A comparative analysis of ConGolog and Concurrent METATEM

Michelle Blom (mlblom)

This report compares the principles underpinning two high level agent programming techniques: ConGolog [2] and Concurrent METATEM [5, 4]. The Golog family of languages, of which ConGolog is a member, aim to find legal executions of user-specified high-level programs. Each legal execution represents a plan that enables a collection of agents in a multi-agent system (MAS) to achieve a given task. ConGolog extends other languages in the Golog family by allowing the concurrent execution of actions by different agents in a system. Concurrent METATEM [5, 4] is a high level programming language for multi-agent systems based on first order temporal logic (FML) [3]. Concurrent METATEM approaches the implementation of multi-agent systems from a behavioural or event-based perspective, rather than being based on logic programming like ConGolog.

The two techniques will be demonstrated and compared using a problem from the cooking domain. This problem requires the implementation of a multi-agent system capable of organising and executing a dinner party for a large number of people. The desired system must be capable of conducting all stages of dinner organisation, from sending invitations, to food preparation, and cleaning up. For a description of the system design, refer to [1].

Specific aspects which will be analysed in the comparison are the perspectives by which both languages approach the implementation of multi-agent systems, the ease with which developers may transition from agent models into implementation, and the suitability of each language for a variety of applications, such as planning. The potential to combine the advantages of each approach will also be investigated.

1 ConGolog

ConGolog programs are described using axioms of the situation calculus [2, 6], together with operators and constructs to encode procedures, concurrent execution, iteration, and nondeterministic choices.

1.1 The Situation Calculus

The situation calculus is a formalism for reasoning about dynamically changing domains or worlds, developed by McCarthy in [9] and extended by Reiter in [10]. The situation calculus enables reasoning about *actions*, which change the state of the world, *situations*, which describe a sequence of actions applied to an initial state[1], and *fluents*, which represent properties of the world. These *fluents* may be *relational* – a predicate taking a situation as a final argument and returning a true or false value – or *functional* – which also take a situation as their final argument, but may return any type of value. For example, *is_man(harry, s)* is a relational fluent returning true if *harry* is a *man* in the situation s. An example of a *functional* fluent is *location(harry, s)* which may return the coordinate location of *harry* in the situation s.

Reasoning about action is achieved through the use of action precondition axioms, successor state axioms, a series of foundational domain independent axioms, and axioms representing the initial state of the system.

Action precondition axioms

Action precondition axioms specify the conditions that must be satisfied before a particular action may be performed. For example,

$$Poss(send\_invitation(i,g,p),s) \leftrightarrow is\_written(i,s) \wedge is\_stamped(i,s) \wedge is\_carrying(i,s) \wedge on\_list(g,p)$$

is a precondition axiom for the action of sending a dinner invitation i for the party p in the situation s to guest g. The action is only possible by an agent if the invitation has already been written and stamped, it is in its possession in situation s, and the invited guest is on the guest list.

[1] In the extended version by Reiter [10].

Successor state axioms

Successor state axioms provide a solution to the frame problem, and define, for each fluent, the ways in which its value may be changed. This is the alternative to frame axioms, which require the explicit specification of which fluents remain unchanged with each possible action. A successor state axiom has the form:

$$Poss(a,s) \rightarrow [F(\vec{x}, do(a,s)) \leftrightarrow \gamma_F^+(\vec{x},a,s) \vee (F(\vec{x},s) \wedge \neg\gamma_F^-(\vec{x},a,s))]$$

where $F(\vec{x},s)$ is the fluent, $\gamma_F^+(\vec{x},a,s)$ represents the ways in which it can be made true, $\gamma_F^-(\vec{x},a,s)$ represents the ways in which it can be made false, and $do(a,s)$ represents the situation that results when action a is performed in situation s. The axiom expresses that if its possible to perform the action a in situation s, the fluent F will be true in the resulting situation $do(a,s)$ if the conditions expressed in $\gamma_F^+(\vec{x},a,s)$ are satisfied, or if the fluent is true in the situation s and the conditions in $\gamma_F^-(\vec{x},a,s)$ do not arise to make it false.

For example, a successor state axiom for the fluent *guest_invited(g, p, s)* might be:

$$\begin{aligned} Poss(a,s) \quad &\rightarrow \quad [guest\_invited(g,p,do(a,s)) \leftrightarrow a = send\_invitation(i,g,p) \wedge post\_operational(g,s) \\ &\vee \quad (guest\_invited(g,p,s) \wedge a \neq uninvite(g,p))] \end{aligned}$$

where the fluent is true if an invitation has been sent to the guest g for the party p in situation s, the postal service in the guest's area is operational, and the guest has not been uninvited to the party.

Foundational axioms

The foundational axioms of the situation calculus describe the properties of situations, independent of the domain being reasoned about. The axioms may be found in [7], and highlight that situations are unique, that no situation exists that is not reachable from the initial situation by performing actions, and that new situations are obtained by adding actions to existing situations.

Initial state axioms

Axioms representing the initial state of the system may include statements such as: *location(x, S_o) = (0, 0)*, denoting that *x* is at location $(0,0)$ in the initial state, and $\forall i \neg sent(i, S_o)$, indicating that no dinner invitations have been sent in the initial state.

1.2 ConGolog operators

The basic operators and constructs present in the Golog programming language are as follows [2, 6]:

$\boldsymbol{\alpha}$	primitive action
$\boldsymbol{\phi?}$	wait for a condition
$\boldsymbol{(\sigma_1;\sigma_2)}$	sequential action execution
$\boldsymbol{(\sigma_1 \vert \sigma_2)}$	nondeterministic choice between actions
$\boldsymbol{\pi x.\sigma}$	nondeterministic choice between arguments
$\boldsymbol{\sigma^*}$	nondeterministic iteration
if $\boldsymbol{\phi}$ **then** $\boldsymbol{\sigma_1}$ **else** $\boldsymbol{\sigma_2}$	conditional
while $\boldsymbol{\phi}$ **do** $\boldsymbol{\sigma}$	loop
proc $\boldsymbol{\beta(\vec{x})}$ $\boldsymbol{\sigma}$ **end;**	procedure definition

These elements may be combined to produce procedures such as the following for inviting guests to a dinner party, where now denotes the current situation:

```
proc InviteGuests
    $\pi\ g$ [OnInviteList($g$, $now$)? ; WriteInvitation($g$, $i$) ; Stamp($i$) ; SendInvitation($i$)]
end ;
InviteGuests*;
$\neg\exists\ guest$ OnInviteList($guest$, $now$)?
```

ConGolog introduces the following new elements [6]:

$(\boldsymbol{\sigma_1} \| \boldsymbol{\sigma_2})$	concurrent execution of actions
$(\boldsymbol{\sigma_1} >> \boldsymbol{\sigma_2})$	concurrency with different priorities
$\boldsymbol{\sigma}''$	concurrent iteration
$< \boldsymbol{\phi} \rightarrow \boldsymbol{\sigma} >$	if ϕ then interrupt and execute σ

Concurrency is achieved in ConGolog by interleaving the execution of actions in multiple programs. At any one point in time, actions from all concurrently executing programs may be performed, depending on the availability of resources.

Developing a ConGolog interpreter

A ConGolog interpreter may be developed and defined with the use of a transition semantics. Such an interpreter is required to find legal executions of high level programs consisting of ConGolog style procedures and operators. A transition semantics is defined which determines whether a system can transition from one situation (*s*) to another (*s'*) by performing one step in a program (σ). For each of the operators and constructs defined above, *Trans* and *Final* predicates are defined to indicate if a transition between two situations is possible by executing a program statement of that type, and whether a program together with a situation represents a final configuration. A final configuration may be one in which the program is empty, indicating that no steps need to be performed. A complete specification of the transition semantics may be found in [2].

The following examples demonstrate the *Trans* and *Final* predicates of some of the operators and constructs of ConGolog.

$$Trans(a, s, \sigma', s') \equiv Poss(a[s], s) \wedge \sigma' = nil \wedge s' = do(a[s], s)$$

denotes that by performing the action $a[s]$ in situation *s*, the resulting situation $do(a[s], s)$ is obtained.

$$Trans(if\ \ \phi\ \ then\ \ \sigma_1\ \ else\ \ \sigma_2, s, \sigma', s') \equiv \phi[s] \wedge Trans(\sigma_1, s, \sigma', s') \vee \neg\phi[s] \wedge Trans(\sigma_2, s, \sigma', s')$$

denotes that by executing the conditional statement, the system may transition from situation *s* to *s'* if $\phi[s]$ is true in situation *s* and the program σ_1 can cause the transition, or $\phi[s]$ is not true in *s* and the program σ_2 can cause the transition.

$$Final(nil, s) \equiv True$$

denotes that an empty program together with situation s is a final configuration as no further steps need be executed.

$$Final(while\ \ \phi\ \ do\ \ \sigma, s) \equiv \neg\phi[s] \vee Final(\sigma, s)$$

denotes that if $\neg\phi[s]$ is true, then the program represented by the while loop and the situation s is a final configuration (as the contents of the while loop are not executed). If $\phi[s]$ is true, then the configuration is only final if the program σ requires no steps to be executed.

ConGolog additionally provides support for the execution of exogenous actions – actions which are not defined as part of a user program, but which may execute in the background. These actions are defined using a predicate denoted Exo. A program designed to nondeterministically select an exogenous action or event to perform is developed and executed concurrently with a user defined program.

2 Concurrent METATEM

Concurrent METATEM [5, 4] is a high level programming language for multi-agent systems based on first order temporal logic (FML) [3]. METATEM refers to the family of temporal logics, which includes FML, that may be executed. FML may be viewed as classical logic with some additional connectives allowing the construction of temporal logic formulas.

2.1 First Order Temporal Logic

First Order Temporal Logic (FML) is obtained by adding the following connectives – until ($\mathcal{U}$) and since ($\mathcal{S}$) – to the connectives of first order classical logic. With these connectives we can express the following statements and define additional modalities [5, 4, 8].

$\phi\,\mathcal{U}\,\psi$	is true now if ϕ will be true until ψ becomes true at a future timepoint.
$\phi\,\mathcal{S}\,\psi$	is true now if ϕ was true until ψ became true at some past timepoint.
$\bigcirc\phi$	is true now if ϕ will be true in the next state.
$\mathbf{O}\phi$	is true now if ϕ was true in the previous state *and* 'now' is not the initial state.
$\bullet\phi$	is true now if ϕ was true in the previous state *or* if 'now' is the initial state.
$\Diamond\phi$	is true now if ϕ will be true sometime in the future.
$\blacklozenge\phi$	is true now if ϕ was true sometime in the past.
$\Box\phi$	is true now if ϕ will be true at *all* future timepoints.
$\blacksquare\phi$	is true now if ϕ was true at *all* past timepoints.
$\boldsymbol{start}$	is true in the initial state.

2.2 Concurrent METATEM systems

A concurrent METATEM system contains a set of concurrently executing elements (or agents) which broadcast messages (asynchronously) to each other to communicate [5, 4]. Each element performs its duties by interpreting a series of rules expressed in the temporal logic described in Section 2.1. Each system is an instance of the Concurrent METATEM Process model – an abstract model that specifies each executing element with an interface and internal definition.

The interface definition, which may be changed dynamically, specifies which broadcast messages an element will recognise and act upon, and which messages it is capable of sending. A supplier agent, for example, who is responsible for providing the planner of a party with supplies, might have the following interface definition:

supplier(order)[accept, refuse, deliver]

This interface definition indicates that supplier agents recognise orders for supplies (*order*) from planner agents, and are able to accept orders (*accept*), refuse them (*refuse*), and deliver supplies (*deliver*).

Within the internal definition of an element, a computational engine which interprets and executes temporal logic formulae is specified. These formulae take the form of *if – then* rules:

$$p \rightarrow f$$

where p and f are temporal logic expressions, with p (the *antecedent*) representing a formula about the past or present, and f (the *consequent*) representing a formula about the future. At a given instance in time, the element will determine which rules have an antecedent that is true at the current timepoint, and must ensure that the consequents of these rules are also true at the current timepoint. This will typically require enforcing a constraint on future states (such that a particular expression is true in a future state, for example). A set of these *if – then* rules forms a METATEM program.

An executing element, or agent, will therefore continually execute the following tasks: listen for broadcast messages that it recognises, determine which of its rules have a true antecedent at the current timepoint, and take action to ensure that the consequents of these rules are true at the current point in time.

The following rules may be created for the supplier agent, for example, specifying its behaviour:

RULE ONE: $\boldsymbol{\bigcirc}$*order(supplier, supplies)* $\wedge$ *available(supplier, supplies)*

$\Longrightarrow$ *[*$\bigcirc$*accept(supplier, order)* $\wedge$ $\Diamond$*deliver(supplier, supplies, order)* $\wedge$ *adjust_stock(supplier, supplies)]*

RULE TWO: $\boldsymbol{\bigcirc}$*order(supplier, supplies)* $\wedge$ $\neg$*available(supplier, supplies)*

$\Longrightarrow$ $\bigcirc$*refuse(supplier, order)*

The first rule indicates that when a customer orders supplies from the supplier, and the required stock is available in the current state, then the supplier must ensure that: an acceptance is sent to the customer in the next state (where the acceptance message provides an indication of when the supplies will be delivered), the levels of stock the supplier has available to sell is adjusted, and at some future time the stock is delivered to the customer. The second rule indicates that when a customer orders supplies from the supplier, and the required stock is not be available in the current state, then the supplier must ensure that it sends a refusal to the customer (specifying that it cannot deliver it the required supplies) in the next state.

3 Comparative Analysis

ConGolog [2] and Concurrent METATEM [5, 4] are two distinct high level programming languages for multi-agent systems. Each language enables the implementation of agent systems from differing perspectives: one from the perspective of logic programming, and the other from the perspective of event-based simulation. These different perspectives gear the languages toward different applications, but are not all together uncomplementary.

3.1 Perspectives

The basic concepts underpinning each language are indeed different, but share some similarities. In ConGolog, action and change is represented by axioms of the situation calculus, and procedures or high level programs are defined outlining what needs to be done to achieve particular goals. A ConGolog interpreter is designed to find legal executions of these programs, utilising the situation calculus axioms which define the domain. Finding these legal executions, which define what each agent in a multi-agent system should do in a given situation to ultimately achieve its goals, is equivalent to finding a solution to a logic program. ConGolog therefore naturally views the problem of programming agents in multi-agent systems from the *future backwards* – determining what actions need to be performed to achieve a desired goal state in the future based on the current situation. Concurrent METATEM systems, on the other hand, describe what an agent should do or what an agent should make happen given that certain properties about the past and present are true. Concurrent METATEM therefore naturally views the programming problem from the *past* or *present forwards*. Another way of expressing this, is that systems programmed in ConGolog decide what to do in a more offline fashion, where the elements in a Concurrent METATEM system make choices incrementally. We can imagine, however, that once a legal execution of a program has been discovered using ConGolog, that agents have a series of rules informing them what actions to perform in various situations. Given this view, ConGolog and Concurrent METATEM are not entirely dissimilar.

The situation calculus versus temporal logic

A more basic difference between the two high level programming languages is that the interpretation of programs in each (procedures in ConGolog and rules in Concurrent METATEM) technique is based on two distinct formalisms: the situation calculus and temporal logic. In ConGolog, the properties of a situation that has transitioned from another with the execution of an action is determined using axioms of the situation calculus, which specify or encode how a particular aspect of an agent's environment may change. In Concurrent METATEM, programs are interpreted by analysing how certain aspects of an agent's environment have changed over time, and how they will change in the future. In the situation calculus, we deal with situations or action histories, and given a desired situation we define which history is required to achieve it. In temporal logics we deal with timepoints, the state of the world at those timepoints, and ensuring the a future state of the world will meet certain constraints. Given a situation, however, we may determine aspects of state using axioms of the situation calculus and resolution. If we view situations as timepoints, the situation calculus and temporal logics are not entirely dissimilar. Where one operates by constraining the past, the other operates by constraining the future.

3.2 Concurrency

In order to maximise the utilisation of available resources, an approach for programming agents in multi-agent systems requires some mechanism for managing the concurrent execution of actions. In ConGolog, legal executions may be found which assign actions to multiple agents to be performed in the same situation. For example, the following ConGolog procedure uses the concurrent execution operator (with priorities) << to emphasis that the line and cold station cooks [1] may begin to prepare the main meal and entree, respectively, at the same time (but that creating the entrees is of higher priority than creating the mains).

```
proc CookMeal($menu$)
    CookEntree($menu$) >> CookMain($menu$) >> CookDessert($menu$)
end ;
CookMeal($menu$)$^{||}$;
$\neg\exists$ $guest$ UnFed($guest$)?
```

The above procedure also uses the concurrent iteration operator. The statement CookMeal($menu$)$^{||}$ indicates that the entree, main and dessert for each guest of the dinner party may be cooked concurrently. Procedures are additionally defined that outline the steps required to cook each of the three components of a meal. For example, the following procedure may be used to make a salad as an entree for each of the guests.

```
proc CookEntree($menu$)
    $\pi$ $dest$ [IsContainer(dest)?
        $\pi$(agt) [IsType($agt$, $cold$)? ChopInto($tomato$, $dest$)] ||
        $\pi$(agt) [ISTYPE($agt$, $cold$)? CHOPINTO($lettuce$, $dest$)] ||

        $\pi$(agt) [ISTYPE($agt$, $cold$)? CHOPINTO($cheese$, $dest$)]]
    **end** ;
    COOKENTREE($menu$);
```

The above procedure nondeterministically selects a container for the salad, and nondeterministically selects cold station cooks to chop tomato, lettuce, and cheese into the container to make the salad. In this case, we assume we have a procedure outlining the required steps to chop an object into a container.

Whilst ConGolog may schedule the concurrent execution of actions, concurrency is achieved in a Concurrent METATEM system by virtue of the fact that it consists of elements which simultaneously interpret and execute their internal definitions. As described in Section 2.2, an agent in a Concurrent METATEM system continuously performs three tasks: listening for messages broadcast from other agents, determining which of the rules in its internal definition have *fired*, and executing the consequents of these *fired* rules. To program the preparation of a meal in Concurrent METATEM, we require definitions for a cold station and line cook, in addition to the definition of a head chef agent. The head chef agent is able to send orders to the cold station and line cooks, requesting that they begin preparing the entrees, main, or dessert courses. Each cold station and line cook is able to recognise these messages, and their reception triggers rules which result in the preparation of food. The definition of a cold station agent may take the following form, for example.

cold_station(order)[deliver] :

$\blacklozenge$*order(chef, entree, menu)* $\wedge$ *unfulfilled(order)*
$\implies$ *[*$\bigcirc$ *find_container(dest)* $\wedge$ $\bigcirc$*initiate_salad(dest, menu)]*

$\blacklozenge$*initiate_salad(dest, menu)* $\wedge$ $\neg$*contains(dest, lettuce)*
$\implies$ *[chop_into(lettuce, dest)]*

$\blacklozenge$*initiate_salad(dest, menu)* $\wedge$ $\neg$*contains(dest, tomato)*
$\implies$ *[chop_into(tomato, dest)]*

$\blacklozenge$*initiate_salad(dest, menu)* $\wedge$ $\neg$*contains(dest, cheese)*
$\implies$ *[chop_into(cheese, dest)]*

$\blacklozenge$*initiate_salad(dest, menu)* $\wedge$ *contains(dest, lettuce)* $\wedge$ *contains(dest, tomato)*
$\wedge$ *contains(dest, cheese)* $\implies$ *[*$\bigcirc$*deliver(order, dest)]*

If we have several cold station cooks with the above definition, then at each time point they will assess if there is a salad being made that is unfinished, and will determine what step should be performed next. Of course, necessary communication is required *between* the cooks (which for simplicity is not modelled in the above definition) to ensure that the actions each cook performs do not conflict.

3.3 Distribution

It is evident in Section 3.2 that ConGolog represents a more centralised approach to programming agent systems, in comparison with the decentralised (and more distributed) approach represented by Concurrent METATEM. In Concurrent METATEM, the behaviour of an agent is programmed in their interface and internal definitions. The behaviour of all agents in a system programmed with ConGolog is programmed in a single user defined procedure or set of procedures. This aspect of ConGolog enables it to be quite suited for planning tasks, however. Having an entire view of the agents in a system and the tasks that need to be performed allows for the optimal allocation of these tasks to be determined more easily.

Although behaviour in Concurrent METATEM systems is determined incrementally, planning may be implemented with some additional machinery. Consider an agent a which desires to construct a plan for achieving a goal g. By defining a process to split the goal into subgoals, processes to generate plans for each of these subgoals, and a process to synthesis or combine these plans, the development of a plan may be achieved in a Concurrent METATEM system. These processes may be individual agents where the invocation of a process represents communication amongst them. Planning is therefore implemented utilising the concept of hierarchical task decomposition, and by incorporating some offline, non-incremental decision making. Moreover, it seems reasonable that ConGolog style procedures may be incorporated into the consequents of rules in a Concurrent METATEM element (agent) to enable ConGolog style generation of plans.

From models to implementation

Concurrent METATEM's approach to programming agents by specifying behaviour in such a decentralised manner, with the encoding of a communication interface and an internal definition, allows for an intuitive mapping from agent models to implementation. Agent models encode behaviour in terms of communication between agents and scenarios which express how an agent should act given the changes it perceives of its environment. Such communication models may be intuitively mapped to the interface definitions of agents, and scenarios and behavioural models mapped to their internal definitions.

3.4 Applications

The differing perspectives of the two programming languages gear the approaches toward different applications. ConGolog – with its *future backwards* perspective – is heavily geared toward applications that require planning and the optimisation of planning tasks. Concurrent METATEM – with its *past forwards* perspective – is geared toward simulating the behaviour of autonomous entities in dynamic environments, and the encoding of action as a reaction to events in such environments. As demonstrated with the implementation of planning in Concurrent METATEM, however, it is evident that additional machinery may be integrated into both approaches to tackle the same problems. By incorporating aspects of ConGolog and Concurrent METATEM together, a more well-rounded approach can be achieved.

4 Conclusion

This report compared the principles underpinning two high level agent programming techniques: ConGolog [2] and Concurrent METATEM [5, 4]. It was found that ConGolog and Concurrent METATEM program agent systems from two different perspectives. ConGolog enables legal executions (action sequences) of high level programs to be found in an offline fashion, where the elements in a Concurrent METATEM system make choices incrementally. The two high level programming languages interpret programs (procedures in ConGolog and rules in Concurrent METATEM) based on two distinct formalisms: the situation calculus (ConGolog) and temporal logic (Concurrent METATEM). Concurrency is scheduled in ConGolog, where in a Concurrent METATEM system it is achieved by virtue of its concurrently executing elements which simultaneously interpret and execute their internal definitions. The decentralised manner in which agents are programmed and their behaviour described makes Concurrent METATEM systems ideal for simulation in dynamic environments, where ConGolog is suited for planning and planning optimisation tasks.

References

[1] Michelle Blom. Dinner manager plus. 433-482 Assignment One, 2008.

[2] Giuseppe de Giacomo, Yves Lesperance, and Hector J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[3] M. Fisher. A normal form for first-order temporal formulae. In *Proceedings of the 11th International Conference on Automated Deduction (CADE'92)*, pages 370–384, 1992.

[4] M. Fisher. Concurrent metatem – a language for modeling reactive systems. In *Proceedings of the 5th International Conference on Parrallel Architectures and Language, Europe (PARLE'93)*, pages 185–196, 1993.

[5] M. Fisher and H. Barringer. Concurrent metatem process – a language for distributed ai. In *Proceedings of the 1991 European Simulation Multiconference*, 1991.

[6] Y. Lespérance, H. J. Levesque, and R. Reiter. A situation calculus approach to modeling and programming agents. *Foundations and Theories of Rational Agency*, pages 275–299, 1999.

[7] Hector J. Levesque, Fiora Pirri, and Raymond Reiter. Foundations for the situation calculus. *Electron. Trans. Artif. Intell.*, 2:159–178, 1998.

[8] Viviana Mascardi, Maurizio Martelli, and Leon Sterling. Logic-based specification languages for intelligent software agents. *Theory and Practice of Logic Programming*, 4(4):429–494, 2004.

[9] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in McC90.

[10] Raymond Reiter. The frame problem in situation the calculus: a simple solution (sometimes) and a completeness result for goal regression. pages 359–380, 1991.