Towards High-Level Programming for Distributed Problem Solving[*]

Ryan F. Kelly and Adrian R. Pearce
NICTA Victoria Research Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne
Victoria, 3010, Australia
{rfk,adrian}@csse.unimelb.edu.au

Abstract

We present a new approach to distributed problem solving based on high-level program execution. While this technique has proven itself for single-agent systems based on the Golog language, several challenges are encountered when moving to a multi-agent setting. Key to our approach is a better representation of the dynamics of multi-agent teams by means of the following features: a robust combination of true concurrency of actions with the interleaved concurrency of ConGolog; an explicit notion of time to assist coordination; and semantic support for predictable exogenous actions (also called "natural actions"). The result is MIndiGolog, a new Golog variant suitable for distributed, cooperative execution by a multi-agent team.

1. Introduction

In distributed problem solving applications, a team of agents must cooperate closely to achieve a shared goal. Such teams can often be conceptualized as a single agent with distributed sensing, reasoning and acting capabilities, which leads to a two-stage technique for programming such teams: adapt a formalism from single-agent programming to specify the tasks to be performed, then provide a coordination strategy allowing the team to execute the specification cooperatively. This approach has been successfully employed by platforms such as STEAM [17], SharedPlans [8], and TAEMS [5], which specify tasks with variants of the Hierarchical Task Networks (HTN) formalism.

While HTN is a popular and powerful approach to task specification, an increasingly popular alternative is *high-level program execution* as embodied by the Golog programming language [11]. By "high-level program" is meant a program whose primitive components are domain-specific actions, connected by standard programming constructs, and that may contain nondeterministic operators. The primary advantage of this approach is *controlled nondeterminism*, allowing some program parts to be fully specified while others may involve arbitrary amounts of nondeterminism, or even goal-based planning. Compared to standard HTN, Golog provides a more natural representation of many tasks and is based on a more sophisticated logic of action [2]. Extensions to Golog have introduced further advantageous features such as concurrent execution by interleaving of actions (ConGolog [3]) and a combination of online execution with offline planning (IndiGolog [4]).

Motivated by these advantages, our research program aims to build a distributed problem solving system based on cooperative execution of shared high-level programs. This paper represents a first step towards that goal. We integrate several existing extensions to the situation calculus into the Golog language to better represent the dynamics of a multi-agent team. Key among these is true concurrency of actions, which we combine with the interleaved concurrency of ConGolog to give a flexible account of concurrent execution. An explicit notion of time is incorporated to enrich the world model and to assist in coordination between agents. The concept of natural actions is also tightly integrated into the language, to allow agents to predict the behavior of their teammates and environment. We name the resulting language "MIndiGolog" for "Multi-Agent IndiGolog".

The paper proceeds as follows: section 2 gives some background on the situation calculus and Golog; section 3 develops the semantics of MIndiGolog and shows how distributed logic programming techniques can facilitate the shared execution of MIndiGolog programs; section 4 discusses related work and section 5 concludes with a summary of our results and ongoing research. The work is motivated and illustrated throughout by examples from a simple multi-agent domain, in which a team of robotic chefs must be programmed to cooperatively prepare a meal.

[*]This research was supported by a grant from the Australian Research Councils (ARCs) Centre for Perceptive and Intelligent Machines in Complex Environments (PIMCE).

2. Background

2.1. The Situation Calculus

The situation calculus is a first-order logic formalism for modeling dynamic worlds, with the following key features: *Actions* are functions denoting individual instantaneous events that can cause the state of the world to change; *Situations* are histories of the actions that have occurred, with S_0 being the initial situation and successive situations built up using the function $do(a, s)$; *Fluents* are predicates or functions representing properties of the world that may change from one situation to another. The special fluent $Poss(a, s)$ indicates when it is possible to perform an action in a situation. For a detailed description consult [14].

A collection of situation calculus statements $\mathcal{D}$ describing a dynamic world is referred to as a *theory of action*, and queries about the behavior or evolution of the world are posed as logical entailment queries relative to this theory.

Concurrency. In the basic situation calculus only a single action can occur at any instant. While suitable for most single-agent domains, this limitation is emphatically not suitable for multi-agent systems - several actions can easily occur simultaneously if performed by different agents. Modeling this *true concurrency* is necessary to avoid problems with conflicting or incompatible actions. There is also the potential to utilize concurrency to execute tasks more efficiently. Clearly a solid account of concurrency is required for programming multi-agent teams.

The work of [12, 16] adds true concurrency to the situation calculus by replacing action terms with sets of actions that are performed simultaneously. All functions and predicates that take an action are modified to take sets of actions instead. For example, $do(a, s)$ becomes $do(\{a_1, a_2, ...\}, s)$.

This seemingly simple modification introduces a complication - a combination of actions is not guaranteed to be possible even if each of the individual actions are. For example, two agents may not be able to acquire the same resource at the same time. This is known as the precondition interaction problem [13] and is an area of ongoing research. For our purposes it is addressed by introducing a predicate $Conflicts(c, s)$ which is true when the actions in c are in conflict and cannot be performed together.

Time. An explicit notion of time can make coordination between agents easier, as joint actions may be performed at a particular time. It also allows a richer description of the world, for example when stating the baking time of a cake.

The standard approach to time in the situation calculus is that of [16, 13]. Each action gains an extra argument indicating the time at which is was performed. Time itself can be represented by any appropriately-behaved sequence, such as integers or reals, whose axiomatisation must be included in the theory of action. The functions $time$ and $start$ are introduced to give the performance time of an action and the start time of a situation respectively. The start time of the initial situation may be defined arbitrarily.

An additional predicate $Coherent$ is defined to ensure that the performance time is the same for all members in a set of concurrent actions. The $Poss$ fluent for concurrent actions can then be defined to ensure that the temporal relation between past and future situations is respected, as well as accounting for conflicting actions[1]:

$$Poss(c, s) \equiv \forall a. [a \in c \rightarrow Poss(a, s)]$$
$$\wedge \neg Conflicts(c, s) \wedge time(c) > start(s) \wedge Coherent(c)$$

This representation is accompanied by a standard approach to actions with a finite duration: they are decomposed into instantaneous $start$ and end actions and a fluent indicating that the action is in progress. For example, a long-running task may be represented by the actions $beginTask$ and $endTask$ along with a fluent $doingTask$.

Natural actions. These are a special class of exogenous actions, those actions which occur outside of an agent's control [16]. They are classified according to the following requirement: natural actions must occur at their predicted times, provided no earlier actions prevent them from occurring. For example, a timer will ring at the time it was set for, unless it is switched off. The action $endTask$ from above is another example - it must occur whenever it is possible, which is at the time when the agent finishes the task. In domains where many agents may be simultaneously engaged in many long-running tasks, strong semantic support for natural actions will therefore be of significant benefit.

Natural actions are indicated by the truth of the predicate $Natural(a)$. The times at which natural actions may occur are specified by the $Poss$ predicate as usual. For example, suppose that the fluent $TimerSet(m, s)$ represents the fact that a timer is set to ring in m minutes in situation s. The possibility predicate for the $ringTimer(t)$ action would be:

$$Poss(ringTimer(t), s) \equiv$$
$$\exists m. [TimerSet(m, s) \wedge t = start(s) + m]$$

The timer may thus ring only at its predicted time. To enforce the requirement that natural actions *must* occur whenever possible, a predicate $Legal(s)$ is introduced which is true only for situations that respect this requirement. Legal situations are the only situations that could be brought about in the real world:

$$Legal(S_0) \equiv True$$
$$Legal(do(c, s)) \equiv Legal(s) \wedge Poss(c, s)$$
$$\wedge \forall a. [Natural(a) \wedge Poss(a, s) \rightarrow [a \in c \vee t < time(a)]]$$

[1] As usual, lower-case terms are variables and free variables are implicitly universally quantified

Table 1. Some Golog operators

Operator	Meaning
a	Execute action a in the world
$\phi?$	Proceed if condition ϕ is true
$\delta_1;\delta_2$	Execute $\delta_1$followed by δ_2
$\delta_1\|\delta_2$	Execute either δ_1 or δ_2
$\pi(x)\delta(x)$	Nondet. select arguments for δ
$\delta*$	Execute δ zero or more times
$\mathbf{if}\,\phi\,\mathbf{then}\,\delta_1\,\mathbf{else}\,\delta_2$	Exec. δ_1 if ϕ holds, δ_2 otherwise
$\mathbf{while}\,\phi\,\mathbf{do}\,\delta$	Execute δ while ϕ holds
$\mathbf{proc}P(\overrightarrow{x})\delta(\overrightarrow{x})\mathbf{end}$	Procedure definition
$\delta_1\|\|\delta_2$	Concurrent execution (ConGolog)
$\Sigma\delta$	Plan execution offline (IndiGolog)

An important concept when dealing with natural actions is the least natural time point (LNTP) of a situation, defined as the earliest time at which a natural action may occur. We assume that the theory of action avoids certain pathological cases, so that absence of an LNTP implies that no natural actions are possible.

$$\begin{aligned}&Lntp(s,t) \equiv\\ &\quad \exists a.\,[Natural(a) \wedge Poss(a,s) \wedge time(a) = t] \wedge\\ &\qquad \forall a.\,[Natural(a) \wedge Poss(a,s) \rightarrow t \leq time(a)]\end{aligned}$$

2.2. Golog

Golog [11] is a declarative agent programming language based on the situation calculus. Testimony to its success are its wide range of applications and many extensions to provide additional functionality ([3, 4, 6]). We use "Golog" to refer to the family of languages based on this technique, including ConGolog [3] and IndiGolog [4].

To program an agent using Golog one specifies a situation calculus theory of action, and a program consisting of actions from the theory connected by programming constructs such as if-then-else, while loops, and nondeterministic choice. Table 1 lists some of the operators available in various incarnations of the language.

In line with the idea of high-level program execution, the agent's control program may be nondeterministic. It is the task of the agent to plan a deterministic instantiation of the program, a sequence of actions that can legally be performed in the world. Such a sequence is called a *legal execution* of the Golog program.

Two predicates $Trans$ and $Final$ define the semantics for each operator. $Trans(\delta, s, \delta', s')$ holds when executing a step of program δ can cause the world to move from situation s to situation s', after which δ' remains to be executed. It thus characterizes single steps of computation. $Final(\delta, s)$ holds when program δ may legally terminate its execution in situation s. We base our work on the semantics of IndiGolog, which builds on ConGolog and is the most feature-full of the standard Golog variants. The full semantics are available in the references, but as an example consider equation (1), which specifies the concurrent-execution operator as an *interleaving* of computation steps. It states that it is possible to single-step the concurrent execution of δ_1 and δ_2 by performing either a step from δ_1 or a step from δ_2, with the remainder γ left to execute concurrently with the other program:

$$\begin{aligned}&Trans(\delta_1||\delta_2, s, \delta', s') \equiv\\ &\qquad \exists\gamma.Trans(\delta_1, s, \gamma, s') \wedge \delta' = (\gamma||\delta_2)\\ &\qquad \vee\ \exists\gamma.Trans(\delta_2, s, \gamma, s') \wedge \delta' = (\delta_1||\gamma) \qquad (1)\end{aligned}$$

Clearly there are two notions of concurrency to be considered: the possibility of performing several actions at the same instant (*true concurrency*), and the possibility of interleaving the execution of several programs (*interleaved concurrency*). These were combined in [1] by modifying Golog to incorporate sets of concurrent actions. However, they give a semantics which may call for actions to be performed that are not possible and which can result in unintuitive program behavior. A key aspect of our work is a more robust integration of these two notions of concurrency.

If the theory of action $\mathcal{D}$ is enriched with $Trans$ and $Final$, planning an execution of a Golog program δ is basically a theorem proving task as shown in equation (2). Here $Trans*$ indicates reflexive transitive closure. The situation s gives a sequence of actions forming a legal execution of the program.

$$\mathcal{D} \models \exists s.\,[Trans*(\delta, S_0, \delta', s) \wedge Final(\delta', s)] \qquad (2)$$

In IndiGolog agents can also proceed without planning a full terminating execution of their program, by searching for a legal "next step" action a such that $\mathcal{D} \models \exists a\,.\,Trans*(\delta, s, \delta', do(a, s))$. The search operator ($\Sigma$) controls which parts of the program are subject to full execution planning, providing fine-grained control over nondeterminism and the amount of planning work required.

As an example of a multi-agent task specification in Golog, consider a program $MakeSalad$ that instructs a team of agents to prepare a simple salad:

$$\begin{aligned}&\mathbf{proc}\; MakeSalad(dest)\\ &\quad [\pi(agt)(ChopTypeInto(agt, Lettuce, dest))\,||\\ &\quad \pi(agt)(ChopTypeInto(agt, Carrot, dest))\,||\\ &\quad \pi(agt)(ChopTypeInto(agt, Tomato, dest))]\ ;\\ &\qquad\quad \pi(agt)\,[acquire(agt, dest)\,;\\ &\qquad\quad beginTask(agt, mix(dest, 1))\,;\\ &\qquad\qquad\qquad release(agt, dest)]\ \mathbf{end} \qquad (3)\end{aligned}$$

The sub-procedure $ChopTypeInto$ (not shown) picks an object of the given type and an available chopping board, chops the object using the board, then transfers it into the destination container. $MakeSalad$ tells the agents to do this for a lettuce, a carrot and a tomato, then mix the ingredients together for 1 minute. Note the nondeterminism in this program - the agent assigned to handling each ingredient is not specified (π construct), nor is the order in which they should be added ($||$ construct). There is thus considerable scope for cooperation between agents to effectively carry out this task.

While this is a valid program in standard IndiGolog, executing it using the existing semantics would be far from ideal. The explicit temporal component described above must to added to IndiGolog to accommodate the $mix(dest, 1)$ task. The lack of true concurrency would mean only one agent could act at a time, while others would remain idle. And since there is no support for natural actions, IndiGolog would fail to find a legal execution of this program: it would find that the final action $release$ cannot be performed after doing $beginTask$, as our theory of action ensures agents can only be doing one thing at a time. But it would not determine that the natural action $endTask$ will occur after one minute and enable to program to finish.

Our new Golog variant, MIndiGolog, is designed to produce executions of such programs in a manner that overcome these limitations, and is thus suitable for specifying tasks to be performed by multi-agent teams in distributed problem solving applications.

3. MIndiGolog

We have integrated three extensions to the situation calculus with the semantics of IndiGolog to better model the dynamics of a multi-agent setting. These extensions allow agents to represent time, concurrently-occurring actions, and natural actions in a robust way.

3.1. Time

It is clear from the background section that the approach of [16] to modeling time is complicated by the presence of concurrent actions. To avoid the need for the $Coherent$ predicate, we attach the temporal argument to each situation rather than to each action. The successor situation function $do(a, s)$ becomes $do(a, t, s)$, to indicate "action a was performed at time t in situation s". The possibility predicate $Poss(a, s)$ likewise becomes $Poss(a, t, s)$. The semantics of IndiGolog trivially accommodate this change, and $Coherent$ and $time$ are no longer needed.

3.2. Concurrency

While it is straightforward to modify the IndiGolog $Trans$ rule for primitive actions to accept sets of concurrent actions, there are deeper implications for the concurrency operator. This is implemented by accepting a transition from either of the two programs as a transition for the pair [3]. In the presence of true concurrency, this is insufficient. Suppose program δ_1 may be transitioned by performing actions c_1, and δ_2 may be transitioned by performing actions c_2. As noted in [1], it should be possible to exploit true concurrency by performing both simultaneously, i.e. $c_1 \cup c_2$. However, this introduces several complications that [1] does not address.

First, precondition interaction means that $c_1 \cup c_2$ may not be possible even if the individual actions are. The transition clause must ensure that the combination of the two sets of actions is possible. Another issue arises when two programs can legitimately be transitioned by executing the same action. Consider the following programs which add ingredients to a bowl:

$$\begin{aligned} \delta_1 &= placeIn(Thomas, Flour, Bowl)\,;\\ &\qquad placeIn(Thomas, Sugar, Bowl)\\ \delta_2 &= placeIn(Thomas, Flour, Bowl)\,;\\ &\qquad placeIn(Thomas, Egg, Bowl) \end{aligned}$$

Executing $\delta_1 || \delta_2$ should result in the bowl containing two units of flour, one unit of sugar and an egg. However, an individual transition for both programs is $c_1 = c_2 = placeIn(Thomas, Flour, Bowl)$. Naively executing $c_1 \cup c_2$ to transition both programs would add only one unit of flour. Alternately, consider two programs waiting for a timer to ring:

$$\begin{aligned} \delta_1 &= ringTimer\,;\, acquire(Thomas, Bowl)\\ \delta_2 &= ringTimer\,;\, acquire(Richard, Board) \end{aligned}$$

Both programs should be allowed to proceed with a single occurrence of the $ringTimer$ action, because it is an aspect of the environment. To avoid unintuitive (and potentially dangerous) behavior, concurrent execution must not be allowed to transition both programs using the same *agent-initiated* action. If an agent-initiated action may be safely skipped, it can be enclosed in an appropriate if-then-else or choice construct.

Taking these factors into account, we develop the improved transition rule for concurrency in equation (4). The first two lines are the original interleaved concurrency clause from ConGolog, while the remainder characterizes the above considerations for true concurrency. This robust combination allows the language to more accurately reflect the concurrency present in multi-agent teams. As with ConGolog and IndiGolog, our semantics make no attempt to maximize concurrency or otherwise differentiate between potential executions, they only state which transitions can be legally performed.

$$\begin{aligned}
&Trans(\delta_1||\delta_2, s, \delta', s') \equiv \\
&\qquad \exists\gamma\,.\,Trans(\delta_1, s, \gamma, s') \wedge \delta' = (\gamma||\delta_2) \\
&\qquad \vee \exists\gamma\,.\,Trans(\delta_2, s, \gamma, s') \wedge \delta' = (\delta_1||\gamma) \\
&\qquad \vee \exists c_1, c_2, \gamma_1, \gamma_2, t\,.\,Trans(\delta_1, s, \gamma_1, do(c_1, t, s)) \\
&\qquad \wedge Trans(\delta_2, s, \gamma_2, do(c_2, t, s)) \wedge Poss(c_1 \cup c_2, t, s) \\
&\qquad \wedge \forall a.\,[a \in c_1 \wedge a \in c_2 \rightarrow Natural(a)] \\
&\qquad \wedge \delta' = (\gamma_1||\gamma_2) \wedge s' = do(c_1 \cup c_2, t, s) \qquad (4)
\end{aligned}$$

3.3. Natural actions

The formalism of [16] is adopted, with simple modifications for our handling of time. While planning with natural actions has previously been done in Golog [15], the programmer was required to explicitly check for any possible natural actions and ensure that they appear in the execution. We significantly lower the burden on the programmer by guaranteeing that all legal program executions result in legal situations. MIndiGolog agents will plan for the occurrence of natural actions without having them explicitly mentioned in the program. They may optionally be included in the program, instructing the agents to wait for the action to occur before proceeding.

This is achieved using a new $Trans$ clause for the case of a single action c, as shown in equation (5). If s has an LNTP t_n and corresponding set of natural actions c_n, a transition can be made in three ways: perform c at a time before t_n (fourth line), perform it along with the natural actions at t_n (fifth line), or wait for the natural actions to occur (sixth line). If there is no LNTP, then c may be performed at any time greater than $start(s)$.

$$\begin{aligned}
&Trans(c, s, \delta', s') \equiv \\
&\qquad \exists t, t_n, c_n\,.\,Lntp(s, t_n) \wedge t \geq start(s) \wedge \\
&\qquad \forall a.\,[Natural(a) \wedge Poss(a, t_n, s) \equiv a \in c_n] \wedge \\
&\qquad [t < t_n \wedge Poss(c, t, s) \wedge s' = do(c, t, s) \wedge \delta' = Nil \\
&\qquad \vee Poss(c \cup c_n, t_n, s) \wedge s' = do(c \cup c_n, t, s) \wedge \delta' = Nil \\
&\qquad \vee\ s' = do(c_n, t_n, s) \wedge \delta' = c] \\
&\qquad \vee \neg\exists t_n\,.\,Lntp(s, t_n) \wedge \exists t\,.\,Poss(c, t, s) \wedge \\
&\qquad t \geq start(s) \wedge s' = do(c, t, s) \wedge \delta' = Nil \qquad (5)
\end{aligned}$$

The occurrence of natural actions may also cause test conditions within the program to become satisfied, so a new $Trans$ clause for $\phi?$ is also required as shown in equation (6). This permits a program consisting of a single test condition to make a transition if the condition is satisfied, or if a natural action occurs: [2]

$$\begin{aligned}
&Trans(\phi?, s, \delta', s') \ \equiv\ \phi[s] \wedge \delta' = Nil \wedge s' = s \\
&\vee\ \exists t_n, c_n\,.\,Lntp(s, t_n) \wedge \delta' = \phi? \wedge s' = do(c_n, t_n s) \\
&\qquad \wedge\ \forall a.\,[Natural(a) \wedge Poss(a, t_n, s) \rightarrow a \in c_n] \qquad (6)
\end{aligned}$$

A MIndiGolog execution will thus contain all natural actions that will occur, regardless of whether they were considered explicitly by the programmer.

Contrast this with the standard handling of exogenous events in IndiGolog, which is achieved by executing the main program concurrently with a program that generates exogenous actions:

$$\delta_{main}\ ||\ (\pi(a)(Exog(a)?\,;\,a))^*$$

This allows the program to make a legal transition regardless of what exogenous actions occur - an approach suitable for dealing with arbitrary exogenous actions which may occur at any time, but suboptimal for handling *predictable* exogenous actions. Our approach allows the agents to directly predict the natural actions that will occur and automatically include them in a planned execution.

3.4. Legality of the semantics

Let a *MIndiGolog Theory of Action* be a theory of action in the situation calculus enhanced with time, true concurrency and natural actions, augmented with the predicates $Trans$ and $Final$ from IndiGolog, modified according to equations (4), (5) and (6). All legal executions of a MIndiGolog program derived from such a theory of action produce legal situations.

Lemma 1. *Let $\mathcal{D}$ be a MIndiGolog theory of action. Then:*

$$\mathcal{D} \models \forall s, s', \delta, \delta'.Legal(s) \wedge Trans(\delta, s, \delta', s') \rightarrow Legal(s')$$

Proof. By induction on the structure of δ. That the theorem holds for the modified $Trans$ clauses of equations (4), (5) and (6) is straightforward, and no other clause constructs new situation terms. □

Theorem 1. *Let $\mathcal{D}$ be a MIndiGolog theory of action. Then:*

$$\mathcal{D} \models \forall s', \delta, \delta'\,.\,Trans*(\delta, S_0, \delta', s') \rightarrow Legal(s')$$

Thus, all legal executions of a MIndiGolog program produce legal situations.

Proof. From lemma 1, the legality of S_0, and the properties of transitive closure. □

[2] $\phi[s]$ should be read as "ϕ holds in situation s"

Like standard IndiGolog, MIndiGolog will only call for actions to be performed if it is actually possible to perform them. MIndiGolog has the added advantage of effortlessly incorporating any natural actions that may occur.

3.5. Example execution

The effect of our new semantics can be seen in figure 1(a), which shows one possible legal execution of the $MakeSalad$ program using the new semantics of MIndiGolog[3] in a domain with thee agents. Note the occurrence of several actions within each **do** statement, demonstrating the integration of true concurrency into the language. Note also the incorporation of natural $endTask$ actions into the program, and the explicit occurrence time for each action. (in this trace the occurrence times have been instantiated to their earliest possible value).

For comparison purposes, we added a temporal element and inserted explicit $endTask$ actions into $MakeSalad$ to allow a legal execution to be found using the standard IndiGolog semantics. Such an execution is shown in figure 1(b). Note that the lack of true concurrency means only a single agent can act at each step, leaving the other two agents completely idle. The ability to reduce idle time by performing actions concurrency is clearly an advantage for distributed problem solving applications.

3.6. Distributed execution planning

As the existing languages based on Golog have demonstrated, an interpreter can be conveniently constructed using a logic programming language such a Prolog. We have followed the style of [3, 4] to build interpreters for our language in both Prolog and Oz [18], a multi-paradigm programming language with strong support for distributed computing.

One powerful feature of Oz is the ability to use several networked computers to search for solutions to a logic program in parallel. Since the task of planning a MIndiGolog execution is encoded as a logic program, this immediately allows a team of agents to distribute the execution planning workload. Below we briefly summarize our implementation; the full version is available from the author's website.

MIndiGolog programs are represented in Oz as record terms (which are similar to Prolog data terms) with the name of the record representing an operator and its features the arguments. Actions are also encoded as records. As in Prolog, uppercase terms in Oz represent variables. For example, the program:

$$\pi(agt)\ [acquire(agt, Bowl); acquire(agt, Knife)]$$

is represented as follows:

```
pi(agt seq(acquire(agt bowl)
           acquire(agt knife)))
```

The predicates $Trans$ and $Final$ have a straightforward encoding as Oz procedures, using the **case** statement to encode each individual clause using pattern matching, and the **choice** statement to explicitly introduce choice points:

```
proc {Trans D S Dp Sp}
  case D of nil then fail
  [] test(C) then {Holds.yes C S} Sp=S Dp=nil
  [] seq(D1 D2) then choice D1r in
                        {Trans D1 S D1r Sp}
                        Dp=seq(D1r D2)
                     []  {Final D1 S}
                        {Trans D2 S Dp Sp}
                     end
  [] pick(D1 D2) then choice
                        {Trans D1 S Dp Sp}
                     []  {Trans D2 S Dp Sp}
                     end
  [] ... <additional cases ommitted> ...
  end
end

proc {Final D S}
  case D of nil then skip
  [] test(Cond) then fail
  [] seq(D1 D2) then {Final D1 S}
                     {Final D2 S}
  [] pick(D1 D2) then choice
                        {Final D1 S}
                      [] {Final D2 S}
                      end
  [] ... <additional cases ommitted> ...
  end
end
```

A procedure $Do(\delta, s, s') \equiv Trans*(\delta, s, \delta', s') \wedge Final(\delta', s')$ is defined that determines a legal execution ***Sp*** for a given program ***D***:

```
proc {TransStar D S Dp Sp}
  choice  Dp=D Sp=S
  []  Dr Sr in {Trans D S Dr Sr}
               {TransStar Dr Sr Dp Sp}
  end
end

proc {Do D S Sp}
  local Dp in
    {TransStar D S Dp Sp}
    {Final Dp Sp}
  end
end
```

[3]A Prolog implementation of the MIndiGolog semantics, from which this trace was obtained, is available at http://www.csse.unimelb.edu.au/˜rfk/golog/

```
do [acquire_object(thomas,lettuce1),          do [place_in(richard,tomato1,board1),
    acquire_object(richard,tomato1),              acquire_object(harriet,bowl1)] at 12
    acquire_object(harriet,carrot1)] at 1     do [begin_task(richard,chop(board1)),
do [acquire_object(thomas,board1),                transfer(harriet,board2,bowl1)] at 13
    acquire_object(harriet,board2)] at 2      do [release_object(harriet,board2),
do [place_in(thomas,lettuce1,board1),             end_task(richard,chop(board1))] at 18
    place_in(harriet,carrot1,board2)] at 3    do [release_object(harriet,bowl1)] at 19
do [begin_task(thomas,chop(board1)),          do [acquire_object(richard,bowl1)] at 20
    begin_task(harriet,chop(board2))] at 4    do [transfer(richard,board1,bowl1)] at 21
do [end_task(thomas,chop(board1)),            do [release_object(richard,board1)] at 22
    end_task(harriet,chop(board2))] at 7      do [release_object(richard,bowl1)] at 23
do [acquire_object(thomas,bowl1)] at 8        do [acquire_object(thomas,bowl1)] at 24
do [transfer(thomas,board1,bowl1)] at 9       do [begin_task(thomas,mix(bowl1,1))] at 25
do [release_object(thomas,board1)] at 10      do [end_task(thomas,mix(bowl1,1))] at 26
do [release_object(thomas,bowl1),             do [release_object(thomas,bowl1)] at 27
    acquire_object(richard,board1)] at 11
```

(a) One possible execution of the $MakeSalad$ program with three agents, using MIndiGolog. Multiple actions occur at each step.

```
do acquire_object(thomas,lettuce1) at 1       do release_object(thomas,board1) at 21
do acquire_object(thomas,board1) at 2         do release_object(thomas,bowl1) at 22
do place_in(thomas,lettuce1,board1) at 3      do acquire_object(thomas,carrot1) at 23
do begin_task(thomas,chop(board1)) at 4       do acquire_object(thomas,board1) at 24
do end_task(thomas,chop(board1)) at 7         do place_in(thomas,carrot1,board1) at 25
do acquire_object(thomas,bowl1) at 8          do begin_task(thomas,chop(board1)) at 26
do transfer(thomas,board1,bowl1) at 9         do end_task(thomas,chop(board1)) at 29
do release_object(thomas,board1) at 10        do acquire_object(thomas,bowl1) at 30
do release_object(thomas,bowl1) at 11         do transfer(thomas,board1,bowl1) at 31
do acquire_object(thomas,tomato1) at 12       do release_object(thomas,board1) at 32
do acquire_object(thomas,board1) at 13        do release_object(thomas,bowl1) at 33
do place_in(thomas,tomato1,board1) at 14      do acquire_object(thomas,bowl1) at 34
do begin_task(thomas,chop(board1)) at 15      do begin_task(thomas,mix(bowl1,1)) at 35
do end_task(thomas,chop(board1)) at 18        do end_task(thomas,mix(bowl1,1)) at 36
do acquire_object(thomas,bowl1) at 19         do release_object(thomas,bowl1) at 37
do transfer(thomas,board1,bowl1) at 20
```

(b) One possible execution of the $MakeSalad$ program with three agents, using IndiGolog. Only one agent acts at each step.

Figure 1. Example executions of the $MakeSalad$ program

This ***Do*** procedure can then be passed to the parallel search functionality to plan a program execution. Here "agent1" and "agent2" are the DNS names of agents in the team, and "Goloz" is an Oz functor (basically, a portable piece of code) that exports the ***Do*** procedure defined above:

```
proc {ParallelDo D Exec}
    PS={New Search.parallel
        init(agent1:1#ssh agent2:1#ssh)}
  in
    Exec={PS one(Goloz $)}
end
```

When this code is run, it will utilize the computational resources of both agents to plan a legal execution of a given MIndiGolog program. This requires that the same information is available to each agent, which restricts the technique to fully-observable domains. We are currently developing an algorithm for cooperative execution of MIndiGolog programs that utilizes such distribution of the planning workload.

4. Related work

That Golog shows promise for multi-agent teams is evidenced by the success of [6] with a RoboCup soccer team executing a shared Golog program. However, the semantics of their Golog variant "ReadyLog" remain largely single-agent and do not address concerns such as: the possibility of performing actions concurrently and the coordination of concurrent actions; differing knowledge or beliefs between team members; sharing the computational workload of planning; and predicting the behavior of teams members and the environment in the face of many concurrently-executing tasks. MIndiGolog overcomes some of these limitations, while our ongoing work on cooperative execution will address the others.

As stated earlier, there has been much promising work on distributed problem solving systems using the Hierarchical Task Networks formalism ([17, 5, 8], among others). We believe high-level program execution to have several clear advantages over HTN, in particular the ready availability of controlled nondeterminism. Combined with familiar programming constructs such as loops and if-then-else, it provides a very powerful formalism for expressing complex behaviors and tasks [7, 2]. Golog also benefits from a logic of action rich enough to capture many challenging aspects of multi-agent domains (such as time and concurrency) while remaining computationally feasible.

Note that this paper focuses on task specification using Golog and does not deal with coordination between team members. We are currently developing techniques for cooperative execution of MIndiGolog programs based on these successful approaches to executing HTN specifications.

5. Conclusions and future work

Our work integrates several important extensions to the situation calculus and Golog to better model the dynamics of multi-agent teams. Specifically, MIndiGolog combines true and interleaved concurrency, an explicit account of time, and seamless integration of natural actions. It defines legal executions of high-level programs that are suitable for cooperative execution by a multi-agent team.

Since the semantics of MIndiGolog are based on first-order logic, existing techniques for distributed logic programming can be used to share the execution-planning workload between agents. In fully-observable domains, the parallel search capabilities of Oz can be used directly. We are currently developing a more sophisticated coordination strategy to augment these techniques and allow cooperative execution of MIndiGolog programs by a team of autonomous agents in partially observable domains.

Such coordination strategies are typically based on explicit mental attitudes such as knowledge and intention. A key aspect of our recent work has been the development of a computationally-feasible account of knowledge in partially observable domains [9]. This requires efficient reasoning about what cannot be changed by certain types of action, and we have developed a technique for answering such "persistence queries" under some simple assumptions [10]. An implementation of these techniques will form the base of our distributed problem solving system.

This paper thus represents a first step towards providing the advantages of Golog (such as controlled nondeterminism, powerful programming constructs, and a rich logic of action) for task specification for multi-agent teams. While significant work remains to be done to produce a full distributed problem solving system, our current implementations of MIndiGolog in Prolog and Oz, particularly combined with distributed logic programming techniques, have already proven useful for programming the behavior of multi-agent teams in fully observable domains.

References

[1] J. Baier and J. Pinto. Integrating true concurrency into the robot programming language GOLOG. In *XIX Int. Conf. of the Chilean C.S. Society*, 1999.

[2] C. Baral and T. Son. Extending ConGolog to allow partial ordering. In *Proc. of the 6th Int. Workshop on Agent Theories, Architectures, and Languages*, 2000.

[3] G. De Giacomo, Y. Lespérance, and H. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *AI*, 121(1-2):109–169, 2000.

[4] G. De Giacomo and H. Levesque. An incremental interpreter for high-level programs with sensing. In *Logical foundation for cognitive agents: contributions in honor of Ray Reiter*. Springer, 1999.

[5] K. Decker and V. Lesser. Designing a family of coordination algorithms. In *Proc. of the 1st Int. Conf. on Multi-Agent Systems (ICMAS'95)*, 1995.

[6] A. Ferrein, C. Fritz, and G. Lakemeyer. Using Golog for deliberation and team coordination in robotic soccer. *KI Kunstliche Intelligenz*, (1), 2005.

[7] A. Gabaldon. Programming hierarchical task networks in the situation calculus. In *AIPS'02 Workshop on On-line Planning and Scheduling*, 2002.

[8] B. Grosz, L. Hunsberger, and S. Kraus. Planning and acting together. *The AI Magazine*, 20(4):23–34, 1999.

[9] R. F. Kelly and A. R. Pearce. Knowledge and observations in the situation calculus. 2006. In Submission.

[10] R. F. Kelly and A. R. Pearce. Property persistence in the situation calculus. 2006. In Submission.

[11] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[12] F. Lin and Y. Shoham. Concurrent actions in the situation calculus. In *Proc. of the National Conference on Artificial Intelligence*, 1992.

[13] J. A. Pinto. *Temporal Reasoning in the Situation Calculus*. PhD thesis, University of Toronto, Toronto, 1994.

[14] F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):325–361, 1999.

[15] F. Pirri and R. Reiter. Planning with natural actions in the situation calculus. In *Logic-Based Artificial Intelligence*. Kluwer Press, 2000.

[16] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *KR'96: Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, 1996.

[17] M. Tambe. Towards flexible teamwork. *Journal of A.I. Research*, 7:83–124, 1997.

[18] P. Van Roy, P. Brand, D. Duchier, S. Haridi, M. Henz, and C. Schulte. Logic programming in the context of multi-paradigm programming: the Oz experience. *Theory and Practice of Logic Programming*, 3(6):717–763, 2003.