Investigating Strategies for Cooperative Planning of Independent Agents through Prototype Evaluation*

E.-E. Doberkat W. Hasselbring C. Pahl

University of Dortmund
Dept. of Computer Science, Informatik 10 (Software Technology)
D-44221 Dortmund, Germany
Tel.: 49-(231)-755-2780/2781, Fax: 49-(231)-755-2061
{doberkat|willi|pahl}@ls10.informatik.uni-dortmund.de

Abstract

This paper discusses the application of the prototyping approach to investigating the requirements on strategies for cooperative planning and conflict resolution of independent agents by means of an example application: the strategic game "Scotland Yard". The strategies for coordinating the agents, which are parallel algorithms, are developed with a prototyping approach using ProSet-Linda. ProSet-Linda is designed for prototyping parallel algorithms.

We concentrate on the techniques employed to elicit the requirements on the algorithms for agent interaction. The example application serves to illustrate the prototyping approach to requirements elicitation by means of a non-trivial instance for investigating algorithms for cooperative planning and conflict resolution.

Keywords: cooperative planning, multi-agent systems, prototyping parallel algorithms, requirements elicitation

*A shortened version to appear in the Proceedings of the First International Conference on Coordination Models and Languages, Cesena, Italy, Springer-Verlag Lecture Notes in Computer Science, April 1996.

1 Introduction

Cooperative planning of independent agents is a realistic problem which requires careful study. For concentrating on the essential aspects (plan generation, conflict resolution) we propose in this paper a prototypical approach which is realized for a strategic game called "Scotland Yard". This game has a number of cooperating detectives who chase a villain through London using different means of public transportation. The villain's moves are only partially visible. Each detective develops for each move a plan which may or may not conflict with the plans of fellow agents; if it does, the conflict has to be resolved before all the agents make their moves. There is no master detective who supervises plan generation in general (and conflict resolution in particular), so the detectives have to come to terms on their own.

Finding a clear and intelligible solution to plan generation and conflict resolution is certainly more important than obtaining directly a very efficient program — once a solution is found through exploration, it may be used as an executable specification for an efficient implementation. Consequently, we concentrate on conceptual aspects and implement our solution in a prototyping language. The language is based on finite sets and multisets. Set theoretic notions come into our game quite naturally: e.g. the collection of all plans may be a multiset, since more than one detective may have formulated the same plan. Each plan must be inspected by every detective, so the plans are written on a blackboard. Technically, this may be thought of as generative communication, so the blackboard is implemented as a *tuple space* in the sense of Linda [10]. This implies that a prototyping language providing sets as well as tuple spaces will suit our purposes well.

Another aspect of prototyping should be mentioned: prototyping means modelling essential features, and strategies, which are certainly essential here, may very well be isolated textually from the rest of the code. Then it is easy to experiment with strategies and, equally important, easy to argue even informally about strategies: this is so since the very high level character of our prototyping language makes the details of a strategy rather transparent (which would not always be the case in programs written in one of the common production languages).

The main technical contribution of this paper is demonstrating the flexibility of incorporating different approaches for planning and conflict resolution strategies for independent agents. This is made possible through the use of a very high-level language and the corresponding techniques for exploratively prototyping algorithms.

Section 2 takes a general look at cooperative planning of independent agents. Section 3 presents our example application and Section 4 discusses some strategies for the agents of this application. Section 5 provides a brief introduction to the prototyping language ProSet-Linda and Section 6 presents the design and implementation of the program for our example application. Section 6 essentially presents the work of our project group "Scotland Yard".[1] The Evaluation of the investigated strategies is discussed in Section 7.

[1]In the computer science curriculum of the University of Dortmund, a project group consists of twelve students working on a project for the duration of a year. We acknowledge the work of our students Klaus Alfert, Oliver Alsbach, Jörn Bodemann, Markus Brameier, Stefan Hedtfeld, Marcus Kirsch, Peter

Section 8 takes a look a related work and Section 9 draws some conclusions and indicates extensions.

2 Cooperative Planning of Independent Agents

Distributed artificial intelligence is concerned with the development and analysis of ensembles of cooperating (intelligent) processes. These processes are called agents. In multi-agent architectures, a set of autonomous agents cooperate to achieve a common goal. The individual agent does not need to construct a plan that solves the whole problem. The agent develops only the part of the plan which applies in his own domain of responsibility or his area of knowledge. An autonomous agent is independent in his decisions from the proposals of other agents, but is constrained by the rules of the problem. The agents are expected to help in building the global plan. The primary goal is the solution of the given main problem. In contrast to centralized planning, in cooperate planning both the problem data and the development of the plan are distributed across several planning components (agents).

Cooperation is the central aspect in distributed artificial intelligence applications. The benefits of distributed problem solving can be capitalized on only through cooperation. The agents are independent of each other in their decisions, but only the cooperation enables an ensemble to achieve the common goal. Cooperation encompasses communication (transfer of information) and synchronization (temporal ordering of actions).

Many cooperation models have been developed in the area of parallel and distributed programming [3] and in the area of distributed artificial intelligence [9, 14]. In distributed artificial intelligence, the blackboard model is often employed [15]. With the blackboard model, the problem-solving data are kept in a global store, the *blackboard*. Agents produce changes to the blackboard, which lead incrementally to a solution to the problem. Communication and synchronization among the agents take place solely through the blackboard.

In this paper, we employ ProSet-Linda's model of coordination with tuple-spaces (see Section 5 for a brief description) to implement a multi-agent system. ProSet-Linda is designed for prototyping parallel algorithms. Tuple spaces have a lot in common with blackboards. Both models provide a shared data space to the cooperating processes, however, the operations to access the shared data space are quite different.

3 An Example Application: Scotland Yard

We discuss the development of cooperative planning algorithms for independent agents by means of an example application in the present paper: the strategic game "Scotland Yard" [21]. In this game, several detectives (agents) have to capture the mysterious villain Mister-X, encircling him on a map of the City of London. The detectives and Mister-X are initially

Neumann, Dirk Niemann, Stefan Schüler, Horst Sdun, and Knuth Waltenberg to realize the presented application program.

positioned at randomly selected transportation stops on the map (for taxi, bus, subway, or ferry resp.). In every step, each detective moves to another station which is connected with his current station by an appropriate vehicle. The detectives have a limited number of tickets for the corresponding means of transportation. In contrast to the detectives, who move visibly, Mister-X just announces the means of transportation he has taken. In regular intervals, Mister-X has to appear on his current station. He disappears with his next move. Every round involves the move of Mister-X together with the moves of each detective.

The detectives are allowed to exchange their plans and ideas. Therefore, this application is well-suited for cooperative planning. No "master" determines the moves of the individual detectives, the decisions are rather to be arrived at by cooperation and negotiation. Hence, before moving, the detectives coordinate their actions by exchanging ideas. Based on the appearance of Mister-X and the knowledge about the tickets he used since his last appearance, the detectives narrow down possible locations for Mister-X and try to catch him.

4 Strategies for the Agents of the Application

The common goal of the detectives is to capture Mister-X. The detectives are autonomous agents, are able to access the same knowledge, and are expected to behave constructively to achieve the common goal. The knowledge they can access to plan their moves are the rules of the game, informations about the locations and available tickets of all detectives, the possible locations of Mister-X, and distances between locations. Planning is the process of selecting a suitable way of proceeding for solving the problems. A strategy is an algorithm used by an agent to develop his own plan.

One strategy, which is based on the ideas presented in [4], tries to minimize the distance between Mister-X and the detectives. Because of the uncertainty with respect to the current location of Mister-X — remember that Mister-X appears only in intervals — the detectives have to take all possible locations of Mister-X into account. If a detective gets close to all possible locations with his next move, this move is assigned a high score. Technically, this works as follows: Mister-X has several possibilities for a current location; the lengths of the shortest paths between a detective's target position and all these locations are summed up. This yields a score of a particular target position, and the position with the lowest score is selected as the next move. For comparability, the scores for the moves selected are fitted then into one uniform scale (of course, other functions than summing the lengths of the shortest paths are possible, e.g. the maximum could be taken). This is only one possible strategy. In Section 7, the investigation of various alternative strategies by means of the evaluation of executable prototypes is discussed.

The moves of the detectives have to be coordinated when conflicts arise or when the total effort should be optimized. Conflicts arise when two or more detectives want to move to the same location. Therefore, each detective computes a set of moves, each move is scored. If two detectives want to move to the same location with their best moves, i.e. their highest scored moves, the scores will determine, which detectives can execute his move and which detective has to select another move. The latter detective is the detective whose loss is smaller when he cannot execute his highest scored move. This loss is the difference between his best and his second best scored move.

5 Prototyping Parallel Algorithms with ProSet-Linda

Before presenting the implementation of our example application, we have a look at ProSet-Linda as the language used for implementation. The procedural, set-oriented language ProSet [8] is a successor to SETL [17]. ProSet is an acronym for PROtotyping with SETs. The high-level structures that SETL and ProSet provide qualify these languages for prototyping [7, 17]. Linda and the sequential kernel of ProSet both provide tuples; thus, it is quite natural to combine both models to form a tool for prototyping parallel algorithms [13].

5.1 Basic Concepts

ProSet provides the data types atom, integer, real, string, Boolean, tuple, set, function, module, and instance. Modules may be instantiated to obtain module instances. It is a *higher-order* language, because functions and modules have first-class rights. ProSet is weakly typed, i.e., the type of an object is in general not known at compile time. Tuples and sets are compound data structures, the components of which may have different types. Sets are unordered collections while tuples are ordered. There is also the undefined value `om` which indicates undefined situations.

As an example consider the expression `[123, "abc", true, {1.4, 1.5}]` which creates a tuple consisting of an integer, a string, a Boolean, and a set of two reals. This is an example of what is called a *tuple former*. As another example consider the set forming expression `{2*x: x in [1..10] | x>5}` which yields the set `{12, 14, 16, 18, 20}`. Sets consisting only of tuples of length two are called maps. There is no genuine data type for maps, because set theory suggests handling them this way.

The control structures show that the language has ALGOL as one of its ancestors. There are `if`, `case`, `loop`, `while`, and `until` statements as usual, and the `for` and `whilefound` loops which are custom tailored for iteration over compound data structures. The quantifiers ($\exists$, $\forall$) of predicate calculus are provided.

5.2 Parallel Programming

Parallel programming is conceptually harder to undertake and to understand than sequential programming, because a programmer often has to focus on more than one process at a time. Consequently, developing parallel algorithms is in general considered an awkward undertaking. The goal of the ProSet-Linda approach is to partially overcome this problem by providing a tool for prototyping parallel algorithms [12]. To support prototyping parallel algorithms, a prototyping language should provide simple and powerful facilities for dynamic creation and coordination of parallel processes.

In ProSet, the concept for process creation via Multilisp's futures [11] is adapted to set-oriented programming and combined with the coordination language Linda [10] to obtain the parallel programming language ProSet-Linda. Linda is a coordination language which provides means for synchronization and communication through so-called tuple spaces. The access unit in tuple spaces is the tuple, similar to tuples in ProSet. A tuple space may contain any number of copies of the same tuple: it is a multiset, not a set. Process communication and synchronization in Linda is called *generative communication*, because tuples are added to, removed from, and read from tuple space concurrently. Synchronization is done implicitly. Reading access to tuples in tuple space is associative and not based on physical addresses, but rather on their expected content described in *templates*. This method is similar to the selection of entries from a data base. Refer to [5] for a full account to programming with Linda. ProSet supports multiple tuple spaces. Several library functions are provided for handling multiple tuple spaces dynamically.

ProSet provides three tuple-space operations: `deposit`, `fetch` and `meet`. The `deposit` operation deposits a tuple into a tuple space. The `fetch` operation tries to fetch and remove a tuple from a tuple space. *Templates* are specified to *match* tuples in a tuple space (associative access). The `fetch` operation blocks until a matching tuple is available (implicit synchronization). The selected tuple is removed from tuple space. The `meet` operation is the same as `fetch`, but the tuple is not removed and may be changed. Changing tuples is done by specifying values into which specific tuple fields will be changed. Tuples which are met in tuple space may be regarded as shared data since they remain in tuple space irrespective of changing them or not. For a detailed discussion of prototyping parallel algorithms in set-oriented languages refer to [13].

6 Design and Implementation of the Application

An important element to be realized in our implementation of the Scotland Yard game is a program structure being supportive of coordinating the program components. These components are a *graphical user interface*, a *rule component*, and finally a *planning component*. The graphical user interface displays the board and handles the communication with the player. The rule component manages the board, supervises the correctness of the moves, and executes the moves. The planning component is realized by autonomous detectives.

The rule and the planning components are implemented in ProSet. The graphical user interface has been realized with `Tcl/Tk`, a public-domain system for developing graphical user interfaces [16]. The main window of the graphical user interface is presented in Figure 1. The user interface displays the map of London with the current locations of the detectives and the last known location of Mister-X. Additionally, interesting information about Mister-X and the detectives is presented below the map (remaining tickets etc). Before the game starts, the user may configure the game, e.g. the number of detectives participating in the game and their start positions. The player determines the individual strategy a detective works with. This supports evaluation of the individual strategies.

The user interface is an independent Unix process. The communication between the user interface und the rule component is realized by *inter process communication (IPC)* on the

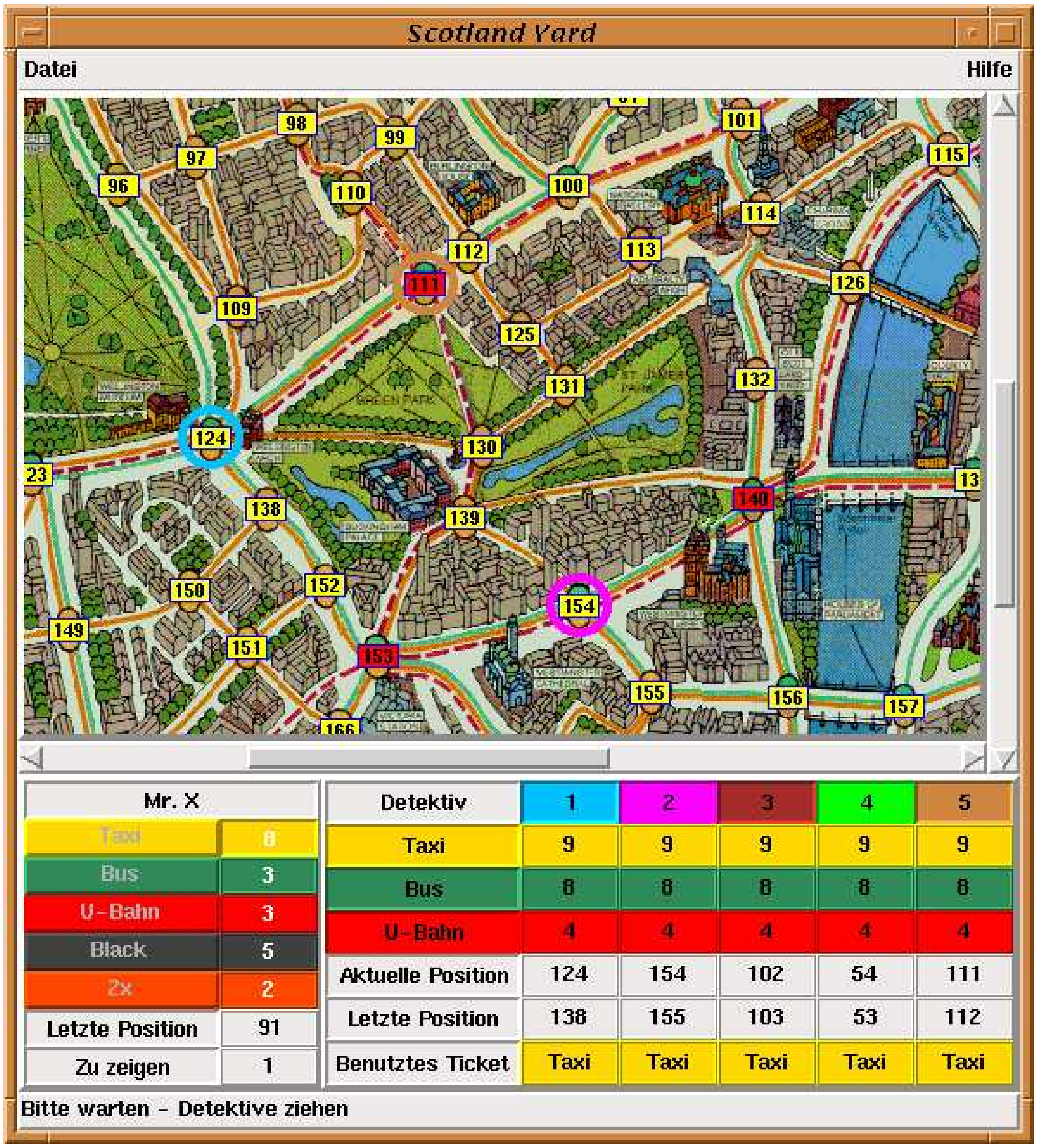

Figure 1: The main window. Example position after the second move of Mister-X.
The language displayed is German. Some Translations: U-Bahn *means* subway, Aktuelle Position *means* actual position, Letzte Position *means* last position, Benutztes Ticket *means* used ticket, zu zeigen *means* to show. Bitte warten — Detektive ziehen *means* please wait — detectives are moving. Mister-X can use up to five *black tickets* without indicating the used means of transportation, and he is allowed to make a double move twice in a game.
Note that on this page only a part of the map of the City of London is displayed (see the scrollbars). The original user interface is a color graphic.

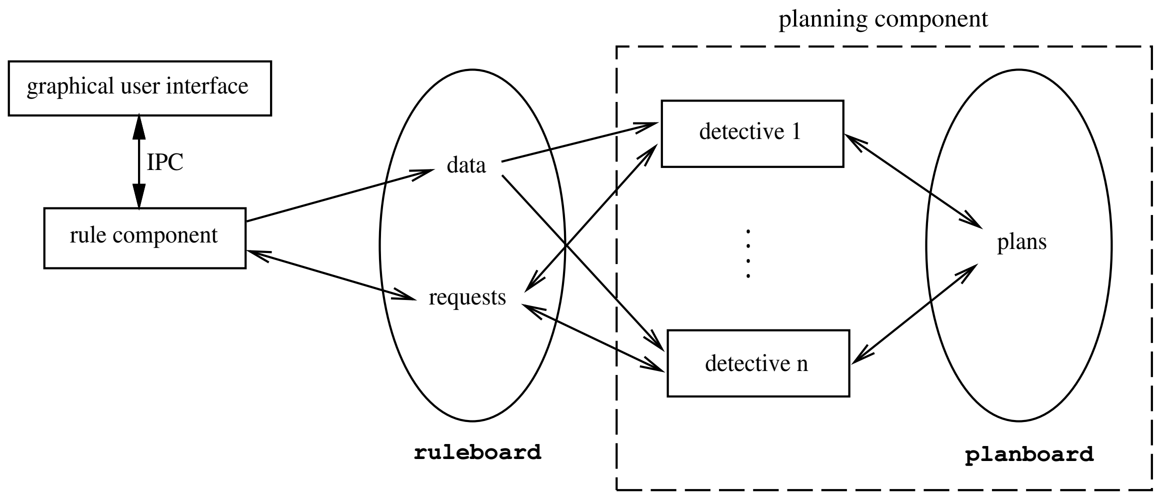

Figure 2: Tuple spaces used for coordination. Simple arrows indicate associative read access to tuple spaces, double arrows indicate read and write access.

Unix-level. The necessary functions are available through a C-language library, and are called from ProSet-programs through a C-language interface. We do not discuss the details of the user interface here, but rather concentrate on the two other components.

6.1 The Program Structure

The main program, consisting of the rule and the planning component, launches the user interface process, and loads a board. Then, the game is initialized by sending standard settings to the user interface and receiving the final ones, possibly changed by the user. The blackboards, which are used for the detectives' communication with the rule component and among each other, are initialized. Blackboards are implemented as tuple spaces. Each detective is started as an independent ProSet process. Having finished the preparation, the game can be played. This is essentially a loop in the rule component receiving Mister-X's move, updating the blackboards, answering questions from the detectives, and executing their plans. The game is finished by perceiving and announcing the winner.

Two separate tuple spaces are used for communication (see Figure 2). Planning among the detectives is done with a tuple space called `planboard`. The interaction with the rule component is realized through a tuple space called `ruleboard`. This includes updating data concerning the game's state by the rule component as well as receiving queries and giving answers between the detectives and the rule component.

6.2 Data Structures for Communication

The rule component has to provide data concerning Mister-X, the tickets used, the location of his last appearance, and his possible moves based on the last known position (among other things). A detective may obtain information depending on his current location, e.g.

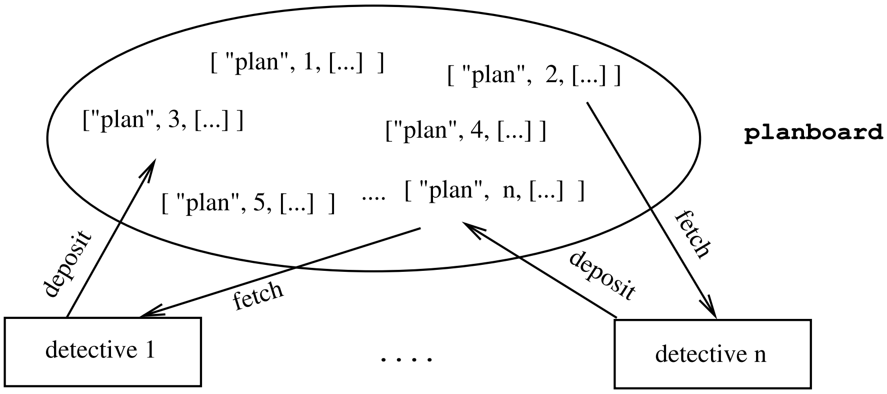

Figure 3: Communication of plans between the detectives on the `planboard`.

the correctness of a planned move, the possible moves from a certain position or the shortest path between two locations, by asking questions to the rule component. After each move of Mister-X, the data concerning him is updated in tuple space `ruleboard` by the rule component.

Let us have a look at the data structures used to model the board, the moves and the plans. The map of London is represented by a quadruple which indicates reachability by a particular means of transportation:

```
LondonMap = [ ["taxi",       { [109, {96,97,110,124}], ...   } ],
              ["bus" ,       { [130, {124,114,139}], ...     } ],
              ["subway",     { [153, {111,140,185,164}], ... } ],
              ["ferry",      { [157, {115,194}], ...         } ] ]
```

Formally, `LondonMap(1)` has the two components:

```
LondonMap(1)(1) = "taxi",
LondonMap(1)(2) = { [109, {96,97,110,124}], ...  }
```

The latter set is technically a map assigning locations to sets of locations, e.g. `109` is mapped to the set {`96,97,110,124`}. The domain of this map is the set of all stations on the map of London from which an agent may take a taxi. For later use we remark that an application of the built-in operator `domain` yields the domain of a map.

Tuples are used for communication via tuple spaces. A plan developed by a detective is a tuple in tuple space `planboard`:

```
["plan", det-nr, [[move, value], ..., [move, value]] ]
```

which contains a sorted scored list of possible moves (higher scored moves first). Plans are fetched from the tuple space `planboard` by an associative access with `fetch` operations. Figure 3 displays the communication of plans between the detectives in tuple space

```
procedure possible_moves(StartPos);
begin
   PosMoves := {}; -- initialize the result set
   for MeansOfTransportation in LondonMap do
      Connections := MeansOfTransportation(2);
      if StartPos in domain(Connections) then
         PosMoves +:= {[x,MeansOfTransportation(1)]: x in Connections(StartPos)};
      end if;
   end for;
   return PosMoves;
end possible_moves;
```

Figure 4: The procedure `possible_moves`.

`planboard`. The strategies to achieve consensus and resolve conflicts are explained in Section 6.4.

An example for ProSet's concise formulation (hence the ease of use for experimenting with algorithms) is the procedure to compute the possible moves from a specific location. It is displayed in Figure 4. `MeansOfTransportation` is a tuple the first component of which denotes a vehicle (as a string). The `for` loop ranges over `LondonMap`, taking each component as a value in turn, it assigns a map to `Connections`, mapping locations to sets of locations reachable from the current location, as indicated above. The conditional statement selects all stations reachable from the given location `StartPos` with the actual means of transportation. `MeansOfTransportation(1)` selects the first component from the tuple `MeansOfTransportation`. `Connections(StartPos)` selects the image of `StartPos` in the map `Connections`. The procedure returns a set of pairs, mapping possible target stations to corresponding vehicles.

6.3 Communication between the Rule Component and the Detectives

Communication between the rule component and the detectives is realized in tuple space `ruleboard`. Information about Mister-X is maintained by the rule component. For example, the list of tickets used by Mister-X is updated by the rule component with changing `meet` operations when Mister-X has moved. This information is accessed by the detectives by corresponding reading `meet` operations. This is the principle of providing data by the rule component to the detectives. Additionally, the `ruleboard` is used for answering the detectives' questions in a *client/server*-model. Each detective imports a module of functions to pose questions and receive available answers. For example, a detective can ask for the shortest path between his current position and a target position. The detectives do not know the connections on the map. This allows for easily experimenting with different maps.

6.4 The Detectives

All detectives follow a common control structure independent of the used strategy. After the initialization, each detective works in a loop. First, he has to read the ticket last used and the position last known of Mister-X. In an inner loop he develops an own plan, deposits it for the other detectives, resolves conflicts and tries to optimize the set of plans in cooperation with the other detectives. When all conflicts have been resolved, the moves are made known to the rule component and then executed. Conflict resolution is discussed below.

Initialization of the Detectives To support prototyping of strategies, the algorithms implementing the detectives should be easy to change. We use ProSet's module concept to realize this requirement. All strategies to be investigated are made available through individual modules. These modules import the above mentioned service module containing the question-and-answer procedures and routines to post moves, resolve conflicts and read data from `ruleboard`. Each module has the same export interface, e.g. the procedure `Calculate_Move`. Modules are made available in ProSet through *instantiation*. The resulting value is of type `instance` and has first class civil rights, in particular it may be assigned to a variable. This has the important consequence that it is possible to dynamically select the strategy to be used. All strategies can be used together in one game. Detectives using different strategies are able to cooperate.

Conflict Solving A conflict occurs when two or more detectives try to move to the same location. Each detective tries to coordinate his plans with the other plans. He reads the other plans by fetching them from tuple space `planboard`. If there is a conflict, it is resolved according to the strategy indicated in Section 4. The highest scored moves are compared. The lower valuated one is deleted from the corresponding plan. The detective writes the modified plan back to tuple space `planboard` with a `deposit` operation. The process of solving conflicts terminates, because in each conflict-solving step a reduction of the number of moves in a plan takes place. If no more conflicts arise, i.e., if the set of plans becomes stable, planning is finished. It is possible that individual detectives are not able to make a move.

7 Strategies and their Evaluation

Our goal is to implement and evaluate executable prototypes of cooperative planning algorithms to exploratively analyze the requirements on such algorithms. Several strategies have been implemented and evaluated. Examples are a simple randomized version for initial testing, the *minimal-distance* strategy tries to minimize the shortest paths from each detective to the possible positions of Mister-X, the *distance-sum* strategy tries to minimize the sum over all such paths, and a variation of the strategy presented in [4] which has been discussed in Section 4. In another strategy, which we call *mixture*, each detective selects for every move randomly among the latter three strategies (excluding the random strategy).

The high level of ProSet-Linda allowed us to easily experiment with different algorithmic variations.

Additional strategies varying the planning depth or the procedure of scoring the moves have been implemented. Others integrate elements of static valuations of positions into their planning algorithms. One strategy which considers not only the position of Mister-X but also the positions of the neighboring detectives to obtain a good distribution for avoiding conflicts has been proven to be most successful in the experimental evaluation at our department. A detailed discussion of the implemented and evaluated strategies is beyond the scope of this paper. This paper emphasizes on the presentation of the employed development technique.

As an aside, we mention that some players studied the individual strategies of the detectives to base their movements on this knowledge to obtain better positions. Against such players, the *mixture* strategy is the most successful, because the player cannot rely on deterministic movements by the detectives.

Also, the graphical user interface changed during the evaluation according to the user's requests. We do not discuss this in detail here, because this paper is concerned with prototyping of parallel algorithms and not with prototyping of user interfaces.

Several extensions come to mind. First, one may want to experiment with more sophisticated strategies for plan generation and for conflict resolution; this could be done by introducing other weight functions for scoring individual positions, and by considering a deeper look-ahead, hereby exploring different methods for tree pruning. Refer to [22] for a detailed discussion of techniques for planning of independent agents. Second, the question of scalability arises: all works well for only a handful of agents with London's public transportation, but suppose the villain works on a global level and is being chased by armies of agents. Then certainly some local arguments in arriving at decisions have to be introduced. We feel that the principle of information hiding and distribution through multiple tuple spaces, which is made use of the tuple spaces `ruleboard` resp. `planboard` in the present paper may be a suitable way to go. The prototyping approach of ProSet-Linda allows to easily experiment with such extensions.

A further extension deals with scalability with respect to the agents' type: we still let the agents be independent of each other, but several types of agents are possible. Going a step further, one might wish to cluster information by giving agents only partial information about the state of information for the other agents. This could be modelled by different tuple spaces (as the source of information for the agents), and by selective exchange between them. Currently, all the agents are sharing all their plans what leads to an expensive communication. It would be interesting to investigate agents sharing only their plans with the agents which are close to them, and only those plans which have some effects on their plan.

8 Related Work

There has been particular attention on multi-agent systems in the last years [19]. The agents are often programmed with the object-based *actor model* [1]. An actor is an object which responds to messages: actors communicate by passing messages to each other. For instance, the model of Agent-Oriented Programming (AOP) [18] is based on the actor model. Agents are described with AOP in a logic which describes the *mental state* of agents and how the states change as a result of interactions with its environment. A mental state describes *beliefs* (logical statements) and *capabilities* (actions the agent is able to perform) among other things. Agents communicate by passing messages. AGENT-0 is a language based on the AOP model [18].

Agents which are collaborating on a problem will ordinarily need to share data, but in the message-passing model data structures are encapsulated within agents, so agents cannot access the others data directly. Instead they exchange messages. When one agent has data for another one, it sends a message to the agent. This scheme adds complexity to the system as a whole: it means that each agent must know how to generate messages and where to send them. Refer to [2] for an extensive discussion of the shortcomings of the message-passing model in parallel and distributed programming. These problems arise accordingly when modelling the interaction of actors with message passing. The *contract net* approach for distributed problem solving [20] overcomes some of the problems with point-to-point message passing, because it additionally supports broadcasting of messages (tasks) to sets of agents.

Parallel applications which use the shared memory model for coordination are usually significantly smaller and easier to understand than equivalent programs that use message passing. In particular, message passing is less suitable when several agents need to coordinate indirectly by sharing global state information.

Note, however, that blackboard architectures are also proposed to coordinate multiple agents [6, 14, 15]. As opposed to most blackboard systems, coordination in ProSet-Linda is carried out through atomic addition, removal, reading, and updates of tuples in tuple space. Synchronization is done implicitly. In accordance with blackboard systems, tuple-space communication allows a high degree of decoupling between the cooperating agents. This decoupling alleviates the development of parallel algorithms considerably. Multiple tuple spaces allow to structure the shared memory.

It is apparent that multi agent systems can profit considerably by the experiences made with parallel programming languages during the last decades since the underlying demands for coordination of parallel activities are fairly similar: shared memory models should be preferred for coordination of agents.

9 Conclusions and Future Work

We presented a case study for the development of algorithms for cooperation strategies of independent agents with ProSet-Linda. The investigated strategies for planning are only sketched in the present paper. The emphasis is on the prototypical development and evaluation of planning algorithms for independent agents.

The evaluation showed that not all algorithmic variations for the planning strategies are good candidates for further (more efficient) implementations with some lower-level language. If we had implemented *all* the algorithms directly with a production language, for example C with extensions for message passing, the implementation effort would have been higher. This is what prototyping is about: experimenting with ideas for algorithms and evaluating them to make the right decisions for the next steps in the development. Purely theoretic evaluations are often not possible in practice. However, the exact savings in time cannot be presented: This would require a similar project without prototyping for comparison.

In the current implementation, the user can only adopt the role of Mister-X. We consider to extend the application to allow users to take over the roles of individual detectives. This way, multiple users could play together. For such an extension we would need a multi-user interface (running on multiple workstations). Each user would have a graphical interface similar to the existing single-user interface presented in Section 6. The individual interfaces would have to translate the user actions into corresponding tuple-space operations. This would allow the human agents to cooperate with the artificial agents. An important question is *how* to present the contents of the tuple spaces (the plans of the other detectives) to the human agents. The plans of the other detectives could be visualized on the map. Labeled arcs could indicate the intention to move to certain stations. Such a multi-user program could serve as a basis to explore the problem solving capabilities of the human-machine couple to analyze whether the investigated strategies for planning and conflict resolution are also applicable to the cooperation between artificial and human agents.

References

[1] G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 37–53. MIT Press, Cambridge, MA, 1987.

[2] H.E. Bal. *Programming Distributed Systems*. Silicon Press, 1990.

[3] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[4] W. Becker and A. Zell. Cooperative planing of independent agents. Technical Report 5/91, University of Stuttgart, IPVR, Stuttgart, Germany, 1991.

[5] N. Carriero and D. Gelernter. *How to write parallel programs*. MIT Press, 1990.

[6] V. Chevrier. GTMAS: a tool for prototyping and assessing design choices in multi-agent systems. In *Proc. Actes Quatorzièmes Journées Internationales d'Avignon (IA '94)*, pages 161–170, Avignon, France, June 1994.

[7] E.-E. Doberkat and D. Fox. *Software Prototyping mit SETL*. Leitfäden und Monographien der Informatik. Teubner-Verlag, 1989.

[8] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, and C. Pahl. ProSet — A Language for Prototyping with Sets. In N. Kanopoulos, editor, *Proc. Third International Workshop on Rapid System Prototyping*, pages 235–248, Research Triangle Park, NC, June 1992. IEEE Computer Society Press.

[9] L. Gasser and M.N. Huhns. *Distributed artificial intelligence, Volume II.* Pitman/Morgan Kaufmann, London, 1989.

[10] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[11] R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[12] W. Hasselbring. Prototyping parallel algorithms with ProSet-Linda. In J. Volkert, editor, *Parallel Computation (Proc. Second International ACPC Conference)*, volume 734 of *Lecture Notes in Computer Science*, pages 135–150, Gmunden, Austria, October 1993. Springer-Verlag.

[13] W. Hasselbring. *Prototyping Parallel Algorithms in a Set-Oriented Language.* Dissertation (University of Dortmund, Dept. Computer Science). Verlag Dr. Kovač, Hamburg, 1994.

[14] M.N. Huhns. *Distributed artificial intelligence.* Pitman/Morgan Kaufmann, London, 1987.

[15] V. Jagannathan, R. Dodhiawala, and L.S. Baum, editors. *Blackboard architectures and applications.* Perspectives in artificial intelligence. Academic Press, Boston, 1989.

[16] J.K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing, 1994.

[17] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets – An Introduction to SETL.* Springer-Verlag, 1986.

[18] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–90, 1993.

[19] M.P. Singh. *Multiagent Systems*, volume 799 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1994.

[20] R.G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, 29(12):1104–1113, December 1980.

[21] Ravensburger Spiele. Scotland Yard. Otto Maier Verlag, Ravensburg, Germany, 1983.

[22] F. von Martial. *Coordinating Plans of Autonomous Agents*, volume 610 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1992.