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 multiagent 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.

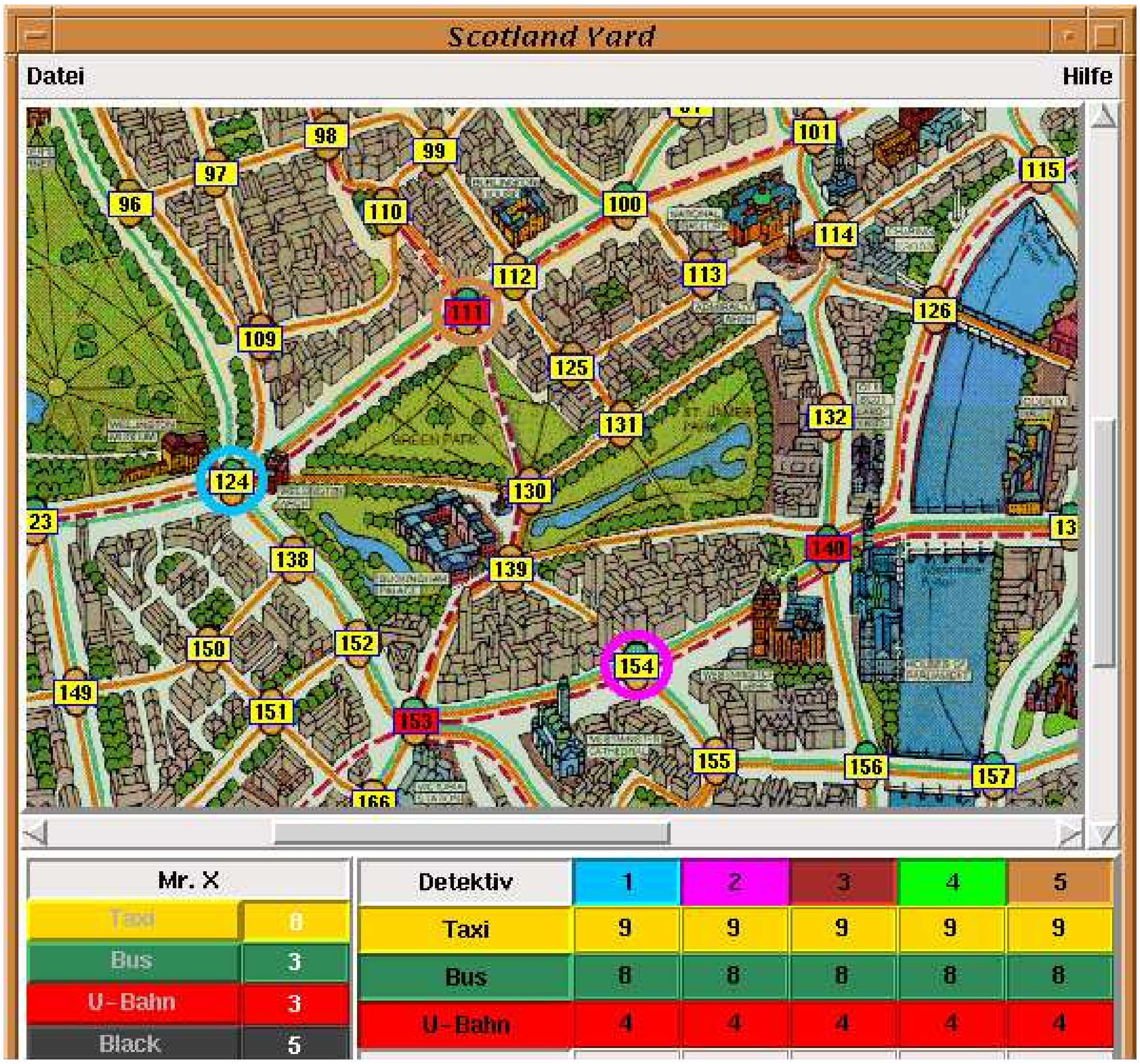
ProSet provides three tuple-space operations: `deposit`, `fetch` and `meet`. The `deposit` operation deposits a tuple into a tuple space. The `fetch` operation tries to fetch and remove a tuple from a tuple space. *Templates* are specified to *match* tuples in a tuple space (associative access). The `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



Mr. X	
Taxi	8
Bus	3
U-Bahn	3
Black	5

Detektiv	1	2	3	4	5
Taxi	9	9	9	9	9
Bus	8	8	8	8	8
U-Bahn	4	4	4	4	4