular node* is a cableset.

Next, we will revisit the definitions of the OK formalism and show how each entity in the Ok formalism can be represented in SNePS.

9.2 Conceptual Entities

Every node in a SNePS network is regarded as representing a conceptual entity. Entities represented by direct instances of the class Conceptual Entity Term in the OK formalism (i.e., those labeled E1, E2, ...) are the ones represented by base nodes in SNePS (i.e., they are labeled B1, B2 ...). Thus, there is a one-to-one correspondence between base nodes in SNePS and instances of the class Conceptual Entity Term in the OK formalism.

9.3 Named Individuals

In SNePS, conceptual entities that have a linguistic name are represented as structured individual molecular nodes. Each such node has one arc labeled LEX pointing to the name of the entity. In the past, some SNePS researchers have regarded the names themselves as basic entities (i.e., base nodes) [Shapiro 1979, Shapiro and Rapaport 1992]. This is consistent with the syntactic definition of the network; however, conceptually, all that these nodes represent are the linguistic strings. It has also been suggested that these are simply the *extensions* of the concept represented by the molecular node into the external world [Shapiro and Rapaport 1987]. For example, the node,

$$\mathsf{M1} : \{\langle \mathsf{LEX}, \mathsf{BLOCK}\rangle\}$$

represents the named individual that is referred to in English by the word "block". The same entity in the OK formalism is represented by an instance (say I2) of the class Individual Term and is written as

BLOCK

Thus, to translate an object that is an instance of the class Individual Term to an equivalent SNePS representation, one simply has to create a molecular node that has a LEX arc coming out of it pointing to the node/string that is the value of the NAME slot. The formal definition of the syntax of a SNePS node representing named individuals is defined below:

Definition S1:

Entity: Named Individual

Syntax: Named individuals are represented using structured molecular nodes with the following wire(s) in the cableset:

$\langle \mathsf{LEX}, \eta \rangle$**:** where η is a base node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{LEX}, \eta \rangle\}$$

is an individual expressed in English as η.

Examples: In Figure 9.1, the nodes M1, M2, M4, M6, M8 represent named individuals.

Instances of the class Individual Term can also be depicted as a semantic network. For example, the entity I2 above can be depicted as a semantic network node labeled I2, with the cableset, $\{\langle \mathsf{NAME}, \mathsf{BLOCK} \rangle\}$. Figure 9.2 shows I2 and some other instances, I1, I3, I4, I5, as a semantic network. Thus, the representations of named individuals in SNePS and OK are isomorphic.[1]

9.4 Variables

Variables in SNePS are the same as the variables in the OK formalism. The NAME slot for variables is not present, since they can simply be referred to by their node labels (i.e., V1, V2,...). In SNePS, variables can be universally, existentially, or numerically quantified (see [Shapiro and Group 1989]).

9.5 Propositions

In SNePS, propositions are represented using molecular nodes. The specific proposition represented by a node is defined by the wires emanating from the node. It is easy to translate instances of the class Proposition Term to molecular proposition nodes in SNePS. Each slot defined for the class becomes the label of the arc coming out of the proposition

[1] We are using the term *isomorphism* in its strictest sense. In addition to implementing the OK Formalism using SNePS we are also showing that the OK Formalism can be treated as an object-oriented definition of the SNePS Formalism.

node. The values of the slots become the respective heads of the arcs. For example, the proposition in the OK formalism that is written as

$$\mathsf{B1} : \mathsf{Isa}(\mathsf{A}, \mathsf{BLOCK})$$

is represented in SNePS as the node:

$$\mathsf{M3} : \{\langle \mathsf{MEMBER}, \{\langle \mathsf{LEX}, \mathsf{A}\rangle\}\rangle, \langle \mathsf{CLASS}, \{\langle \mathsf{LEX}, \mathsf{BLOCK}\rangle\}\rangle\}$$

Formally, all propositions of this form are defined as follows:

Definition S2:

Entity: The Isa proposition.

Syntax: All Isa propositions are represented using structured molecular nodes with the following wire(s) in the cableset:

$\langle \mathsf{MEMBER}, \mu\rangle$**:** where μ is a node.

$\langle \mathsf{CLASS}, \sigma\rangle$**:** where σ is a node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{MEMBER}, \mu\rangle, \langle \mathsf{CLASS}, \sigma\rangle\}$$

is a proposition that the entity represented by μ is a member of the class of entities represented by σ.

Examples: In Figure 9.1, the nodes M3 and M5 represent Isa propositions.

When a proposition is asserted (in SNePS), we write an exclamation mark following the label of the node, i.e.

$$\mathsf{M3!} : \{\langle \mathsf{MEMBER}, \{\langle \mathsf{LEX}, \mathsf{A}\rangle\}\rangle, \langle \mathsf{CLASS}, \{\langle \mathsf{LEX}, \mathsf{BLOCK}\rangle\}\rangle\}$$

SNeBR, the SNePS system for Belief Revision [Martins and Shapiro 1988], is an integral part of the SNePS system. It records TMS-related information with every proposition asserted in the network. In SWM, the logic underlying SNeBR, every proposition is called a *supported wff* (or *swff* for short). Associated with each swff is a support set. For example, the node M3 shown above, could have the support

$$\langle \mathsf{HYP}, \{\mathsf{M3}\}, \{\}\rangle$$

Each support set is a triple consisting of an *origin tag* (which can be HYP for hypotheses or DER for derived propositions) an *origin set* (which contains those, and only those, hypotheses used in the derivation of the swff) and a *restriction set* (which records inconsistency information). A proposition is asserted in a belief space if its origin set is a subset of that context.

Figure 9.1 shows a network representing four propositions, "A is a block,""B is a block," "A is clear," and "A is on B." Notice that the arc labels are the same as the defined slots of the subclasses Isa, On, and Clear (see Examples 1–3 in Chapter 4). The following SNePS trace creates a belief space that contains these propositions:

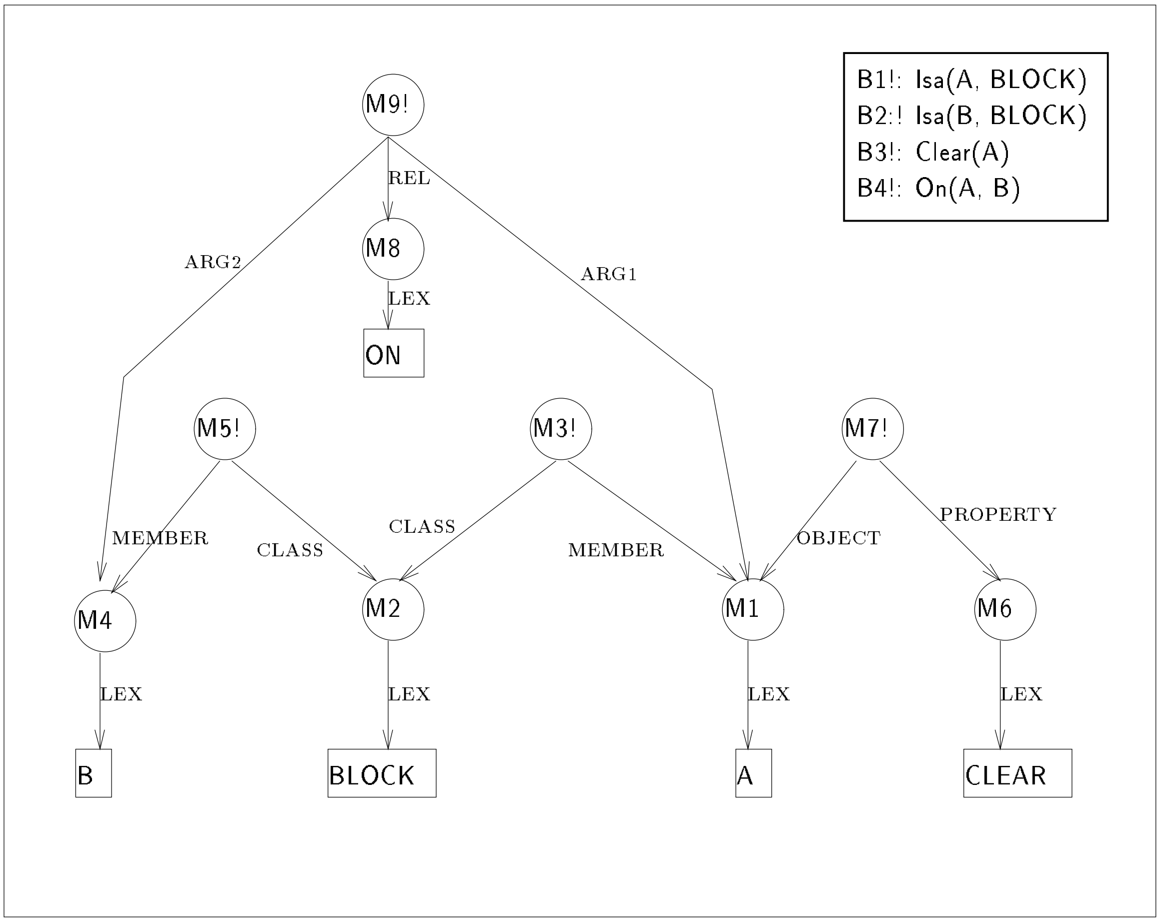

Figure 9.1: A SNePS network equivalent to the OK propositions shown in the inset box.

```
> (sneps)
  Welcome to SNePS-2.1
  Copyright 1984, 88, 89 by
  Research Foundation of State University of New York
  5/24/1993 16:22:51

; Define arc labels
* (define member class
          rel arg1 arg2
          object property
          lex)
(MEMBER CLASS REL ARG1 ARG2 OBJECT PROPERTY LEX)
 CPU time : 0.11     GC time : 0.00
;
; A is a block
* (assert member (build lex "A") = A
          class (build lex "BLOCK") = BLOCK)
(M3!)
 CPU time : 0.05     GC time : 0.00
;
; B is a block
* (assert member (build lex "B") = B
          class *BLOCK)
(M5!)
 CPU time : 0.05     GC time : 0.00
;
; A is clear
* (assert object *A property (build lex "CLEAR"))
(M7!)
 CPU time : 0.04     GC time : 0.00
;
; A is on B
* (assert rel (build lex "ON")
          arg1 *A
          arg2 *B)
(M9!)
 CPU time : 0.02     GC time : 0.00
;
; Show all the nodes in the belief space
* (describe *nodes)

(A)
(B)
(BLOCK)
(CLEAR)
(M1 (LEX A))
(M2 (LEX BLOCK))
(M3! (CLASS (M2)) (MEMBER (M1)))
(M4 (LEX B))
(M5! (CLASS (M2)) (MEMBER (M4)))
(M6 (LEX CLEAR))
(M7! (OBJECT (M1))
     (PROPERTY (M6)))
(M8 (LEX ON))
```

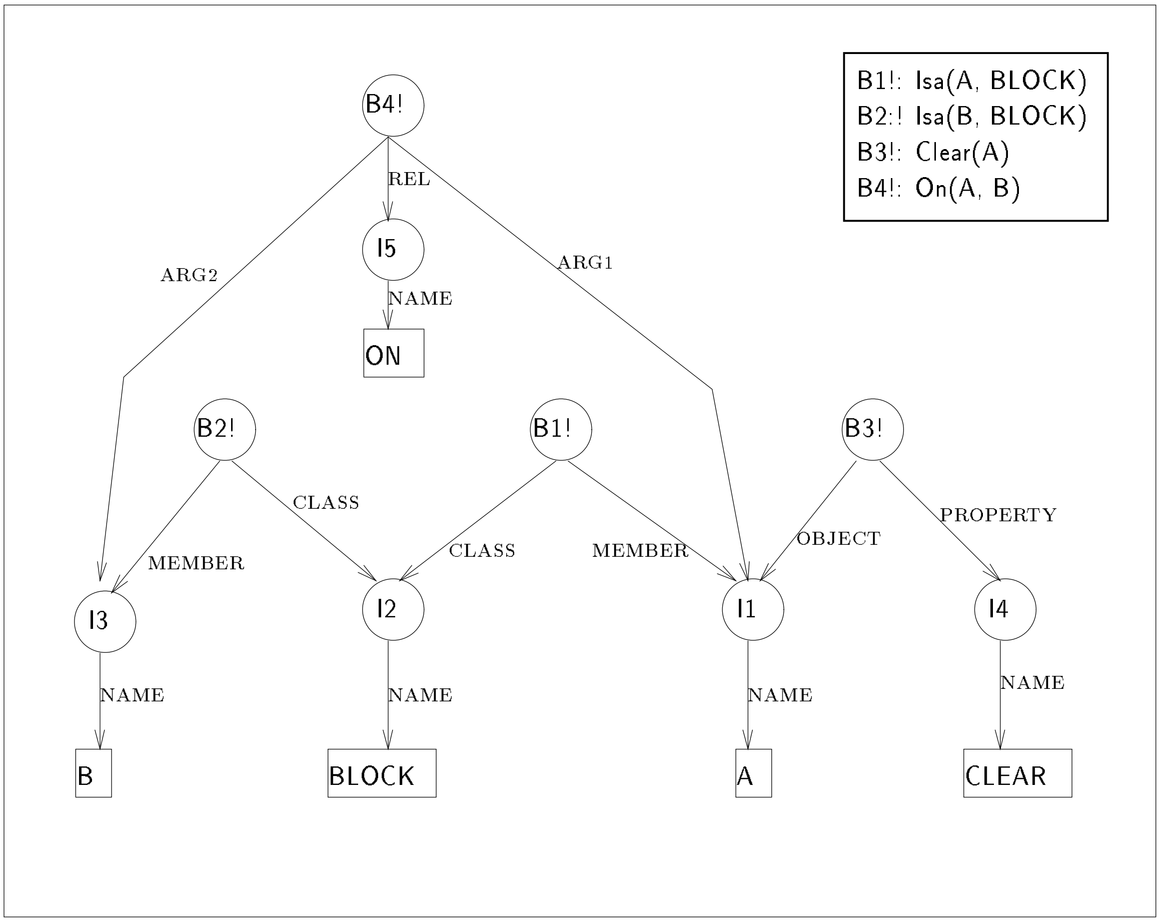

Figure 9.2: OK propositions shown in the inset box depicted as a semantic network.

```
(M9! (ARG1 (M1))
     (ARG2 (M4))
     (REL (M8)))

(A B BLOCK CLEAR M1 M2 M3! M4 M5! M6 M7! M8 M9! ON)
 CPU time : 0.13     GC time : 0.00
*
```

One could also depict the instances of the subclasses of the class Proposition Term as a semantic network. Figure 9.2 shows a semantic network representation of the instances B1, B2, B3, B4. Although the SNePS representations are isomorphic to those of the OK formalism, notice that in SNePS there is no explicit connection between the definitions of arc labels and their usage. One can erroneously create structured nodes using any kinds of arcs. Conceptually, they may not represent anything meaningful. In the OK formalism, each subclass explicitly defines its slot names. Thus, the interpretation applied to the subclass applies uniformly to all instances of that subclass. It is not possible to create instances that do not belong to an undefined class of propositions.

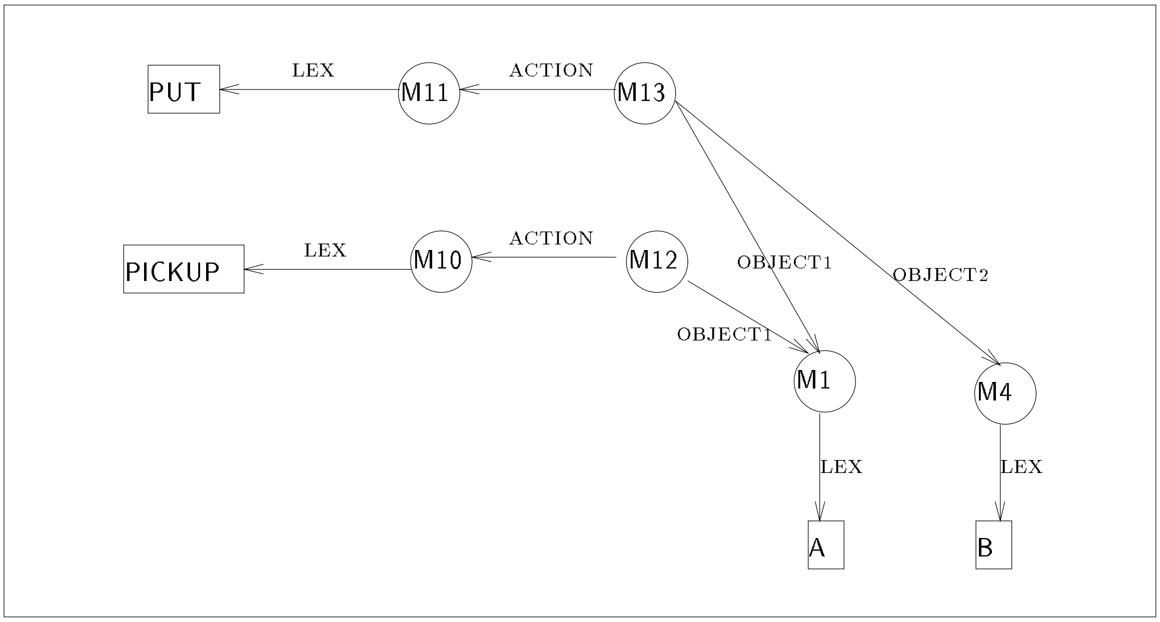

Figure 9.3: A SNePS network showing the representations of the acts PICKUP(A) and PUT(A, B).

9.6 Acts

Acts in SNePS are represented using structured individual molecular nodes. Arc labels on the arcs emanating from a node representing an act correspond, like propositions, to the slots defined for instances of specific subclasses of acts.

Definition S3:

Entity: Acts

Syntax: Acts are represented using structured molecular nodes with the following wire(s) in the cableset:

$\langle \mathsf{ACTION}, \alpha \rangle$**:** where α is a named individual node.

$\langle \mathsf{OBJECT1}, \mathsf{o}_1 \rangle$**:** where o_1 is a node.

$\langle \mathsf{OBJECT2}, \mathsf{o}_2 \rangle$**:** where o_2 is a node.

...

$\langle \mathsf{OBJECTn}, \mathsf{o}_\mathsf{n} \rangle$**:** where o_n is a node.

The number of OBJECTi arcs corresponds to the number of objects an action has.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \alpha \rangle, \langle \mathsf{OBJECT1}, \mathsf{o}_1 \rangle, \ldots, \langle \mathsf{OBJECTn}, \mathsf{o}_\mathsf{n} \rangle\}$$

is an act whose action is represented by α, and the entities represented by $\mathsf{o}_1, \ldots, \mathsf{o}_\mathsf{n}$ are its objects.

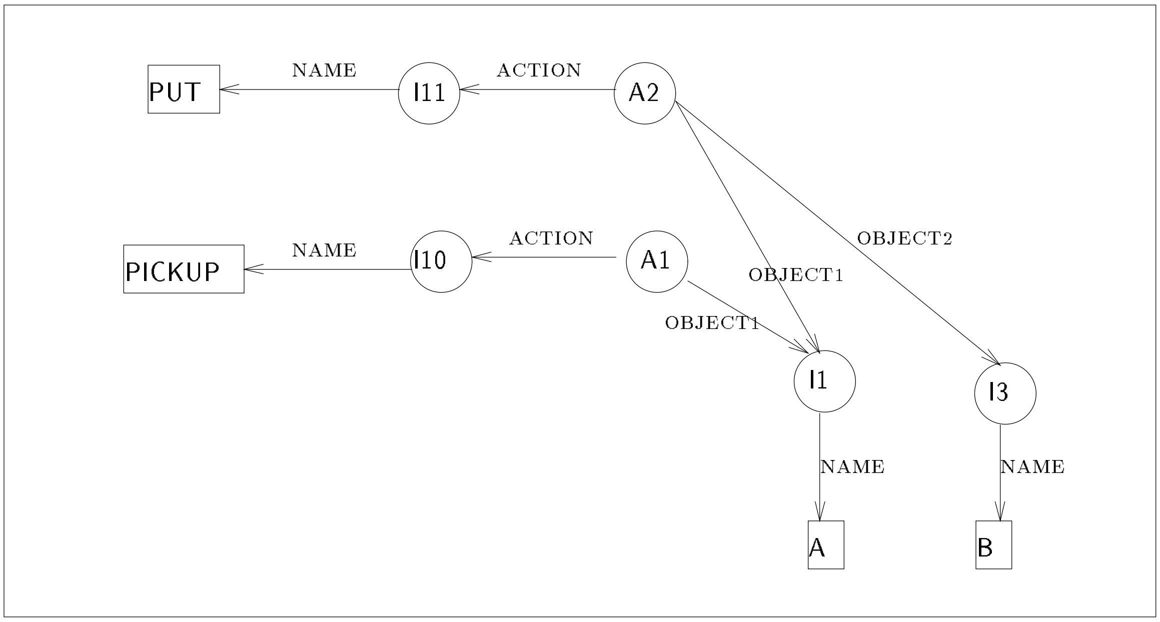

Figure 9.4: Acts in OK depicted as semantics networks.

Examples: In Figure 9.3, the nodes M12 and M13 represent acts.

For example, the acts PICKUP(A) and PUT(A, B) (as defined in Examples 1 and 2, Chapter 7) would be represented by nodes M12 and M13, respectively, as shown in Figure 9.3. Thus, a structured individual molecular node representing the act PICKUP will have two arcs, ACTION and OBJECT1, coming out of it. Figure 9.4 shows the OK instances of the same acts depicted in a semantic network form.

All the specific subclasses of the class Act Term have equivalent representations as SNePS networks (and vice versa). Definitions of the acts defined in Chapter 7 are presented next.

9.6.1 Mental Acts

SNePS representations for the mental acts BELIEVE and DISBELIEVE are defined below:

Definition S4:

Entity: The mental act of believing a proposition (BELIEVE).

Syntax: Nodes representing the act of believing a proposition have the following wire(s) in their cableset:

$\langle \mathsf{ACTION}, \{\langle \mathsf{LEX}, \mathsf{BELIEVE}\rangle\}\rangle$**:**

$\langle \mathsf{OBJECT1}, \phi\rangle$**:** where ϕ is a proposition node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \{\langle \mathsf{LEX}, \mathsf{BELIEVE}\rangle\}\rangle, \langle \mathsf{OBJECT1}, \phi\rangle\}$$

is the mental act of believing the proposition represented by ϕ.

Examples: The node

$\{\langle \mathsf{ACTION}, \{\langle \mathsf{LEX}, \mathsf{BELIEVE}\rangle\}\rangle,$
$\langle \mathsf{OBJECT1}, \{\langle \mathsf{PROPERTY}, \{\langle \mathsf{LEX}, \mathsf{CLEAR}\rangle\}\rangle, \langle \mathsf{OBJECT}, \{\langle \mathsf{LEX}, \mathsf{A}\rangle\}\rangle\}\rangle\}$

represents the act

$$\mathsf{BELIEVE}(\mathsf{Clear}(\mathsf{A}))$$

Definition S5:

Entity: The mental act of disbelieving a proposition (DISBELIEVE).

Syntax: Nodes representing the act of believing a proposition have the following wire(s) in their cableset:

$\langle \mathsf{ACTION}, \{\langle \mathsf{LEX}, \mathsf{DISBELIEVE}\rangle\}\rangle$**:**

$\langle \mathsf{OBJECT1}, \phi\rangle$**:** where ϕ is a proposition node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \{\langle \mathsf{LEX}, \mathsf{DISBELIEVE}\rangle\}\rangle, \langle \mathsf{OBJECT1}, \phi\rangle\}$$

is the mental act of disbelieving the proposition represented by ϕ.

Examples: The node

$\{\langle \mathsf{ACTION}, \{\langle \mathsf{LEX}, \mathsf{DISBELIEVE}\rangle\}\rangle,$
$\langle \mathsf{OBJECT1}, \{\langle \mathsf{PROPERTY}, \{\langle \mathsf{LEX}, \mathsf{CLEAR}\rangle\}\rangle, \langle \mathsf{OBJECT}, \{\langle \mathsf{LEX}, \mathsf{A}\rangle\}\rangle\}\rangle\}$

represents the act

$$\mathsf{DISBELIEVE}(\mathsf{Clear}(\mathsf{A}))$$

The effector components of mental acts are written using the SNeBR operations add-to-context and remove-from-context. These operations add or remove propositions from a belief space. Removing a hypothesis from a belief space using remove-from-context also removes all derived propositions that have the removed hypothesis in their support set. Similarly, while adding a hypothesis to a belief space, it is possible for other propositions that had been derived from the added hypothesis to be reincluded into the belief space (see [Kumar and Shapiro 1993]).

9.6.2 Plans and Control Acts

In this section, we present SNePS representations of the various control acts defined in Chapter 7. Sequencing acts in SNePS are defined as:

Definition S6:

Entity: Sequencing acts

Syntax: Nodes representing sequencing acts have the following wire(s) in their cableset:

$\langle \mathsf{ACTION}, \mathsf{SNSEQUENCE}\rangle$**:**

$\langle \mathsf{OBJECT1}, \alpha_1\rangle$**:** where α_1 is an act node.

$\langle \mathsf{OBJECT2}, \alpha_2\rangle$**:** where α_2 is an act node.

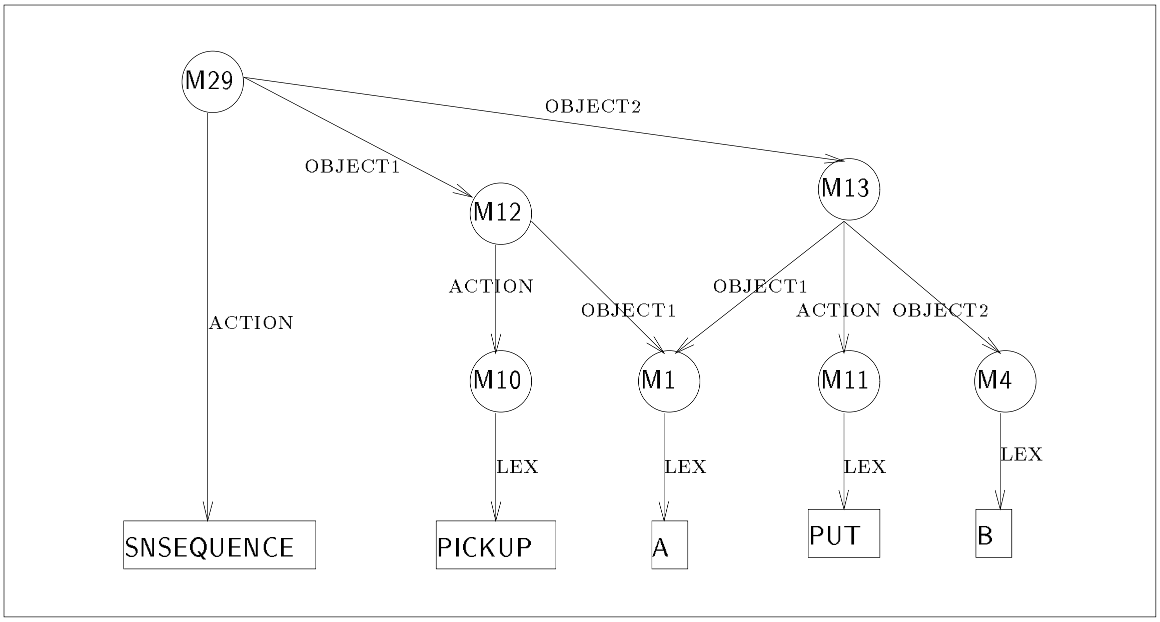

Figure 9.5: A SNePS network showing the representation of a sequencing act.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \mathsf{SNSEQUENCE}\rangle, \langle \mathsf{OBJECT1}, \alpha_1\rangle, \langle \mathsf{OBJECT2}, \alpha_2\rangle\}$$

is the act of performing the act represented by α_1 followed by the act represented by α_2.

Examples: Figure 9.5 shows a representation of the act

SEQUENCE(PICKUP(A), PUT(A, B))

In Figure 9.5, notice the node at the end of the ACTION arc coming out of node M29. Since SNSEQUENCE is not a lexical concept, it is represented as a base node labeled SNSEQUENCE. This is equivalent to the null definition of the ACTION methods for all subclasses of control acts (see Chapter 7).

When specifying a sequence of three or more actions, one has a choice of representations. For example, the act

$$\mathsf{SEQUENCE(PUT(C, TABLE), SEQUENCE(PUT(B, C), PUT(A, B)))}$$

can also be represented as the act

$$\mathsf{SEQUENCE(SEQUENCE(PUT(C, TABLE), PUT(B, C)), PUT(A, B))}$$

Pictorially, using a NOAH-like procedural net formalism [Sacerdoti 1977], they would appear the same (as shown in Figure 7.1). However, it turns out that in certain circumstances the latter representation is more desirable [Kumar *et al.* 1990].

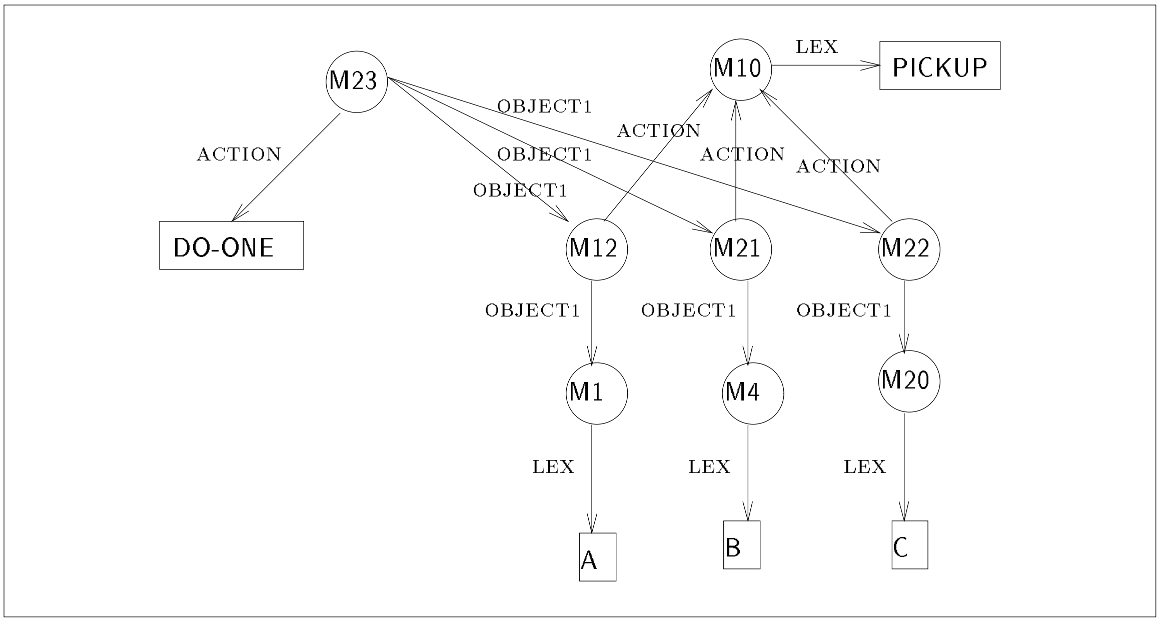

Figure 9.6: A SNePS network showing the representations of the act DoONE((PICKUP(A) PICKUP(B) PICKUP(C))).

Definition S7:

Entity: Non-deterministic Choice

Syntax: Nodes representing acts for non-deterministic choice have the following wire(s) and cable(s) in their cableset:

$\langle \mathsf{ACTION}, \mathsf{DO-ONE} \rangle$**:**

$\langle \mathsf{OBJECT1}, \alpha \rangle$**:** where α is a set of act nodes, $\{\alpha_1 \ldots \alpha_\mathsf{n}\}$.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \mathsf{DO-ONE} \rangle, \langle \mathsf{OBJECT1}, \{\alpha_1 \ldots \alpha_\mathsf{n}\} \rangle\}$$

is the act of performing only one of the acts represented by $\alpha_1 \ldots \alpha_\mathsf{n}$.

Example: Figure 9.6 is the SNePS representation of the act

DoONE((PICKUP(A) PICKUP(B) PICKUP(C)))

Definition S8:

Entity: Non-deterministic Ordering

Syntax: Nodes representing acts for non-deterministic ordering have the following wire(s) and cable(s) in their cableset:

$\langle \mathsf{ACTION}, \mathsf{DO-ALL} \rangle$**:**

$\langle \mathsf{OBJECT1}, \alpha \rangle$**:** where α is a set of act nodes, $\{\alpha_1 \ldots \alpha_\mathsf{n}\}$.

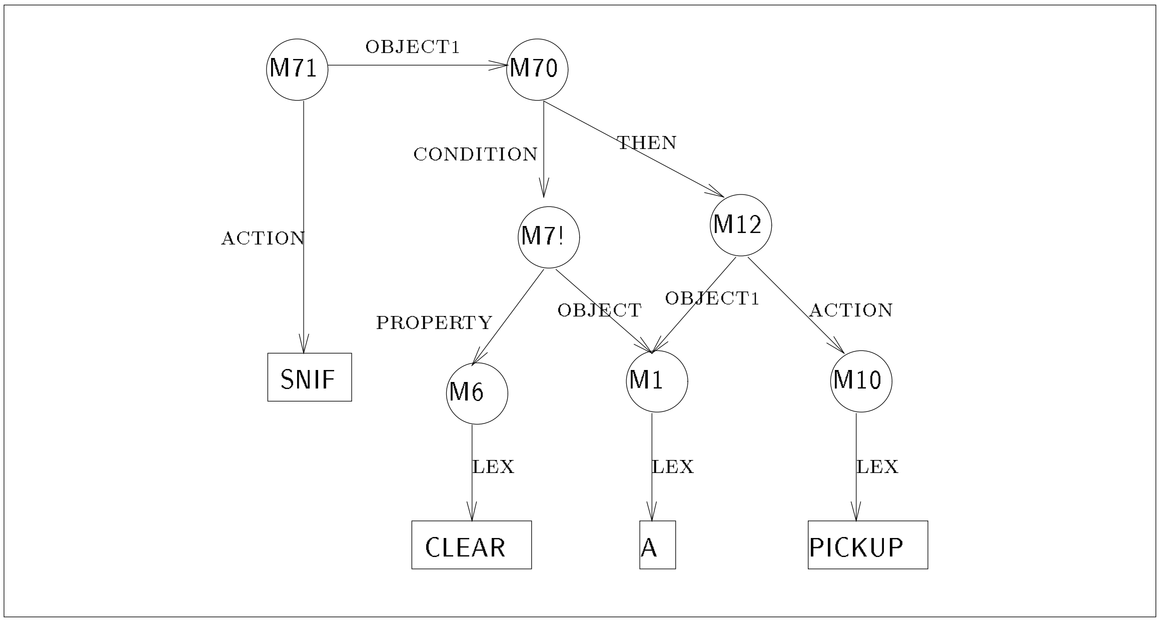

Figure 9.7: A SNePS network showing the representations of a conditional act

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \mathsf{DO-ALL}\rangle, \langle \mathsf{OBJECT1}, \{\alpha_1 \ldots \alpha_\mathsf{n}\}\rangle\}$$

is the act of performing all of the acts represented by $\alpha_1 \ldots \alpha_\mathsf{n}$ in some non-deterministic order.

Recall that the act representing non-deterministic choice is equivalent to an **or-split** and a non-deterministic ordering act is equivalent to an **and-split** in the procedural network formalism. Next, we define SNePS representations for conditional and iterative acts:

Definition S9: ______

Entity: Conditional acts

Syntax: Nodes representing conditional acts have the following wire(s) and cable(s) in their cable-set:

$\langle \mathsf{ACTION}, \mathsf{SNIF}\rangle$:

$\langle \mathsf{OBJECT1}, \alpha\rangle$: where α is a set of guarded-act nodes (see below), $\{\gamma_1 \ldots \gamma_\mathsf{n}\}$.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \mathsf{SNIF}\rangle, \langle \mathsf{OBJECT1}, \{\gamma_1 \ldots \gamma_\mathsf{n}\}\rangle\}$$

is a conditional act.

Definition S10: ______

Entity: Guarded acts

Syntax: Nodes representing guarded acts have the following wire(s) and cable(s) in their cableset:

> $\langle \mathsf{CONDITION}, \phi \rangle$**:** where ϕ is a proposition node.
>
> $\langle \mathsf{THEN}, \alpha \rangle$**:** where α is an act node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{CONDITION}, \phi \rangle, \langle \mathsf{THEN}, \alpha \rangle\}$$

is a guarded act.

Example: The guarded act $[\mathsf{Clear(A)} \leadsto \mathsf{PICKUP(A)}]$ is represented as

$$\{\langle \mathsf{CONDITION}, \{\langle \mathsf{OBJECT}, \{\langle \mathsf{LEX}, \mathsf{A} \rangle\}\rangle, \langle \mathsf{PROPERTY}, \{\langle \mathsf{LEX}, \mathsf{CLEAR} \rangle\}\rangle\}\rangle,$$
$$\langle \mathsf{THEN}, \{\langle \mathsf{ACTION}, \{\langle \mathsf{LEX}, \mathsf{PICKUP} \rangle\}\rangle, \langle \mathsf{OBJECT1}, \{\langle \mathsf{LEX}, \mathsf{A} \rangle\}\rangle\}\rangle\}$$

Figure 9.7 shows the SNePS representation of the act

$$\mathsf{IF}((\mathsf{[Clear(A)} \leadsto \mathsf{PICKUP(A)]}))$$

Definition S11: ________________

Entity: Iterative acts

Syntax: Nodes representing iterative acts have the following wire(s) and cable(s) in their cableset:

> $\langle \mathsf{ACTION}, \mathsf{SNITERATE} \rangle$**:**
>
> $\langle \mathsf{OBJECT1}, \alpha \rangle$**:** where α is a set of guarded-act nodes, $\{\gamma_1 \ldots \gamma_n\}$.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \mathsf{SNITERATE} \rangle, \langle \mathsf{OBJECT1}, \{\gamma_1 \ldots \gamma_n\}\rangle\}$$

is a an iterative act.

Definition S12: ________________

Entity: Achieve acts

Syntax: Nodes representing achieve acts have the following wire(s) and cable(s) in their cableset:

> $\langle \mathsf{ACTION}, \mathsf{ACHIEVE} \rangle$**:**
>
> $\langle \mathsf{OBJECT1}, \alpha \rangle$**:** where α is a proposition node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \mathsf{ACHIEVE} \rangle, \langle \mathsf{OBJECT1}, \alpha \rangle\}$$

is the act of achieving the proposition represented by the node α.

Example: The act $\mathsf{ACHIEVE(Clear(A))}$ is represented by the node

$$\{\langle \mathsf{ACTION}, \mathsf{ACHIEVE} \rangle,$$
$$\langle \mathsf{OBJECT1}, \{\langle \mathsf{PROPERTY}, \{\langle \mathsf{LEX}, \mathsf{CLEAR} \rangle\}\rangle, \langle \mathsf{OBJECT}, \{\langle \mathsf{LEX}, \mathsf{A} \rangle\}\rangle\}\rangle\}$$

Definition S13:

Entity: Single-object qualifier acts

Syntax: Nodes representing single-object qualifier acts have the following wire(s) and cable(s) in their cableset:

$\langle \mathsf{ACTION}, \mathsf{WITHSOME} \rangle$**:**

$\langle \mathsf{VARS}, \nu \rangle$**:** where ν is a set of variable nodes.

$\langle \mathsf{SUCHTHAT}, \phi \rangle$**:** where ϕ is a proposition node.

$\langle \mathsf{DO}, \alpha \rangle$**:** where α is an act node.

All variables in the set ν should be dominated by ϕ.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \mathsf{WITHSOME} \rangle, \langle \mathsf{VARS}, \nu \rangle, \langle \mathsf{SUCHTHAT}, \phi \rangle, \langle \mathsf{DO}, \alpha \rangle\}$$

is a single-object qualifier act.

Example: Figure 9.8 shows the SNePS representation of the act

$$\mathsf{WITHSOME(?x)[Isa(?x, BLOCK)]PICKUP(?x)}$$

Definition S14:

Entity: Multiple-object qualifier acts

Syntax: Nodes representing multiple-object qualifier acts have the following wire(s) and cable(s) in their cableset:

$\langle \mathsf{ACTION}, \mathsf{WITHALL} \rangle$**:**

$\langle \mathsf{VARS}, \nu \rangle$**:** where ν is a set of variable nodes.

$\langle \mathsf{SUCHTHAT}, \phi \rangle$**:** where ϕ is a proposition node.

$\langle \mathsf{DO}, \alpha \rangle$**:** where α is an act node.

All variables in the set ν should be dominated by ϕ.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \mathsf{WITHALL} \rangle, \langle \mathsf{VARS}, \nu \rangle, \langle \mathsf{SUCHTHAT}, \phi \rangle, \langle \mathsf{DO}, \alpha \rangle\}$$

is a multiple-object qualifier act.

For the qualifier acts, the proposition node ϕ can be a simple proposition (as shown in the example in Figure 9.8) or a proposition formed by using connectives. SNePS supports several non-standard connectives—*andor* and *non-derivable*, both of which can be used to combine simple propositions. The andor connective, written as

$$\bowtie_{\mathsf{i}}^{\mathsf{j}}\{\phi_1, \ldots, \phi_{\mathsf{n}}\}$$

means that at least i and at most j of the n propositions are asserted. The syntax of an andor in SNePS is defined as follows:

Definition S15:

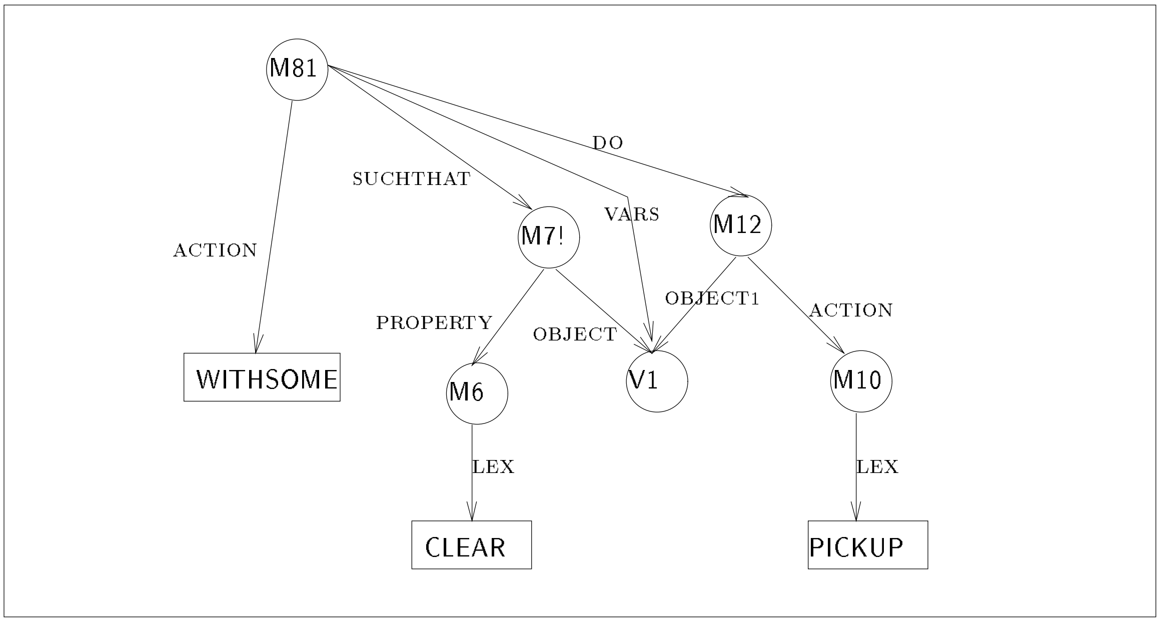

Figure 9.8: A SNePS network showing the representation of a single-qualifier act.

Entity: The AndOr Connective

Syntax: Nodes representing propositions formed using the andor connectives have the following wire(s) and cable(s) in their cableset:

$\langle \mathsf{MIN}, \mathsf{n}_1 \rangle$**:** where n_1 is a number.

$\langle \mathsf{MAX}, \mathsf{n}_2 \rangle$**:** where n_2 is a number.

$\langle \mathsf{ARG}, \eta \rangle$**:** where η is a set of proposition nodes.

Semantics: The entity represented by the node

$$\{\langle \mathsf{MIN}, \mathsf{n}_1 \rangle, \langle \mathsf{MAX}, \mathsf{n}_2 \rangle, \langle \mathsf{ARG}, \eta \rangle\}$$

is an andor proposition.

Example: When $i = j = n$ it represents the logical AND connective. $i = j = 0$ is a generalization of NOR; and $i = j = 1$ is a generalization of EXCLUSIVE OR.

For example, the act

$$\mathsf{WITHSOME(?x)[(Isa(?xBLOCK)} \wedge \mathsf{Clear(?x))]PICKUP(?x)}$$

can be represented using an andor connective. Since we claimed that the logic underlying the OK formalism is extendible, here is an opportunity to show how it would be done. We can define the andor connective in OK as follows:

Example 1: ______

Class: AndOr

Superclass(es): Proposition Term, Conceptual Entity Term

Slots: The following slots are defined for all instances of this class:

> MIN: Can be a number greater than or equal to 0.
>
> MAX: Can be a number greater than or equal to 0.
>
> ARG: Contains a set of proposition nodes.

Methods: If o is an instance of this class, then

> MIN(o): Returns the number in the MIN slot.
>
> MAX(o): Returns the number in the MAX slot.
>
> ARG(o): Returns the contents of the ARG slot.

Syntax: Instances of this class are expressed by writing the symbol $\bigvee\!\!\!\bigwedge_{i}^{j}$, where i is the value of the MIN slot, and j is the value of the MAX slot, followed by the set of arguments.

Description: The entity represented by o is an andor of its argument propositions.

Example Instances: The proposition "A is clear and B is clear" can be represented by an object written as:

$$\mathsf{B14!} : \bigvee\!\!\!\bigwedge_{2}^{2}\{\mathsf{Clear(A)}\ \mathsf{Clear(B)}\}$$

Both the qualifier acts serve to represent descriptions of the entities that can be objects of actions. If one employs structured variable representations as in the work of [Ali 1993a, Ali 1993b, Ali 1993c], the qualifications on the entities of the actions are specified directly in the representation of the variable that is in the object position for the action. The resulting representation of the action is more natural; i.e., it supports the claims made by [Ali 1993a].

9.7 Transformers

Like the OK formalism, transformers in SNePS are also propositions. Transformers are used by SNeRE during backward- and forward-chaining inference and acting. The SNePS formalism already had several belief–belief transformers (based on the SWM logic). In this chapter, we present only those that are equivalent to the ones defined in the OK formalism. It should, however, be evident by now that the OK formalism can be extended to represent all the SNePS belief-belief transformers.

9.7.1 Belief–Belief Transformers

The SNePS reasoning rules form the belief–belief transformers of the SNePS BDI architecture. The logic underlying SNePS, called SWM logic, defines several non-standard connectives—*and-entailment, or-entailment, numerical entailment, andor, thresh, and non-derivable* [Shapiro and Group 1989]. The definition of an or-entailment rule is as follows:

Definition S16:

Entity: Or-Entailment rule

Syntax: Nodes representing or-entailment rules have the following wire(s) and cable(s) in their cableset:

$\langle \Theta, \nu \rangle$: where Θ can be the universal quantifier, FORALL, or the existential, EXISTS, and ν is a set of variable nodes.

$\langle \mathsf{ANT}, \phi \rangle$: where ϕ is a set of proposition nodes.

$\langle \mathsf{CQ}, \sigma \rangle$: where σ is is a set of proposition nodes.

Semantics: The entity represented by the node

$$\{\langle \mathsf{FORALL}, \nu \rangle, \langle \mathsf{ANT}, \phi \rangle, \langle \mathsf{CQ}, \sigma \rangle\}$$

is an or-entailment rule with universally quantified variables in the set ν.

The or-entailment transformer is the same as the AntCq transformer. For example, the transformer

$$\mathsf{B5!} : \forall \mathsf{x}[\mathsf{Isa}(\mathsf{x}, \mathsf{BLOCK}) \rightarrow \mathsf{Isa}(\mathsf{x}, \mathsf{SUPPORT})]$$

representing the sentence, "all blocks are supports" is represented in SNePS as an or-entailment rule:

```
(M3! (FORALL V2)
     (ANT (P1 (CLASS (M1 (LEX BLOCK)))
              (MEMBER V2)))
     (CQ (P2 (CLASS (M2 (LEX SUPPORT)))
             (MEMBER V2))))
```

The sentence, "if a block is on one block then it is not on another block", which in the OK formalism is represented as

$$\mathsf{B17!} : \forall \mathsf{x} \forall \mathsf{y} \forall \mathsf{z}[(((\mathsf{Isa}(\mathsf{x}, \mathsf{BLOCK}) \wedge \mathsf{Isa}(\mathsf{y}, \mathsf{BLOCK})) \wedge \mathsf{Isa}(\mathsf{z}, \mathsf{BLOCK})) \wedge \mathsf{On}(\mathsf{x}, \mathsf{y})) \rightarrow \neg \mathsf{On}(\mathsf{x}, \mathsf{z})]$$

will be represented using an and-entailment rule:

```
(M5! (FORALL V13 V14 V15)
     (&ANT (P15 (CLASS (M1 (LEX BLOCK)))
                (MEMBER V13))
           (P16 (CLASS (M1)) (MEMBER V14))
           (P17 (CLASS (M1)) (MEMBER V15))
           (P18 (ARG1 V13)
                (ARG2 V14)
                (REL (M4 (LEX ON)))))
     (CQ
       (P20 (MIN 0)
            (MAX 0)
            (ARG (P19 (ARG1 V13)
                      (ARG2 V15)
                      (REL (M4)))))))
```

One could also represent the antecedent of the above rule as an andor and then use the or-entailment rule

$$\mathsf{B20!} : \forall \mathsf{x} \forall \mathsf{y} \forall \mathsf{z}[\bowtie_4^4 \{\mathsf{Isa}(\mathsf{x}, \mathsf{BLOCK})\ \mathsf{Isa}(\mathsf{y}, \mathsf{BLOCK})\ \mathsf{Isa}(\mathsf{z}, \mathsf{BLOCK})\ \mathsf{On}(\mathsf{x}, \mathsf{y})\} \rightarrow \neg \mathsf{On}(\mathsf{x}, \mathsf{z})]$$

We have been purposely using different notations in our discussions—the OK formalism, the SNePS cableset notation, SNePS diagrams, and SNePSUL notation. Since they are all isomorphic to each other, we will, as we have been, use any of these to express our representations. This serves to make the reader familiar with all the notations, as well as emphasize the isomorphism.

9.7.2 Belief–Act Transformers

The proposition representing preconditions of actions was already present in the SNePS acting system. The same representations will be used. However, in the acting process it is defined as a backward chaining transformer. We will see the details of this transformation process in the next chapter. Here is the SNePS definition of the PreconditionAct transformer:

Definition S17:

Entity: The PreconditionAct transformer

Syntax: All PreconditionAct propositions are represented using structured molecular nodes with the following wire(s) in the cableset:

$\langle \mathsf{PRECONDITION}, \phi \rangle$**:** where ϕ is a proposition node.

$\langle \mathsf{ACT}, \alpha \rangle$**:** where α is an act node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{PRECONDITION}, \phi \rangle, \langle \mathsf{ACT}, \alpha \rangle\}$$

is a proposition that the entity represented by ϕ is a precondition of the act represented by α.

Examples: Figure 9.9 shows SNePS representations for the following propositions:

$$\mathsf{PreconditionAct(Clear(B), PUT(A, B))}$$

$$\mathsf{PreconditionAct(Held(A), PUT(A, B))}$$

They are represented by proposition nodes, M66 and M67, respectively.

The forward chaining transformer WhenDo is a new transformer introduced in the SNePS formalism. It is used to represent general desires of the agent to react to certain situations. It is defined as follows:

Definition S18:

Entity: The WhenDo transformer

Syntax: All WhenDo propositions are represented using structured molecular nodes with the following wire(s) in the cableset:

$\langle \Theta, \nu \rangle$**:** (optional cableset) where Θ can be the universal quantifier, FORALL, or the existential, EXISTS, and ν is a set of variable nodes.

$\langle \mathsf{WHEN}, \phi \rangle$**:** where ϕ is a proposition node.

$\langle \mathsf{DO}, \alpha \rangle$**:** where α is an act node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{WHEN}, \phi \rangle, \langle \mathsf{DO}, \alpha \rangle\}$$

is a proposition that represents the agent's desire to react to a situation in which the proposition represented by ϕ is asserted by intending to perform the act represented by α.

Examples: Figure 9.10 shows SNePS representations for the following proposition:

$$\mathsf{WhenDo(Clear(A), PICKUP(A))}$$

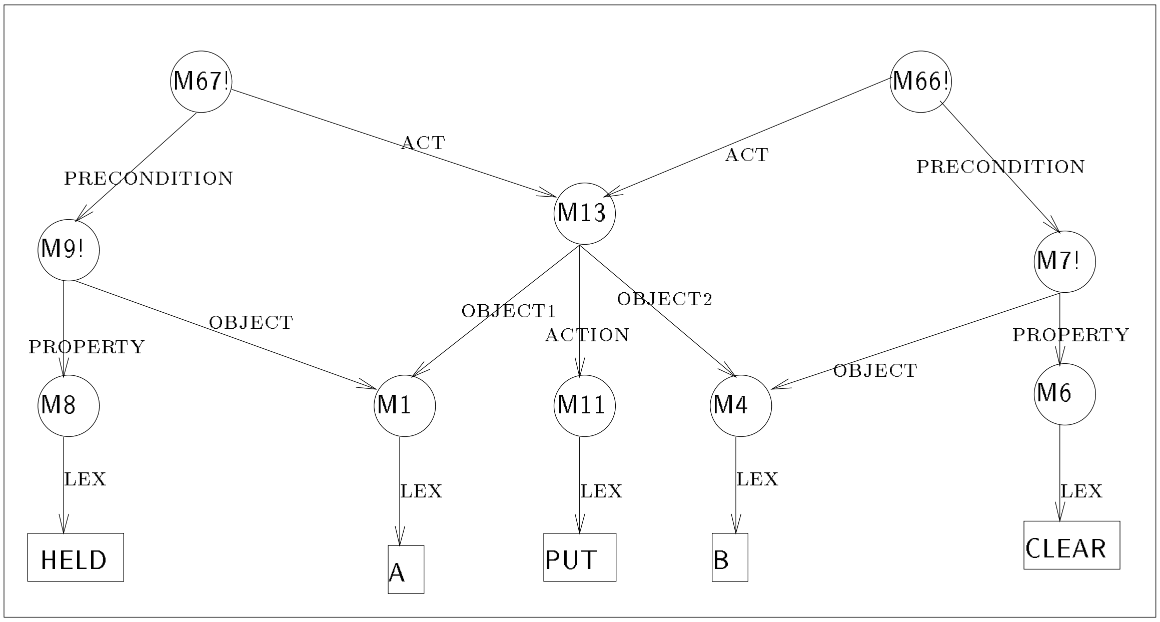

Figure 9.9: A SNePS network showing the preconditions of the act PUT(A, B).

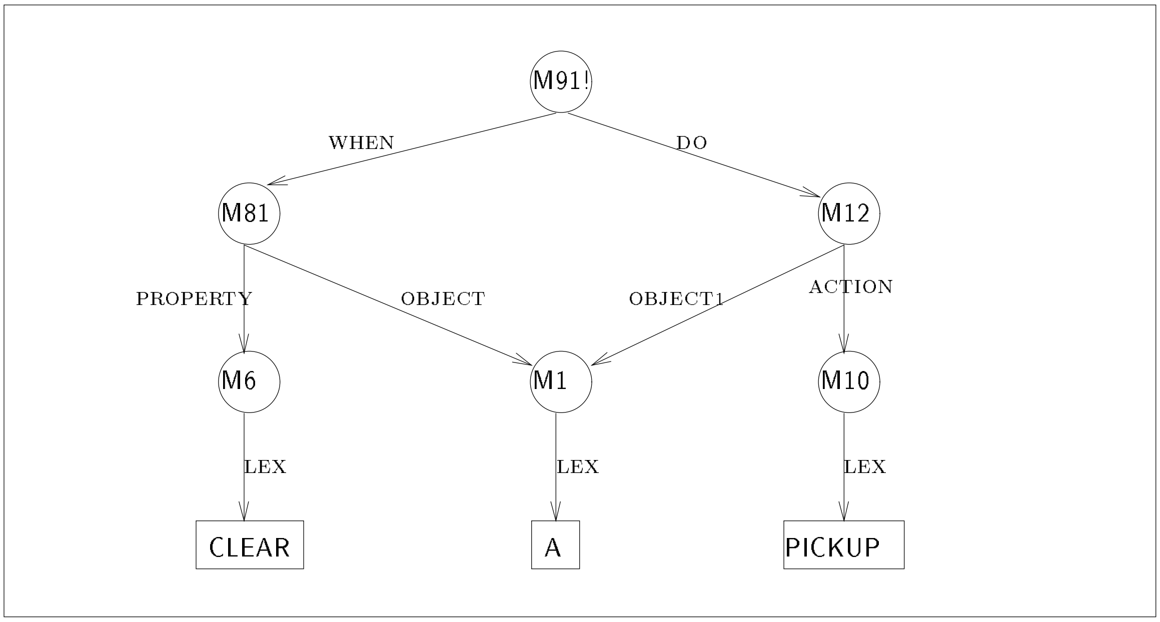

Figure 9.10: A SNePS network showing a WhenDo transformer.

We will see some more examples of these transformers as well as their usage in later chapters. This transformer extends the SNePS ontology in several respects. It allows for a representation of general reactive desires of the agent as propositions. They are used only when the agent believes them. Since they are represented as molecular nodes, they also form terms in the SWM logic. Thus, the agent can not only represent and use the proposition denoted, it can also reason about it. It can represent its own reactive desires as well as those of others, as in "John believes that whenever he sees a clear block, he will pick it up". This transformer can also be used to represent desires such as "whenever someone performs some act then believe the act's effects," thus keeping an updated belief space consistent with the outside world.

9.7.3 Act–Belief Transformers

The forward chaining act-effect transformer that represents the effects of acts is the same as the proposition defined by the SNePS acting system and is defined as follows:

Definition S19:

Entity: The ActEffect transformer

Syntax: All ActEffect propositions are represented using structured molecular nodes with the following wire(s) in the cableset:

$\langle \mathsf{ACT}, \alpha \rangle$**:** where α is an act node.

$\langle \mathsf{EFFECT}, \phi \rangle$**:** where ϕ is a proposition node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACT}, \alpha \rangle, \langle \mathsf{EFFECT}, \phi \rangle\}$$

is a proposition that the entity represented by ϕ is an effect of the act represented by α.

The following propositions represent the effects of the act represented by the node M18:

```
(M68!
  (ACT
    (M58 (ACTION (M18 (LEX PUT)))
         (OBJECT1 (M1 (LEX A)))
         (OBJECT2 (M4 (LEX B))))
  (EFFECT
    (M64 (REL (M8 (LEX ON)))
         (ARG1 (M1))
         (ARG2 (M4))))))
(M69!
  (ACT
    (M58 (ACTION (M18))
         (OBJECT1 (M1))
         (OBJECT2 (M4)))
  (EFFECT
    (M63 (OBJECT (M1))
         (PROPERTY (M6 (LEX CLEAR)))))))
(M70!
```

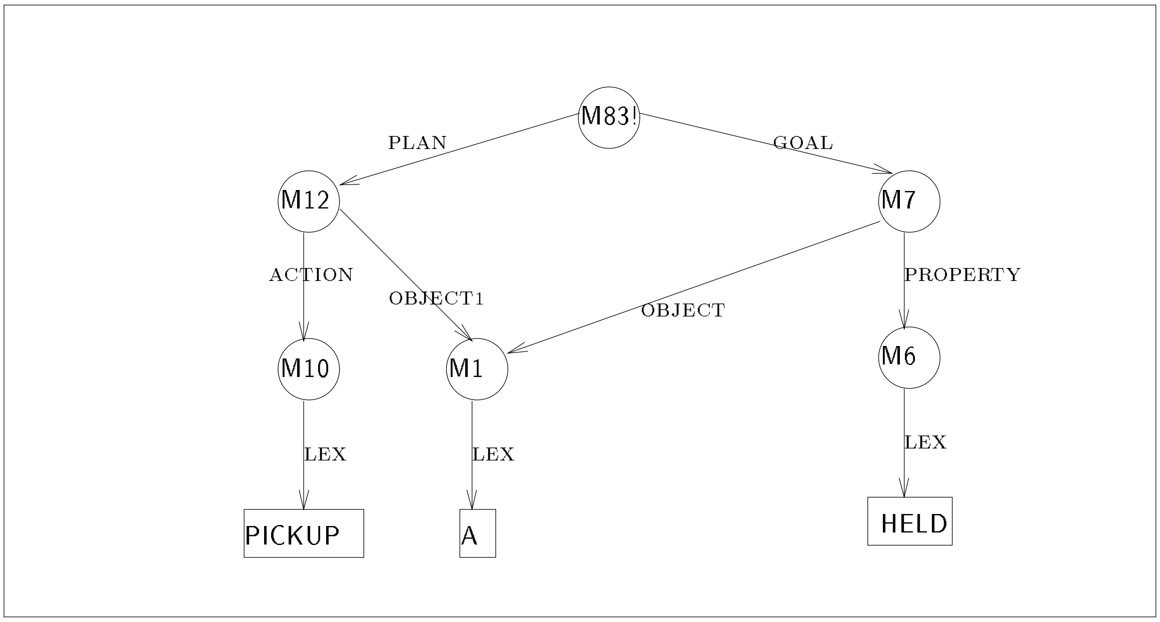

Figure 9.11: A SNePS network showing a PlanGoal transformer.

```
(ACT
  (M58 (ACTION (M18))
       (OBJECT1 (M1))
       (OBJECT2 (M4)))
(EFFECT
  (M60 (MIN 0)
       (MAX 0)
       (ARG (M73 (OBJECT (M4))
                 (PROPERTY (M6))))))))
```

The backward chaining PlanGoal transformer in SNePS is defined next.

Definition S20:

Entity: The PlanGoal transformer

Syntax: All PlanGoal propositions are represented using structured molecular nodes with the following wire(s) in the cableset:

$\langle \mathsf{PLAN}, \alpha \rangle$**:** where α is an act node.

$\langle \mathsf{GOAL}, \phi \rangle$**:** where ϕ is a proposition node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{PLAN}, \alpha \rangle, \langle \mathsf{GOAL}, \phi \rangle\}$$

is a proposition that the entity represented by ϕ is a goal that can be achieved by intending to perform the plan (act) represented by α.

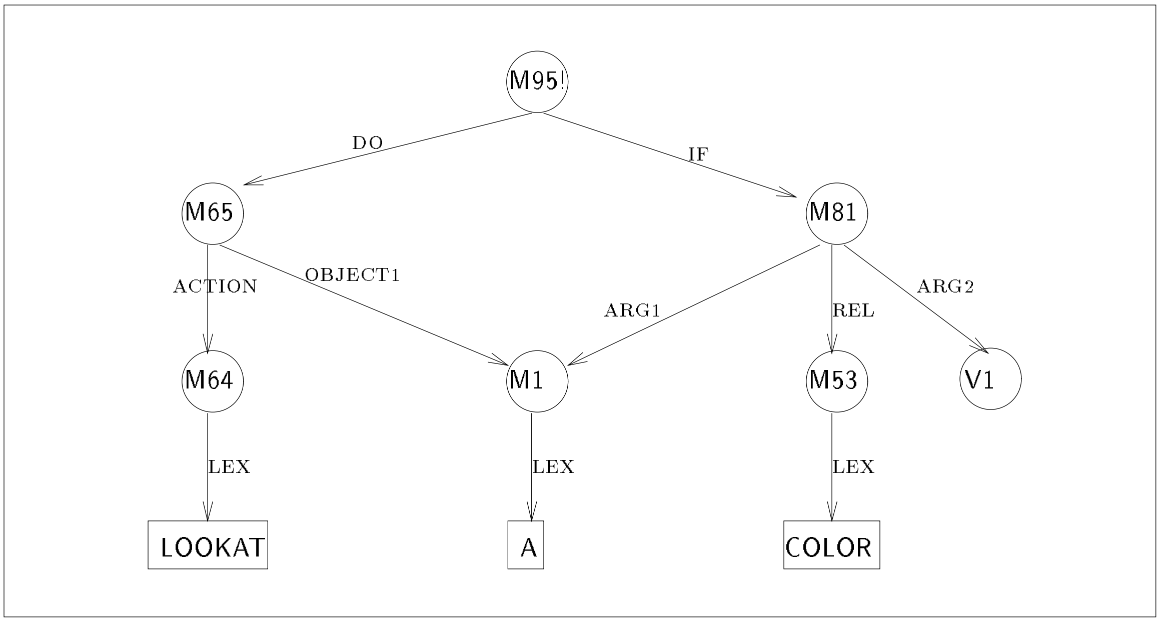

Figure 9.12: A SNePS network showing a DoWhen transformer.

Example: Figure 9.11 shows the SNePS representation of the proposition

$$\mathsf{PlanGoal}(\mathsf{PICKUP}(\mathsf{A}), \mathsf{Held}(\mathsf{A}))$$

Definition S21:

Entity: The DoWhen transformer

Syntax: All DoWhen propositions are represented using structured molecular nodes with the following wire(s) in the cableset:

$\langle \Theta, \nu \rangle$**:** (optional cableset) where Θ can be the universal quantifier, FORALL, or the existential, EXISTS, and ν is a set of variable nodes.

$\langle \mathsf{DO}, \alpha \rangle$**:** where α is an act node.

$\langle \mathsf{IF}, \phi \rangle$**:** where ϕ is a proposition node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{DO}, \alpha \rangle, \langle \mathsf{IF}, \phi \rangle\}$$

is a proposition that represents the agent's desire to perform the act α whenever it needs to know the assertional status of the proposition represented by ϕ.

Examples: Figure 9.12 shows SNePS representations for the following proposition:

$$\mathsf{DoWhen}(\mathsf{LOOKAT}(\mathsf{A}), \mathsf{Color}(\mathsf{A}, \mathsf{V1}))$$

We have changed an arc label here. Instead of using the label WHEN we have used IF. This is because a SNePS node is simply a cableset, i.e., the node

$$\{\langle \mathsf{MEMBER}, \{\langle \mathsf{LEX}, \mathsf{A}\rangle\}\rangle, \langle \mathsf{CLASS}, \{\langle \mathsf{LEX}, \mathsf{BLOCK}\rangle\}\rangle\}$$

can also be written as

$$\{\langle \mathsf{CLASS}, \{\langle \mathsf{LEX}, \mathsf{BLOCK}\rangle\}\rangle, \langle \mathsf{MEMBER}, \{\langle \mathsf{LEX}, \mathsf{A}\rangle\}\rangle\}$$

Similarly, the node

$$\{\langle \mathsf{DO}, \alpha\rangle, \langle \mathsf{WHEN}, \phi\rangle\}$$

is the same as the node

$$\{\langle \mathsf{WHEN}, \phi\rangle, \langle \mathsf{DO}, \alpha\rangle\}$$

Thus, it would be impossible to apply two different interpretations to the same node—a forward chaining interpretation (to the WhenDo transformer) that represents a reactive rule and a backward chaining interpretation (to the DoWhen transformer) that represents a rule for acquiring a belief. We have resolved this situation as shown above by defining the DoWhen transformer using an arc labeled IF. This was unnecessary in the OK formalism, because the classes WhenDo and DoWhen are separate subclasses, even though they have the same names for their slots.

9.7.4 Act–Act Transformers

The SNePS acting system's representations for the propositions that represent decompositions of complex acts now correspond to the act-act transformer (PlanAct).

Definition S22:

Entity: The PlanAct transformer

Syntax: All PlanAct propositions are represented using structured molecular nodes with the following wire(s) in the cableset:

$\langle \mathsf{PLAN}, \alpha\rangle$**:** where α is an act node.

$\langle \mathsf{ACT}, \sigma\rangle$**:** where σ is a complex act node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{PLAN}, \alpha\rangle, \langle \mathsf{ACT}, \sigma\rangle\}$$

is a proposition that the entity represented by σ is a complex act that can be decomposed into the the plan (act) represented by α.

Example: Figure 9.13 shows the SNePS representation of the proposition

$$\mathsf{PlanAct(PILE(A, B, C), SEQUENCE(SEQUENCE(PUT(C, TABLE), PUT(B, C)), PUT(A, B)))}$$

9.8 Discussion

In this chapter we have done two things: given the SNePS definitions for the ontology defined in the OK formalism and shown that the two formalisms are isomorphic. Next, we will present the operational module—the SNePS rational engine (SNeRE) that interprets the representations defined here during acting and inference.

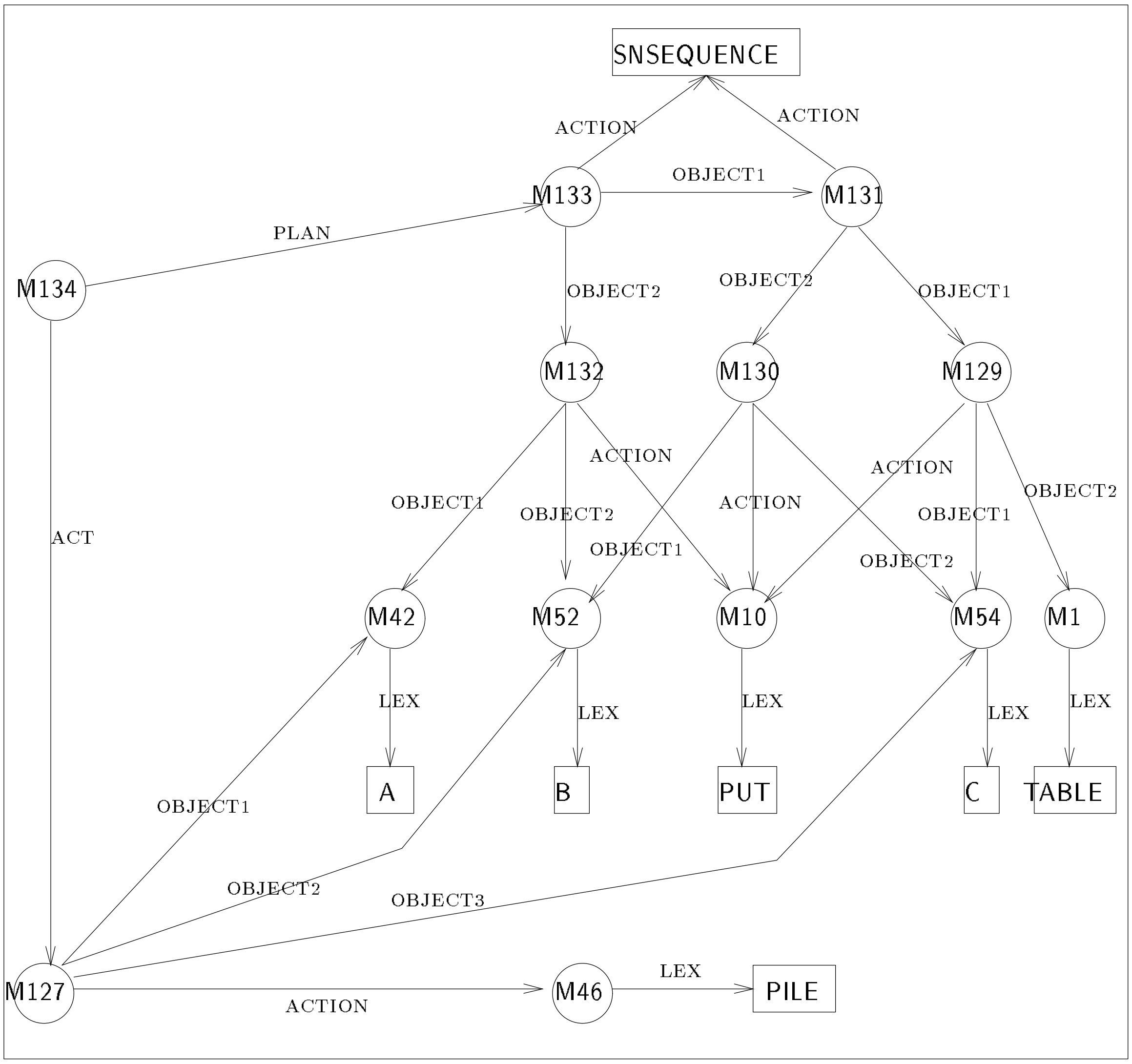

Figure 9.13: A SNePS network showing a PlanAct transformer.

Chapter 10

The Blocksworld Domain

Most of the examples presented so far have been from a blocksworld domain. In this chapter, we will demonstrate how to model the beliefs, desires, and intentions of an agent in the blocksworld using the SNePS BDI formalism. We will model a blocksworld where there is a robot arm that can do two primitive actions, PICKUP and PUT. Our model of the blocksworld consists of some rectangular blocks on a table. The blocks are identified by names like A, B, C, etc. (see Figure 10.1).

10.1 Beliefs

Beliefs about the current state of the blocksworld are represented using the propositions in Table 10.1. The class of propositions Isa, On, and Clear were defined in the OK formalism in Chapter 4 (Examples 1–3). The SNePS representation for Isa was defined in Chapter 9 (Definition S2). Here are the SNePS definitions for Clear, Held, and On:

Definition S23:

Entity: The OBJECT-PROPERTY proposition.

Syntax: All OBJECT-PROPERTY propositions are represented using structured molecular nodes with the following wire(s) in the cableset:

$\langle$OBJECT$, \phi\rangle$**:** where ϕ is a node.

$\langle$PROPERTY$, \sigma\rangle$**:** where σ is a node representing the name of a property.

Predicate	Interpretation
Clear(x)	x is clear (*i.e.*, there is nothing on top of x).
On(x,y)	x is on top of y.
Held(x)	x is currently being held.
Isa(x, y)	x is a member of the class y

Table 10.1: Belief propositions for the blocksworld

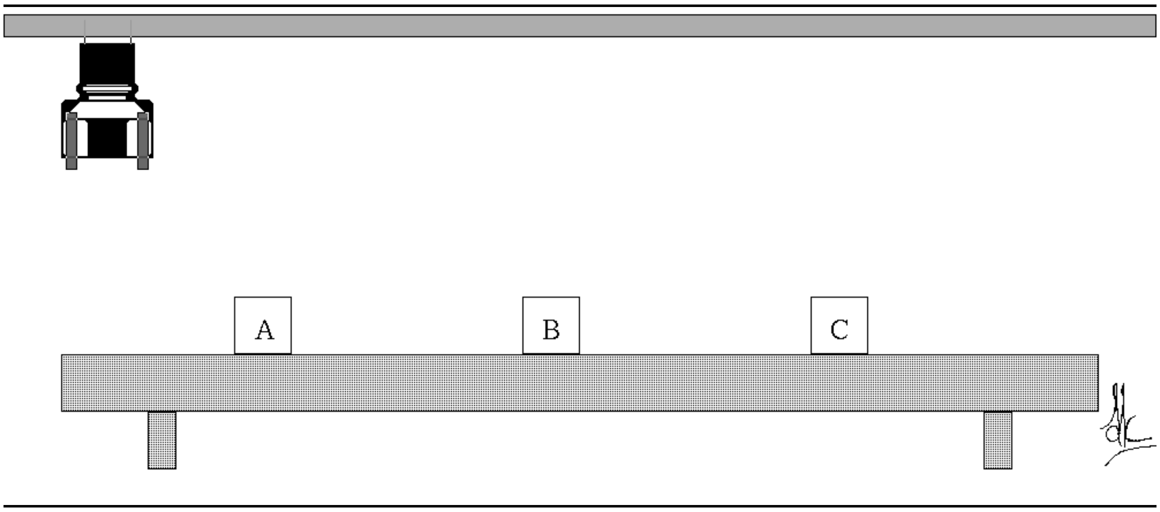

Figure 10.1: The blocksworld

Semantics: The entity represented by the node

$$\{\langle \mathsf{OBJECT}, \phi\rangle, \langle \mathsf{PROPERTY}, \sigma\rangle\}$$

is a proposition that the entity represented by ϕ has the property represented by σ.

Examples: Both Clear and Held are properties. The propositions "A is clear" and "B is held" are represented as:

$$\{\langle \mathsf{OBJECT}, \{\langle \mathsf{LEX}, \mathsf{A}\rangle\}\rangle, \langle \mathsf{PROPERTY}, \{\langle \mathsf{LEX}, \mathsf{CLEAR}\rangle\}\rangle\}$$

$$\{\langle \mathsf{OBJECT}, \{\langle \mathsf{LEX}, \mathsf{B}\rangle\}\rangle, \langle \mathsf{PROPERTY}, \{\langle \mathsf{LEX}, \mathsf{HELD}\rangle\}\rangle\}$$

Definition S24:

Entity: The On proposition.

Syntax: All On propositions are represented using structured molecular nodes with the following wire(s) in the cableset:

$\langle \mathsf{REL}, \{\langle \mathsf{LEX}, \mathsf{ON}\rangle\}\rangle$**:**

$\langle \mathsf{ARG1}, \phi\rangle$**:** where ϕ is a node.

$\langle \mathsf{ARG2}, \sigma\rangle$**:** where σ is a node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{REL}, \{\langle \mathsf{LEX}, \mathsf{ON}\rangle\}\rangle, \langle \mathsf{ARG1}, \phi\rangle, \langle \mathsf{ARG2}, \sigma\rangle\}$$

is a proposition that the entity represented by ϕ is on top of the entity represented by σ.

Examples: The proposition "A is on B" is represented by the node:

$$\{\langle \mathsf{REL}, \{\langle \mathsf{LEX}, \mathsf{ON}\rangle\}\rangle, \langle \mathsf{ARG1}, \{\langle \mathsf{LEX}, \mathsf{A}\rangle\}\rangle, \langle \mathsf{ARG2}, \{\langle \mathsf{LEX}, \mathsf{B}\langle\}\rangle\}$$

The following paragraph describes a state of the blocksworld:

Label: Proposition	Support
M19! : $\mathsf{Isa}(\mathsf{B1}, \mathsf{TABLE})$	$\langle \mathsf{HYP}, \{\mathsf{M19!}\}, \{\} \rangle$
M21! : $\mathsf{Isa}(\mathsf{B1}, \mathsf{SUPPORT})$	$\langle \mathsf{HYP}, \{\mathsf{M21!}\}, \{\} \rangle$
M23! : $\forall \mathsf{V2}[\mathsf{Isa}(\mathsf{V2}, \mathsf{BLOCK}) \rightarrow \mathsf{Isa}(\mathsf{V2}, \mathsf{SUPPORT})]$	$\langle \mathsf{HYP}, \{\mathsf{M23!}\}, \{\} \rangle$
M51! : $\mathsf{Isa}(\mathsf{A}, \mathsf{BLOCK})$	$\langle \mathsf{HYP}, \{\mathsf{M51!}\}, \{\} \rangle$
M53! : $\mathsf{Isa}(\mathsf{B}, \mathsf{BLOCK})$	$\langle \mathsf{HYP}, \{\mathsf{M53!}\}, \{\} \rangle$
M55! : $\mathsf{Isa}(\mathsf{C}, \mathsf{BLOCK})$	$\langle \mathsf{HYP}, \{\mathsf{M55!}\}, \{\} \rangle$
M56! : $\mathsf{Clear}(\mathsf{C})$	$\langle \mathsf{HYP}, \{\mathsf{M56!}\}, \{\} \rangle$
M57! : $\mathsf{On}(\mathsf{C}, \mathsf{B1})$	$\langle \mathsf{HYP}, \{\mathsf{M57!}\}, \{\} \rangle$
M58! : $\mathsf{Clear}(\mathsf{B})$	$\langle \mathsf{HYP}, \{\mathsf{M58!}\}, \{\} \rangle$
M59! : $\mathsf{On}(\mathsf{B}, \mathsf{B1})$	$\langle \mathsf{HYP}, \{\mathsf{M59!}\}, \{\} \rangle$
M60! : $\mathsf{Clear}(\mathsf{A})$	$\langle \mathsf{HYP}, \{\mathsf{M60!}\}, \{\} \rangle$
M61! : $\mathsf{On}(\mathsf{A}, \mathsf{B1})$	$\langle \mathsf{HYP}, \{\mathsf{M61!}\}, \{\} \rangle$

Figure 10.2: A belief space showing the beliefs of the agent about a blocksworld. The notation employed is the OK syntax for the SNePS formalism.

> 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. C is on the table. B is clear. B is on the table. A is clear. A is on the table.

This state can be represented in SNePS using the following SNePS assertions:

```
* (describe *assertions)
(M19! (CLASS (M18 (LEX table))) (MEMBER B1))
(M21! (CLASS (M20 (LEX support))) (MEMBER B1))
(M23! (forall V2)
      (ANT (P1 (CLASS (M22 (LEX block))) (MEMBER V2))
      (CQ (P2 (CLASS (M20)) (MEMBER V2)))))
(M51! (CLASS (M22)) (MEMBER (M42 (LEX A))))
(M53! (CLASS (M22)) (MEMBER (M52 (LEX B))))
(M55! (CLASS (M22)) (MEMBER (M54 (LEX C))))
(M56! (OBJECT (M54)) (PROPERTY (M26 (LEX clear))))
(M57! (ARG1 (M54)) (ARG2 B1) (REL (M29 (LEX on))))
(M58! (OBJECT (M52)) (PROPERTY (M26)))
(M59! (ARG1 (M52)) (ARG2 B1) (REL (M29)))
(M60! (OBJECT (M42)) (PROPERTY (M26)))
(M61! (ARG1 (M42)) (ARG2 B1) (REL (M29)))
```

The propositions shown above can be directly translated from their English descriptions. In fact, the natural language component of the SNePS acting system does just that [Kumar *et al.* 1988, Kumar *et al.* 1990, Shapiro *et al.* 1989b]. The above propositions can also be expressed in the OK formalism notation. This is shown in Figure 10.2.

10.2 Physical Acts

We will define two primitive acts, PICKUP and PUT, and one complex act, PILE. The SNePS definitions for these acts are given below:

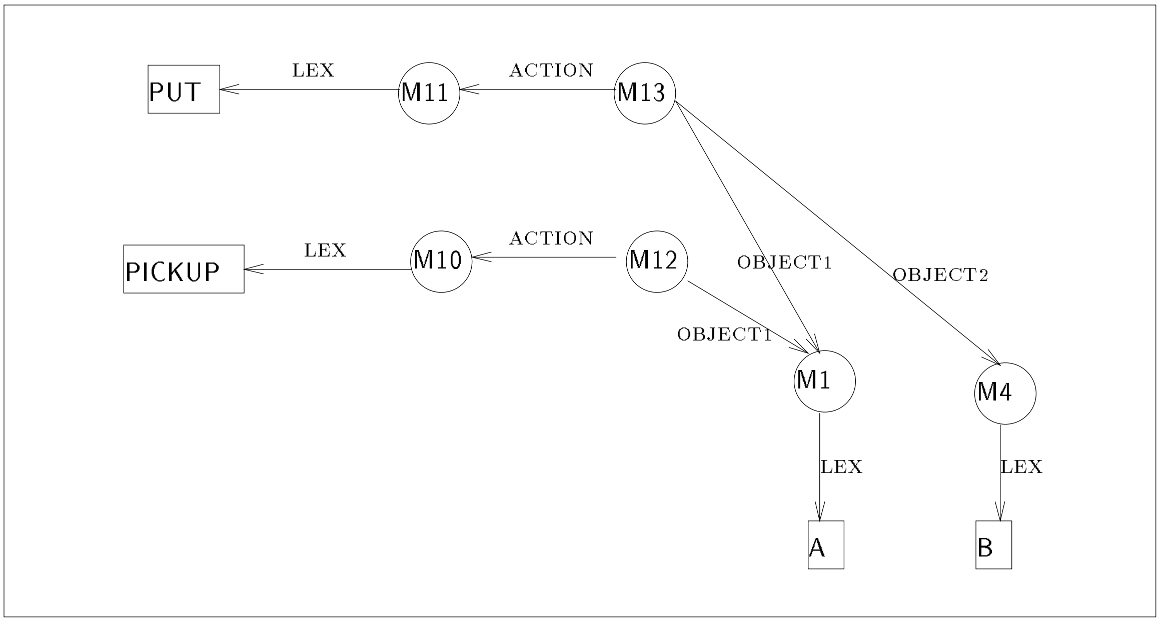

Figure 10.3: A SNePS network showing the representations of the acts PICKUP(A) and PUT(A, B).

Definition S25:

Entity: The physical act of picking up an object.

Syntax: Nodes representing the act of picking up an object have the following wire(s) in their cableset:

$\langle \mathsf{ACTION}, \{\langle \mathsf{LEX}, \mathsf{PICKUP}\rangle\}\rangle$**:**

$\langle \mathsf{OBJECT1}, \phi\rangle$**:** where ϕ is a node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \{\langle \mathsf{LEX}, \mathsf{PICKUP}\rangle\}\rangle, \langle \mathsf{OBJECT1}, \phi\rangle\}$$

is the physical act of picking up the entity represented by ϕ.

Examples: In Figure 10.3 the node M12 represents the act of picking up A.

Definition S26:

Entity: The physical act of putting an object on a support.

Syntax: Nodes representing the act of putting an object on a support have the following wire(s) in their cableset:

$\langle \mathsf{ACTION}, \{\langle \mathsf{LEX}, \mathsf{PUT}\rangle\}\rangle$**:**

$\langle \mathsf{OBJECT1}, \phi_1\rangle$**:** where ϕ_1 is a node.

$\langle \mathsf{OBJECT2}, \phi_2\rangle$**:** where ϕ_2 is a node.

Label: Proposition	Support
M24! : $\mathsf{Isa(PICKUP, PRIMITIVE)}$	$\langle \mathsf{HYP}, \{\mathsf{M24!}\}, \{\}\rangle$
M25! : $\mathsf{Isa(PICKUP, PRIMITIVE)}$	$\langle \mathsf{HYP}, \{\mathsf{M25!}\}, \{\}\rangle$
M27! : $\forall \mathsf{V2}[\mathsf{Isa(V2, BLOCK)} \rightarrow \mathsf{PreconditionAct(Clear(V2), PICKUP(V2))}]$	$\langle \mathsf{HYP}, \{\mathsf{M27!}\}, \{\}\rangle$
M28! : $\forall \mathsf{V2}[\mathsf{Isa(V2, BLOCK)} \rightarrow \mathsf{ActEffect(PICKUP(V2)}, \neg \mathsf{Clear(V2))}]$	$\langle \mathsf{HYP}, \{\mathsf{M27!}\}, \{\}\rangle$
M31! : $\forall \mathsf{V2V4}[\mathsf{Isa(V2, BLOCK)} \wedge \mathsf{Isa(V4, SUPPORT)} \wedge \mathsf{On(V2, V4)}$ $\rightarrow \mathsf{ActEffect(PICKUP(V2)}, \neg \mathsf{On(V2, V4))}]$	$\langle \mathsf{HYP}, \{\mathsf{M31!}\}, \{\}\rangle$
M32! : $\forall \mathsf{V2V4}[\mathsf{Isa(V2, BLOCK)} \wedge \mathsf{Isa(V4, SUPPORT)} \wedge \mathsf{On(V2, V4)}$ $\rightarrow \mathsf{ActEffect(PICKUP(V2)}, \neg \mathsf{Clear(V4))}]$	$\langle \mathsf{HYP}, \{\mathsf{M32!}\}, \{\}\rangle$
M34! : $\forall \mathsf{V2}[\mathsf{Isa(V2, BLOCK)} \rightarrow \mathsf{ActEffect(PICKUP(V2)}, \neg \mathsf{Held(V2))}]$	$\langle \mathsf{HYP}, \{\mathsf{M34!}\}, \{\}\rangle$
M35! : $\forall \mathsf{V2V4}[\mathsf{Isa(V2, BLOCK)} \wedge \mathsf{Isa(V4, SUPPORT)}$ $\rightarrow \mathsf{PreconditionAct(Held(V2), PUT(V2, V4))}]$	$\langle \mathsf{HYP}, \{\mathsf{M35!}\}, \{\}\rangle$
M36! : $\forall \mathsf{V2V4}[\mathsf{Isa(V2, BLOCK)} \wedge \mathsf{Isa(V4, SUPPORT)}$ $\rightarrow \mathsf{PreconditionAct(Clear(V4), PUT(V2, V4))}]$	$\langle \mathsf{HYP}, \{\mathsf{M36!}\}, \{\}\rangle$
M37! : $\forall \mathsf{V2V4}[\mathsf{Isa(V2, BLOCK)} \wedge \mathsf{Isa(V4, SUPPORT)}$ $\rightarrow \mathsf{ActEffect(PUT(V2, V4)}, \neg \mathsf{Held(V2))}]$	$\langle \mathsf{HYP}, \{\mathsf{M37!}\}, \{\}\rangle$
M38! : $\forall \mathsf{V2V4}[\mathsf{Isa(V2, BLOCK)} \wedge \mathsf{Isa(V4, SUPPORT)}$ $\rightarrow \mathsf{ActEffect(PUT(V2, V4), Clear(V2))}]$	$\langle \mathsf{HYP}, \{\mathsf{M38!}\}, \{\}\rangle$
M39! : $\forall \mathsf{V2V4}[\mathsf{Isa(V2, BLOCK)} \wedge \mathsf{Isa(V4, SUPPORT)}$ $\rightarrow \mathsf{ActEffect(PUT(V2, V4), On(V2, V4))}]$	$\langle \mathsf{HYP}, \{\mathsf{M39!}\}, \{\}\rangle$
M41! : $\forall \mathsf{V2V5}[\mathsf{Isa(V2, BLOCK)} \wedge \mathsf{Isa(V5, BLOCK)}$ $\rightarrow \mathsf{ActEffect(PUT(V2, V5)}, \neg \mathsf{Clear(V5))}]$	$\langle \mathsf{HYP}, \{\mathsf{M41!}\}, \{\}\rangle$

Figure 10.4: A belief space showing the beliefs of the agent about the preconditions and effects of the the acts PICKUP and PUT.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \{\langle \mathsf{LEX}, \mathsf{PUT}\rangle\}\rangle, \langle \mathsf{OBJECT1}, \phi_1\rangle, \langle \mathsf{OBJECT2}, \phi_2\rangle\}$$

is the physical act of putting the entity represented by ϕ_1 on top of the entity represented by ϕ_2.

Examples: In Figure 10.3 the node M13 represents the act of putting A on B.

The following sentences describe the agent's beliefs about these primitive acts:

> Picking up is a primitive action. Putting is a primitive action. Before picking up a block the block must be clear. After picking up a block the block is not 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 the 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 not held. 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.

Figure 10.4 shows the resulting propositions. Where possible, we will try to use the OK notation for expressing SNePS propositions. The act of piling three blocks is defined below:

Definition S27:

Entity: The physical act of piling three blocks on a table.

Syntax: Nodes representing the act of piling three block have the following wire(s) in their cableset:

$\langle \mathsf{ACTION}, \{\langle \mathsf{LEX}, \mathsf{PILE}\rangle\}\rangle$:

$\langle \mathsf{OBJECT1}, \phi_1\rangle$: where ϕ_1 is a node.

$\langle \mathsf{OBJECT2}, \phi_2\rangle$: where ϕ_2 is a node.

$\langle \mathsf{OBJECT3}, \phi_3\rangle$: where ϕ_3 is a node.

Semantics: The entity represented by the node

$$\{\langle \mathsf{ACTION}, \{\langle \mathsf{LEX}, \mathsf{PILE}\rangle\}\rangle, \langle \mathsf{OBJECT1}, \phi_1\rangle, \langle \mathsf{OBJECT2}, \phi_2\rangle, \langle \mathsf{OBJECT3}, \phi_3\rangle\}$$

is the physical act of piling the entity represented by ϕ_1 on top of the entity represented by ϕ_2 on top of the entity represented by ϕ_3.

10.3 Plans

In order to perform a complex act, the agent has to decompose it into a plan. This can be done via some planning algorithm. The agent can also be told generic plan decompositions which it can use in order to carry out complex acts. The following paragraph shows some plan decompositions:

> 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. A plan to pile a block on another block on a third block is to put the third block on the table and then put the second block on the third block and then put the first block on the second block.

The propositions that represent these sentences are shown in Figure 10.5. In the following chapters, whenever we refer to the blocksworld, we will assume the definitions presented in this chapter. We will present several examples of inference and acting. Unless otherwise specified, we will assume that the agent's belief space contains all of the propositions listed in Figures 10.2, 10.4, 10.5. We may also extend the domain by adding more propositions as well as acts.

Label: Proposition	Support
M43! : $\forall$V5[Isa(V5, BLOCK) $\rightarrow$ PlanGoal(PICKUP(V5), Held(V5))]	$\langle$HYP, {M43!}, {}$\rangle$
M44! : $\forall$V4V5[Isa(V5, BLOCK) $\wedge$ Isa(V4, SUPPORT) $\rightarrow$ PlanGoal(PUT(V5, V4), On(V4, V5))]	$\langle$HYP, {M44!}, {}$\rangle$
M45! : $\forall$V4V5[Isa(V5, BLOCK) $\wedge$ Isa(V4, SUPPORT) $\wedge$ On(V5, V4) $\rightarrow$ PlanGoal(SNSEQUENCE(PICKUP(V5), PUT(V5, B1)), Clear(V4))]	$\langle$HYP, {M45!}, {}$\rangle$
M50! : $\forall$V2V5V7[Isa(V2, BLOCK) $\wedge$ Isa(V5, BLOCK) $\wedge$ Isa(V7, BLOCK) $\rightarrow$ PlanAct(SNSEQUENCE(PUT(V7, B1), SNSEQUENCE(PUT(V2, V7), PUT(V5, V2))), PILE(V5, V2, V7))]	$\langle$HYP, {M50!}, {}$\rangle$

Figure 10.5: A belief space showing the beliefs of the agent about the some generic plans.

Chapter 11

SNeRE—The SNePS Rational Engine

In Chapter 6, we defined a rational engine for the OK architecture. In Chapter 8, we addressed concerns for implementing a concurrent rational engine. We presented a quasi-concurrent model using MULTI that is employed by SNIP, the SNePS Inference package. In this chapter, we present details of SNeRE, the SNePS Rational Engine, an extension of SNIP that incorporates acting as well as other transformations. Inference and acting in SNeRE is accomplished by message passing between nodes in the semantic network. In this chapter, we present details of SNeRE that constitute the extensions made to SNIP.

11.1 SNePSUL Commands

SNIP classifies messages as *requests* or *reports*. A request corresponds to a BELIEVE? message when it is sent to a proposition node and a TRANSFORM? message when it is sent to a transformer by a node in the consequent position. A report corresponds to a BELIEVE! message when sent to a proposition node and a TRANSFORM! message when sent to a transformer by a node in the antecedent position. In addition, SNeRE also incorporates the INTEND message, which is essentially a request to perform some act. At the user level, messages are sent to nodes in the network by the following SNePSUL commands:

(add {*relation nodeset*}* *context-specifier*)
: This is defined in the SNePS 2.1 User's Manual [Shapiro and Group 1989]. It "puts a node in the network with an arc labeled *relation* to each node in the *nodeset*. The node is asserted (as a hypothesis), added to the hypothesis set of the context specified by *context-specifier*, activated as a report, and triggers forward inference with a fresh set of MULTI queues."

(dynamic-add {*relation nodeset*}* *context-specifier*)
: Similar to add, it puts a node in the network with an arc labeled *relation* to each node in the *nodeset*. The node is asserted (as a hypothesis), added to the hypothesis set of the context specified by *context-specifier*, and activated as a report on the existing set of MULTI queues.

(deduce [*numb*] {*relation nodeset*}* *context-specifier*)

This is defined in the SNePS 2.1 User's Manual [Shapiro and Group 1989]. It "puts a node in the network with an arc labeled *relation* to each node in the *nodeset*. A user process is activated, as is the node. The process corresponding to the node is a request (i.e., starts a backward chaining inference in the specified context). The user process receives the answers and returns all nodes that satisfy the specification. The *numb* argument is optional. If *numb* is omitted, then deduce continues until no more answers can be derived. If *numb* is a single integer, it specifies the total number of answers requested. If *numb* is zero, no inference is done—only answers already in the network are returned. Otherwise, *numb* must be a list of two numbers, (*npos nneg*), and deduction terminates after at least *npos* positive and *nneg* negative instances are derived."

(perform *actnodeset*)

The *actnodeset* is a specification of an act node. The act node is activated as a process with priority INTEND and scheduled on the queue A so that the act can be performed as specified in the definition of SNeRE.

Thus, the deduce command corresponds to a BELIEVE? message; add and dynamic-add correspond to a BELIEVE! message; and perform corresponds to an INTEND message.

11.2 The Control Paradigm

The control paradigm employed for the rational engine is a quasi-concurrent message passing scheme as described in Chapter 8. Any time a node receives a message, it gets activated as a process and enters the MULTI loop, which was presented in Chapter 8 and is reproduced below:

while there is a non-empty queue loop
  choose the highest priority non-empty queue
  remove a process from it
  dispatch the process to a processor
endloop

We will use three queues—L (a low-priority FIFO queue), H (a high-priority FIFO queue), and A (the lowest-priority act queue). Of these, L and H were already defined and used by SNIP. All messages corresponding to *requests* get scheduled on L, messages corresponding to *reports* get scheduled on H, and act nodes receiving INTEND messages get scheduled on A.

11.3 Channels

In Chapter 8, we mentioned that each message contains the label for the sender as well as the receiver. In SNeRE, when a message is sent from one node to another, it is sent along a "channel". We define two types of channels that may exist between activations of any two nodes in the network:

Match Channels: When the molecular structure under two nodes unifies, we have a match channel between them.

Transformer Channels: The two nodes are connected to each other by a transformer.

Each channel is a 5-tuple consisting of a filter, switch, context, destination, and a valve. The filter and switch components of a channel take care of variable bindings, the context is the label of the context name, the destination is a node, and the valve can be open or closed for controlling inference (see [Hull 1986, Ferreira *et al.* 1989] for more details). In SNIP, the transformer channels are called *rule-use-channels*. Ideally, channels are set up on a permanent basis. Message passing over a channel is controlled by the valve. However, in the current implementation, channels are created on the fly, and processes are destroyed once a chain of inference is completed. Information about existing channels to or from a node is stored in process registers. These are presented next.

11.4 Processes and Registers

Activations of nodes are called processes. A process is an executable entity consisting of an *action* and a set of *registers* and *bindings* (representing the state of the computation). The action part of a process is simply a LISP function. Thus, dispatching a process to a processor corresponds to invoking the function corresponding to the action of the process to the process's bindings. Different types of nodes have different action functions as well as registers. SNIP defines three types of processes—*non-rule*, *rule*, and *user*. Non-rule processes correspond to activations of non-rule proposition nodes, rule processes correspond to activations of transformer proposition nodes, and user processes are used to accumulate results in order to communicate them to the user. In SNeRE, we additionally define *act* processes that correspond to activations of act nodes. Each type of process has a different set of registers, which are presented below.

11.4.1 Non-rule processes

Non-rule processes correspond to activations of non-rule proposition nodes. As defined in [Hull 1986, Ferreira *et al.* 1989], non-rule processes have the following registers along with descriptions of their bindings:

`*NAME*` the value of this register is the name of the action function (`NON-RULE`),

`*NODE*` the label of the node whose activation this process represents,

`*KNOWN-INSTANCES*` the collection of instances of this node (both positive and negative) that are currently asserted,

`*REPORTS*` reports received,

`*REQUESTS*` requests received,

`*INCOMING-CHANNELS*` the set of channels which will be feeding instance reports to this process,

Message	Response
BELIEVE?(p) (i.e., a request)	If p is asserted, send message BELIEVE!(p) to s. If p is not asserted then send message BELIEVE?(p) to all match channels and send TRANSFORM?(p) to all applicable transformer channels (CQ, IF), if any (Backward Chaining).
BELIEVE!(m) (i.e., a report)	If p = m then Assert it. If p is a transformer and the report was a response to a request then process earlier pending messages. Send BELIEVE!(m) to all requesters, all match channels, and send TRANSFORM!(p) to all applicable transformer channels (ANT, WHEN), if any (Forward Chaining).

Table 11.1: Message processing by proposition nodes (p is the receiver, and s the sender)

`*OUTGOING-CHANNELS*` the set of channels to which this node is to report instances that are discovered,

`*PENDING-FORWARD-INFERENCES*` temporary storage for those reports which contained instances which had not been requested,

`*PRIORITY*` is either `HIGH` or `LOW`.

Each process is identified by a process label, which is a symbol beginning with the letter p followed by some number. For example, a process labeled p405 may have the following bindings:

```
*PRIORITY* = SNIP:LOW
*PENDING-FORWARD-INFERENCES* = NIL
*OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN))
*INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P2 OPEN)
                       ((NIL) DEFAULT-DEFAULTCT V1 OPEN))
*REQUESTS* = NIL
*REPORTS* = NIL
*KNOWN-INSTANCES* = NIL
*NODE* = M6
*NAME* = SNIP::NON-RULE
```

The process shown above corresponds to an activation of the node labeled M6 (value of `*NODE*` register). The priority of the process is `LOW`, and it has an outgoing channel to a process, USER, to which it will send any reports that it receives. It also has two incoming channels, one from P2 and the other from V1. These two nodes may feed reports to it in the future.

When a non-rule node receives a message (i.e., it is activated, scheduled, and then finally executed), it determines the type of message (request or a report) and processes the reports first, followed by requests. These are described in detail in [Hull 1986, Ferreira *et al.* 1989]. Processing of a request is similar to the BELIEVE! message defined for the OK rational engine for the class Proposition Term. Processing a report is similar to the BELIEVE? method. Table 11.1 summarizes the message processing activity.

11.4.2 Rule-processes

Processes that are activations of transformers are called rule processes, since they denote inference rules (belief–belief transformers), reactive rules (WhenDo transformers), and DoWhen transformers. Rule processes have all the registers that are defined for non-rule processes (since transformers are also propositions!). The value of the `*NAME*` register is set to the function `RULE`. Additionally, as defined by [Hull 1986, Ferreira *et al.* 1989, Choi 1993], rule-processes also have the following transformer-related registers:

`*TYPE*` designates the type of transformer which can be one of *thresh, nor, and, and-or, or-entailment, and-entailment, num-quant, non-deriv, when-do, and do-if.*

`*RULE-USE-CHANNELS*` the set of outgoing channels which travel parallel to the rule arcs. For example, in the case of the *and-or*, they go to nodes considered to be in the consequent position with regard to the andor rule,

`*INTRODUCTION-CHANNELS*` the set of channels to raised hypotheses in a subproof during implication introduction,

`*RULE-HANDLER*` the function used to determine when enough information is available to draw a new conclusion,

`*USABILITY-TEST*` a function to determine whether or not an instance of the rule can be used based on the sign and the quantifier present.

For example, a process representing an activation of an or-entailment rule

$$\mathsf{M3!} : \forall \mathsf{V1}[\mathsf{Isa}(\mathsf{V1}, \mathsf{BLOCK}) \rightarrow \mathsf{Isa}(\mathsf{V1}, \mathsf{SUPPORT})]$$

represented in SNePS as

```
(M3! (FORALL V1)
     (ANT (P1 (CLASS (M1 (LEX BLOCK))) (MEMBER V1)))
     (CQ (P2 (CLASS (M2 (LEX SUPPORT))) (MEMBER V1))))
```

may have the following bindings:

```
*USABILITY-TEST* = SNIP::USABILITY-TEST.V-ENT
*RULE-HANDLER* = SNIP::RULE-HANDLER.V-ENT
*PRIORITY* = SNIP:LOW
*PENDING-FORWARD-INFERENCES* = NIL
```

Message	Response
BELIEVE?(p) (i.e., a request)	Sent by a node (p) over a CQ- or IF- arc (transformer channel). If t is asserted and it is a belief-belief transformer then send a request BELIEVE?(A_i) to all the antecedent nodes, (A_i's), over transformer channels (Backward Chaining). If t is asserted and it is a backward-chaining act-belief transformer then Send INTEND(α) to the act component α of the transformer. If t is not asserted and is a consequent of some belief-belief transformer then Send TRANSFORM?(t) to the transformer.
BELIEVE!(m) (i.e., a report)	If t is asserted and the report is unsolicited then Send a BELIEVE?(A_i) to all the remaining antecedents, A_i's. (starting a forward inference). If t is asserted and it is in response to an earlier request then If the firing criteria is satisfied then Apply the appropriate transformation ... If t is a belief-belief transformer then Send BELIEVE!(C_i) to all the consequents (C_i's) If it is a belief-act transformer then Send INTEND(α) to the act component, α If t is not asserted and is a consequent of some belief-belief transformer then send TRANSFORM?(t) to the transformer.

Table 11.2: Message processing by rule nodes (t is the rule node receiving the message).

```
*INTRODUCTION-CHANNELS* = NIL
*RULE-USE-CHANNELS* = NIL
*OUTGOING-CHANNELS* = NIL
*INCOMING-CHANNELS* = NIL
*REQUESTS* = ((((V1 . M4)) NIL DEFAULT-DEFAULTCT P2 OPEN))
*REPORTS* = NIL
*KNOWN-INSTANCES* = NIL
*NODE* = M3!
*TYPE* = SNIP::OR-ENTAILMENT
*NAME* = SNIP::RULE
```

The state of the process can be determined by looking at the values of its registers. For example, the above process has a low priority and has received a request from the node P2 (Isa(V1, SUPPORT)) with the binding {M4/V1}. When a rule node receives a request from a node in the consequent position (i.e., it is a TRANSFORM? message), the rule is first tested to see if it is applicable (by the function USABILITY-TEST.V-ENT in the case above). The rest proceeds as described in the various transformation methods in Chapter 6. Table 11.2 summarizes the abstract behavior of rule-processes.

11.4.3 Act processes

Act-processes are created when the agent has to perform an act (i.e., the act node receives an INTEND message). The modeled agent can perform an act in several situations—an explicit request by a user (using the perform command), firing of some belief-act transformer during forward chaining, or backchaining through some act-belief transformer. All act processes have the following registers:

`*NAME*` the name of the action function (ACT),

`*NODE*` the act node corresponding to this activation,

`*CONTEXT-NAME*` the context in which the act was intended,

`*REPORTS*` reports received,

`*PLANS*` known plans that decompose the act (if it is complex) in the current context,

`*PRECONDITIONS*` proposition believed to be the act's preconditions in the current context,

`*EFFECTS*` proposition nodes believed to be the act's effects in the current context,

`*AGENDA*` the current agenda for the act process (see below),

`*PRIORITY*` is set to the value INTEND (lowest priority).

For example, an act process for some node labeled M15, may have the following bindings:

```
*NAME* = SNIP::ACT
*NODE* = M15
*CONTEXT-NAME* = NIL
*REPORTS* = (M16! M17!)
*PLANS* = NIL
```

```
*PRECONDITIONS* = NIL
*EFFECTS* = NIL
*AGENDA* = SNIP::FIND-PRECONDITIONS
*PRIORITY* = SNIP::INTEND
```

The above process has just received a report. Given that the node's agenda (see below) is FIND-PRECONDITIONS, the reports must be the agent's beliefs about the act's preconditions. Thus, M16 and M17 are believed to be the preconditions of the act represented by M15 (i.e., M16 and M17 are PreconditionAct propositions).

Since the act queue A is the lowest priority queue, an act process gets executed only when there is nothing pending on the other two queues. This, as we explained in section 8.4.3, is because an intention to perform an act requires inference in order to determine preconditions, effects, and plans. Also, this conforms to the "think before you act" strategy. Whenever an intended act starts an inference, say to determine its preconditions, it receives a response only if the act is believed to have preconditions. This causes a synchronization problem—how does an act process "know" if the inference it had started has terminated (successfully or otherwise) so it can go on to the next step (say, apply precondition-act transformations) towards the fulfillment of the intention? For that purpose, we introduce the notion of an act's agenda.

11.4.4 Agendas

The agenda of an act process defines the act's current role towards the fulfillment of the intention to perform the act. The agenda of an act process is specified by the value of the `*AGENDA*` register which can be any of the following:

`START`: Signifies the beginning of an attempt by the agent to perform the act,

`FIND-PRECONDITIONS`: the agent is in the process of determining the act's preconditions,

`TEST-PRECONDITIONS`: the agent is testing to see if it believes the preconditions of the act,

`FIND-EFFECTS`: the agent is trying to determine the effects of the act,

`FIND-PLAN`: the agent is trying to find a plan decomposition for the act,

`EXECUTE`: the agent is ready to execute the actuator component of the act,

`DONE`: the intention to perform the act has been fulfilled.

When an act node (say α) receives an INTEND message, an act process is created (i.e., the act node is activated) with the agenda `START` and scheduled on the act queue. When the process is dispatched to a processor, its agenda helps define the agent's behavior. For instance, if it is `START`, the process initiates an inference denoted by the message

$$\mathsf{BELIEVE?}(\mathsf{PreconditionAct}(?\mathsf{p}, \alpha))$$

on the low priority queue, L. It also changes its agenda to `FIND-PRECONDITIONS` and reschedules itself on the act queue. Since L has a higher priority than A, the BELIEVE? message is dispatched next. If the inference succeeds in deriving some preconditions, a report is sent to the act process. The act process will be dispatched again when the inference queues are flushed (i.e., the inference has concluded). Since the act's agenda is FIND-PRECONDITIONS, the act process knows what to do next—look at the value of the *REPORTS* register and transfer its contents to the *PRECONDITIONS* register. Next, it should try to determine if its preconditions are satisfied. This constitutes inference(s) of the form

$$\mathsf{BELIEVE?}(\phi_{\mathsf{i}})$$

where $\phi_1 \ldots \phi_n$ are the preconditions of the act α. Once again, the act process reschedules itself on the act queue (after changing its agenda to TEST-PRECONDITIONS), and the inference is carried out.

11.4.5 Actuators for Primitive Actions

All primitive actions have corresponding procedures, called *actuators*, that get executed when an action is performed. The actuator is the agent's link to the state being affected by the act. Thus, when a physical act is performed, the actuator brings about changes in the physical world; when a mental act is performed, its actuator brings about a change in the belief space of the agent; and when a control action is performed, its actuator brings about a change in the agent's intentions. In this section, we describe the implementational details of associating actuators with their primitive actions. All actuators are basically LISP functions with a single argument—the node representing the act. They are defined using the following command:

(define-primaction *primaction act-node* & body *forms*)

Creates a single argument (*act-node*) function called *primaction* with the *forms* in its body.

For example, the actuator for PICKUP may be defined as follows:

```
(define-primaction pickup (pickup-act)
  "The actuator for the primitive action PICKUP"
  (let (; find the entity at the end of the OBJECT1 arc of pickup-act.
          (entity (choose.ns (pathfrom '(object1 lex) (node pickup-act)))))
     (format t ``~& Now doing:  Pickup ~A'' entity)
     ; code for directing the arm to pick up the object represented by entity
     (blocksworld-pickup (eval (node-na entity)))))
```

All control actions are also primitive. We will give a description of their actuators in the next section. Whenever the act process is ready to execute the actuator of a primitive action, it applies the corresponding actuator function on the act's node.

11.5 Acting Using Message Passing

The function ACT, corresponding to the act process that is executed when an act process is dispatched, follows the algorithm for the INTEND method given in Section 6.3, which is reproduced below:

```
Method
    INTEND(α : Act)
is
        Find preconditions ...
        PRECONDS ← BELIEVE?(PreconditionAct(?x,α))

        Test and achieve that the preconditions are believed ...
        loop
            exit loop when ⋀_{p∈PRECONDS} BELIEVE?(p)
            INTEND(DoALL((ACHIEVE(p_1) ... ACHIEVE(p_n)))), where n =| PRECONDS |
        endloop

        Find the believed effects of the act ...
        EFFECTS ← BELIEVE?(ActEffect(α,?x))

        Execute the act's actuator if it is primitive ...
        if BELIEVE?(Isa(α,PRIMITIVE)) then
            execute the actuator for α
            Believe the act's effects ...
            for each e ∈ EFFECTS loop
                TRANSFORM!(e)
            endloop
        else
            Derive a plan that decomposes the act ...
            PLANS ← BELIEVE?(PlanAct(?x,α))

            Execute one of the plans ...
            if PLANS then
                PLANS ← { PLAN(p_i) | ∀p_i ∈ PLANS}
                INTEND(DoONE(PLANS))
                for each e ∈ EFFECTS loop
                    TRANSFORM!(e)
                endloop
            else
                use a planning algorithm to derive a plan if possible
            endif
        endif
end INTEND
```

We have included the skeptical version of achieving the preconditions (see Section 6.3). We have also annotated the algorithm by some comments. As explained above, the agendas of an act process determine the progress of the fulfillment of the intention to perform the act. The ACT function that implements the above algorithms can be described as follows:

```
IF *NODE* is a control action THEN
        Execute its actuator
ELSE
        CASE *AGENDA* OF
                START
                        *AGENDA* ← FIND-PRECONDITIONS
                        Schedule the act process on the act queue
                        Schedule a request to PreconditionAct(?p, *NODE*) on
                                the low priority queue
                FIND-PRECONDITIONS
```

```
11.                 IF any reports received in *REPORTS* THEN
12.                     process them into *PRECONDITIONS*
13.                     *AGENDA* ← TEST-PRECONDITIONS
14.                     Schedule the act process on the act queue
15.                     Schedule a request to all the act's preconditions
16.                 ELSE
17.                     *AGENDA* ← FIND-EFFECTS
18.                     Schedule the act process on the act queue
19.                     Schedule a request to ActEffect(*NODE*, ?e)
20.                 ENDIF
21.         TEST-PRECONDITIONS
22.                 IF *REPORTS* tally with the preconditions THEN
23.                     *AGENDA* ← FIND-EFFECTS
24.                     Schedule the act process on the act queue
25.                     Schedule a request to ActEffect(*NODE*, ?e)
26.                 ELSE
27.                     *AGENDA* ← START
28.                     Schedule the act process on the act queue
29.                     Schedule the act DoALL(ACHIEVE(*PRECONDITIONS*))
30.                 ENDIF
31.         FIND-EFFECTS
32.                 IF any reports received in *REPORTS* THEN
33.                     process them into *EFFECTS*
34.                     Schedule the act DoALL(BELIEVE(*EFFECTS*))
35.                 ENDIF
36.                 *AGENDA* ← EXECUTE
37.                 Schedule the act process on the act queue
38.         EXECUTE
39.                 IF the act is primitive THEN
40.                     Execute the act's actuator
41.                 ELSE
42.                     *AGENDA* ← FIND-PLANS
43.                     Schedule the act process on the act queue
44.                     Schedule a request to PlanAct(?p, *NODE*)
45.                 ENDIF
46.         FIND-PLANS
47.                 IF any reports received in *REPORTS* THEN
48.                     process them into *PLANS*
49.                     Schedule the act DoONE(*PLANS*) on the act queue
50.                 ELSE
51.                     Use a classical planning algorithm to find plans
52.                 ENDIF
53.     ENDCASE
54. ENDIF
```

If an act is a control action, it is directly dispatched to its corresponding actuator (lines 1–2). This is because we do not need to deliberate about control actions the same way we deliberate about physical actions. For example, there is no need to do anything for a SEQUENCE act other than scheduling the two object acts on the act queue in the proper order. Control acts do not, typically, have any specific beliefs about their preconditions and effects. Control acts that do need to deliberate (for instance identifying satisfied guards for guarded acts) may require additional agendas for their processes. The actuators for control actions are presented later in this section.

All the deliberations carried out in order to perform a physical act are described in lines 3–54 above. Each time an act process is dispatched, its agenda determines the next task that needs to be carried out. Lines 5–9 describe the tasks that are carried out for act processes with an agenda START—the agenda is changed to FIND-PRECONDITIONS, the act is rescheduled on the act queue, and a request is sent to find out the act's preconditions. As described earlier, since the act queue has the lowest priority, the inference will conclude before the act is dispatched again. The next time the act is dispatched, it has the agenda FIND-PRECONDITIONS (see lines 10–20). Any results of the inference will be found in the *REPORTS* register. The reports are processed into the *PRECONDITIONS* register, the agenda of the act process is changed to TEST-PRECONDITIONS, and the act is rescheduled, followed by the scheduling of a request to see if the act's preconditions are believed (lines 11–15). In case the act did not receive any reports (lines 16–20), the act's agenda is changed to FIND-EFFECTS, and it is rescheduled, followed by the scheduling of a request to find the act's effects.

When the act process gets dispatched after the inference to determine if its preconditions are believed (lines 21–30), the reports are tallied with the preconditions. If the act's preconditions are believed, its agenda is changed to FIND-EFFECTS, it is rescheduled, and inference to find out its effects begins (lines 22–25). In case the reports do not tally (lines 26–30), we reset the act's agenda to START, reschedule it, and schedule the act of achieving all of the act's preconditions. This corresponds to a backward chaining belief-act transformation. The next time the act is attempted, it will be a fresh start.

Upon conclusion of the inference issued to determine the act's effects, the act process is dispatched with the agenda FIND-EFFECTS. Lines 31–37 specify that if the act has any effects, the act of believing them is scheduled. This corresponds to a forward chaining act-belief transformation. Subsequently, the act's agenda is set to EXECUTE, and the act is rescheduled. At this moment, the act queue will have at least two acts—the intended act on the front followed by the act of believing all its effects.

When an act process is dispatched with the agenda EXECUTE, the agent has determined the act's preconditions and that they are satisfied, and it has also scheduled the act of believing all of the act's effects. Before the act's effects are executed, the agent has to determine if the act is primitive (lines 39–40), in which case its actuator is executed. If the act is a complex act (lines 41–45), its agenda is set to FIND-PLANS and is rescheduled, followed by a request to a PlanAct transformer to determine any derivable plan decompositions for it. If any decompositions are found, an act to perform one of them is scheduled, else a classical planning algorithm can be employed to derive a plan (lines 47–52). This completes a description of the acting component. Actuators for control acts are presented next.

11.6 Control Acts

As mentioned earlier, all control acts are also primitive acts. In this section, we present the details of their actuators. Some control acts have simple actuators in that they do not need to do any deliberations. Control acts that require some deliberation have additional agendas defined for them.

Sequencing Acts:

The actuator for a sequencing act is similar to its INTEND method defined in Section 7.4.1:

Schedule the act at the end of OBJECT2 *arc on the act queue*
Schedule the act at the end of OBJECT1 *arc on the act queue*

Since the act queue is a LIFO queue, the next act to be dispatched would be the act at the end of the OBJECT1 arc of the sequencing act.

Nondeterministic Choice and Ordering:

The actuator for DoONE acts can be defined as follows:

Randomly pick one of the object acts and schedule it on the act queue

Similarly, the DoALL act can be implemented as follows:

Schedule all of the object acts on the act queue

The above implements a nondeterministic ordering. It is possible to implement these acts using an "eager-beaver" strategy. If all the object acts of a DoALL are physical acts, the actuator could be made to schedule the object acts as well as processes for determining the preconditions of those acts. Thus, acts whose preconditions get determined would be performed on a first-come, first-served basis. Similarly, in the case of a DoONE act, the object act performed would be the one whose preconditions were already achieved. We have chosen to implement the simple strategy shown above for two reasons—it is not guaranteed that all of the object acts would be physical acts; the eager-beaver strategy forces a deterministic interpretation on the DoONE and DoALL acts. There are domains where other factors may determine the choice—cost, complexity, etc. Such acts are conceptually different from pure nondeterministic versions of the acts presented. They should be defined and implemented separately.

Conditional and Iterative Acts:

Both SNIF and SNITERATE need to deliberate about the guards of their guarded object acts. The actuators for these acts employ two agendas:

START Signifies the beginning of the attempt by the agent to perform the act.

TEST The agent is trying to determine the guards that are believed.

The actuator for SNIF is defined as follows:

```
CASE *AGENDA* OF
    START
        *AGENDA* ← TEST
        Schedule the act process on the act queue
        GUARDS ← propositions that are the guards of all the object acts
```

```
6.          FOREACH G ∈ GUARDS LOOP
7.              Schedule a request to see if G is believed
8.          ENDLOOP
9.      TEST
10.         *AGENDA* ← DONE
11.         IF any reports received in *REPORTS* THEN
12.             initialize ACTS ← {}
13.             FOREACH G ∈ *REPORTS* LOOP
14.                 ACTS ← ACTS ∪ the act whose guard is G
15.             ENDLOOP
16.             Schedule the act DoONE(ACTS) on the act queue
17.         ENDIF
18. ENDCASE
```

The actuator for SNITERATE is similar except line 16 is replaced with the following:

```
16.             Schedule the act SEQUENCE(DoONE(ACTS), *NODE*)
```

Achieve Acts:

The actuator for ACHIEVE also employs two agendas—START and FIND-PLANS; it is defined as follows:

```
CASE *AGENDA* OF
    START
        IF *NODE* is asserted THEN
            RETURN (it is already achieved)
        ELSE
            *AGENDA* ← FIND-PLANS
            Schedule the act process on the act queue
            Schedule a request to PlanGoal(?p, OBJECT1(*NODE*))
        ENDIF
    FIND-PLANS
        IF any reports received THEN
            process them into *PLANS*
            Schedule the act DoONE(*PLANS*) on the act queue
        ELSE
            Use a classical planning algorithm to find plans
        ENDIF
ENDCASE
```

Qualifier Acts:

The following agendas are defined for qualified acts:

START Signifies the beginning of the attempt by the agent to perform the act.

IDENTIFY-OBJECTS The act is trying to identify specific objects given their descriptions.

The actuator for WITHSOME is then defined as follows:

In the following, D is the node at the end of the SUCHTHAT arc,
and A is the node at the end of the DO arc.

```
1.  CASE *AGENDA* OF
2.          START
3.                  *AGENDA* ← IDENTIFY-OBJECTS
4.                  Schedule the act process on the act queue
5.                  Schedule a request to the designator node (D) of the act
6.          IDENTIFY-OBJECTS
7.                  IF any reports received in *REPORTS* THEN
8.                          *AGENDA* ← DONE
9.                          ACTS ← {}
10.                         FOREACH R ∈ *REPORTS* LOOP
11.                                 σ ← UNIFY(D, R)
12.                                 ACTS ← ACTS ∪ Aσ
13.                         ENDLOOP
14.                         Schedule the act DoONE(ACTS) on the act queue
15.                 ENDIF
16. ENDCASE
```

The actuator for the WITHALL act differs from the above in line 14 which reads:

```
14.                         Schedule the act DoALL(ACTS) on the act queue
```

11.7 From Beliefs to Intentions

SNIP has also been extended to perform forward chaining belief-act transformations using the WhenDo transformer. A WhenDo transformer is represented by a node structure defined in Definition S18 (see Chapter 9). An activation of a node representing a WhenDo transformer is also a rule process. When a node at the end of a WHEN arc receives a report, it forwards it to the transformer node. If the transformer is asserted, and its usability test succeeds, its rule-handler function is invoked. The function applies the bindings obtained from the report to the act at the end of the DO arc to instantiate a specific reaction that gets scheduled on the act queue, thus completing a forward chaining belief-act transformation. The extended description of the rule process is already summarized in Table 11.2. We will show examples of use of this transformer in the next chapter.

11.8 From Intentions to Beliefs

SNeRE also includes the rule process corresponding to the DoWhen transformer. The node, defined in Chapter 9, Definition S21, also gets activated as a rule process. When it receives a request from a node at the end of the IF arc, it schedules an intention to perform an instance of the act obtained by applying the substitutions sent through the request. This is also summarized in Table 11.2. We will see examples in the next chapter.

11.9 Discussion

In this chapter, we have presented details of SNeRE, which is an extension of SNIP. SNeRE is the rational engine for the SNePS BDI architecture, which is depicted in Figure 11.1. The

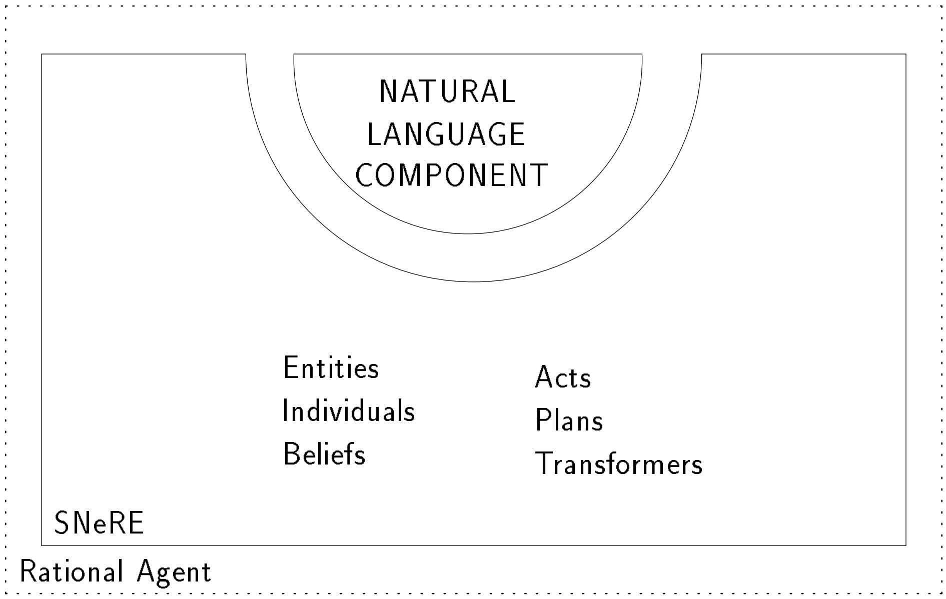

Figure 11.1: A Schematic of the SNePS BDI Architecture-based rational agent.

result is an architecture that uses a uniform representational formalism for representing entities, individuals, beliefs, acts, plans, and transformers. The unified view of acting and inference is implemented in the rational engine. The rational engine is implemented using a quasi-concurrent control paradigm. The rational engine is responsible for inference, acting, plan decomposition, reacting, and knowledge acquisition. The next chapter is devoted to examples of a rational agent that exhibits these faculties.

Chapter 12

Examples

In this chapter, we will present several examples illustrating various capabilities of a rational cognitive agent modeled using the SNePS BDI architecture. We have already shown how to represent a blocksworld agent's beliefs, acts, and plans (see Chapter 10). In what follows, we will assume that the agent already has the beliefs listed in Figures 10.2, 10.4, 10.5.

12.1 Asking Queries—Inference

The agent can be queried about its beliefs at any point. The following is an example of a query about a belief the agent already has:

```
; Is A clear?
* (deduce object (build lex "A")
          property (build lex "clear"))

I know
((M54! (OBJECT (M36 (LEX (A))))
       (PROPERTY (M20 (LEX (clear))))))

(M54!)

 CPU time : 0.05     GC time : 0.00
*
```

The message generated by deduce was a request to the node M54, which responded with a report, since it is asserted. Queries to the agent can lead to backward chaining inference, as depicted below:

```
; How would you pile B on C on A?
* (deduce act (build action (build lex "pile")
                     object1 (build lex "B")
                     object2 (build lex "C")
                     object3 (build lex "A"))
          plan $plan)

I wonder if
```

```
((P313
  (ACT
   (M56 (ACTION (M40 (LEX (pile)))) (OBJECT1 (M46 (LEX (B))))
    (OBJECT2 (M48 (LEX (C)))) (OBJECT3 (M36 (LEX (A))))))
  (PLAN V12)))
holds within the BS defined by context DEFAULT-DEFAULTCT
```

At this stage a request has been sent to the pattern representing query. The request gets forwarded to the consequent of the rule represented by M44 (see Figure 10.5). The consequent requests the rule to perform the transformation, which in turn sends requests to all its antecedents:

```
I wonder if
((P200 (CLASS (M14 (LEX (block)))) (MEMBER (V5 <-- M46))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P298 (CLASS (M14 (LEX (block)))) (MEMBER (V7 <-- M36))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P9 (CLASS (M14 (LEX (block)))) (MEMBER (V2 <-- M48))))
holds within the BS defined by context DEFAULT-DEFAULTCT
```

Specific instances of the antecedents respond by sending reports:

```
I know
((M47! (CLASS (M14 (LEX (block)))) (MEMBER (M46 (LEX (B))))))

I know
((M45! (CLASS (M14 (LEX (block)))) (MEMBER (M36 (LEX (A))))))

I know
((M49! (CLASS (M14 (LEX (block)))) (MEMBER (M48 (LEX (C))))))
```

The reports are tallied by the rule, and the rule fires; i.e., a plan has been derived:

```
Since
((M44! (FORALL (V2 <-- M48) (V5 <-- M46) (V7 <-- M36))
  (&ANT (P200 (CLASS (M14 (LEX (block)))) (MEMBER (V5 <-- M46)))
   (P298 (CLASS (M14 (LEX (block)))) (MEMBER (V7 <-- M36)))
   (P9 (CLASS (M14 (LEX (block)))) (MEMBER (V2 <-- M48))))
  (CQ
   (P297
    (ACT
     (P286 (ACTION (M40 (LEX (pile)))) (OBJECT1 (V5 <-- M46))
      (OBJECT2 (V2 <-- M48)) (OBJECT3 (V7 <-- M36))))
    (PLAN
     (P296 (ACTION (M1 (LEX (snsequence))))
      (OBJECT1
       (P295 (ACTION (M1 (LEX (snsequence))))
        (OBJECT1
```

```
        (P289 (ACTION (M18 (LEX (put)))) (OBJECT1 (V7 <-- M36))
         (OBJECT2 (B1))))
       (OBJECT2
        (P291 (ACTION (M18 (LEX (put)))) (OBJECT1 (V2 <-- M48))
         (OBJECT2 (V7 <-- M36))))))
     (OBJECT2
      (P293 (ACTION (M18 (LEX (put)))) (OBJECT1 (V5 <-- M46))
       (OBJECT2 (V2 <-- M48))))))))))
and
((P200 (CLASS (M14 (LEX (block)))) (MEMBER (V5 <-- M46))))
and
((P298 (CLASS (M14 (LEX (block)))) (MEMBER (V7 <-- M36))))
and
((P9 (CLASS (M14 (LEX (block)))) (MEMBER (V2 <-- M48))))
I infer
((P297
  (ACT
   (P286 (ACTION (M40 (LEX (pile)))) (OBJECT1 (V5 <-- M46))
    (OBJECT2 (V2 <-- M48)) (OBJECT3 (V7 <-- M36))))
  (PLAN
   (P296 (ACTION (M1 (LEX (snsequence))))
    (OBJECT1
     (P295 (ACTION (M1 (LEX (snsequence))))
      (OBJECT1
       (P289 (ACTION (M18 (LEX (put)))) (OBJECT1 (V7 <-- M36))
        (OBJECT2 (B1))))
      (OBJECT2
       (P291 (ACTION (M18 (LEX (put)))) (OBJECT1 (V2 <-- M48))
        (OBJECT2 (V7 <-- M36))))))
    (OBJECT2
     (P293 (ACTION (M18 (LEX (put)))) (OBJECT1 (V5 <-- M46))
      (OBJECT2 (V2 <-- M48))))))))

(M62!)

 CPU time : 2.22     GC time : 0.00
```

The derived plan is represented by the node M62, which is asserted in the agent's belief space as follows:

```
(M62!
 (ACT
  (M56 (ACTION (M40 (LEX pile))) (OBJECT1 (M46 (LEX B)))
   (OBJECT2 (M48 (LEX C))) (OBJECT3 (M36 (LEX A)))))
 (PLAN
  (M61 (ACTION (M1 (LEX snsequence)))
   (OBJECT1
    (M59 (ACTION (M1))
     (OBJECT1 (M57 (ACTION (M18 (LEX put))) (OBJECT1 (M36)) (OBJECT2 B1)))
     (OBJECT2 (M58 (ACTION (M18)) (OBJECT1 (M48)) (OBJECT2 (M36))))))
   (OBJECT2 (M60 (ACTION (M18)) (OBJECT1 (M46)) (OBJECT2 (M48))))))
 (DER ((ASSERTIONS (M44! M45! M47! M49!)) (RESTRICTION NIL) (NAMED NIL))))
```

It is a derived proposition and has the underlying assumptions M44!, M45!, M47!, M49!. The node M62 represents a plan decomposition of the act of making a pile using the blocks B, C, and A. I.e.,

$$\mathsf{M62!} : \mathsf{PlanAct}(\mathsf{SEQUENCE}(\mathsf{SEQUENCE}(\mathsf{PUT}(\mathsf{A}, \mathsf{TABLE}),\ \mathsf{PUT}(\mathsf{C}, \mathsf{A})),\ \mathsf{PUT}(\mathsf{B}, \mathsf{C})), \mathsf{PILE}(\mathsf{B},\ \mathsf{C},\ \mathsf{A})) \quad \langle \mathsf{DER}, \{\mathsf{M44!\ M45!\ M47!\ M49!}\}, \{\}\rangle$$

The derived proposition has been asserted in the agent's belief space. Thus, the next time the agent is asked the same query, it will be answered without the need for carrying out any inference:

```
; How would you pile B on C on A?
* (deduce act (build action (build lex "pile")
                     object1 (build lex "B")
                     object2 (build lex "C")
                     object3 (build lex "A"))
          plan $plan)

I wonder if
((P315
  (ACT
   (M56 (ACTION (M40 (LEX (pile)))) (OBJECT1 (M46 (LEX (B))))
    (OBJECT2 (M48 (LEX (C)))) (OBJECT3 (M36 (LEX (A))))))
  (PLAN V14)))
holds within the BS defined by context DEFAULT-DEFAULTCT

I know
((M62!
  (ACT
   (M56 (ACTION (M40 (LEX (pile)))) (OBJECT1 (M46 (LEX (B))))
    (OBJECT2 (M48 (LEX (C)))) (OBJECT3 (M36 (LEX (A))))))
  (PLAN
   (M61 (ACTION (M1 (LEX (snsequence))))
    (OBJECT1
     (M59 (ACTION (M1 (LEX (snsequence))))
      (OBJECT1
       (M57 (ACTION (M18 (LEX (put)))) (OBJECT1 (M36 (LEX (A))))
        (OBJECT2 (B1))))
      (OBJECT2
       (M58 (ACTION (M18 (LEX (put)))) (OBJECT1 (M48 (LEX (C))))
        (OBJECT2 (M36 (LEX (A))))))))
    (OBJECT2
     (M60 (ACTION (M18 (LEX (put)))) (OBJECT1 (M46 (LEX (B))))
      (OBJECT2 (M48 (LEX (C))))))))))

(M62!)

 CPU time : 0.77     GC time : 0.00
*
```

Thus, the deductive efficiency property of SNIP [Kumar and Shapiro 1993] is still preserved. M62 will continue to be asserted in the agent's belief space as long as its supporting

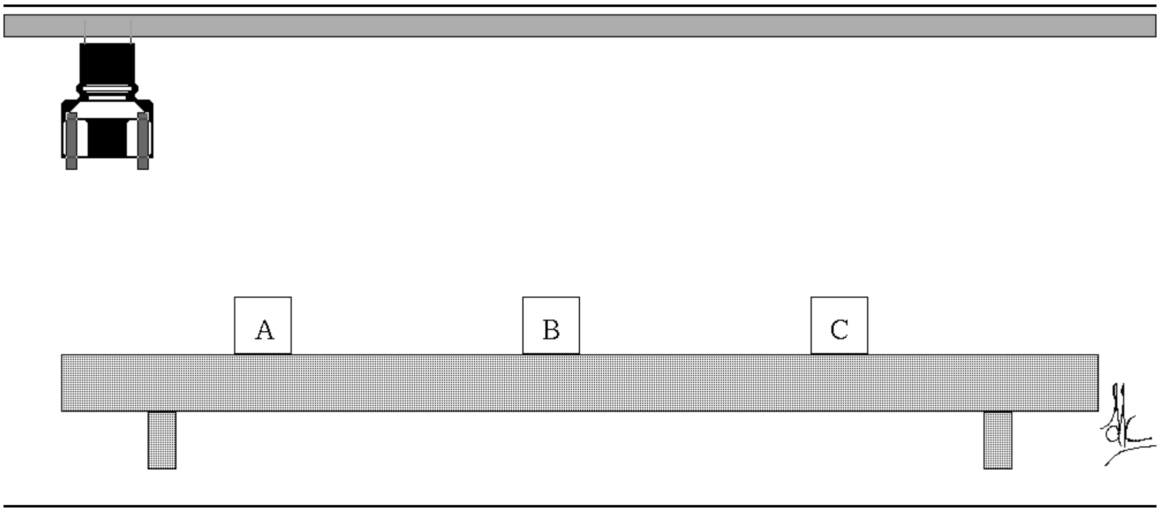

Figure 12.1: The blocksworld.

assumptions M44!, M45!, M47!, and M49! are also asserted. Using a belief held using a DoWhen transformer, it is possible for backward chaining to lead to an intention to perform an action. This will be presented later. First we present examples of fulfilling intentions.

12.2 Acting

There are several ways an agent can form an intention to do some act—it could be an explicit request from a user (using perform), an act-precondition transformation in order to fulfill a precondition of some already intended act, an act-effect transformation in order to believe the effects of some recently fulfilled intention, a plan-act transformation leading to the intention to carry out a plan, a reaction to some situation, or an act to acquire new information in order to answer a query. We will present examples of acting in all these situations. The agent can be explicitly asked to perform an act by a user as shown below:

```
; Pick up C.
* (perform (build action (build lex "pickup")
                  object1 (build lex "C")))

About to do
((M56 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M48 (LEX (C))))))
```

The act (represented by the node M56) gets activated and scheduled on the act queue. When the act process gets dispatched, the first thing it has to do is determine the act's preconditions (if it is a physical act). This is shown in the next section.

12.2.1 Precondition-Act Transformations

We will continue with the example showing how an act's preconditions are derived. The act's agenda is changed to FIND-PRECONDITIONS; it is rescheduled on the act queue; and a message is sent to the proposition, PreconditionAct(?p, PICKUP(C)), which is represented by the node P313 below:

```
I wonder if the act
((M56 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M48 (LEX (C))))))

has any preconditions...
I wonder if
((P313 (ACT (M56 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M48 (LEX (C))))))
  (PRECONDITION V12)))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P9 (CLASS (M14 (LEX (block)))) (MEMBER (V2 <-- M48))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I know
((M49! (CLASS (M14 (LEX (block)))) (MEMBER (M48 (LEX (C))))))

Since
((M21! (FORALL V2) (ANT (P9 (CLASS (M14 (LEX (block)))) (MEMBER V2)))
  (CQ
   (P10 (ACT (P6 (ACTION (M16 (LEX (pickup)))) (OBJECT1 V2)))
    (PRECONDITION (P8 (OBJECT V2) (PROPERTY (M20 (LEX (clear))))))))))
and
((P9 (CLASS (M14 (LEX (block)))) (MEMBER (V2 <-- M48))))
I infer
((P10 (ACT (P6 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (V2 <-- M48))))
  (PRECONDITION (P8 (OBJECT (V2 <-- M48)) (PROPERTY (M20 (LEX (clear))))))))
```

As we can see above, the rule represented by the node M21 performs a belief-belief transformation to derive the precondition-act proposition, which is represented by the node M57. This is reported to the act process:

```
The act
((M56 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M48 (LEX (C))))))

has a precondition:
((M57! (ACT (M56 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M48 (LEX (C))))))
  (PRECONDITION
   (M50! (OBJECT (M48 (LEX (C)))) (PROPERTY (M20 (LEX (clear))))))))
```

Next, the act process saves the derived precondition in its *PRECONDITIONS* register. It changes its agenda to TEST-PRECONDITIONS, reschedules itself, and sends a request to the node M50 to confirm that the precondition is satisfied. At this stage, the various registers of the act process have the following contents:

```
*PRIORITY* = SNIP::INTEND
*AGENDA* = SNIP::TEST-PRECONDITIONS
*PLANS* = NIL
*EFFECTS* = NIL
*PRECONDITIONS* = (M57!)
*REPORTS* = NIL
*CONTEXT-NAME* = NIL
*NODE* = M56
*NAME* = SNIP::ACT
```

In a situation where the preconditions are already satisfied (as is the case in the example shown), the act process will receive the reports and go on to find the effects of the action.

```
I know
((M50! (OBJECT (M48 (LEX (C)))) (PROPERTY (M20 (LEX (clear))))))

It is satisfied.
```

In situations where the preconditions are not satisfied, a precondition-act transformation results in the agent intending to do the act of achieving them. For example, consider the request below:

```
; Put A on B.
* (perform (build action (build lex "put")
                  object1 (build lex "A")
                  object2 (build lex "B")))

About to do
((M75 (ACTION (M18 (LEX (put)))) (OBJECT1 (M36 (LEX (A))))
  (OBJECT2 (M46 (LEX (B))))))

I wonder if the act
((M75 (ACTION (M18 (LEX (put)))) (OBJECT1 (M36 (LEX (A))))
  (OBJECT2 (M46 (LEX (B))))))

has any preconditions...

...
The act
((M75 (ACTION (M18 (LEX (put)))) (OBJECT1 (M36 (LEX (A))))
  (OBJECT2 (M46 (LEX (B))))))

has the following preconditions:
((M77!
  (ACT
   (M75 (ACTION (M18 (LEX (put)))) (OBJECT1 (M36 (LEX (A))))
    (OBJECT2 (M46 (LEX (B))))))
  (PRECONDITION
   (M52! (OBJECT (M46 (LEX (B)))) (PROPERTY (M20 (LEX (clear)))))))
 (M78!
  (ACT
   (M75 (ACTION (M18 (LEX (put)))) (OBJECT1 (M36 (LEX (A))))
    (OBJECT2 (M46 (LEX (B))))))
  (PRECONDITION (M76 (OBJECT (M36 (LEX (A)))) (PROPERTY (M27 (LEX (held))))))))

I know
((M52! (OBJECT (M46 (LEX (B)))) (PROPERTY (M20 (LEX (clear))))))

I wonder if
((M76 (OBJECT (M36 (LEX (A)))) (PROPERTY (M27 (LEX (held))))))
holds within the BS defined by context DEFAULT-DEFAULTCT

They are not satisfied.
```

This results in precondition–act transformations—since one of the act's preconditions is not satisfied, the agent schedules to perform the act of achieving *all* of the act's preconditions.

```
Intending to do
((M82 (ACTION (M73 (LEX (do-all))))
  (OBJECT1
   (M80 (ACTION (M4 (LEX (achieve))))
    (OBJECT1 (M52! (OBJECT (M46 (LEX (B)))) (PROPERTY (M20 (LEX (clear)))))))
   (M81 (ACTION (M4 (LEX (achieve))))
    (OBJECT1 (M76 (OBJECT (M36 (LEX (A)))) (PROPERTY (M27 (LEX (held)))))))))

Now doing: DO-ALL
((M80 (ACTION (M4 (LEX (achieve))))
  (OBJECT1 (M52! (OBJECT (M46 (LEX (B)))) (PROPERTY (M20 (LEX (clear)))))))
 (M81 (ACTION (M4 (LEX (achieve))))
  (OBJECT1 (M76 (OBJECT (M36 (LEX (A)))) (PROPERTY (M27 (LEX (held))))))))
```

At this point, the intention to perform the act of putting A on B is still on the act queue. The precondition-act transformation has led the agent to form new intentions. Once these intentions are fulfilled, the the act of putting A on B will again be attempted, with its agenda set to START. Let us return to the earlier example of picking up C where the agent had determined the act's precondition and that it was satisfied. The next task for the act process is to determine the act's effects, i.e., the act-effect transformers.

12.2.2 Act-Effect Transformations

For determining the effects of the act of picking up C, the act process first sets its agenda to FIND-EFFECTS, reschedules itself, and activates the node representing the pattern Act-Effect(PICKUP(C), ?e):

```
I wonder if
((P314 (ACT (M56 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M48 (LEX (C))))))
  (EFFECT V13)))
holds within the BS defined by context DEFAULT-DEFAULTCT

Since
((M22! (FORALL V2) (ANT (P9 (CLASS (M14 (LEX (block)))) (MEMBER V2)))
  (CQ
   (P21 (ACT (P6 (ACTION (M16 (LEX (pickup)))) (OBJECT1 V2)))
    (EFFECT
     (P20 (MIN 0) (MAX 0)
      (ARG (P8 (OBJECT V2) (PROPERTY (M20 (LEX (clear))))))))))))
and
((P9 (CLASS (M14 (LEX (block)))) (MEMBER (V2 <-- M48))))
I infer
((P21 (ACT (P6 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (V2 <-- M48))))
  (EFFECT
   (P20 (MIN 0) (MAX 0)
    (ARG (P8 (OBJECT (V2 <-- M48)) (PROPERTY (M20 (LEX (clear))))))))))
```

```
Since
((M28! (FORALL V2) (ANT (P9 (CLASS (M14 (LEX (block)))) (MEMBER V2)))
  (CQ
   (P95 (ACT (P6 (ACTION (M16 (LEX (pickup)))) (OBJECT1 V2)))
    (EFFECT (P94 (OBJECT V2) (PROPERTY (M27 (LEX (held))))))))))
and
((P9 (CLASS (M14 (LEX (block)))) (MEMBER (V2 <-- M48))))
I infer
((P95 (ACT (P6 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (V2 <-- M48))))
  (EFFECT (P94 (OBJECT (V2 <-- M48)) (PROPERTY (M27 (LEX (held))))))))

I wonder if
((P46 (ARG1 (V2 <-- M48)) (ARG2 V4) (REL (M23 (LEX (on))))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P48 (CLASS (M12 (LEX (support)))) (MEMBER V4)))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P47 (MIN 0) (MAX 0)
  (ARG (P46 (ARG1 (V2 <-- M48)) (ARG2 V4) (REL (M23 (LEX (on))))))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I know
((M51! (ARG1 (M48 (LEX (C)))) (ARG2 (B1)) (REL (M23 (LEX (on))))))

I know
((M49! (CLASS (M14 (LEX (block)))) (MEMBER (M48 (LEX (C))))))

I know
((M47! (CLASS (M14 (LEX (block)))) (MEMBER (M46 (LEX (B))))))

I know
((M45! (CLASS (M14 (LEX (block)))) (MEMBER (M36 (LEX (A))))))

I know
((M13! (CLASS (M12 (LEX (support)))) (MEMBER (B1))))

Since
((M26! (FORALL (V2 <-- M48) (V4 <-- B1))
  (&ANT (P46 (ARG1 (V2 <-- M48)) (ARG2 (V4 <-- B1)) (REL (M23 (LEX (on)))))
   (P48 (CLASS (M12 (LEX (support)))) (MEMBER (V4 <-- B1)))
   (P9 (CLASS (M14 (LEX (block)))) (MEMBER (V2 <-- M48))))
  (CQ
   (P79 (ACT (P6 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (V2 <-- M48))))
    (EFFECT (P78 (OBJECT (V4 <-- B1)) (PROPERTY (M20 (LEX (clear))))))))))
and
((P46 (ARG1 (V2 <-- M48)) (ARG2 (V4 <-- B1)) (REL (M23 (LEX (on))))))
and
((P48 (CLASS (M12 (LEX (support)))) (MEMBER (V4 <-- B1))))
and
((P9 (CLASS (M14 (LEX (block)))) (MEMBER (V2 <-- M48))))
I infer
```

```
((P79 (ACT (P6 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (V2 <-- M48))))
  (EFFECT (P78 (OBJECT (V4 <-- B1)) (PROPERTY (M20 (LEX (clear))))))))

Since
((M25! (FORALL (V2 <-- M48) (V4 <-- B1))
  (&ANT (P46 (ARG1 (V2 <-- M48)) (ARG2 (V4 <-- B1)) (REL (M23 (LEX (on)))))
   (P48 (CLASS (M12 (LEX (support)))) (MEMBER (V4 <-- B1)))
   (P9 (CLASS (M14 (LEX (block)))) (MEMBER (V2 <-- M48))))
  (CQ
   (P49 (ACT (P6 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (V2 <-- M48))))
    (EFFECT
     (P47 (MIN 0) (MAX 0)
      (ARG
       (P46 (ARG1 (V2 <-- M48)) (ARG2 (V4 <-- B1))
        (REL (M23 (LEX (on))))))))))))
and
((P46 (ARG1 (V2 <-- M48)) (ARG2 (V4 <-- B1)) (REL (M23 (LEX (on))))))
and
((P48 (CLASS (M12 (LEX (support)))) (MEMBER (V4 <-- B1))))
and
((P9 (CLASS (M14 (LEX (block)))) (MEMBER (V2 <-- M48))))
I infer
((P49 (ACT (P6 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (V2 <-- M48))))
  (EFFECT
   (P47 (MIN 0) (MAX 0)
    (ARG
     (P46 (ARG1 (V2 <-- M48)) (ARG2 (V4 <-- B1)) (REL (M23 (LEX (on))))))))))
```

As we can see above, several belief-belief transformations, through the rules represented by M22, M28, M26, and M25, lead to the derivation of four effects of the act of picking up A—that C is no longer clear, it is held, the table is clear, and C is no longer on the table. The act process, upon receiving these reports, processes them into the *EFFECTS* register, performs an act-effect transformation (schedules the act of believing all of the act's effects), changes its agenda to EXECUTE, and reschedules itself. Thus, after the act is scheduled, there are at least two act processes on the act queue—the act of picking up C followed by the process corresponding to a DoALL act of believing the act's effects.

Next, the pickup act is again dispatched; its agenda is EXECUTE. Since the act is primitive, the act's actuator is executed. The next act on the act queue is to believe all of the act's effects. This is shown below:

```
Now  doing: Pickup C.

Now doing: DO-ALL
((M69 (ACTION (M5 (LEX (believe))))
  (OBJECT1
   (M58 (MIN 0) (MAX 0)
    (ARG (M50! (OBJECT (M48 (LEX (C)))) (PROPERTY (M20 (LEX (clear)))))))))
 (M70 (ACTION (M5 (LEX (believe))))
  (OBJECT1 (M59 (OBJECT (M48 (LEX (C)))) (PROPERTY (M27 (LEX (held)))))))
 (M71 (ACTION (M5 (LEX (believe))))
  (OBJECT1 (M65 (OBJECT (B1)) (PROPERTY (M20 (LEX (clear)))))))
 (M72 (ACTION (M5 (LEX (believe))))
```

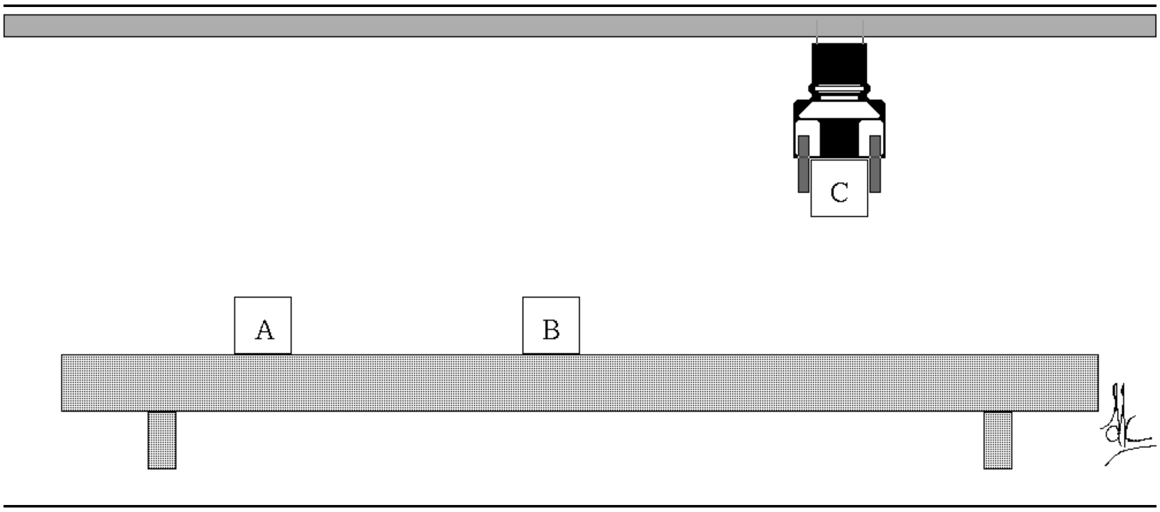

Figure 12.2: The blocksworld after PICKUP(C) is performed.

```
  (OBJECT1
   (M67 (MIN 0) (MAX 0)
    (ARG (M51! (ARG1 (M48 (LEX (C)))) (ARG2 (B1)) (REL (M23 (LEX (on)))))))))

Disbelieve
(M51! (ARG1 (M48 (LEX C))) (ARG2 B1) (REL (M23 (LEX on))))

Believe
(M65! (OBJECT B1) (PROPERTY (M20 (LEX clear))))

Believe
(M59! (OBJECT (M48 (LEX C))) (PROPERTY (M27 (LEX held))))

Disbelieve
(M50! (OBJECT (M48 (LEX C))) (PROPERTY (M20 (LEX clear))))
```

Thus, act-effect transformations are carried out to believe the act's effects. This concludes the fulfillment of the intention to perform the act of picking up C. The state of the blocksworld is depicted in Figure 12.2.

12.2.3 Plan-Goal Transformations

In the example in Section 11.2.1, where the agent was asked to perform the act of putting A on B, we showed the derivations of the act's preconditions and that they were not satisfied. That led to precondition-act transformations where the agent had formed the intention to achieve the act's preconditions. These are reproduced below:

```
Intending to do
((M82 (ACTION (M73 (LEX (do-all))))
  (OBJECT1
   (M80 (ACTION (M4 (LEX (achieve))))
    (OBJECT1 (M52! (OBJECT (M46 (LEX (B)))) (PROPERTY (M20 (LEX (clear)))))))
```

```
   (M81 (ACTION (M4 (LEX (achieve))))
    (OBJECT1 (M76 (OBJECT (M36 (LEX (A)))) (PROPERTY (M27 (LEX (held))))))))))

Now doing: DO-ALL
((M80 (ACTION (M4 (LEX (achieve))))
  (OBJECT1 (M52! (OBJECT (M46 (LEX (B)))) (PROPERTY (M20 (LEX (clear)))))))
 (M81 (ACTION (M4 (LEX (achieve))))
  (OBJECT1 (M76 (OBJECT (M36 (LEX (A)))) (PROPERTY (M27 (LEX (held))))))))
```

The agent, in carrying out the DO-ALL act, first tries to achieve that A is held. This becomes its current goal, and it tries to derive a plan that, if carried out, would achieve the goal. Derivation of the plan begins with a query to a plan-goal transformer. This is shown below:

```
Want to achieve
((M76 (OBJECT (M36 (LEX (A)))) (PROPERTY (M27 (LEX (held))))))

Want to find a plan to achieve
((M76 (OBJECT (M36 (LEX (A)))) (PROPERTY (M27 (LEX (held))))))

...
The goal can be achieved by a plan:
((M84! (GOAL (M76 (OBJECT (M36 (LEX (A)))) (PROPERTY (M27 (LEX (held))))))
  (PLAN (M83 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M36 (LEX (A))))))))
```

This leads the agent to form the intention to carry out the plan. Recall that it is possible that several plans may be derived. The agent schedules to perform one of them. This represents a plan-goal transformation—an intention to achieve a goal led to an intention to perform a plan that it believes achieves that goal:

```
Intending to do
((M86 (ACTION (M85 (LEX (do-one))))
  (OBJECT1 (M83 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M36 (LEX (A))))))))

Now doing: Do-ONE
((M83 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M36 (LEX (A))))))

Chose to do the act
((M83 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M36 (LEX (A))))))

About to do
((M83 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M36 (LEX (A))))))

...
Now  doing: Pickup A.

...
Disbelieve
(M55! (ARG1 (M36 (LEX A))) (ARG2 B1) (REL (M23 (LEX on))))

Believe
(M76! (OBJECT (M36 (LEX A))) (PROPERTY (M27 (LEX held))))
```

```
Disbelieve
(M54! (OBJECT (M36 (LEX A))) (PROPERTY (M20 (LEX clear))))

I already believe
(M65! (OBJECT B1) (PROPERTY (M20 (LEX clear))))

Want to achieve
((M52! (OBJECT (M46 (LEX (B)))) (PROPERTY (M20 (LEX (clear))))))

Already Achieved.
```

At this point the agent is again ready to try to put A on B. This time its preconditions are satisfied, so the act is performed:

```
About to do
((M75 (ACTION (M18 (LEX (put)))) (OBJECT1 (M36 (LEX (A))))
  (OBJECT2 (M46 (LEX (B))))))

...
The act
((M75 (ACTION (M18 (LEX (put)))) (OBJECT1 (M36 (LEX (A))))
  (OBJECT2 (M46 (LEX (B))))))

has the following preconditions:
((M77!
  (ACT
   (M75 (ACTION (M18 (LEX (put)))) (OBJECT1 (M36 (LEX (A))))
    (OBJECT2 (M46 (LEX (B))))))
  (PRECONDITION
   (M52! (OBJECT (M46 (LEX (B)))) (PROPERTY (M20 (LEX (clear)))))))
 (M78!
  (ACT
   (M75 (ACTION (M18 (LEX (put)))) (OBJECT1 (M36 (LEX (A))))
    (OBJECT2 (M46 (LEX (B))))))
  (PRECONDITION
   (M76! (OBJECT (M36 (LEX (A)))) (PROPERTY (M27 (LEX (held))))))))

I know
((M52! (OBJECT (M46 (LEX (B)))) (PROPERTY (M20 (LEX (clear))))))

I know
((M76! (OBJECT (M36 (LEX (A)))) (PROPERTY (M27 (LEX (held))))))

They are satisfied.

...
Now doing: Put A on B.

...
Disbelieve
(M52! (OBJECT (M46 (LEX B))) (PROPERTY (M20 (LEX clear))))

Believe
```

```
(M98! (ARG1 (M36 (LEX A))) (ARG2 (M46 (LEX B))) (REL (M23 (LEX on))))

Believe
(M54! (OBJECT (M36 (LEX A))) (PROPERTY (M20 (LEX clear))))

Disbelieve
(M76! (OBJECT (M36 (LEX A))) (PROPERTY (M27 (LEX held))))
```

12.2.4 Plan-Act Transformations

When the agent forms an intention to perform a complex act, it decomposes it into a plan. This constitutes a plan-act transformation. The transformation is performed if the agent is able to derive a specific PlanAct proposition that decomposes the complex act. Earlier, in Section 11.1.1, we showed how an agent can derive a specific plan. This time, we will ask the agent to perform the complex act. It will first derive the plan and then form the intention to execute the plan (i.e., a plan-act transformation) in order to carry out the complex act:

```
; Pile B on C on A.
* (perform (build action (build lex "pile")
                  object1 (build lex "B")
                  object2 (build lex "C")
                  object3 (build lex "A")))

About to do
((M56 (ACTION (M40 (LEX (pile)))) (OBJECT1 (M46 (LEX (B))))
  (OBJECT2 (M48 (LEX (C)))) (OBJECT3 (M36 (LEX (A))))))

I wonder if the act
((M56 (ACTION (M40 (LEX (pile)))) (OBJECT1 (M46 (LEX (B))))
  (OBJECT2 (M48 (LEX (C)))) (OBJECT3 (M36 (LEX (A))))))

has any preconditions...
The act
((M56 (ACTION (M40 (LEX (pile)))) (OBJECT1 (M46 (LEX (B))))
  (OBJECT2 (M48 (LEX (C)))) (OBJECT3 (M36 (LEX (A))))))

has no preconditions:
The act
((M56 (ACTION (M40 (LEX (pile)))) (OBJECT1 (M46 (LEX (B))))
  (OBJECT2 (M48 (LEX (C)))) (OBJECT3 (M36 (LEX (A))))))

has a plan:
((M62!
  (ACT
   (M56 (ACTION (M40 (LEX (pile)))) (OBJECT1 (M46 (LEX (B))))
    (OBJECT2 (M48 (LEX (C)))) (OBJECT3 (M36 (LEX (A))))))
  (PLAN
   (M61 (ACTION (M1 (LEX (snsequence))))
    (OBJECT1
     (M59 (ACTION (M1 (LEX (snsequence))))
      (OBJECT1
       (M57 (ACTION (M18 (LEX (put)))) (OBJECT1 (M36 (LEX (A))))
        (OBJECT2 (B1))))
```

```
      (OBJECT2
       (M58 (ACTION (M18 (LEX (put)))) (OBJECT1 (M48 (LEX (C))))
        (OBJECT2 (M36 (LEX (A))))))))
    (OBJECT2
     (M60 (ACTION (M18 (LEX (put)))) (OBJECT1 (M46 (LEX (B))))
      (OBJECT2 (M48 (LEX (C))))))))))

Intending to do
((M64 (ACTION (M63 (LEX (do-one))))
  (OBJECT1
   (M61 (ACTION (M1 (LEX (snsequence))))
    (OBJECT1
     (M59 (ACTION (M1 (LEX (snsequence))))
      (OBJECT1
       (M57 (ACTION (M18 (LEX (put)))) (OBJECT1 (M36 (LEX (A))))
        (OBJECT2 (B1))))
      (OBJECT2
       (M58 (ACTION (M18 (LEX (put)))) (OBJECT1 (M48 (LEX (C))))
        (OBJECT2 (M36 (LEX (A))))))))
    (OBJECT2
     (M60 (ACTION (M18 (LEX (put)))) (OBJECT1 (M46 (LEX (B))))
      (OBJECT2 (M48 (LEX (C))))))))))
```

The intention to carry out the plan will be fulfilled as described earlier.

12.2.5 Qualifier Acts

The act "pick up a clear block" can be represented by a single-object qualifier act (WITHSOME). When the agent is asked to perform the act, first it tries to identify specific objects via inference:

```
; Pick up a clear block.
* (perform
   (build action (build lex "withsome")
          vars $x
          suchthat (build min 2 max 2
                          arg ((build member *x class (build lex "block"))
                               (build object *x property (build lex "clear"))))
          do (build action (build lex "pickup")
                    object1 *x)))

Now doing: WITHSOME
((M57 (ACTION (M56 (LEX (withsome))))
  (DO (P316 (ACTION (M16 (LEX (pickup)))) (OBJECT1 V12)))
  (SUCHTHAT
   (P315 (MIN 2) (MAX 2)
    (ARG (P313 (CLASS (M14 (LEX (block)))) (MEMBER V12))
     (P314 (OBJECT V12) (PROPERTY (M20 (LEX (clear))))))))
  (VARS V12)))

I wonder if
((P315 (MIN 2) (MAX 2)
```

```
  (ARG (P313 (CLASS (M14 (LEX (block)))) (MEMBER V12))
   (P314 (OBJECT V12) (PROPERTY (M20 (LEX (clear))))))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P313 (CLASS (M14 (LEX (block)))) (MEMBER V12)))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P314 (OBJECT V12) (PROPERTY (M20 (LEX (clear))))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I know
((M49! (CLASS (M14 (LEX (block)))) (MEMBER (M48 (LEX (C))))))

I know
((M47! (CLASS (M14 (LEX (block)))) (MEMBER (M46 (LEX (B))))))

I know
((M45! (CLASS (M14 (LEX (block)))) (MEMBER (M36 (LEX (A))))))

I know
((M54! (OBJECT (M36 (LEX (A)))) (PROPERTY (M20 (LEX (clear))))))

Since
((P313 (CLASS (M14 (LEX (block)))) (MEMBER (V12 <-- M36))))
and
((P314 (OBJECT (V12 <-- M36)) (PROPERTY (M20 (LEX (clear))))))
I infer
((P315 (MIN 2) (MAX 2)
  (ARG (P313 (CLASS (M14 (LEX (block)))) (MEMBER (V12 <-- M36)))
   (P314 (OBJECT (V12 <-- M36)) (PROPERTY (M20 (LEX (clear))))))))

I know
((M52! (OBJECT (M46 (LEX (B)))) (PROPERTY (M20 (LEX (clear))))))

Since
((P313 (CLASS (M14 (LEX (block)))) (MEMBER (V12 <-- M46))))
and
((P314 (OBJECT (V12 <-- M46)) (PROPERTY (M20 (LEX (clear))))))
I infer
((P315 (MIN 2) (MAX 2)
  (ARG (P313 (CLASS (M14 (LEX (block)))) (MEMBER (V12 <-- M46)))
   (P314 (OBJECT (V12 <-- M46)) (PROPERTY (M20 (LEX (clear))))))))

I know
((M50! (OBJECT (M48 (LEX (C)))) (PROPERTY (M20 (LEX (clear))))))

Since
((P313 (CLASS (M14 (LEX (block)))) (MEMBER (V12 <-- M48))))
and
((P314 (OBJECT (V12 <-- M48)) (PROPERTY (M20 (LEX (clear))))))
I infer
((P315 (MIN 2) (MAX 2)
```

```
  (ARG (P313 (CLASS (M14 (LEX (block)))) (MEMBER (V12 <-- M48)))
   (P314 (OBJECT (V12 <-- M48)) (PROPERTY (M20 (LEX (clear)))))))
```

At this point, the agent has derived three propositions—M58, M59, M60—that satisfy the designator represented by P315:

```
The designator
((P315 (MIN 2) (MAX 2)
  (ARG (P313 (CLASS (M14 (LEX (block)))) (MEMBER V12))
   (P314 (OBJECT V12) (PROPERTY (M20 (LEX (clear)))))))

is effective on the following:
((M58! (MIN 2) (MAX 2)
  (ARG (M45! (CLASS (M14 (LEX (block)))) (MEMBER (M36 (LEX (A)))))
   (M54! (OBJECT (M36 (LEX (A)))) (PROPERTY (M20 (LEX (clear))))))
 (M59! (MIN 2) (MAX 2)
  (ARG (M47! (CLASS (M14 (LEX (block)))) (MEMBER (M46 (LEX (B)))))
   (M52! (OBJECT (M46 (LEX (B)))) (PROPERTY (M20 (LEX (clear))))))
 (M60! (MIN 2) (MAX 2)
  (ARG (M49! (CLASS (M14 (LEX (block)))) (MEMBER (M48 (LEX (C)))))
   (M50! (OBJECT (M48 (LEX (C)))) (PROPERTY (M20 (LEX (clear)))))))

for the act:
((P316 (ACTION (M16 (LEX (pickup)))) (OBJECT1 V12)))
```

The agent has determined that since A, B, and C are all clear, it can fulfill the intention to pick up a clear block by picking up any one of the identified blocks:

```
I intend to do:
((M65 (ACTION (M64 (LEX (do-one))))
  (OBJECT1 (M61 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M36 (LEX (A)))))
   (M62 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M46 (LEX (B)))))
   (M63 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M48 (LEX (C))))))))

Now doing: DO-ONE
((M61 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M36 (LEX (A)))))
 (M62 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M46 (LEX (B)))))
 (M63 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M48 (LEX (C))))))

Chose to do the act
((M63 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M48 (LEX (C))))))

...
Now  doing: Pickup C.
...
```

Let us repeat the same request again and see what happens:

```
; Pick up a clear block.
* (perform
   (build action (build lex "withsome")
          vars $x
```

```
          suchthat (build min 2 max 2
                          arg ((build member *x class (build lex "block"))
                               (build object *x property (build lex "clear"))))
          do (build action (build lex "pickup")
                    object1 *x)))

Now doing: WITHSOME
((M84 (ACTION (M56 (LEX (withsome))))
  (DO (P324 (ACTION (M16 (LEX (pickup)))) (OBJECT1 V15)))
  (SUCHTHAT
   (P323 (MIN 2) (MAX 2)
    (ARG (P321 (CLASS (M14 (LEX (block)))) (MEMBER V15))
     (P322 (OBJECT V15) (PROPERTY (M20 (LEX (clear)))))))
  (VARS V15)))

The designator
((P323 (MIN 2) (MAX 2)
  (ARG (P321 (CLASS (M14 (LEX (block)))) (MEMBER V15))
   (P322 (OBJECT V15) (PROPERTY (M20 (LEX (clear))))))))

is effective on the following:
((M58! (MIN 2) (MAX 2)
  (ARG (M45! (CLASS (M14 (LEX (block)))) (MEMBER (M36 (LEX (A)))))
   (M54! (OBJECT (M36 (LEX (A)))) (PROPERTY (M20 (LEX (clear)))))))
 (M59! (MIN 2) (MAX 2)
  (ARG (M47! (CLASS (M14 (LEX (block)))) (MEMBER (M46 (LEX (B)))))
   (M52! (OBJECT (M46 (LEX (B)))) (PROPERTY (M20 (LEX (clear))))))))

for the act:
((P324 (ACTION (M16 (LEX (pickup)))) (OBJECT1 V15)))
```

Thus, this time only two objects were identified (since C is no longer clear).

```
I intend to do:
((M85 (ACTION (M64 (LEX (do-one))))
  (OBJECT1 (M61 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M36 (LEX (A)))))
   (M62 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M46 (LEX (B))))))))

...
Now  doing: Pickup B
...
```

Similarly, the act, "Put on the table all blocks that are being held", which is represented by a multiple-qualifier act (WITHALL), is carried out as shown below:

```
; Put on the table all the blocks that are being held
* (perform
    (build action (build lex "withall")
           vars $x
           suchthat (build min 2 max 2
                           arg ((build member *x class (build lex "block"))
                                (build object *x property (build lex "held"))))
           do (build action (build lex "put")
```

```
                    object1 *x
                    object2 B1)))

Now doing: WITHALL
((M99 (ACTION (M98 (LEX (withall))))
  (DO (P332 (ACTION (M18 (LEX (put)))) (OBJECT1 V16) (OBJECT2 (B1))))
  (SUCHTHAT
   (P331 (MIN 2) (MAX 2)
    (ARG (P329 (CLASS (M14 (LEX (block)))) (MEMBER V16))
     (P330 (OBJECT V16) (PROPERTY (M27 (LEX (held))))))))
  (VARS V16)))

The designator
((P331 (MIN 2) (MAX 2)
  (ARG (P329 (CLASS (M14 (LEX (block)))) (MEMBER V16))
   (P330 (OBJECT V16) (PROPERTY (M27 (LEX (held))))))))

is effective on the following:
((M100! (MIN 2) (MAX 2)
  (ARG (M49! (CLASS (M14 (LEX (block)))) (MEMBER (M48 (LEX (C)))))
   (M88! (OBJECT (M48 (LEX (C)))) (PROPERTY (M27 (LEX (held)))))))
 (M101! (MIN 2) (MAX 2)
  (ARG (M47! (CLASS (M14 (LEX (block)))) (MEMBER (M46 (LEX (B)))))
   (M68! (OBJECT (M46 (LEX (B)))) (PROPERTY (M27 (LEX (held))))))))

for the act:
((P332 (ACTION (M18 (LEX (put)))) (OBJECT1 V16) (OBJECT2 (B1))))
```

Since both B and C are held, the agent decides to form the intention of putting them on the table:

```
I intend to do:
((M104 (ACTION (M82 (LEX (do-all))))
  (OBJECT1
   (M102 (ACTION (M18 (LEX (put)))) (OBJECT1 (M48 (LEX (C)))) (OBJECT2 (B1)))
   (M103 (ACTION (M18 (LEX (put)))) (OBJECT1 (M46 (LEX (B)))) (OBJECT2 (B1))))))

...
Now doing: Put B on table.
...
Now doing: Put C on table.
...
```

12.2.6 Reactivity—Belief–Act Transformations

One of the examples we used in Section 1.2 to motivate the reasons for integrated reactivity was that the agent should be able to react to actions performed by other agents. For example, the response to "John picked up block A" should at least prompt the agent to update its beliefs about the world. Starting with the belief space defined in Chapter 10, we add the following beliefs:

```
; John is an agent.
```

```
* (assert member (build lex "john") class (build lex "agent"))

(M58!)

 CPU time : 0.10     GC time : 0.00

; If some agent performs some act and the act has some effects then
; perform the act of believing those effects.
* (assert forall ($agent $act $effect)
          &ant ((build act *act effect *effect)
                (build member *agent class (build lex "agent")))
          cq (build when (build agent *agent act *act)
                    do (build action (build lex "believe")
                              object1 *effect)))

(M59!)

 CPU time : 0.17     GC time : 0.00
```

We can ask the agent what it would do if someone performed some specific act:

```
; What happens when John picks up A?
* (describe (deduce when (build agent (build lex "john")
       act (build action (build lex "pickup")
  object1 (build lex "A")))
do $do))

(M68!
 (DO (M67
      (ACTION (M5 (LEX believe)))
      (OBJECT1 (M63 (OBJECT (M36 (LEX A))) (PROPERTY (M27 (LEX held)))))))
 (WHEN (M61 (ACT (M60 (ACTION (M16 (LEX pickup))) (OBJECT1 (M36))))
        (AGENT (M56 (LEX john))))))
(M70!
 (DO (M69
      (ACTION (M5))
      (OBJECT1
       (M62 (MIN 0) (MAX 0)
        (ARG (M54! (OBJECT (M36)) (PROPERTY (M20 (LEX clear)))))))))
 (WHEN (M61)))

(M78! (DO (M77 (ACTION (M5)) (OBJECT1 (M73 (OBJECT B1) (PROPERTY (M20))))))
 (WHEN (M61)))
(M80!
 (DO (M79
      (ACTION (M5))
      (OBJECT1
       (M75 (MIN 0) (MAX 0)
        (ARG (M55! (ARG1 (M36)) (ARG2 B1) (REL (M23 (LEX on)))))))))
 (WHEN (M61)))

(M68! M70! M78! M80!)

 CPU time : 5.08     GC time : 0.00
```

The agent responds with propositions M68, M70, M78, M80, which represent WhenDo transformers for believing the effects of the act of picking up A. We can tell the agent that John did pick up A:

```
; John picked up A.
* (add agent (build lex "john")
       act (build action (build lex "pickup")
                   object1 (build lex "A")))

Since
((M68!
  (DO (M67
       (ACTION (M5 (LEX (believe))))
       (OBJECT1 (M63 (OBJECT (M36 (LEX (A)))) (PROPERTY (M27 (LEX (held)))))))
  (WHEN (M61!
         (ACT (M60 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M36 (LEX (A))))))
         (AGENT (M56 (LEX (john))))))))

I will perform
((M67 (ACTION (M5 (LEX (believe))))
  (OBJECT1 (M63 (OBJECT (M36 (LEX (A)))) (PROPERTY (M27 (LEX (held))))))))

Since
((M70!
  (DO (M69
       (ACTION (M5 (LEX (believe))))
       (OBJECT1
        (M62 (MIN 0) (MAX 0)
         (ARG
          (M54! (OBJECT (M36 (LEX (A)))) (PROPERTY (M20 (LEX (clear))))))))))
  (WHEN (M61!
         (ACT (M60 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M36 (LEX (A))))))
         (AGENT (M56 (LEX (john))))))))

I will perform
((M69 (ACTION (M5 (LEX (believe))))
  (OBJECT1
   (M62 (MIN 0) (MAX 0)
    (ARG (M54! (OBJECT (M36 (LEX (A)))) (PROPERTY (M20 (LEX (clear))))))))))

Since
((M78!
  (DO (M77
       (ACTION (M5 (LEX (believe))))
       (OBJECT1 (M73 (OBJECT (B1)) (PROPERTY (M20 (LEX (clear))))))))
  (WHEN (M61!
         (ACT (M60 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M36 (LEX (A))))))
         (AGENT (M56 (LEX (john))))))))

I will perform
((M77 (ACTION (M5 (LEX (believe))))
  (OBJECT1 (M73 (OBJECT (B1)) (PROPERTY (M20 (LEX (clear))))))))

Since
```

```
((M80!
  (DO (M79
       (ACTION (M5 (LEX (believe))))
       (OBJECT1
        (M75 (MIN 0) (MAX 0)
         (ARG
          (M55! (ARG1 (M36 (LEX (A)))) (ARG2 (B1)) (REL (M23 (LEX (on))))))))))
  (WHEN (M61!
         (ACT (M60 (ACTION (M16 (LEX (pickup)))) (OBJECT1 (M36 (LEX (A))))))
         (AGENT (M56 (LEX (john))))))))

I will perform
((M79 (ACTION (M5 (LEX (believe))))
  (OBJECT1
   (M75 (MIN 0) (MAX 0)
    (ARG (M55! (ARG1 (M36 (LEX (A)))) (ARG2 (B1)) (REL (M23 (LEX (on))))))))))

Disbelieve
(M55! (ARG1 (M36 (LEX A))) (ARG2 B1) (REL (M23 (LEX on))))

Believe
(M73! (OBJECT B1) (PROPERTY (M20 (LEX clear))))

Disbelieve
(M54! (OBJECT (M36 (LEX A))) (PROPERTY (M20 (LEX clear))))

Believe
(M63! (OBJECT (M36 (LEX A))) (PROPERTY (M27 (LEX held))))

 CPU time : 1.32     GC time : 0.00
```

The above shows that the agent is able to perform forward chaining belief-act transformations that represent reactions.

12.2.7 Act–Belief Transformations

It is now possible for the modeled agent to perform actions in the service of reasoning. The agent can form an intention to acquire new information in order to answer some query. We will define a knowledge acquisition act, called LOOKAT, that represents the act of looking at a block in order to determine its color. The agent has the following additional beliefs:

```
; Lookat is a primitive action.
* (assert member (build lex "lookat")
          class (build lex "primitive"))

(M57!)

 CPU time : 0.13     GC time : 0.00
; If you want to know the what the color of some block is, look at it.
* (assert forall ($x $y)
          ant (build member *x class (build lex "block"))
          cq (build do (build action (build lex "lookat")
```

```
                              object1 *x)
                    if (build rel (build lex "color")
                              arg1 *x
                              arg2 *y)))

(M59!)

 CPU time : 0.18     GC time : 0.00
; All red colored blocks are wooden.
* (assert forall $a
          &ant ((build rel (build lex "color")
                       arg1 *a
                       arg2 (build lex "red"))
                (build member *a class (build lex "block")))
          cq (build object *a property (build lex "wooden")))

(M62!)

 CPU time : 0.18     GC time : 0.00
```

At this point, the agent knows about three blocks—A, B, and C—but it has no beliefs about their colors. Let us ask it if it believes that A is wooden.

```
; Is A wooden?
* (deduce object (build lex "A") property (build lex "wooden"))

I wonder if
((M63 (OBJECT (M36 (LEX (A)))) (PROPERTY (M61 (LEX (wooden))))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P317 (ARG1 (V14 <-- M36)) (ARG2 (M60 (LEX (red)))) (REL (M58 (LEX (color))))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P318 (CLASS (M14 (LEX (block)))) (MEMBER (V14 <-- M36))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I know
((M45! (CLASS (M14 (LEX (block)))) (MEMBER (M36 (LEX (A))))))
```

The agent's quest for the proposition that A is red will backchain into the DO-IF transformer. However, since the transformer is not asserted, it will first attempt to assert it:

```
I wonder if
((P316 (DO (P314 (ACTION (M56 (LEX (lookat)))) (OBJECT1 (V12 <-- M36))))
  (IF (P315 (ARG1 (V12 <-- M36)) (ARG2 (V13 <-- M60))
       (REL (M58 (LEX (color))))))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P313 (CLASS (M14 (LEX (block)))) (MEMBER (V12 <-- M36))))
holds within the BS defined by context DEFAULT-DEFAULTCT
```

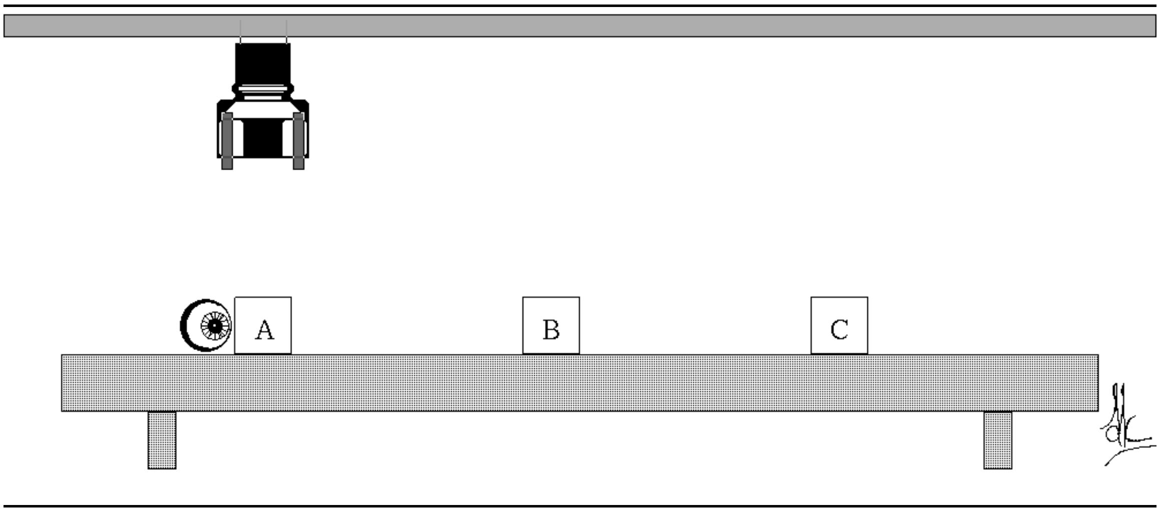

Figure 12.3: The act LOOKAT(A) being performed in the blocksworld.

```
Since
((M59! (FORALL V12 V13) (ANT (P313 (CLASS (M14 (LEX (block)))) (MEMBER V12)))
  (CQ
   (P316 (DO (P314 (ACTION (M56 (LEX (lookat)))) (OBJECT1 V12)))
    (IF (P315 (ARG1 V12) (ARG2 V13) (REL (M58 (LEX (color))))))))))
and
((P313 (CLASS (M14 (LEX (block)))) (MEMBER (V12 <-- M36))))
I infer
((P316 (DO (P314 (ACTION (M56 (LEX (lookat)))) (OBJECT1 (V12 <-- M36))))
  (IF (P315 (ARG1 (V12 <-- M36)) (ARG2 (V13 <-- M60))
       (REL (M58 (LEX (color))))))))

Since
((P316 (DO (P314 (ACTION (M56 (LEX (lookat)))) (OBJECT1 V12)))
  (IF (P315 (ARG1 V12) (ARG2 V13) (REL (M58 (LEX (color))))))))

I will perform
((M65 (ACTION (M56 (LEX (lookat)))) (OBJECT1 (M36 (LEX (A))))))
```

The agent has successfully performed the act-belief transformation. Its reasoning had led it to form an intention based on its desire to do so. Figure 12.3 shows the agent in the middle of the act of looking at the block A.

```
About to do
((M65 (ACTION (M56 (LEX (lookat)))) (OBJECT1 (M36 (LEX (A))))))

I wonder if the act
((M65 (ACTION (M56 (LEX (lookat)))) (OBJECT1 (M36 (LEX (A))))))

has any preconditions...

The act
((M65 (ACTION (M56 (LEX (lookat)))) (OBJECT1 (M36 (LEX (A))))))
```

```
has no preconditions:

Now doing: Look at A

* (dynamic-add rel (build lex "color") arg1 M36 arg2 (build lex "red") :context
               default-defaultct)

(M64!)

 CPU time : 0.08     GC time : 0.00

Since
((M62! (FORALL (V14 <-- M36))
  (&ANT
   (P317 (ARG1 (V14 <-- M36)) (ARG2 (M60 (LEX (red))))
    (REL (M58 (LEX (color)))))
   (P318 (CLASS (M14 (LEX (block)))) (MEMBER (V14 <-- M36))))
  (CQ (P319 (OBJECT (V14 <-- M36)) (PROPERTY (M61 (LEX (wooden))))))))
and
((P317 (ARG1 (V14 <-- M36)) (ARG2 (M60 (LEX (red)))) (REL (M58 (LEX (color))))))
and
((P318 (CLASS (M14 (LEX (block)))) (MEMBER (V14 <-- M36))))
I infer
((P319 (OBJECT (V14 <-- M36)) (PROPERTY (M61 (LEX (wooden))))))

(M63!)
 CPU time : 4.65     GC time : 0.00

* (describe m63)

(M63! (OBJECT (M36 (LEX A))) (PROPERTY (M61 (LEX wooden))))

(M63!)
 CPU time : 0.05     GC time : 0.00
```

The result of performing the LOOKAT action was a new belief—that A is red—that was dynamically added to the agent's belief space. The result of dynamically adding the proposition is that the proposition is activated as a report and is sent to all the processes (if any) that may have requested it. This ends up completing the earlier query, which results in the proposition M63 being asserted and returned as the answer.

12.3 Discussion

In this chapter, we have shown the various capabilities of a rational agent modeled in the SNePS BDI architecture. We have shown various capabilities of SNeRE and how its unified inference and acting engine in conjunction with various transformers results in reasoning, acting, reacting, and knowledge acquisition behaviors.

Chapter 13

Conclusions

We started this dissertation with the observation that most "good" planning/acting systems are "bad" knowledge representation and reasoning systems, and vice versa. We identified the need for a unified representational formalism that makes proper ontological distinctions between beliefs, acts, and plans. While a unified BDI formalism enables the inference engine to reason about beliefs, acts, and plans, one still needs a different module for acting. The acting executive forces the inference engine to a subordinate role—the system performs inference only in the service of acting (and querying). We recognized that it should be possible for reasoning to lead to acting. We achieved that by making a semantic clarification between inference and acting—that inference is basically akin to mental acting. This led us to formulate the model of a rational engine as a unified reasoning/acting module that reasons as well as acts (physically as well as mentally).

The unified view of inference and acting led us to generalize the idea of a deduction rule into transformers. We showed that it was possible to have transformers that transformed beliefs (and acts) into acts (and beliefs). This enabled us to extend the unified ontology to include various transformers. We defined the roles of various transformers in the context of forward and backward chaining. Consequently, using transformers, we were able to model deduction rules, preconditions and effects of actions, hierarchical plan decompositions for complex acts, plans for achieving goals, reactive rules, and knowledge acquisition rules. We called the resulting architecture that includes the unified (and extended) BDI formalism and the rational engine module, a BDI architecture.

We defined an object-oriented BDI formalism, called the OK formalism, where the class hierarchies were defined based on the ontological components of the BDI formalism. The rational engine was defined using methods that directly corresponded to the attitudes of belief and intention. The methods can be inherited, or specialized to individual classes. This makes the architecture naturally extendible to different logics, as well as action theories. We also showed that the resulting object-oriented rational engine was inherently amenable to concurrency. We also presented a quasi-concurrent computational model for sequential platforms.

We showed that the OK formalism was isomorphic to the SNePS formalism by extending the SNePS ontology. This was not surprising, since the fundamental principles underlying the OK formalism were adapted from the SNePS formalism. Based on this, and the fact that it is easy to change the external representations of objects, we claim that objects of the OK formalism are canonical representations of intensional entities. All instances of the

OK formalism can be expressed in a linear logical notation as well as a SNePS-like semantic network notation. This can inherently lead to applications where knowledge bases can be distributed (and shared). A canonical instance makes the job of translation between formalisms easier. Moreover, the DoWhen transformers provide a natural extension for inference to query other knowledge bases (see example in Section 6.1). We then extended the SNePS inference engine to SNeRE, the SNePS Rational Engine, which employs a quasi-concurrent computational model. We presented examples of inference, acting, reacting, and knowledge acquisition demonstrating the underlying claims. The SNePS BDI architecture builds upon the capabilities of the SNePS acting system. It reformulates the representations for preconditions and effects of actions, and plan decompositions, as transformers that are used in forward and backward chaining. It provides additional transformers that represent the agent's desires for reacting and acquiring new information. It preserves the ontology of control actions of the SNePS acting system. However, the semantics of the actions is operationally defined as an integral component of the rational engine (as opposed to an acting executive). With the unification of acting and inference, it was also possible to provide qualifier acts that inherently employ reasoning. Qualifier acts enable the representation, as well as use, of generic acts, something that is lacking in most acting and planning systems.

13.1 Contributions

The main contributions of this research are summarized below:

1. We have clarified the semantic distinction between inference and acting—that inference is a kind of acting. Architectures that employ this view provide a uniform platform for modeling all kinds of behavior—inference, acting, plan decomposition, reaction, intentional knowledge acquisition, etc. The module that implements the unified perspective is called a *rational engine* (as opposed to an inference engine).

2. The unified view of acting and inference introduces the generalized representational notion of a *transformer*. Transformers explicitly represent deduction rules, preconditions and effects of actions, plan decompositions, reactive rules, and rules for acquiring new beliefs. The semantics of transformers is defined operationally in terms of forward/backward chaining procedures of the rational engine.

3. We defined the OK BDI architecture that has the following constituents:

   OK BDI Architecture = OK Formalism + OK Rational Engine

   The OK formalism is a conceptual, extendible, object-oriented hierarchy of classes that correspond to the various ontological components of a knowledge representation system—individuals, propositions, and acts. While several object-based AI systems are already in existence, this is the first one that makes an object-oriented commitment to the entities of a KR formalism itself. The OK rational engine is defined using methods inherited or specialized by the classes of the OK formalism. The rational engine is inherently concurrent in nature.

4. We extended the ontology of SNePS so that it can now be considered an instance of the OK formalism. We extended the SNePS inference engine into SNeRE, the

SNePS Rational Engine, which implements the unified view of acting and inference. The resulting SNePS BDI architecture preserves all the features of the SNePS acting system. It reformulates several representations as transformers as well as provides new transformers that represent desires for reacting and acquiring new information. The acting executive is no longer required since the rational engine is responsible for all reasoning and acting. The semantics of all control acts was redefined using the rational engine. This work also introduced the idea of qualifier acts that enable representation and carrying out of generic actions.

13.2 Future Work

This work has opened up several interesting areas for future research. First, and foremost, it includes the specification of an object-based AI architecture. Given that concurrency and object-oriented design are one of the most promising emerging technologies, the design of the OK architecture represents a promising alternative for building AI architectures. It has been claimed about object-based design that the resulting programs are extremely scalable and amenable to extensions. Both scalability to larger systems and extendibility are integral goals of all AI systems. We have designed the OK formalism to be extremely extendible—in its underlying ontology and logic, as well as action theory. No system implemented today can make such a claim. However, we don't as yet have concurrent object-oriented technology that would enable us to build such an architecture. We have mentioned some viable paradigms (like Actor systems) that could be used.

We have shown that the OK formalism is isomorphic to the SNePS formalism. Another way to look at this would be to say that the OK formalism is an object-oriented definition of SNePS. Actually, we have gone beyond that and claimed that the OK formalism is canonical in the sense that it may facilitate easy translation across other formalisms. While it was trivial to show this in the case of SNePS, this hypothesis still needs to be tested. There are, however, two things that led to this claim—we incorporated several principles underlying knowledge representational formalisms (specifically from SNePS) and the external interface of an object-oriented system is reprogrammable via its print methods. Thus, if the underlying principles are universal, it may be possible to change the external interface of such systems by replacing its print methods. This can also be substantiated by the SNePS example. Objects in the OK formalism can be expressed as SNePS networks as well as a linear logical notation (which is actually similar to the SNePSLOG notation [McKay and Martins 1981, Shapiro *et al.* 1981, Shapiro and Group 1989]).

The above two have been more or less architectural and engineering issues. At the modeling level, there are issues relating to the ontology, the underlying logic, rules of inference (or transformation), etc., that need to be formalized. The dynamic sensory interface can be explored and evolved into an integrated theory of attention. We have not addressed the plan generation issue in this work. While several plan structures are possible, there isn't a one-to-one correspondence between them and known planning algorithms. We have also mentioned the utility of structured representations of variables. Not only does this impact the nature of the underlying logic and class of inferences, it also affects the representations of qualifier acts.

We have also shown in this dissertation that it is conducive, when building integrated architectures, that logical, semantic, as well as architectural concerns be addressed in cohesion. Thus, for all the extensions of the architecture, it would be interesting to find out their impact on concurrency.

Bibliography

[Agha and Hewitt 1987] 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.

[Agha 1986a] Gul Agha. *Actor: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

[Agha 1986b] Gul Agha. An Overview of Actor Laguages. *ACM SIGPLAN Notices*, 21(10):58–67, October 1986.

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

[Agre and Chapman 1987] Philip E. Agre and David Chapman. Pengi: An Implementation of a Theory of Activity. In *Proceedings of the 6th National Conference on AI (AAAI-87)*, pages 268–272. AAAI, 1987.

[Agre and Chapman 1990] Philip E. Agre and David Chapman. What Are Plans for? In Patti Maes, editor, *New Directions for Autonomous Agents: Task-Level Decomposition and Emergent Functionality*, pages 17–34. MIT Press, Cambridge, MA, 1990.

[Ali 1993a] 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.

[Ali 1993b] Syed S. Ali. A Structured Representation for Noun Phrases and Anaphora. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, 1993.

[Ali 1993c] Syed S. Ali. Natural Language Processing Using Propositional Semantic Networks. *Minds and Machines*, 1993. Special Issue on Knowledge Representation for Natural Language Processing, (to appear).

[Allen and Kautz 1985] James F. Allen and Henry A. Kautz. A model of naive temporal reasoning. In Jerry R. Hobbs and Robert C. Moore, editors, *Formal Theories of the Commonsense World*, pages 251–269. Ablex Publishing Corp., Norwood, NJ, 1985.

[Allen and Koomen 1983] James F. Allen and J. A. Koomen. Planning using a temporal world model. In *Proceedings of 8th IJCAI 1983*. IJCAI, 1983.

[Allen 1984] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–54, 1984.

[Allen 1991] James Allen. The RHET System. In Charles Rich (Guest Editor), editor, *SIGART BULLETIN Special Issue on Implemented KRR Systems*, pages 1–7, June 1991.

[Almeida 1987] M. J. Almeida. *Reasoning About the Temporal Structure of Narratives.* PhD thesis, Technical Report 87–10, Department of Computer Science, SUNY at Buffalo, 1987.

[ANSI, Inc. 1983] ANSI, Inc. *The Programming Language Ada Reference Manual.* Number 155 in Lecture Notes in Computer Science. Springer-Verlag, 1983.

[Beetz and McDermott 1992] Michael Beetz and Drew McDermott. Declarative Goals in Reactive Plans. In James Hendler, editor, *Proceedings of the 1st Conference on AI Planning Systems (AIPS)*, pages 3–12, San Mateo, CA, 1992. Morgan Kaufmann Publishers.

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

[Bratman 1987] M. Bratman. *Intentions, Plans and Practical Reason.* Harvard University Press, 1987.

[Bratman 1990] Michael E. Bratman. What IS Intention? In Philip R. Cohen, Jerry Morgan, and Martha E. Pollack, editors, *Intentions in Communication*, chapter 2, pages 15–31. MIT Press, 1990.

[Brooks 1986] R. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, 1986.

[Castañeda 1975] Hector-Neri Castañeda. *Thinking and Doing.* D. Reidel, Dordrecht, Holland, 1975.

[Chin and Chanson 1991] Roger S. Chin and Samuel T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[Choi 1993] Joongmin Choi. *Eperience-Based Learning in Deductive Reasoning Systems.* PhD thesis, State University of New York at Buffalo, May 1993.

[Cohen and Levesque 1987] P. R. Cohen and H. J. Levesque. Persistence, Intention, and Commitment. In Michael P. Georgeff and Amy L. Lansky, editors, *Reasoning about Actions and Plans - Proceedings of the 1986 Workshop*, pages 297–340, Los Altos, CA, 1987. AAAI and CSLI, Morgan Kauffmann.

[Cohen and Levesque 1991] Philip R. Cohen and Hector J. Levesque. Teamwork. *Noûs*, XXV(4):487–512, September 1991.

[Cravo and Martins 1993] M. Cravo and J. P. Martins. SNePSwD: A Newcomer to the SNePS Family. *JETAI*, 5(2), 1993.

[Currie and Tate 1991] Ken Currie and Austin Tate. O-Plan: the open planning architecture. *Artificial Intelligence*, 52:49–86, 1991.

[Dijkstra 1976] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Drummond and Bresina 1990] Mark Drummond and John Bresina. Anytime Synthetic Projection: Maximizing the Probability of Goal Satisfaction. In *Proceedings of the 8th National Conference on AI (AAAI)*, pages 138–144, 1990.

[Drummond 1985] M. Drummond. Refining and extending the procedural net. In *Proceedings of the 9th IJCAI 1985*, pages 528–531. IJCAI, August 1985.

[Drummond 1987] Mark E. Drummond. A representation of action and belief for automatic planning systems. In Michael P. Georgeff and Amy L. Lansky, editors, *Reasoning about Actions and Plans - Proceedings of the 1986 Workshop*, pages 189–212, Los Altos, CA, 1987. AAAI and CSLI, Morgan Kauffmann.

[Ferreira *et al.* 1989] Carlos P. Ferreira, Nuno J. Mamede, and Joao P. Martins. SNIP 2.1 — The SNePS Inference Package. Technical Report TR GIA 89/5, Instituto Superior Technico, Lisbon, Portugal, December 1989.

[Fikes and Nilsson 1971] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5:189–208, 1971.

[Firby 1987] R. James Firby. An Investigation into reactive Planning in Complex Domains. In *Proceedings of the 6th National Conference on AI (AAAI)*, pages 202–206, 1987.

[Firby 1989] R. James Firby. Coping with realistic sensory data. In Ernest Davis, Leora Morgenstern, and Kate Sanders, editors, *Working Notes of The IJCAI Workshop on Knowledge, Perception, and planning*, August 1989.

[Georgeff and Lansky 1987] Michael. P. Georgeff and Amy. Lansky. Procedural knowledge. Technical Note 411, AI Center, SRI International, 1987.

[Georgeff *et al.* 1985] M. P. Georgeff, A. Lansky, and P. Bessiere. A procedural logic. In *Proceedings of the 9th IJCAI*, 1985.

[Georgeff *et al.* 1987] M. P. Georgeff, A. Lansky, and M. Schoppers. Reasoning and planning in dynamic domains: An experiment with a mobile robot. Technical Note 380, AI Center, SRI International, 1987.

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

[Georgeff 1988] Michael P. Georgeff. An embedded reasoning and planning system. In Jay Webber, Josh Tenenberg, and James Allen, editors, *Advance Proceedings of The Rochester Planning Workshop–From Formal Systems to Practical Systems*, pages 79–101, October 1988.

[Goldberg and Robson 1983] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

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

[Hewitt 1972] C. Hewitt. Description and theoretical analysis (using schemata) of planner: A language for proving theorems and manipulating models in a robot. Technical Report 258, MIT, April 1972.

[Hull 1986] R. G. Hull. A new design for SNIP the SNePS inference package. SNeRG Technical Note 14, Department of Computer Science, SUNY at Buffalo, 1986.

[Israel 1983] David J. Israel. Interpreting Network Formalisms. In N. Cercone, editor, *Computational Linguistics*, pages 1–13. Oxford: Pergammon Press, 1983.

[Kaelbling 1986] L. Kaelbling. An architecture for intelligent reactive systems. In *Proceedings Workshop on Planning and Reasoning about Action.* AAAI, 1986.

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

[Kumar and Shapiro 1991b] Deepak Kumar and Stuart C. Shapiro. Modeling a rational cognitive agent in SNePS. In P. Barahona, L. Moniz Pereira, and A. Porto, editors, *EPIA 91: 5th Portugese Conference on Artificial Intelligence, Lecture Notes in Artificial Intelligence 541*, pages 120–134. Springer-Verlag, Heidelberg, 1991.

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

[Kumar *et al.* 1988] Deepak Kumar, Syed Ali, and Stuart C. Shapiro. Discussing, using and recognizing plans in SNePS preliminary report—SNACTor: An acting system. In P V S Rao and P Sadanandan, editors, *Modern Trends in Information Technology: Proceedings of the Seventh Biennial Convention of South East Asia Regional Computer Confederation*, pages 177–182. Tata McGraw-Hill, New Delhi, India, 1988.

[Kumar *et al.* 1990] D. Kumar, S. S. Ali, J. Haas, and S. C. Shapiro. The SNePS acting system. In K. E. Bettinger and G. Srikantan, editors, *Proceedings of the Fifth Annual University at Buffalo Graduate Conference on Computer Science*, pages 91–100, 1990.

[Lenat and Guha 1991] Douglas B. Lenat and Ramanathan V. Guha. The Evolution of CYCL, The CYC Representation Language. In Charles Rich (Guest Editor), editor, *SIGART BULLETIN Special Issue on Implemented KRR Systems*, pages 84–87, June 1991.

[Lifschitz 1987] Vladimir Lifschitz. On the semantics of STRIPS. In Michael P. Georgeff and Amy L. Lansky, editors, *Reasoning about Actions and Plans - Proceedings of the 1986 Workshop*, pages 1–10, Los Altos, CA, 1987. AAAI and CSLI, Morgan Kauffmann.

[Liskov *et al.* 1977] B. Liskov, A. Snyder, A. Atkinson, and C. Schaffert. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[Maida and Shapiro 1982] A. S. Maida and S. C. Shapiro. Intensional concepts in propositional semantic networks. *Cognitive Science*, 6(4):291–330, 1982. Reprinted in R. J. Brachman and H. J. Levesque, eds. *Readings in Knowledge Representation*, Morgan Kaufmann, Los Altos, CA, 1985, 170–189.

[Martins and Shapiro 1986a] J. P. Martins and S. C. Shapiro. Belief revision in SNePS. In *Proceedings of the Sixth Canadian Conference on Artificial Intelligence*, pages 230–234. Presses de l'Université du Québec, 1986.

[Martins and Shapiro 1986b] J. P. Martins and S. C. Shapiro. Theoretical foundations for belief revision. In J. Y. Halpern, editor, *Theoretical Aspects of Reasoning About Knowledge*, pages 383–398. Morgan Kaufmann Publishers, Los Altos, CA, 1986.

[Martins and Shapiro 1988] J. P. Martins and S. C. Shapiro. A model for belief revision. *Artificial Intelligence*, 35(1):25–79, 1988.

[McCarthy 1986] John McCarthy. Mental situation calculus. In Joseph Y. Halpern, editor, *Theoretical Aspects of Reasoning about Knowledge—Proceedings of the 1986 Conference*, page 307, 1986.

[McDermott 1978] D. McDermott. Planning and acting. *Cognitive Science*, 2:78ff, 1978.

[McDermott 1991] Drew McDermott. A Reactive Plan Language. Technical Report YALEU/CSD/RR #864, Yale University Department of Computer Science, August 1991.

[McDermott 1992] Drew McDermott. Robot Planning. *AI Magazine*, 13(2):55–79, 1992.

[McKay and Martins 1981] D. P. McKay and J. Martins. *SNePSLOG User's Manual.* Department of Computer Science, SUNY at Buffalo, 1981.

[McKay and Shapiro 1980a] D. P. McKay and S. C. Shapiro. MULTI: A LISP based multiprocessing system. Technical Report 164, Department of Computer Science, SUNY at Buffalo, 1980.

[McKay and Shapiro 1980b] D. P. McKay and S. C. Shapiro. MULTI — a LISP based multiprocessing system. In *Proceedings of the 1980 LISP Conference*, pages 29–37. Stanford University, Stanford, CA, 1980.

[McKay and Shapiro 1981] D. P. McKay and S. C. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 368–374, Los Altos, CA, 1981. Morgan Kaufmann.

[Moore 1980] R. Moore. Reasoning about knowledge and action. Technical Report 191, SRI AI Center, 1980.

[Moore 1985] Robert C. Moore. A formal theory of knowledge and action. In Jerry R. Hobbs and Robert C. Moore, editors, *Formal Theories of the Commonsense World*, pages 319–358. Ablex Publishing Corp., Norwood, NJ, 1985.

[Morgado and Shapiro 1983] E. J. Morgado and S. C. Shapiro. Believing and acting: An approach to meta-knowledge and meta-reasoning. SNeRG Technical Note 11, Department of Computer Science, SUNY at Buffalo, 1983.

[Morgado 1986] E. J. M. Morgado. Semantic networks as abstract data types. Technical Report 86–19, Department of Computer Science, SUNY at Buffalo, 1986.

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

[Nilsson 1988] Nils J. Nilsson. Action networks. In Jay Webber, Josh Tenenberg, and James Allen, editors, *Advance Proceedings of The Rochester Planning Workshop–From Formal Systems to Practical Systems*, pages 21–52, October 1988.

[Peot and Smith 1992] Mark A. Peot and David E. Smith. Conditional Nonlinear Planning. In James Hendler, editor, *Proceedings of the 1st Conference on AI Planning Systems (AIPS)*, pages 189–197, San Mateo, CA, 1992. Morgan Kaufmann Publishers.

[Pollack 1991] Martha E. Pollack. Overloading Intentions for Efficient Practical Reasoning. *Noûs*, XXV(4):513–536, September 1991.

[Rao and Georgeff 1991a] Anand S. Rao and Michael P. Georgeff. Asymmetry Thesis and Side-Effect Problems in Linear-Time and Branching-Time Intention Logics. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91*, pages 498–505. IJCAI and NCAIES/ACS, August 1991.

[Rao and Georgeff 1991b] Anand S. Rao and Michael P. Georgeff. Modeling Rational Agents within a BDI-Architecture. In *Principles of Knowledge Representation and Reasoning— Proceedings of the Second International Conference(KR91*, pages 473–485. AAAI, IJCAI, CSCSI, April 1991.

[Rao and Georgeff 1992] 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.

[Rapaport 1986] W. J. Rapaport. Syntactic semantics: Foundations of computational natural-language understanding. Technical Report 86–24, Department of Computer Science, SUNY at Buffalo, 1986.

[Reiter 1980] R. Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Rich 1991a] Charles Rich. CAKE: An Implemented Hybrid KR and Limited Reasoning System. In Charles Rich (Guest Editor), editor, *SIGART BULLETIN Special Issue on Implemented KRR Systems*, pages 120–127, June 1991.

[Rich 1991b] Charles Rich. Special Issue on Implemented Knowledge Representation and Reasoning Systems—Letter from the Guest Editor. *SIGART Bulletin*, 2(3), June 1991.

[Rosenschein and Kaelbling 1985] S. J. Rosenschein and L. P. Kaelbling. A formal approach to the design of intelligent embedded systems. In *Proceedings Conf on Theoretical Aspects of Reasoning*, 1985.

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

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

[Shapiro and Martins 1990] S. C. Shapiro and J. P. Martins. Recent advances and developments – the SNePS 2.1 report. In D. Kumar, editor, *Current Trends in SNePS–Semantic Network Processing System: Proceedings of the First Annual SNePS Workshop*, pages 1–13, Buffalo, NY, 1990. Springer-Verlag.

[Shapiro and McKay 1980] S. C. Shapiro and D. P. McKay. Inference with recursive rules. In *ProceedIngs of the First Annual National Conference on Artificial Intelligence*, pages 151–153, Los Altos, CA, 1980. Morgan Kaufmann.

[Shapiro and Rapaport 1987] 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.

[Shapiro and Rapaport 1991] Stuart C. Shapiro and William J. Rapaport. Models and minds: Knowledge representation for natural-language competence. In Robert Cummins and John Pollock, editors, *Philosophy and AI: Essays at the Interface*, pages 215–259. MIT Press, Cambridge, MA, 1991.

[Shapiro and Rapaport 1992] Stuart C. Shapiro and William J. Rapaport. The SNePS family. *Computers & Mathematics with Applications*, 23(2–5):243–275, January–March 1992.

[Shapiro *et al.* 1981] S. C. Shapiro, D. P. McKay, J. Martins, and E. Morgado. SNePSLOG: A "higher order" logic programming language. SNeRG Technical Note 8, Department of Computer Science, SUNY at Buffalo, 1981. Presented at the Workshop on Logic Programming for Intelligent Systems, R.M.S. Queen Mary, Long Beach, CA.

[Shapiro *et al.* 1989a] S. C. Shapiro, D. Kumar, and S. Ali. A propositional network approach to plans and plan recognition. In *Proceedings of the 1988 Workshop on Plan Recognition*, page 21, Los Altos, CA, 1989. Morgan Kaufmann.

[Shapiro *et al.* 1989b] Stuart C. Shapiro, Beverly Woolf, Deepak Kumar, Syed S. Ali, Penelope Sibun, David Forster, and Scott Anderson. Discussing, using, and recognizing plans–Annual Report for 1988. Technical report, North-East Artificial Intelligence Consortium, 1989.

[Shapiro 1979] S. C. Shapiro. The SNePS semantic network processing system. In N. V. Findler, editor, *Associative Networks: The Representation and Use of Knowledge by Computers*, pages 179–203. Academic Press, New York, 1979.

[Shapiro 1986] S. C. Shapiro. Symmetric relations, intensional individuals, and variable binding. *Proceedings of the IEEE*, 74(10):1354–1363, 1986.

[Shapiro 1989] S. C. Shapiro. Formal foundations of an intensional propositional semantic network. Presented at the Workshop on Formal Aspects of Semantic Networks, Santa Catalina Island, CA, 1989.

[Shapiro 1991a] Stuart C. Shapiro. Cables, paths and "subconsious" reasoning in propositional semantic networks. In John F. Sowa, editor, *Principles of Semantic Networks*, chapter 4, pages 137–156. Morgan Kaufmann, San Mateo, CA, 1991.

[Shapiro 1991b] Stuart C. Shapiro. Case studies of SNePS. *SIGART Bulletin*, 2(3):128–134, June 1991.

[Shubin 1981] H. Shubin. Inference and control in multiprocessing environments. Technical Report 186, Department of Computer Science, SUNY at Buffalo, 1981.

[Steele 1990] Guy L. Steele. *Common LISP.* Digital Press, second edition edition, 1990.

[Stroustrup 1987] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1987.

[Tate 1977] A. Tate. Generating project networks. In *Proceedings 5th IJCAI*, pages 888–93, 1977.

[Tate 1987] Austin Tate. Goal structure, holding periods and "clouds". In Michael P. Georgeff and Amy L. Lansky, editors, *Reasoning about Actions and Plans - Proceedings of the 1986 Workshop*, pages 267–278, Los Altos, CA, 1987. AAAI and CSLI, Morgan Kauffmann.

[Vere and Bickmore 1990] Steven Vere and Timothy Bickmore. A basic agent. *Computational Intelligence*, 6:41–60, 1990.

[Vere 1985] Steven A. Vere. Temporal scope of assertions and window cutoff. In *Proceedings of 9th IJCAI*, pages 1055–1059. IJCAI, 1985.

[Wegner 1987] Peter Wegner. Dimensions of Object-Based Language Design. *Proceedings of OOPSLA-87*, pages 168–182, 1987.

[Wilkins 1983] D. E. Wilkins. Representation in a domain-independent planner. In *Proceedings of the 8th IJCAI*, 1983.

[Wilkins 1984] D. E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22:269ff, 1984.

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

[Wirth 1988] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, fourth edition edition, 1988.

[Yonezawa and Tokoro 1987a] Akinori Yonezawa and Mario Tokoro. *Object-Oriented Concurrent Programming.* MIT Press, Cambridge, MA, 1987.

[Yonezawa and Tokoro 1987b] Akinori Yonezawa and Mario Tokoro. Object-oriented concurrent programming– an introduction. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 1–8. MIT Press, Cambridge, MA, 1987.