A Cooperation Language

Michael Kolb
German Research Center for Artificial Intelligence Inc.
PO-Box 2080
67608 Kaiserslautern
Germany
kolb@dfki.uni-kl.de

Abstract

This paper introduces CooL, a programming language for building cooperative applications. It combines the expressiveness of a high-level AOP-language with the efficiency required by industrial applications. It integrates the support for planning and scheduling with efficient execution on the single agent as well as multi-agent levels (cooperation). CooL's knowledge representation and execution facilities are introduced, yielding the mechanisms that allow for easy programming of cooperations on the basis of cooperation primitives with a formal semantics.

Introduction

Research on multi-agent systems nowadays faces its biggest challenge, the transfer of the established methodologies and technologies into industrial systems. While there is great progress on the theoretical foundations of MASs and a steadily growing number of prototypical systems which implement them, one can find a discrepancy between concepts and implementations when it comes down to real industrial applications.

CooL, the Cooperation Language, is intended as a first step towards bridging the gap between research and product development. It combines the expressiveness of a prototypical research language with the efficiency and universality of an established programming language such as C or C++. It was developed in the research project CoMMA, a cooperation between DFKI and SIEMENS AG, as a continuation of the prior work on the MAI²L language.[1]

The purpose of this paper is to highlight the key features of CooL. The underlying concepts and their realisation are exemplified using excerpts of the implementation of an urban traffic scenario (Haugeneder & Steiner 1993). It involves several different types of agents, namely cars, car parks and traffic guidance systems, which together form a multi-agent system allowing the cars to park at car parks according to goals given by the user.

All the agents in the scenario share a common design. According to (Steiner, Mahling, & Haugeneder 1990) an agent is divided into three parts, a communicator, a head and a body. The body includes the agent's basic functionality, like driving the car (in a very abstract sense). The head covers the reasoning of the agent. It takes the user's goals and maps them to executable plans, possibly involving cooperations with other agents, schedules and executes this plans. During execution, the head stays responsive to unforeseen events that may occur in the world. The communicator allows the head to abstract from communication specific details during its cooperations with other agents. (Haugeneder 1994) and (Steiner et al. 1993) give a more detailed description of the agent architecture.

With respect to this architecture, CooL covers the head of an agent and its communicator. Clearly, it can also be used to implement the body of an agent, or alternatively interface to an external library to agentify an existing system.

In this paper the knowledge representation and processing aspects of CooL at the single agent level are presented. Then an example of how CooL is used to support cooperation among agents is given. The details of the language introduced in this work only serve to clarify the examples. A full specification of CooL is given in (Kolb 1995).

Elements of CooL

CooL can be characterised as a multi-threaded, object-oriented procedural language. Procedures can be converted into data and vice versa in order to alter them or add new ones dynamically. It comes with an integrated class hierarchy to implement agents and cooperations.

CooL provides simple data types including symbols, strings, numbers, files, threads. Simple data can be structured by two different types of sets used for knowledge representation. Arbitrarily related data is stored in an *association*, an ordered multi-set mathematically spoken. The syntax for an association is:

```
[any ... ].
```

On the other hand, *contexts* are sets consisting of only associations. A particular association can directly be accessed in a context by using its first element as an index. Every evaluation in CooL refers to an environment which is defined by an association of contexts. The environment

[1] Developed in the ESPRIT project IMAGINE.

is processed from left to right by the search operators and if a new context is opened, it is added as the first element to the environment. The foremost context can be removed from the environment by closing it. There is always one context, the system context, in the environment, which can not be removed. The following example illustrates the use of contexts (single quotes mark symbols in contrast to variables):

```
'c1 := new <<
  'a := 5
>>
```

A new context including the association `[a 5]` is created and associated with `c1` in the current context, the first context in the environment.

```
'a := "Hello World\n"
```

`[a "Hello World\n"]` is added to the current context. The evaluation:

```
print(? 'a)
```

yields the output of the string `Hello World`. Now, the context `c1` is opened, and a is evaluated again:

```
? 'c1 <<
print(? 'a)
```

yielding the output `5`. The environment now consists of the two contexts `[<<[a 5]>>` *system_context*`]`.

The object system of CooL is based on contexts and the use of the environment. Every class and instance is represented as a context. An association of contexts can be assigned to each object to represent the classes it inherits from, its inheritance environment. When a message is sent to an object, the inheritance environment and the context representing the object itself are added to the current environment. The method is now evaluated wrt. the new environment. When the method is done, the environment before the send operation is restored.

Execution is defined by procedures in CooL, which are represented internally as executable associations. A procedure consists of a sequence of statements and may return one or more values. Variables have not to be declared within a procedure; a variable can hold any value.

The following procedure computes the cost of a car to park at a specific car park:

```
proc cost_to_park (car_park) {
  d = get_distance(car_park)
  f = car_park\'fee
  g = get_guarantee(car_park)
  return(d * f / g)
}
```

The internal representation of a procedure is an association containing a sequence of statements. The representation of the above procedure is:

```
[cost_to_park [car_park]
  [assign d [get_distance car_park]]
  [assign f [get car_park 'fee]]
  [assign g [get_guarantee car_park]]
  [assign _1 [* d f]]
  [assign _2 [/ _1 g]]
  [return _2]]
```

The value of `f` is computed by retrieving the value associated with `fee` in the context `car_park`. The `\`-operator used in the example is defined in the following way:

Context \ *Index* = *X*, *if* [*Index X*] ∈ *Context*
[*any* ...], *otherwise*

The procedure is split into single statements yielding a flat structure that is easier to analyse and directly reflects the order in which execution of the actions takes place.

Within the body of a procedure, a set of basic control structures is available to the programmer. They allow for composition such as conditional branching, looping and recursion, as well as synchronised parallel execution and forking in addition to the usual sequentialisation. Whilst execution immediately resumes after a procedure is forked, synchronised parallel execution merges the control flow after every branch of the parallel structure is completed. If the parallel branches return values they are ordered according to the order of branches.

CooL also supports reactive behaviour of agents by *demons* that specify a condition upon which a procedure is executed. The condition either relates to some absolute or relative *timepoint*, or specifies a *state* of an agent's environment (its world model). A demon suspends the current thread as soon as its condition holds. The thread is resumed upon termination of the demon's procedure. The following example shows the activation of a demon that reacts to traffic lights, while a car is driving:

```
proc drive_path (path) {
  on {exists 'red_traffic_light} {
    wait_for_green()
  }
  foreach nav_pt in path {
    drive_to(nav_pt)
  }
}
proc wait_for_green {
  while {exists 'red_traffic_light} {
  }
}
```

The example relies on the existence of another thread, which by some means monitors the real world and adds new observations to the current context.

External library functions that are loaded dynamically into the system can be called from within a procedure. There is also a set of C library functions available to access the CooL knowledge representation from other programming languages.

Agents in CooL

Agent are defined as objects, ie. instances of a class according to the type of agent. There is a set of predefined agent classes such as ADS agents and monitor agents which are required to implement, debug and maintain a multi-agent system. Since agents are objects, their knowledge base is the instance-context, holding information about their world model, their internal state etc.

Activities about which an agent has to reason can be specified as *plans* in CooL. Plans are defined as objects,

which hold information about the plan in addition to the plan procedure. The information is intended to be used by a planner and a scheduler that manages an agent's activities and includes preconditions, effects, duration etc. Unlike procedures, which are evaluated step-by-step according to the sequence of calls in their body, a call to a plan is recursively un-rolled, i.e. every subsequent call from within the plan's procedure is determined immediately. The unroll-mechanism stops, when the resulting structure consists only of procedure calls. The resulting structure is a partially ordered set of events, which is inserted into the agent's schedule (see (Kowalski & Sergot 1986) on event calculus). Its expressiveness provides a good basis for reasoning about an agent's schedule or planning its future course of activities, however it must be used with care regarding parameters generated by user-procedures, plan recursion etc.

A typical example for the use of planning in the UTS is the entrance procedure of a car at a particular car park. While the car is executing its plan to get to a selected car park, it negotiates with the car park about the protocol it has to follow upon arrival. The following plan procedure may result from such a negotiation:

```
proc enter_car_park () {
  cp = ? 'car_park
  query_ticket(cp)
  p = query_free_place(cp)
  drive_to(p)
},
```

where `? 'car_park` is a KB-query to retrieve the actual car_park in the current environment upon execution of the plan. In order to schedule the individual actions of this procedure the car has to send the `schedule` message to the plan object containing the procedure.

If a plan is scheduled without any constraints, it is inserted immediately in parallel to the existing schedule. Obviously, the search for the appropriate insertion point in the schedule may be expensive and might fail because of inappropriate plan usage. Therefore, to maintain maximum efficiency, plans are used only for domain-related activities that have to be coordinated with other agents.

While there is no specific planning engine directly included in CooL, the use of plans and schedules provides a solid basis for a planning system to work on. In combination with demons, the implementation of reactive behaviour that is related to plan execution as described in (Dabija 1993) is straightforward.

So far, the presented concepts focused on the single agent aspect of agent oriented programming; the next section deals with CooL's support for cooperating agents in multi-agent systems.

Cooperations in CooL

CooL allows for a high level specification of cooperation methods with powerful operations that involve a lot of default activity. Yet it provides flexibility by allowing the programmer to override the default behaviour. The key features are:

- cooperation methods to describe complex protocols in a single procedure
- characters to represent actual agents in a cooperation
- extension of object-oriented message passing to agent communication by the communicator
- cooperation primitives with formal semantics giving meaning to individual messages independently from protocols

The cooperation model of CooL is based on the ideas in (Steiner et al. 1993) and (Lux & Steiner 1995). Agents cooperate by negotiating a cooperation object, e.g. a goal, a plan, a schedule, or by synchronising mutual execution of a plan. A fixed cooperation protocol is specified by a *cooperation method*, which can be looked upon as a domain-independent multi-agent plan, represented as a plan or procedure, that incorporates message exchange by *cooperation primitives* among agents as well as the execution of procedures individually by the involved agents. A cooperation method must be specified independently from the particular cooperative setting.

CooL uses contexts to abstract from the particularities of a cooperation in order to keep the cooperations universally applicable. Each cooperation is executed in a new context. Before the cooperation is executed, the necessary parameters are defined in this context. During the cooperation, persistent changes of an agent's state has to be moved from the cooperative context to the agent itself, since the cooperative context is removed after the cooperation method is finished.

Within a multi-agent plan, each statement is preceded by a *character*, that identifies the agent which executes the statement. The character of an agent describes its role in the cooperative game. The contract-net cooperation method (Smith 1980) initially involves two characters, a manager and a bidder, where the bidder character represents a set of bidding agents. During a contract net negotiation, one of the bidders becomes the contractor, so the character of an agent can dynamically change during a cooperation.

The cooperative context holds an association for each character which contains the name(s) of the agent(s) identified by this character. Upon execution of the method the system checks, whether the character of the next statement is bound to the agent that executes the method. If they match, the statement is executed, otherwise it is skipped.

The following example shows a multi-agent plan executed by the two characters `car` and `car_park` to enter a car park:

```
proc enter () {
  car: drive_to_entrance()
  car: request_entrance()
  car_park: open_barrier()
  car: drive_to_space()
}
```

To execute this plan, the car has to build up the appropriate

context by defining:

```
new <<
  car := self\'name
  car_park := cp_1     % name of an agent
  % now the procedure is called in this
  % environment
  enter()
>>
```

In the above example, the communication to synchronise the execution is handled within the called procedures.

Communication

Communication among agents is done via a set of cooperation primitives that are exchanged locally or over a network. A cooperation primitive consists of a type, which specifies the intention behind the communication (Searle 1969), and a cooperation object.

In CooL message exchange among agents closely resembles message passing to objects. When an agent communicates the first time with another agent, its communicator queries its Agent Directory Service (ADS) about information on the network address and process number of the recipient. The communicator then creates a new instance of the class `acquaintance` which stores this information.

In the acquaintance object all cooperation primitives are defined as methods. It represents the send-target on the sender side and does the necessary network communication to the actual recipient. The actual message contains additional information, that is added automatically by the communicator of the sender:

- a unique id to identify the ongoing cooperation and its context
- the character name and the agent name of the sender and the recipient
- the name of the cooperation method, if the method is sent within a fixed protocol
- the type of the primitive indicating the intention behind the message.

This information must be available in the cooperative context of the initiator of a cooperation method, the one that sends the first message, before the cooperation is started.

On the recipient side, the communicator examines every incoming message and decides on whether it belongs to an existing cooperation or a new one. If it is a new cooperation, the method name is examined and depending on whether it is known or not, the method is forked in a new cooperative context. If the method is unknown, the communicator forks a new cooperative context and forks the method according to the type of the message. Provided the agent has sufficient planning capabilities, it can participate the cooperation without knowing the protocol.

The communicator puts all the additional information contained in the message into the newly defined context.

Each cooperation primitive has a semantics on the sender and the recipient side, which specifies the change of the state of the corresponding agent embedded in a planning context. The exact definition of cooperation primitives and their formal semantics in MECCA is given in (Lux & Steiner 1995). The semantics of the cooperation primitives is defined by the methods on both sides of the communication process. However, it is the communicator on the recipient side that decides on whether the cooperation method is unknown or unspecified and the default semantics is used. If the cooperation method is known, the semantics of the primitive is defined in the method itself, and the message object is bound to a variable.

Cooperation Methods

The initiator of a cooperation starts the execution of the cooperation method by another agent with the first cooperation primitive sent to that agent. The latter's communicator creates a new context, stores information about the involved characters in this context and forks the cooperation method as requested by the sender.

Each agent registers at its agent directory service upon start-up. The ADS stores information about agent names, types, network addresses and capabilities. Other agents use this information explicitly to find appropriate agents for a particular task, or implicitly when communicating with the agent.

The following cooperation method describes the registration of an agent with its ADS:

```
proc register () {
  % order a task to the ads(reg_server)
  reg_server:task <-
    reg_client:order(['task
      ['add_kb ['agent
                  self\'name self\'type
                  self\'host self\'port
                ]]])
  reg_server:exec(task\'object)
}
```

with the appropriate call on the client side:

```
new <<
  reg_client := self\'name
  reg_server := self\'ads\'name
  % now the call:
  register()
>>
```

The general form of the cooperation primitives is:

Recipient:*Variable* <- *Sender*:*Type* (*Object*) ,

where Recipient and Sender represent character names.

The cooperation primitive `order` sends the following knowledge to the agent associated with the `reg_server` character:

```
<<order [object [task ...]]
  [type order]
  [coop_id c0815]
  [to my_ads reg_server]
  [from my_name reg_client]
  [method register]]>>
```

which is bound to the variable `task` on the `reg_server` side. The ADS then executes the task associated to

object.
The cooperation is finished and the cooperative context is deleted. It should be noted that the add_kb procedure has to take care of adding the knowledge about the client to the agent itself rather than to the cooperative context.

Figure 1 gives the implementation of the canonical *contract-net* cooperation method. The contract-net protocol is one of the cooperation methods used by the cars in the UTS to find an appropriate car_park. It is, however, completely domain-independent, since it only relies on the existence of domain-related evaluation functions in the cooperative context, as the following call shows:

```
new <<
  manager := ? 'name
  bidder := ? 'selected_car_parks
  timeout := 20
  eval_func := ? 'eval_parking_bid
  prepare_task := ? 'drive_to_car_park
>>
```

The car defines itself as the manager, and the agents associated with `selected_car_parks` as the bidders; the method to evaluate the bids is defined and the procedure to prepare the task. Both are specific to the UTS scenario.

The manager initiates the contract-net by sending proposals to all the bidders. The bidders try to refine the call for bids and send them back to the manager. The manager only evaluates the bids that were received within the timeout of 20 seconds that is specified in the cooperative context above. After the evaluation, the manager sends the accept and reject messages to the bidders in parallel.
Now the accepted bidder dynamically changes its character. The manager has to update its own knowledge about the character of the accepted bidder, since it sends the next message to the accepted bidder. If the bidder had sent the next message, the update would have been automatically done by the communicator. Then the execution of the task is prepared, the task is executed and the result is reported to the manager.

In addition to what is explained in this paper, the dynamic cooperation model in CooL supports the following features:

- several different ongoing cooperations related to the same context using common knowledge
- merging of independent contexts upon realisation of interdependencies (such as conflicts due to usage of common resources)
- arbitrary incorporation of new partners in an ongoing cooperative process

The semantic embedding of the cooperation primitives provides a solid basis for future research on dynamically created or altered cooperation methods.

Conclusion and Outlook

The CooL programming paradigm gives a knowledge and execution perspective on agents rather than describing them by mental states (Shoham 1993), (Thomas 1993). The intention behind this is to be able to support a wide

```
proc contract_net () {
  bidder:c_f_b <- manager:propose(? 'bid)
  bidder:ref = c_net_refine_bid(c_f_b)
  manager:bids (? 'timeout)<- bidder:refine(ref)
  manager:evaluated_bids = call(? 'eval_func,[bids])
  manager:acc_bid = first(evaluated_bids)
  manager:rej_bids = rest(evaluated_bids)
  bidder:reply <-
         manager:accept([['object acc_bid\'object],
                 ['to acc_bid\'from]]),   % parallel to the next statement
         foreach eb in rej_bids {
           reject([['object eb\'object]
                   ['to eb\'from]])
         }
  manager:acc_bidder := acc_bid\'from
  bidder: if (reply\'coop_type == 'accept) {
     acc_bidder := self\'name
  }
  manager:exec(? 'prepare_task,[acc_bid\'object])
  acc_bidder:task <- manager:order(['object acc_bid\'object])
  acc_bidder:result = exec(task)
  manager:result <- acc_bidder:tell(['result result])
  manager:return(result\'object)}
```

Figure 1: Contract-Net Example

variety of agent architectures. In particular BDI-approaches (Rao & Georgeff 1991) will be incorporated in a multi-layered agent architecture developed in the CoMMA project at the DFKI. On the other hand, CooL provides a higher level of abstraction wrt. agent design than languages such as ACTORS (Agha 1986), April (McCabe & Clark 1994) and Oz (Henz, Smolka, & Wuertz 1993).

CooL is implemented in C. It will be used to do prototypical implementations of a Personal Traffic Assistant, providing multi-modal trip support, and a Personal Intelligent Secretary, providing automatic meeting scheduling.

Further work will investigate mapping CooL messages onto existing knowledge representation and exchange formats, such as KIF, KQML and CORBA. It is also planned to implement an event calculus based planning engine with abduction in CooL, which will be integrated with our agent architecture.

Acknowledgements

I would like to thank SIEMENS AG for supporting this work as well as the members of the CoMMA group at the DFKI, especially Andreas Lux for his assiduous work on fixing the semantics of the cooperation primitives.

References

Agha, G. 1986. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press.

Chu, D.; and Clark, K. 1993. I.C. Prolog II: a Multi-threaded Prolog System. In Proceedings of the ICLP'93 Post Conference Workshop on Concurrent, Distributed & Parallel Implementations of Logic Programming Systems.

Dabija, V. G. 1993. Deciding Whether to Plan to React. Ph.D. diss., Dept. of Computer Science, Stanford University.

Haugeneder, H. ed. 1994. *IMAGINE Final Report.* SIEMENS AG, Munich.

Haugeneder, H. and Steiner, D. 1993. A Multi-Agent Approach to Cooperation in Urban Traffic. In Proceedings of the Conference on Cooperating Knowledge Based Systems (CKBS). Keele, UK.

Henz, M.; Smolka, G.; and Wuertz, J. 1993. Oz - a programming language for multi-agent systems. In Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93). Chambéry.

Kolb, M. 1995. CooL Specification, Technical Report, SIEMENS AG.

Kowalski, R. A.; and Sergot, M. 1986. A logic based calculus of events. *New Generation Computing 4*: 67-95.

Lux, A. 1992. A Multi-Agent Approach towards Group Scheduling, Research Report, RR-92-41. DFKI, Kaiserslautern.

Lux, A.; and Steiner, D. 1995. Agents and Cooperation in MECCA - A Formal Operational View. In Proceedings of the First International Conference on Multi-Agent Systems. San Francisco.

McCabe, F. G.; and Clark, K. L. 1994. April - Agent Process and Interaction Language. In Proceedings of the 1994 Workshop on Agent Theories, Architectures, and Languages. Amsterdam.

Rao, A. S.; and Georgeff, M. P. 1991. Modelling rational agents within a BDI architecture. In Proceedings of Knowledge Representation and Reasoning (KR-91), 473-484.

Searle, J. R. 1969. *Speech Acts.* Cambridge University Press, Cambridge.

Shoham, Y. 1993. Agent-oriented programming. *Artificial Intelligence* 60:51-92.

Smith, R.G. 1980. The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers* 29: 1104-1113.

Steiner, D.; Burt, A.; Kolb, M.; and Lerin, C. 1993. The Conceptual Framework of MAI^{2}L. In Proceedings of MAAMAW-93. Neuchatel, Switzerland.

Steiner, D.; Mahling, D.; and Haugeneder, H. 1990. Human Computer Cooperative Work. In Proceedings of the 10th International Workshop on Distributed Artificial Intelligence. MCC Technical Report, ACT-AI-355-90, Austin/TX.

Thomas, S. R. 1993. PLACA, an Agent Oriented Programming Language. Ph.D. diss., Dept. of Computer Science, Stanford University.