To appear in *Intelligent Agent Systems: Theoretical and Practical Issues (Based on a Workshop Held at PRICAI '96 Cairns, Australia, August 1996)*, Cavedon, L., Rao, A., and Wobcke, W. (Eds.), LNAI volume 1209, Springer-Verlag, 1997.

An Experiment in Using Golog to Build a Personal Banking Assistant*

Yves Lespérance[1], Hector J. Levesque[2], and Shane J. Ruman[2]

[1] Department of Computer Science, Glendon College, York University, 2275 Bayview Ave., Toronto, ON, Canada M4N 3M6
lesperan@yorku.ca

[2] Department of Computer Science, University of Toronto
Toronto, ON, Canada M5S 1A4
{hector,shane}@ai.toronto.edu

Abstract. Golog is a new programming language based on a theory of action in the situation calculus that can be used to develop multi-agent applications. The Golog interpreter automatically maintains an explicit model of the agent's environment on the basis of user supplied axioms about the preconditions and effects of actions and the initial state of the environment. This allows agent programs to query the state of the environment and consider the effects of various possible courses of action before deciding how to act. This paper discusses a substantial multi-agent application developed in Golog: a system to support personal banking over computer networks. We describe the overall system and provide more details on the agent that assists the user in responding to changes in his financial situation. The advantages and limitations of Golog for developing multi-agent applications are discussed and various extensions are suggested.

1 Introduction

Golog is a new logic programming language for developing intelligent systems that are embedded in complex environments and use a model of the environment in deciding how to act [7, 4]. It is well suited to programming expert assistants, software agents, and intelligent robots. The language is based on a formal theory of action specified in an extended version of the situation calculus. The Golog interpreter automatically maintains an explicit model of the system's environment on the basis of user supplied axioms about the preconditions and effects of actions and the initial state of the environment. This allows programs to query the state of the environment and consider the effects of various possible courses of action before committing to a particular alternative. The net effect is that

* This research was made possible by financial support from the Information Technology Research Center, the Natural Science and Engineering Research Council, and the Institute for Robotics and Intelligent Systems.

programs may be written at a much higher level of abstraction than is usually possible. A prototype implementation in Prolog has been developed.

In this paper, we discuss the most substantial experiment done so far in using Golog to develop an application. The application is a system that assists its users in doing their personal banking over computer networks. The system is realized as a collection of Golog agents that interact. Users can perform transactions using the system. They can also have the system monitor their financial situation for particular conditions and take action when they arise, either by notifying them or by performing transactions on their behalf. Currently, the system only works in a simulated financial environment and has limited knowledge of the domain. But with more than 2000 lines of Golog code, it is certainly more than a toy.

The personal banking application made an excellent test domain for a number of reasons. A shift from branch-based to PC-based banking would have far reaching and largely positive effects for customers and banks alike. Most banks and other financial institutions are actively investigating PC-based banking and electronic commerce. However, successful implementations of home banking will undoubtedly require more flexibility and power than a simple client-monolithic server environment can provide. Software agents can fill this need. Financial systems need to be extremely flexible given the number of options available, the volatility of markets, and the diversity of the user needs. Software agents excel at this type of flexibility. Furthermore, the distributed nature of information in the financial world (i.e. banks do not share information, users do not trust outside services, etc.) requires that applications have a distributed architecture. As well, Golog's solid logical foundations and suitability for formal analysis of the behavior of agents are attractive characteristics in domains that involve financial resources. Finally, personal banking applications raise interesting problems which vary substantially in complexity; this experiment resulted in a system that has some interesting capabilities; it is clear that it could be extended to produce a very powerful application.

In the next section, we outline the theory of action on which Golog is based. Then, we show how complex actions can be defined in the framework and explain how the resulting set of complex action expressions can be viewed as a programming language. In section 5, we present the overall structure of the personal banking system and then focus on the agent that assists the user in responding to changes in his financial situation, giving more details about its design. In the last section, we reflect on this experiment in the use of Golog: what were the advantages and limitations of Golog for this kind of multi-agent application? We also mention some extensions of Golog that are under development and address some of the limitations encountered.

2 A Theory of Action

Golog is based on a theory of action expressed in the situation calculus [11], a predicate calculus dialect for representing dynamically changing worlds. In this framework, the world is taken to be in a certain state (or situation). That state can only change as a result of an action being performed. The term $do(a, s)$ represents the state that results from the performance of action a in state s. For example, the formula $\textsc{On}(\mathrm{B}_1, \mathrm{B}_2, do(\textsc{putOn}(\mathrm{B}_1, \mathrm{B}_2), s))$ could mean that B_1 is on B_2 in the state that results from the action $\textsc{putOn}(\mathrm{B}_1, \mathrm{B}_2)$ being done in state s. Predicates and function symbols whose value may change from state to state (and whose last argument is a state) are called *fluents*.

An action is specified by first stating the conditions under which it can be performed by means of a *precondition axiom* using a distinguished predicate $Poss$. For example,

$$\begin{array}{l} Poss(\textsc{createAlert}(alertMsg, maxPrio, monID), s) \equiv \\ \neg\exists alertMsg', maxPrio'\ \textsc{Alert}(alertMsg', maxPrio', monID, s) \end{array} \tag{1}$$

means that it is possible in state s to create an alert with respect to the monitored condition $monID$ with the alert message $alertMsg$ and the maximum priority $maxPrio$, provided that there is not an alert for the condition $monID$ already; creating an alert leads the agent to send alert messages to the user with a degree of obtrusiveness rising up to $maxPrio$ until it gets an acknowledgement. Secondly, one specifies how the action affects the world's state with *effect axioms*. For example,

$$\begin{array}{l} Poss(\textsc{createAlert}(alertMsg, maxPrio, monID), s) \supset \\ \textsc{Alert}(alertMsg, maxPrio, monID, \\ \qquad do(\textsc{createAlert}(alertMsg, maxPrio, monID), s)) \end{array}$$

says that if the action $\textsc{createAlert}(alertMsg, maxPrio, monID)$ is possible in state s, then in the resulting state an alert is in effect with respect to the monitored condition $monID$ and the user is to be alerted using the message $alertMsg$ sent at a priority up to $maxPrio$.

The above axioms are not sufficient if one wants to reason about change. It is usually necessary to add frame axioms that specify when fluents remain unchanged by actions. The frame problem [11] arises because the number of these frame axioms is of the order of the product of the number of fluents and the number of actions. Our approach incorporates a treatment of the frame problem due to Reiter [14] (who extends previous proposals by Pednault [12], Schubert [16] and Haas [3]). The basic idea behind this is to collect all effect axioms about a given fluent and assume that they specify all the ways the value of the fluent may change. A syntactic transformation can then be applied to obtain a *successor state axiom* for the fluent, for example:

$$\begin{array}{l} Poss(a, s) \supset \\ [\textsc{Alert}(alertMsg, maxPrio, monID, do(a, s)) \equiv \\ \ a = \textsc{createAlert}(alertMsg, maxPrio, monID) \vee \\ \ \textsc{Alert}(alertMsg, maxPrio, monID, s) \wedge \\ \ a \neq \textsc{deleteAlert}(monID)]. \end{array} \tag{2}$$

This says that an alert is in effect for monitored condition $monID$ with message $alertMsg$ and maximum priority $maxPrio$ in the state that results from action

a being performed in state s iff either the action a is to create an alert with these attributes, or the alert already existed in state s and the action was not that of canceling the alert on the condition. This treatment avoids the proliferation of axioms, as it only requires a single successor state axiom per fluent and a single precondition axiom per action.[3] It yields an effective way of determining whether a condition holds after a sequence of primitive actions, given a specification of the initial state of the domain.

3 Complex Actions

Actions in the situation calculus are primitive and determinate. They are like primitive computer instructions (e.g., assignments). We need complex actions to be able to describe complex behaviors, for instance that of an agent.

Complex actions could be treated as first class entities, but since the tests that appear in forms like **if** ϕ **then** δ_1 **else** δ_2 **endIf** involve formulas, this means that we would have to reify fluents and formulas. Moreover, it would be necessary to axiomatize the correspondence between these reified formulas and the actual situation calculus formulas. This would result in a much more complex theory.

Instead we treat complex action expressions as abbreviations for expressions in the situation calculus logical language. They may thus be thought of as macros that expand into the genuine logical expressions. This is done by defining a predicate Do as in $Do(\delta, s, s')$ where δ is a complex action expression. $Do(\delta, s, s')$ is intended to mean that the agent's doing action δ in state s leads to a (not necessarily unique) state s'. Do is defined inductively on the structure of its first argument as follows:

- *Primitive actions:*
$$Do(a, s, s') \stackrel{\text{def}}{=} Poss(a, s) \wedge s' = do(a, s).$$
- *Test actions:*
$$Do(\phi?, s, s') \stackrel{\text{def}}{=} \phi[s] \wedge s = s'$$
$\phi[s]$ denotes the situation calculus formula obtained from ϕ by restoring situation variable s as the suppressed situation argument.
- *Sequences:*
$$Do([\delta_1; \delta_2], s, s') \stackrel{\text{def}}{=} \exists s^* (Do(\delta_1, s, s^*) \ \wedge \ Do(\delta_2, s^*, s')).$$
- *Nondeterministic choice of action:*
$$Do((\delta_1 \mid \delta_2), s, s') \stackrel{\text{def}}{=} Do(\delta_1, s, s') \ \vee \ Do(\delta_2, s, s').$$
- *Nondeterministic choice of argument:*
$$Do(\pi x \, \delta(x), s, s') \stackrel{\text{def}}{=} \exists x \, Do(\delta(x), s, s').$$

[3] This discussion ignores the ramification and qualification problems; a treatment compatible with our approach has been proposed by Lin and Reiter [9].

– *Nondeterministic iteration:*[4]

$$
\begin{aligned}
Do(\delta^*, s, s') \stackrel{\text{def}}{=} \forall P\{ & \\
& \forall s_1[P(s_1, s_1)] \quad \wedge \\
& \forall s_1, s_2, s_3[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)]\,\} \\
& \qquad \supset \ P(s, s').
\end{aligned}
$$

There is another case to the definition that handles procedure definitions (including recursive ones) and procedure calls. The complete definition appears in [7].

As in dynamic logic [13], conditionals and while-loops can be defined in terms of the above constructs as follows:

$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} \stackrel{\text{def}}{=} [\phi?; \delta_1] | [\neg\phi?; \delta_2],$$

$$\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile} \stackrel{\text{def}}{=} [[\phi?; \delta]^*; \neg\phi?].$$

We also define an iteration construct **for** $x : \phi(x)[\delta(x)]$ that performs $\delta(x)$ for all x's such that $\phi(x)$ holds (at the beginning of the loop).[5]

4 Golog

The theoretical framework developed above allows us to define a programming language called Golog (alGOl in LOGic). A Golog program includes both a declarative component and a procedural component. The declarative component specifies the primitive actions available to the agent, when these actions are possible and what effects they have on the agent's world, as well as the initial state of the world. The programmer supplies:

– precondition axioms, one per primitive action,
– successor state axioms, one per fluent,
– axioms specifying what fluents holds in the initial state S_0.

The procedural part of a Golog program defines the behavior of the agent. This behavior is specified using an expression in the language of complex actions introduced in the previous section (typically involving several procedure definitions

[4] We use second order logic here to define $Do(\delta^*, s, s')$ as the transitive closure of the relation $Do(\delta, s, s')$ — transitive closure is not first order definable. A first order version is used in the implementation of Golog, but it is insufficient for proving that an iterative program does not terminate.

[5] **for** $x : \phi(x)[\delta(x)]$ is defined as:

[**proc** $P(Q)$ /* where P is a new predicate variable */
  if $\exists y\, Q(y)$ **then** $\pi\, y, R\, [Q(y) \wedge \forall z (R(z) \equiv Q(z) \wedge z \neq y)?; \delta(y); P(R)]$ **endIf**
endProc;
$\pi\, Q\, [\forall z (Q(z) \equiv \phi(z))?; P(Q)]$]

followed by one or more calls to these procedures). Here's a simple example of a Golog program to get an elevator to move to the ground floor of a building:

$$
\begin{array}{l}
\textbf{proc } \text{DOWN}(n) \\
\quad (n = 0)? \mid \text{DOWNONEFLOOR}; \text{DOWN}(n-1) \\
\textbf{endProc}; \\
\pi\, m\, [\text{ATFLOOR}(m)?; \text{DOWN}(m)]
\end{array}
$$

In the next section, we will see a much more substantial example.

Golog programs are evaluated with a theorem prover. In essence, to execute a program, the Golog interpreter attempts to prove

$$Axioms \models \exists s Do(program, S_0, s).$$

that is, that there exists a legal execution of the program. If a (constructive) proof is obtained, a binding for the variable $s = do(a_n, \ldots do(a_2, do(a_1, S_0)) \ldots)$ is extracted, and then the sequence of actions $a_1, a_2, \ldots, a_n$ is sent to the primitive action execution module.

The declarative part of the Golog program is used by the interpreter in two ways. The successor state axioms and the axioms specifying the initial state are used to evaluate the conditions that appear in the program (test actions and **if/while/for** conditions) as the program is interpreted. The action preconditions axioms are used (with the other axioms) to check whether the next primitive action is possible in the state reached so far. Golog programs are often nondeterministic and a failed precondition or test action causes the interpreter to backtrack and try a different path through the program. For example, given the program $(a; P?) \mid (b; c)$, the Golog interpreter might determine that a is possible in the initial state S_0, but upon noticing that P is false in $do(a, S_0)$, backtrack and return the final state $do(c, do(b, S_0))$ after confirming that b is possible initially and that c is possible in $do(b, S_0)$.

Another way to look at this is that the Golog interpreter automatically maintains a model of the world state using the axioms and that the program can query this state at run time. If a program is going to use such a model, it seems that having the language maintain it automatically from declarative specifications would be much more convenient and less error prone than the user having to program such model updating from scratch. The Golog programmer can work at a much higher level of abstraction.

This use of the declarative part of Golog programs is central to how the language differs from superficially similar "procedural languages". A Golog program together with the definition of Do and some foundational axioms about the situation calculus *is* a formal theory about the possible behaviors of an agent in a given environment. And this theory is used explicitly by the Golog interpreter. In contrast, an interpreter for an ordinary procedural language does not use its semantics explicitly. Nor do standard semantics of programming languages refer to aspects of the environment in which programs are executed [1].

Note that our approach focuses on high-level programming rather than plan synthesis at run-time. But sketchy plans are allowed; nondeterminism can be used to infer the missing details. For example, the plan

$$\textbf{while } \exists b\, \textsc{OnTable}(b) \textbf{ do } \pi b\, \textsc{remove}(b) \textbf{ endWhile}$$

leaves it to the Golog interpreter to find a legal sequence of actions that clears the table.

Before moving on, let us clarify one point about the interpretation of Golog programs. The account given earlier suggests that the interpreter identifies a final state for the program before any action gets executed. In practice, this is unnecessary. Golog programs typically contain fluents that are evaluated by sensing the agent's environment. In the current implementation, whenever the interpreter encounters a test on such a fluent, it commits to the primitive actions generated so far and executes them, and then does the sensing to evaluate the test. One can also add directives to Golog programs to force the interpreter to commit when it gets to that point in the program. As well, whenever the interpreter commits and executes part of the program, it rolls its database forward to reflect the execution of the actions [8, 10].

We have developed a prototype Golog interpreter that is implemented in Prolog. This implementation requires that the program's precondition axioms, successor state axioms, and axioms about the initial state be expressible as Prolog clauses. Note that this is a limitation of the implementation, not the theory. For a much more complete presentation of Golog, its foundations, and its implementation, we refer the reader to [7].

Prior to the personal banking system documented here, the most substantial application developed in Golog had been a robotics one [4]. The robot's task was mail delivery in an office environment. A high-level controller for the robot was programmed in Golog and interfaced to a software package that supports path planning and local navigation. The system currently works only in simulation mode, but we are in the process of porting it to a real robot.

Golog does not in itself support multiple agents or even concurrent processes; all it provides is a sequential language for specifying the behavior of a system. A straightforward way of using it to implement a multi-agent system however, is to have each agent be a Golog process (under Unix with its own copy of the interpreter), and provide these agents with message passing primitive actions that are implemented using TCP/IP or in some other way. This is what was done for the personal banking system described below.

5 The Personal Banking Application

5.1 System Overview

As discussed in the introduction, the multi-agent paradigm is well suited to the decentralized nature of network banking applications. In our experimental system, different agents correspond to the different institutions involved and/or to major subtasks to be handled by the system. The different types of agents in our system and the kind of interactions they have are represented in figure 1. We have:

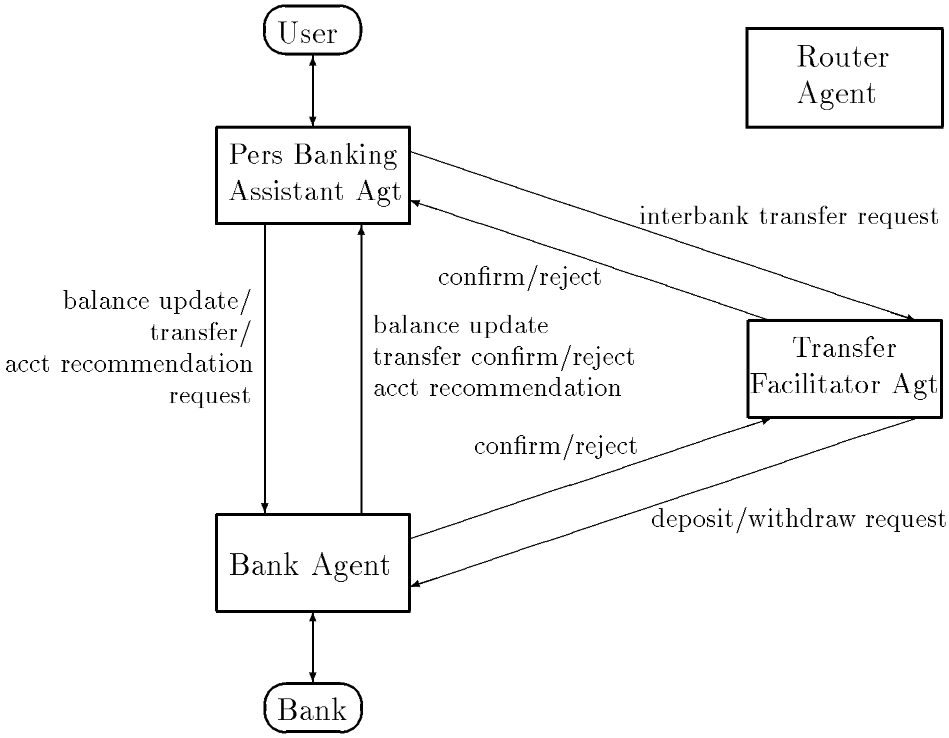

Fig. 1. System components.

- *personal banking assistant agents*, which perform transactions under the direction of their user and monitor his account balances for problem conditions; when the agent detects such a condition, it tries to alert the user or resolve the problem on his behalf;
- *bank agents*, which perform operations on users' accounts at the agent's bank in response to requests; they also provide information on the types of accounts available at the bank;
- *transfer facilitator agents*, which take care of funds transfer between different institutions;
- *router agents*, which keep track of agents' actual network addresses and dispatch messages;
- *automated teller machine agents*, which provide a simple ATM-like interface to bank agents (not represented on the figure).

Figure 2 shows the system's graphical user interface, which is implemented by attaching C and Tcl/Tk procedures to Golog's primitive actions for the domain. Currently, the personal banking assistant agents are the most interesting part of the system and are the only agents to have any kind of sophisticated behavior. We describe their design in detail in the next section. A complete description of

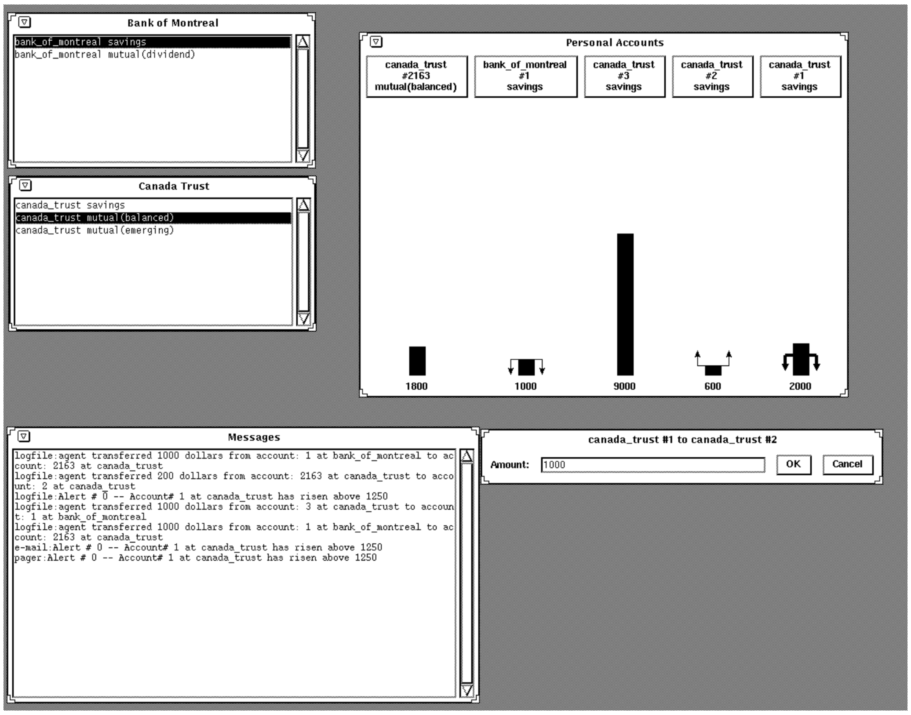

Fig. 2. System user interface.

the whole system appears in [15].

A number of assumptions and simplifications were made, chiefly regarding communication: messages exchanged between agents, users, and other system components always reach their destination in a relatively timely manner; communicating processes do not change their addresses without telling the router, and reliable connections can be made quickly and at any time. Furthermore, each agent assumes that the other agents and processes it relies on are available most of the time, and respond to queries and commands within certain time bounds (often very generous). We also ignore all issues associated with security and authentication. Undoubtedly, these assumptions would have to be lifted and overall robustness improved in order to move beyond our experimental prototype.

5.2 The Personal Banking Assistant Agents

Monitoring account balances is the primary duty of the personal banking assistant (PBA) agent. The user specifies "monitors" by giving the account and institution, the limit at which the balance going below or above triggers action, and the action to be taken. The PBA refreshes the balances of monitored accounts by sending messages to the bank agents where the accounts are held.

The frequency of account balance queries depends on the user-specified volatility of the account. The PBA agent checks all monitors on an account when an institution replies with the current balance for the account.

Two types of actions in response to a monitor trigger are currently supported: alerting its owner of the trigger, and actually resolving the trigger via account transfers. When the PBA is to resolve the trigger by itself, its action depends on whether the trigger involved an account balance going above a maximum limit or below a minimum limit. In the latter case, it arranges to bring all account balances up-to-date, and then chooses an account to transfer funds from, considering: rate of return, owner instructions on moving funds out of that account, minimum allowed balance in that account (user or bank restricted), and current account balance.

In response to an account being above its maximum limit, the PBA first gathers information about accounts at various financial institutions. This is accomplished by asking the router agent to broadcast a message to all financial institutions containing its owner's investment profile (risk level, liquidity needs, and growth-versus-income needs) and requesting an account recommendation. The PBA then waits for account recommendations (the length of the wait depends on the overall urgency of monitors and can be controlled by the user). When the waiting period has elapsed, the agent examines all of the relevant account recommendations and chooses the best. If the user already has an account of this type at the given institution, the PBA orders a transfer into it; if not it opens an account of this type for the user and then orders the transfer.

Some of the fluents that PBA agents use to model the world are:

- $\textsc{UserAccount}(type, bank, account, balance, lastUpdate, rateOfReturn, moveFunds, minBalance, penalty, refreshRate, s)$: the agent's owner has an *account* of *type* at *bank* in situation s,
- $\textsc{Monitor}(type, bank, account, limit, lowerOrHigher, priority, response, monID, s)$: the agent is monitoring *account* of *type* at *bank* in situation s,
- $\textsc{WaitingUpdt}(bank, account, s)$: the agent is expecting an update on *account* at *bank* in situation s,
- $\textsc{Alert}(alertMessage, maxPriority, monID, s)$: the agent must alert its owner that monitor $monID$ has been triggered,
- $\textsc{AlertSent}(medium, priority, timeStamp, monID, s)$: an alert has been sent via *medium* at *timeStamp* in situation s,
- $\textsc{AlertAcknowledged}(monID, s)$: the alert on monitor $monID$ has been acknowledged by the agent's owner in situation s,
- $\textsc{message}(s)$: the last message read by the agent in situation s.

Note that many of these fluents, for example WaitingUpdt, represent properties of the agent's mental state. This is because the agent must be able to react quickly to incoming messages from its owner or other agents. Golog does not support interrupts or any other mechanism that would allow the current process to be suspended when an event that requires immediate attention occurs (although the more recent concurrent version of the language, ConGolog, does [5]). So to get reactive behavior from the agent, one must explicitly program it to monitor its communications ports, take quick action when a message arrives, and return to monitoring its ports. When a new message arrives, whatever associated actions can be performed immediately are done. The agent avoids waiting in the middle of a complex action whenever possible; instead, its mental state is altered using fluents such as WaitingUpdt. This ensures that subsequent events are interpreted correctly, timely reaction to conditions detected occurs (e.g., sending out a new alert when the old one becomes stale), and the agent is still able to respond to new messages.

Among the primitive actions that PBA agents are capable of, we have:

- sendMessage($method, recipient, message$): send *message* to *recipient* via *method*,
- startWaitingUpdt($bank, account$): note that the agent is expecting an update on *account* from *bank*,
- stopWaitingUpdt($bank, account$): note that the agent is no longer expecting an update on *account* from *bank*,
- createAlert($message, maxPriority, monID$): note that the user must be alerted with *message* and keep trying until *maxPriority* is reached,
- sendAlert($priority, message, medium, monID$): send an alert to the user with *priority* via *medium* containing *message*,
- deleteAlert($monID$): stop attempting to alert the user that monitor *monID* was triggered,
- readCommunicationsPort($channel$): check the communications *channel* (TCP/IP, e-mail, etc.) for messages and set the fluent message appropriately,
- updateBalance($bank, account, balance, return, time$): update the balance on the user's *account* at *bank*.

PBA agents are programmed in Golog as follows. First, we provide the required axioms: a precondition axiom for each primitive action, for instance, axiom (1) from section 2 for the action createAlert; a successor state axiom for each fluent, for instance, axiom (2) for Alert; and axioms specifying the initial state of the system — what accounts the user has, what conditions to monitor, etc.[6]

Then, we specify the behavior of the agent in the complex actions language. The main procedure the agent executes is controlPBA:

[6] In this case, the axioms were translated into Prolog clauses by hand. We have been developing a preprocessor that does this automatically. It takes specifications in a high level notation similar to Gelfond and Lifschitz's $\mathcal{A}$ [2]. Successor state axioms are automatically generated from effects axioms.

```
proc controlPBA
  while True do
    refreshMonitoredAccts;
    handleCommunications(TCP/IP);
    generateAlerts
  endWhile
endProc
```

Thus, the agent repeatedly does the following: first request balance updates for all monitored accounts whose balance has gotten stale, second process all new messages, and third send out new messages alerting the user of monitor triggers whenever a previous alert has not been acknowledged after an appropriate period. The procedure for requesting balance updates for monitored accounts appears below:

```
proc refreshMonitoredAccts
  for type, where, account, lastUpdate, refreshRate :
      UserAccount(type, where, account, _, lastUpdate, _, _, _, _, refreshRate)
      ∧ Monitor(type, where, account, _, _, _, _, _) ∧
      Stale(lastUpdate, refreshRate) ∧ ¬ WaitingUpdt(where, account)[
    /* ask for balance */
    composeRequestForInformation(type, request)?;
    sendMessage(TCP/IP, where, request);
    startWaitingUpdt(where, account)
]endProc
```

Note how the fluent WaitingUpdt is set when the agent requests an account update; this ensures that the request will only be made once.

The following procedure handles the issuance of alert messages:

```
proc generateAlerts
  for msg, maxPriority, monID: alert(msg, maxPriority, monID)[
    if ∃medium, lastPriority, time (
        alertSent(medium, lastPriority, time, monID) ∧
    ¬ alertAcknowledged(monID) ∧ stale(lastPriority, medium, time)
        ∧ lastPriority < maxPriority) then
      π p, newMedium, lastPriority [
         (alertSent(_, lastPriority, _, monID) ∧ p is lastPriority + 1 ∧
          appropriateMedium(p, newMedium) ∧ worthwhile(newMedium))?;
        sendAlert(p, msg, newMedium, monID)]
    endIf
]endProc
```

It directs the agent to behave as follows: for every monitor trigger of which the user needs to be alerted, if the latest alert message sent has not been acknowledged and is now stale, and the maximum alert priority has not yet been reached, then send a new alert message at the next higher priority via an appropriate medium.

```
proc handleCommunications(channel)[
  readCommunicationsPort(channel);
  while ¬ Empty(message) do [
    if type(message) = StartMonitor ∧
          sender(message) = PersonalInterface then
      π type, bank, account, limit, lh, prio, resp, monID [
        args(message) = [type, bank, account, limit, lh, prio, resp, monID]?;
        startMonitor(type, bank, account, limit, lh, prio, resp, monID)]
    else if type(message) = AcknowledgeAlert ∧
          sender(message) = PersonalInterface then
      π monID [args(message) = [monID]?; acknowledgeAlert(monID)]
    else if type(message) = UpdateConfirmation ∧
          IsBank(sender(message)) then
      π from, account, amount, rate, time [
        args(message) = [from, account, amount, rate, time]?;
        handleUpdateConfirmation(from, account, amount, rate, time)]
    else if type(message) = UpdateReject ∧
          IsBank(sender(message)) then
      log(UpdateReject,sender(Message),args(Message))
    /* other cases handling message types StopMonitor, */
    /* TransferConfirmation, AccountOpened, and */
    /* RecommendedAccount are omitted */
    endIf
    readCommunicationsPort(channel)
  ]endWhile
]endProc
```

Fig. 3. Main message handling procedure.

The procedure in figure 3 contains the main message handling loop. It repeatedly reads a message from the port and dispatches to the appropriate action depending on the type of message and sender, for as long as the message queue is not empty. For instance, when a StartMonitor message is received from the personal interface, the agent starts monitoring the condition of interest; and when an AcknowledgeAlert message is received, the agent notes the acknowledgement. Some cases are left out for brevity.

The most interesting case involves UpdateComfirmation messages from banks. Here, the agent must note the account's updated balance, check whether this triggers a monitor, and take action if it does. handleCommunications sets this in motion by dispatching to the procedure handleUpdateConfirmation which appears in figure 4. When a monitor trigger is detected in the above procedure, the reaction of the agent depends on whether the account balance has gone lower or higher than the limit and whether it is directed to alert the user or solve the problem itself. The solveLowBalance procedure in figure 5 is invoked when the PBA must solve an account balance going below its limit. The

```
proc handleUpdateConfirmation(bank, account, amount, rate, time)[
  updateBalance(bank, account, amount, rate, time);
  stopWaitingUpdt(bank, account);
  /* check monitors on the account */
  for type, limit, lh, prio, resp, monID:
     Monitor(_, bank, account, limit, lh, prio, resp, monID)[
    if (lh = Lower ∧ amount < limit ∧ ¬alert(_, _, monID)) then
      if resp = solve then
        solveLowBalance(bank, account, amount, limit, lh, prio, monID)
      else
        alertLowBalance(bank, account, amount, limit, lh, prio, monID)
      endIf
    else if (lh = Higher ∧ amount > limit ∧ ¬alert(_, _, monID) then
      if resp = Solve then
        solveHighBalance(bank, account, amount, limit, lh, prio, monID)
      else
        alertHighBalance(bank, account, amount, limit, lh, prio, monID)
      endIf
    else if alert(_, _, monID) ∧ ((lh = Lower ∧ amount ≥ limit) ∨
          (lh = Higher ∧ amount ≤ limit)) then
      handleMonitorUntrip(resp, monID) endIf
]]endProc
```

Fig. 4. Procedure treating update confirmation messages.

agent picks the account with the highest score based on the account's minimum balance and associated penalties, its rate of return, the user-specified mobility of funds in the account, whether the account's balance is sufficient, and whether a monitor exists on the account and what its limit is. If the score of the chosen account is above a certain limit that depends on the monitor's priority, a transfer request is issued, otherwise the user is alerted. It would be interesting to try to generalize this to handle cases where one must transfer funds from several accounts to solve the problem.

6 Discussion

One conclusion that can be drawn from this experiment in developing the personal banking assistance system (2000 lines of Golog code), is that it is possible to build sizable applications in the language. How well did Golog work for this? Golog's mechanism for building and maintaining a world model proved to be easy to use and effective. It provides much more structure to the programmer for doing this — precondition and successor state axioms, etc. — than ordinary programming or scripting languages, and guarantees soundness and expressiveness.

proc $\text{SOLVELOWBALANCE}(bank, account, amount, limit, prio, monID)$
  π $bankFrom, accountFrom, score, amtReq$ [
    $\text{CHOOSEBESTACCOUNT}(bankFrom, accountFrom, score, amtReq, bank,$
        $account, monID)?;$
    if $score > \text{TRANSFERLIMIT}(prio)$ **then**
      $\text{TRANSFERFUNDS}(bankFrom, accountFrom, bank, account, amtReq)$
    else π msg [
      $\text{COMPOSEMSG}(\text{FAILEDTOSOLVE}, bank, account, amtReq, monID, msg)?;$
      $\text{CREATEALERT}(msg, prio, monID)$]
    endIf
]**endProc**

Fig. 5. Procedure to resolve a low balance condition.

The current implementation of the banking assistance system did not make much use of Golog's lookahead and nondeterminism features. But this may be because the problem solving strategies used by its agents are relatively simple. For instance, if we wanted to deal with cases where several transactions are necessary to bring an account balance above its limit, the agent would need to search for an appropriate sequence of transactions. We are experimenting with Golog's nondeterminism as a mechanism for doing this kind of "planning".

We also used Golog's situation calculus semantics to perform correctness proofs for some of the PBA agent's procedures, for instance REFRESHMONITOREDACCTS; the proofs appear in [15]. This was relatively easy, since there was no need to move to a different formal framework and Golog programs already include much of the specifications that are required for formal analysis.

From a software engineering point of view, we found Golog helpful in that it encourages a layered design where Golog is used to program the knowledge-based aspects of the solution and C or Tcl/Tk procedures attached to the Golog primitive actions handle low-level details.

On the negative side, we found a lot of areas where Golog still needs work. Some things that programmers are accustomed to be able to do easily are tricky in Golog: performing arithmetic or list processing operations, assigning a value to a variable without making it a fluent, etc. The language does not provide as much support as it should for distinctions such as fluent vs. non-fluent, fluents whose value is updated using the successor state axioms vs. sensed fluents, the effects of actions on fluents vs. their effects on unmodeled components of the state, etc. The current debugging facilities are also very limited. Some standard libraries for things like agent communication and reasoning about time would also be very useful.

Perhaps the most serious limitation of Golog for agent programming applications is that it does not provide a natural way of specifying event-driven, reactive behavior. Fortunately, an extended version of the language called ConGolog which supports concurrent processes, priorities, and interrupts is under development. This extended language allows event-driven behavior to be specified very naturally using interrupts. In [5], we describe how ConGolog could be used to develop a simple meeting scheduling application.

Finally, there are some significant discrepancies between the Golog implementation and our theories of agency in the way knowledge, sensing, exogenous events, and the relation between planning and execution are treated. We would like to develop an account that bridges this gap. The account of planning in the presence of sensing developed in [6] is a step towards this.

References

1. Michael Dixon. *Embedded Computation and the Semantics of Programs.* PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, 1991. Also appeared as Xerox PARC Technical Report SSL-91-1.
2. M. Gelfond and Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17(301–327), 1993.
3. Andrew R. Haas. The case for domain-specific frame axioms. In F.M. Brown, editor, *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*, pages 343–348, Lawrence, KA, April 1987. Morgan Kaufmann Publishing.
4. Yves Lespérance, Hector J. Levesque, F. Lin, Daniel Marcu, Raymond Reiter, and Richard B. Scherl. A logical approach to high-level robot programming – a progress report. In Benjamin Kuipers, editor, *Control of the Physical World by Intelligent Agents, Papers from the 1994 AAAI Fall Symposium*, pages 109–119, New Orleans, LA, November 1994.
5. Yves Lespérance, Hector J. Levesque, F. Lin, Daniel Marcu, Raymond Reiter, and Richard B. Scherl. Foundations of a logical approach to agent programming. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents Volume II — Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95)*, Lecture Notes in Artificial Intelligence, pages 331–346. Springer-Verlag, 1996.
6. Hector J. Levesque. What is planning in the presence of sensing? In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1139–1146, Portland, OR, August 1996.
7. Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. To appear in the *Journal of Logic Programming,* special issue on Reasoning about Action and Change, 1996.
8. Fangzhen Lin and Raymond Reiter. How to progress a database (and why) I. logical foundations. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference*, pages 425–436, Bonn, Germany, 1994. Morgan Kaufmann Publishing.
9. Fangzhen Lin and Raymond Reiter. State constraints revisited. *Journal of Logic and Computation*, 4(5):655–678, 1994.
10. Fangzhen Lin and Raymond Reiter. How to progress a database II: The STRIPS connection. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 2001–2007, Montréal, August 1995. Morgan Kaufmann Publishing.

11. John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, UK, 1979.
12. E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In R.J. Brachman, H.J. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, Toronto, ON, May 1989. Morgan Kaufmann Publishing.
13. V.R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proc. of the 17th IEEE Symp. on Foundations of Computer Science*, pages 109–121, 1976.
14. Raymond Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.
15. Shane J. Ruman. GOLOG as an agent-programming language: Experiments in developing banking applications. Master's thesis, Department of Computer Science, University of Toronto, Toronto, ON, 1996.
16. L.K. Schubert. Monotonic solution to the frame problem in the situation calculus: An efficient method for worlds with fully specified actions. In H.E. Kyberg, R.P. Loui, and G.N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer Academic Press, Boston, MA, 1990.

This article was processed using the LaTeX macro package with LLNCS style