

Formal Specification and Implementation of a Multi-agent Information System Using Schema Based Reasoning¹

Torrii Murphy and Albert Esterline

NASA ACE

Dept. of Computer Science

North Carolina A&T State University

Greensboro, NC 27411

{torrii,esterlin}@ncat.edu

1. Introduction

Our work deals with a project currently pursued by the Automation Technology Section at NASA Goddard Space Flight Center (GSFC). One area targeted for initial agent technology development is information management [GSFC]. We here report our formalization and prototype of a system of agents for generating reports consisting of references to the literature on several technical areas. We follow the multi-agent system representation of the Agent-based FLight operations Associate (AFLOAT) system [Ma96] being developed at GSFC. In AFLOAT, each agent can be viewed as a miniature expert system, or an expert system in a limited domain. The schema-based reasoning (SBR) paradigm [Tu94] is followed. Our prototype implementation uses the CLIPS expert system shell [Gi93]. Finally, our formalization is presented in the Z++ object-oriented specification language [La95].

The technical areas on which our system generates references are exactly the areas we have applied to our work: CLIPS, Z++, SBR, and multi-agent systems. Our model involves an interface agent that generates a report and a specialist agent for each technical area. A user interacts directly with each specialist agent, which asks the user a series of questions to determine the topics of interest. Inter-agent communication is currently simulated by hand: when an agent indicates that its output is to be communicated to other agents, a user supplies that output as input to the other agents. Specialist agents communicate with each other to reduce redundant user interaction, which arises because the technical areas overlap. The most significant items communicated among agents, however, are the literature references that the specialist agents communicate to the interface agent, which compiles them into a report.

Section 2 of this paper gives a brief introduction to schema-based reasoning and discusses the schemas we have developed. Section 3 is the main section of the paper; after presenting background on Z++ and CLIPS, it discusses the specification and implementation of our system. Section 4 is the conclusion.

2. Schema-Based Reasoning

2.1. Background

In schema-based reasoning [Tu94], declarative knowledge structures called schemas are used to explicitly store an agent's problem-solving knowledge. There are three kinds of schemas: procedural schemas, contextual schemas, and strategy schemas. Procedural schemas, or *p-schemas*, are similar to hierarchical plans or scripts. They are interpreted by the agent's reasoning processes to carry out actions to achieve an agent's goals, and they are interruptible and resumable. Each p-schema step can be either a primitive action, a sub-goal to be achieved, or another p-schema. Contextual schemas, or *c-schemas*, represent contexts the agent may encounter and guide the agent within those contexts. In particular, a contextual schema holds information about responding to events, focusing attention, setting behavior parameters, and selecting actions to use to achieve goals. Strategic schemas, or *s-schemas*, are like domain-independent contextual schemas and hold information that defines the agent's problem-solving strategies for various kinds of problems. The heart of schema-based reasoning centers around a process that applies p-schemas to take action, while remaining ready to respond to the ever-changing wishes of current problem-solving situation. The reasoner finds an active goal using the current contextual and strategic schemas. To achieve the goal, a procedural schema, specifying steps to take based on the current context, is then applied. The process repeats until all goals are satisfied.

¹ The first author has been supported by grant NAG 5-4102, "Formal Foundations of Agents," from NASA/GSFC. The second author has been supported by the same grant and by grant NAG 2-1150, "Motion Planning in a Society of Intelligent Mobile Agents," from NASA/ARC.

2.2. Schema Outlines

Each of the four specialist agents in our system is an information specialist in its given area: Z++, SBR, multi-agent systems (MAS), or CLIPS. Each agent is able to adapt to provide meaningful references to either an expert or a novice in its area. In addition, the Z++ agent adapts to how much progress the user has made in specifying his system in Z++; the user could be either just beginning or almost finished specifying his system. Furthermore, the SBR agent can adapt to the type of the system the user wishes to seek reference information on; in this regard, systems are either simple or complex.

We do not use s-schemas. For each specialist agent, we specify one c-schema pattern, where conceptually there is a distinct c-schema for each set of values for the parameters in the pattern. For example, the following is the c-schema for the Z++ agent. There are two possible values for the parameter *level* (*expert* and *novice*) and two possible values for the parameter *progress* (*almost finished* and *beginning*). The combination of *novice* and *almost finished* is held not to be useful, so three instances of this pattern supply the possible c-schemas for the Z++ agent.

Actors: Z++ software agent, user agent

behavior parameters: determined by user brought about by calling various schemas based on the current context

objects: instances of Z_P_BEGIN_COMPLEX, Z_P_BEGIN_EXPERT, and Z_P_ALMOST_FINISHED_EXPERT

setting: clips environment

<u>level</u>	<u>progress</u>
expert	almost finished
expert	beginning
novice	beginning

goals: generation of Z++ references to include

states: findings of Z++ references to include

events: discovery of a finding of reference numbers to include

strategy: none

The following is an outline of the p-schema giving the Z++ agent's knowledge regarding a novice user who is currently beginning to specify his system with features of concurrency and real-time.

Goal: find concurrency and real time references

Conditions: user has Z++ Rigorous references

Results: user has concurrency and real time references

Steps:

get references for concurrent & real time(y/n)

...

get references for concurrent & real time(y/n)

(using p-Concurrent & Real Time Issues of Z++ Beginner & Novice)

This schema prompts the user about references for various subtopics. A negative response bars proceeding any further with the subtopic. A positive response, however, triggers other prompts needed for the subtopic. A p-schema for an expert presents the list of subtopics in a menu without the explanation provided to novices; it is assumed that an expert can locate subtopics of interests without explanation.

3. Z++ Specifications and CLIPS Implementation

We now present Z++ specifications of our multi-agent system and the CLIPS implementations. We start with brief introductions to Z++ (§3.1) and CLIPS (§3.2). The system is specified as a set of class specifications: several classes for schemas (§3.4), a class for the report system as a whole (§3.5), a class for the specialist agents (§3.6), and a class for the interface agent; space restrictions prohibit discussion of the latter. We discuss the implementation of a class along with the specification of the class. The implementations of the system as a whole and of the agents do not use object-oriented features; it is informative to see why object-oriented specification is appropriate here. We begin the specification, in §3.3, with certain preliminaries that fall outside all the class specifications.

3.1. Z++

Z++ [La95] is an object-oriented extension of the model-based specification language Z. A Z++ specification consists of a collection of class definitions of the following form, where some less-used features are ignored:

```
CLASS C
OWNS
```

c

```

OPERATIONS
    m : Inm -> OUTm;
    ...
ACTIONS
    Prem,c &
        m x? y! ==> Defm,c;
    ...
INVARIANT
    Invc
HISTORY
    Hc
END CLASS

```

Each clause is optional, although an *ACTIONS* clause requires an *OPERATIONS* clause. *c* contains instance variable declarations, in the style of variable declarations in Z. The *OPERATIONS* list declares the types of the operations, as functions from a sequence of input domains to an output domain. The *INVARIANT* gives a predicate that specifies the properties of the internal state which must hold at any time point where the object is interacting with other objects or external agents. There are also features to express inheritance and type parameters.

The *ACTIONS* list gives the definitions of the various operations declared in the *OPERATIONS* clause. Input parameters (ending with “?”) are listed before the output parameters (ending with “!”). Z predicates and method invocations can be used to define methods. Operations are given explicit preconditions by the notation

$$m\ x?\ y!\ ==>\ PRE\ Pre_{m,c}\ THEN\ Def_{m,c}\ END$$

The preconditions of a method *m* of a class *C* are denoted by $Pre_{m,c}$. The definition of a method *m* of the class *C* is denoted by $Def_{m,c}$. In this definition, if α is a variable, then α' denotes the value of α after *m* is executed, while α without the dash denotes the value of α before *m* is executed.

The *HISTORY* clause of a class is a Real-Time Logic, or RTL, predicate. For each method *m* of a class *C*, there is in RTL an associated set of events expressed in terms of an invocation instance of *m*. The expressions represent method initiation and termination and arrival of a request at the object. For a given class *C*, there are a number of terms that can occur in the formulae of its RTL Language. Variables, attributes, and applications of *n*-ary function symbols can be expressed in RTL. Event occurrences are of the form (E,i) , indicating the *i*th occurrence of event *E*. Other terms, including time-valued terms, can be expressed. RTL formulae are constructed from terms, predicates, temporal operators, and quantifiers.

We shall explain more about Z++ as the need arises.

3.2. CLIPS

CLIPS [Gi93] is an expert system tool that represents knowledge in three ways: rules, functions, and object-oriented programming. The left-hand side of a rule consists of one or more patterns. The rule is activated when each pattern is matched against a fact or an instance of a user-defined class. When a rule is activated, it is put on the agenda; CLIPS automatically determines which activation on the agenda is appropriate to fire. When a rule fires, the actions in its left-hand side are executed. Typical actions are asserting and retracting a fact, invoking a function, and sending a message to an object.

Data are generally recorded as facts, each consisting of one or more fields enclosed within parentheses. Facts with the same first field are generally thought of as extensionally defining the relation denoted by that field. Facts are typically asserted and retracted throughout a run. Objects may also be used to represent data and, like facts, may match rule patterns, but the set of objects, unlike the set of facts, is generally not volatile. We shall explain more about CLIPS as the need arises.

3.3. Preliminaries

To begin with, we need the following Z-style “given sets”:

$$[TAG, TOPIC, TEXT, S_AGENT_NAME]$$

TAG is the set of names used for reference topics within a specialist agent. *TOPIC* is the same set, but as used within the interface agent. *TEXT* is the set of literature references. And *S_AGENT_NAME* is the set of names of specialist agents. We use Z-style data-type declarations to enumerate aspects of the system the user intends to develop:

$$Level ::= expert \mid novice \mid don't\ care$$

$$SystemProgress ::= beginning \mid almost\ finished \mid don't\ care$$

SystemType ::= simple | complex | don't care

Aspects of the system that do not require explicit specification are expressed with *don't care*.

3.3. Schema Classes

Classes *SpecialistAgent* and *InterfaceAgent* deal directly with SBR schemas. Class *C_Schema* has instance variables for the aspects of the system to be designed and the user:

```
CLASS C_Schema
OWNS
  level: Level;
  sytem_progress: SystemProgress;
  system_type: SystemType;
END
```

There is only one instance of class *C_Schema* for each specialist agent. The effect of different c-schemas is obtained by varying the values of the instance variables. Class *P_Schema* has all the instance variables of *C_Schema* (since a change of context requires new p-schemas). It also has a *tag* variable indicating the reference topic and a *subtopics* variable for the subtopics of the topic given by *tag*:

```
tag: TAG;
subtopics: P TAG
```

Each specialist agent has a number of instances of *P_Schema*. For each context (values for the instance variables *level*, *system_progress*, and *system_type*), there is a hierarchy of *P_Schema* objects (that is formally a forest) described by identifying the values of the *subtopics* variable of a *P_Schema* object with the values of the *tag* variables of its child *P_Schema* objects. The interaction between a specialist agent and a user generally follows this hierarchy from topic to subtopic.

To implement these two classes in CLIPS, we define (using the `defclass` command) an abstract class SCHEMA that is a direct subclass of the predefined USER class. We define two subclasses of SCHEMA: C_SCHEMA and P_SCHEMA. These have as slots all the specified instance variables, except that the *subtopics* variable of *P_Schema* is implemented as a multislot, which has zero or more values. An indirect subclass of P_SCHEMA that is concrete (can be instantiated) is defined for each significant context. For each specialist domain, for each significant context there is an instance of the concrete class for each topic in that domain; these instances are defined together using CLIPS' `definstances` command. Each `definstances` command occupies its own file.

The *InterfaceAgent* object, instead of using objects of class *P_Schema*, uses objects of class *Gentext*, which is similar to *P_Schema* but has *TOPIC*s instead of *TAG*s and adds a *nexttopic* instance variable that allows the driver for the interface agent to jump from one node in the hierarchy of topics to a sibling so that the entire hierarchy may be traversed. The *Gentext* class is implemented as the abstract GENTEXT class in CLIPS, similar to the P_SCHEMA class.

3.4. Class ReportSystem

The overall system is specified as *CLASS ReportSystem*. The *OWNS* clause of this class specifies three instance variables, and there is an invariant that directly relates to the interpretation of the first instance variable:

```
OWNS
  tt: TOPIC >@ TAG;
  interf_agent : InterfaceAgent;
  spec_agents: P SpecialistAgent
INVARIANTS
  "t: TAG · $ tp : TOPIC · tp = tt(t)
```

The type associated with *tt* indicates that it is a total injection from *TOPIC* to *TAG*. The invariant shown here indicates that *tt* is a surjection. So, combining, we thus specify that *tt* is a bijection between *TOPIC* and *TAG*. Instance variable *interf_agent* is an instance of class *InterfaceAgent* and instance variable *spec_agents* is a set (see "P") of instances of class *SpecialistAgent* – the *ReportSystem* class is a client of these classes.

Instance variable *interf_agent* corresponds, in the CLIPS implementation, to the CLIPS process executing the set of files defining the interface agent: there is no CLIPS construct corresponding to this agent. Similarly, the set-valued instance variable *spec_agents* corresponds to the set of CLIPS processes executing another set of files. The bijection *tt* between *TOPIC* (interface agent) and *TAG* (specialist agents) is realized by maintaining the required one-one correspondence, but there is no direct CLIPS implementation of it.

Besides an initialization method (which invokes the initialization methods of the *InterfaceAgent* and *SpecialistAgent* classes), *ReportSystem* has a *transmit* method that takes as an argument a five-tuple (a “fact”) and returns a similar five-tuple. In the *OPERATIONS* clause we have

$$\text{transmit } S_AGENT_NAME \times TOPIC \times Level \times SystemProgress \times SystemType @ \\ S_AGENT_NAME \times TOPIC \times Level \times SystemProgress \times SystemType$$

In the *ACTIONS* clause we have

$$\text{transmit } f? \text{ } f! ==> PRE \text{ true } THEN \text{ } f! = f? \text{ } END$$

So *transmit* takes a tuple consisting of the source specialist agent’s name, the topic, and the various system attributes and returns the same – specifically, to the other specialist agents and the interface agent, thus communicating the features handled by the source agent. The *transmit* method is a specification of what is done by hand; there is no code that implements it.

The invariant clause of *ReportSystem* specifies that the tags of specialist agents correspond to the topics of the interface agent, that function *relates-to* gives relationships between pairs of specialist agents, that sets of tags from different specialist agents are disjoint, and that every p-schema belongs to a specialist agent. The history clause for *ReportSystem* includes the following conjunct.

There is a bound on the delay between when a fact enters the fact list of a specialist agent and when the interface agent receives the fact, and no fact can be received by the interface agent unless it has been transmitted.

The second part of this is expressed as follows.

$$\begin{array}{l} \text{-----} \quad \text{-----} \\ "sa : SpecialistAgent ; ia : InterfaceAgent ; ir: TOPIC \times Level \times SystemProgress \times SystemType / \\ sa \in spec_agents \wedge ia = interf_agent \cdot \\ \square \uparrow (receive_fact(cons(sa.name,ir)), ia, 1) \geq \uparrow transmit(cons(sa.name,ir), 1) \end{array}$$

Here $cons(sa.name,ir)$ is the result of prefixing the name of specialist agent *sa* to the fact *ir*, giving essentially an extended fact. $\square \uparrow (receive_fact(cons(sa.name,ir)), ia, 1)$ is the time (\square) the *receive-fact* method with argument $cons(sa.name,ir)$ was first (1) invoked (\uparrow) by *ia*. $\uparrow transmit(cons(sa.name,ir), 1)$ is the time (no \square needed here) the report system first invoked the *transmit* method with argument $cons(sa.name,ir)$. So this clause states that, for any specialist agent *sa*, interface agent *ia*, and fact *ir*, the time *ir* invokes the method to receive an extended fact consisting of *sa*’s name and *ir* can be no earlier than the time the system invokes the transmit method for that extended fact. Note that the constraint (between the “|” and the “•”) requires that *sa* be a member of the value of the *spec_agents* instance variable of the report system and that *ia* be the value of its *interf_agent* instance variable.

3.5. Class SpecialistAgent

The instance variables for class *SpecialistAgent* are

$$\begin{array}{l} agent\text{-name}: S_AGENT_NAME; \\ initial\text{-fact}: Boolean; \\ candidates: \mathbb{P} TAG; \\ initial\text{-candidates}: (Level \times SystemProgress \times SystemType) > @ \mathbb{P} TAG; \\ relates\text{-to}: (S_AGENT_NAME \times TAG) @ \mathbb{P} TAG; \\ incl\text{-refs}: \mathbb{P}(TAG \times (Level \times SystemProgress \times SystemType)); \\ p_schs : (Level \times SystemProgress \times SystemType) > @ \mathbb{P} P_Schema; \\ c_sch: C_Schema \end{array}$$

Here *agent-name* is the name of the specialist agent instantiating this class. Variable *initial-fact* is true after initialization but before the initial context and initial set of candidates has been established. Variable *candidates* contains the tags for all topics the user can currently choose; when a subtopic is chosen, its tag is removed from this set and the tags of its subtopics are included. Function *relates-to* is used when another specialist agent communicates its name and a topic it has just included; this function relates the agent and topic to a (often empty) set of topics that this agent will include as candidates. *incl-refs* is the set of tag-context pairs that the interface agent will translate into references; it is progressively constructed. *p_schs* contains hierarchies (structured with the subtopics instance variable of class *P-Schema*) of p-schems, one for each context. Finally, *c-sch* is the c-schema object that represents different c-schemas with different values of its instance variables.

Variable *initial-fact* is implemented by the CLIPS system, which asserts a fact (*initial-fact*) as part of the initialization done when the *reset* command is executed. Each tag *tg* that is a member of *candidates* is represented as a CLIPS fact (*candidate tg*). Similarly, instance variable *incl-refs* is implemented as a set of CLIPS facts (*inc-refs <tag> <level>*). (Set-valued instance variables often are conveniently implemented as sets of CLIPS facts satisfying a given template.) The functional instance variable *initial-candidates* corresponds to the initial set of candidate facts asserted when the initial context is established – so the notion of a mapping from a context to a set of tags is preserved. Variable *p_schs* is implemented by files of P_SCHEMA objects, as described above, where each file corresponds to a different significant context. *c_sch* is implemented as an instance of class C_SCHEMA, and *s_schs* is not implemented. Implementation of *relates-to* is more complex so will be ignored here.

Specification class *SpecialistAgent* has an initialization method to set the *p_schs* instance variable, a method that establishes the initial c-schema and enables the driver, a broadcast method that outputs a fact (a topic-context pair), and a method to receive a fact and (using *relates-to*) possibly updating the *candidates* set. The most important method is the driver, whose signature (in the *OPERATIONS* clause) is

driver: TAG @

and whose definition (in the *ACTIONS* clause) is

```

driver t? ==>
PRE
  t? ∈ candidates
THEN
  incl-refs  $\mathcal{C}$  = incl-refs  $\cup$  {(t?, c_sch.state)}
   $\wedge$  candidates  $\mathcal{C}$  = candidates  $\setminus$  {t?}  $\cup$  tag_to_object(t?, c_sch.state).subtopics
END

```

The precondition here is that the tag supplied as input, *t?*, be a member of the current set of candidate tags. If this is met, then the new set of reference facts (*incl-refs* \mathcal{C}) is the old set with the addition of a fact consisting of this tag and the current context; also, the new set of candidate tags is the old set, with the input tag, *t?*, removed, augmented with the set of subtopic tags of *t?*. Function *tag_to_object* returns the p-schema object with a given tag and context; we access the *subtopics* instance variable of the object returned.

In the implementation, the CLIPS *reset* command is used for initialization. A rule *start-it* is used to establish the initial c-schema; it prompts the user for a context, establishes that context, and establishes the initial set of candidates. The methods to broadcast a fact, to receive a fact, and to change context are all implemented as part of the driver, which itself is actually implemented as two rules, one for novice-level contexts and one for expert-level contexts. For broadcasting a fact, a driver rule, when it asserts an *incl-refs* fact, also outputs that fact so that a user may supply it as input to another specialist agent. For receiving facts, the driver rules, responding to an option selected by the user, input and assert *incl-refs* facts and essentially apply the *relates-to* function to them. The only component of the context that can be changed is the level; responding to a user-selected option, a driver rule updates the context and switches to a different p-schema hierarchy.

4. Conclusion

The work reported here is one of the few cases where a formal specification has been given of a multi-agent system or where a formal specification has been implemented in an expert system tool. Our approach nonetheless has been successful for quite good reasons. Our formal specification language, Z++, lends itself to concurrent agents as it captures concurrent features with RTL and readily models agents as instances of classes. Again, encodings of expert systems generally aim to be declarative, like formal specifications, which emphasize “what” not “how”.

Many of the object-oriented features in the formal specification are implemented without object-oriented constructs. This is notably the case for the agents, where, however, the implementation follows quite naturally from the specification: the unit of definition is the file, not the class, and instance variables are implemented as sets of facts, objects, and even functions. CLIPS rules, it turns out, provide a natural means to implement Z++ methods: the left-hand side of a rule is like the preconditions and the right-hand side is like the definition or postconditions.

Schemas are naturally specified as instances of Z++ classes and implemented as CLIPS objects. We found that much of the functionality specified for the agents is most naturally implemented with message handlers (methods) for schema classes. This suggests that it may be advantageous to view schemas as more active than is generally thought. This more active view can be reconciled with the declarative nature of schemas if we concentrate on the formal specification.

References

- [Gi93] Giarratano, J.C., *CLIPS User's Guide* (CLIPS Version 6.0). NASA L.B. Johnson Space Center, Information Systems Directorate, Software Technology Branch, 1993.
- [GSFC] *Intelligent Agent-Based Systems in the NASA/GSFC Context* (Draft). <http://groucho.gsfc.nasa.gov/agents/documents/code522/agentpaper.html>.
- [La95] Lano, K., *Formal Object-Oriented Development*. London: Springer-Verlag, 1995.
- [Ma96] Maynard-Reid, P., *Architectural, Knowledge Representation, and Formalization Issues in AFLOAT*. Automation Technology Section, NASA Goddard Space Flight Center, 1996.
- [Tu94] Turner, R., *Adaptive Reasoning for Real-World Problems: A Schema-Based Approach*. Hillsdale, NJ: Lawrence Erlbaum, 1994.