4 Human Problem Solving

In this chapter, we shall focus on discussing the application of EM to support the human problem solving that is arguably the most important activity in system development. In the first section, we shall discuss the importance of developing principles and techniques to promote computer support for human problem solving. In the second section, we shall explore the relationship between problem solving and programming. We shall introduce some relevant researches on general problem solving and the psychology of programming. We shall argue that conventional programming paradigms fail to give comprehensive support for problem solving. In section 4.3, we shall propose EM as a better approach for supporting problem solving, and illustrate this with the reference to solving a particular type of recreational puzzle. In section 4.4, we shall describe a case study based on a real life timetabling problem.

4.1 Human problem solving using computers

In the early development of computer science, much research focused on formalising the capability of computers to solve problems automatically. For example, Computability and Feasibility research (e.g. [Gar79]) focused on formalising what can and cannot be solved algorithmically by computers in theory and practice. Such research is commonly regarded as the foundation of computer science. University computer science courses traditionally start with teaching the mathematics and logic that underlie this formalised notion of computing.

However, this mainstream conception of computer science has recently been challenged by a number of researchers. The main argument is that the traditional formalist theories of computer problem solving do not take the environmental and human aspects into account (e.g. [Bey92b, Weg97, Wes97]). The agenda of computer science should not only be considering *computer problem solving* but the broader topic of *human problem solving using computers*. Notice that, in this shift of

perspective, the emphasis changes from the computer to the human. This implies a change of focus from research on abstract concerns such as proofs and reducibility to research on more concrete situated activities that reflect the dynamic nature of the world and human cognitive processes. Of course, we agree with West [Wes97] that, whilst we must acknowledge the usefulness of formalism, we also need to be more conscious of its limitations.

Human problem solving is described by Martinez [Mar98b] as "…the process of moving toward a goal when the path to that goal is uncertain". He further explains that "we solve problems every time we achieve something without having known beforehand how to do so". This definition of problem solving excludes the process of achieving goals with prescribed steps. For the purposes of our discussion, this is the most appropriate characterisation of human problem solving. Too often traditional computer science research focuses on knowledge representation and proofs associated with executing prescribed recipes that are far removed in spirit from human problem solving.

The question then becomes: what support can computers give humans in problem solving? This chapter will propose an answer based on EM principles. Edmonds et al. [Edm95] acknowledge that there is a tension between 'automating expertise' and 'amplifying human creativity' in computer-supported problem solving. Ideally, both aspects are equally important. To automate expertise, the human problem solver is provided with a set of predefined functionalities that are common in solving a particular class of problems. To amplify human creativity, the problem solver is provided with features for customising predefined functionalities or even for defining new functionalities. Too often research tends to stress the automation of expertise, rather than promoting human involvement.

Gallopoulos et al. use the term "problem-solving environment" (PSE) to describe a computer-based system that "provides all the computational facilities necessary to solve a target class of problems" [Gal94]. The facilities include predefined solution methods, automatic or semiautomatic selection of solution methods, and ways to add new solution methods. Many PSEs are widely used by industries already. Examples are SPSS for statistical analysis, LabView for electronic engineering, Matlab for science and engineering, Mathematica for symbolic manipulation and visualisation, spreadsheets for finance and word processors for publishing. PSEs are usually domain-specific. Their usefulness is bound to their intended domain with the arguable exceptions of spreadsheets and word processors.

By contrast, the purpose of this chapter is to discuss support for general human problem solving using the computer. It is more difficult to provide problem solving support for general computer users than for domain experts because we cannot presume a particular model of the intended users.

There seem to be several options for a general computer user to solve a problem using a computer. An obvious option is to use a software package that is specifically designed for solving the problem at hand. This is the best choice provided that the user can find what they are looking for, and provided that the problem is common enough for software to be already constructed by others. Unfortunately, this is not always the case in reality. Since the notion of problem is subjective and changes over time [Hoc90], it is hard to find a software package to suit for a given purpose. Even if the user can actually find one, the dynamic nature of a problem context may quickly render the software package unsuitable without introducing modifications. In addition, as has been discussed in [Ped97], traditional user interfaces of conventional software packages are usually so rigid that they hinder the ability of the user to approach a problem differently.

Apart from finding and using existing software packages, general computer users still have two other options: to use one of the conventional end-user programming paradigms such as the spreadsheet paradigm, or to construct a program using a conventional programming languages such as Java. However, we shall argue that even these two options are not satisfactory for supporting human problem solving using computers.

In the following section, we shall discuss the complexity of providing adequate support for problem solving using the computer. We shall explore the fundamental difficulties from both traditional spreadsheet and programming paradigms for supporting general human problem solving. The nature and relationship of programming and human problem solving is also explored with reference to some recent research in psychology of programming and problem solving.

4.2 Problem solving and programming

4.2.1 Programming as search in problem spaces

Programming can be viewed as *fulfilling a requirements specification* generated from requirement analysis as part of a conventional software development cycle. For the

purpose of our discussion, however, we shall take the broader view that programming is *a form of problem solving*.

Following Newell and Simon [New72], human problem solving can be thought of as navigation from an initial state to a desired goal state in a *problem space*. A problem space represents all possible states of knowledge of the problem solver related to the problem. Problem solving is viewed as a transformation from the initial state to the goal state using a set of cognitive operators. This idea of searching in a problem space has proved to be useful and widely accepted by many researchers. It has also been extended for describing various phenomena of human problem solving. For example, Simon and Lea [Sim74] characterise an induction task that involves developing general rules from particular instances as a search in two interrelated problem spaces: *rule* space and *instance* space (a 'dual-space search'). The problem solver generates possible rules from instances, and tests the rules with new instances; this in turn may invalidate the rules, and result in the generation of new rules. This process goes on and on until the desired rules are found. The concept of dual-space search was later adapted by Klahr and Dunbar [Kla88] in their explanation of scientific discovery, where scientists search in hypothesis (rule) space and experiment (instance) space.

In their research on scientific discovery, Schunn and Klahr [Sch95] suggest a 4-space search. This research was adapted by Kim et al. [Kim97a, Kim97b] for exploring the strategies used by programmers in problem solving. Kim et al. characterise programming as a search in 4-spaces: the *rule*, *instance*, *representation* and *paradigm* spaces. In programming terms, searching in rule space is related to writing program statements; searching in instance space is related to designing and conducting test cases for checking the correctness of the program; searching in representation space is related to maintaining a mental model of the programmer's understanding of the problem; and searching in paradigm space is related to selecting a suitable paradigm for navigating other spaces. Empirical studies with programmers conducted by Kim et al. revealed that the difficulty of programming can be attributed to programmers' intensive searches in representation and paradigm spaces.

Kim et al. has explained the difficulty of programming as a particular kind of problem solving activity. To discuss the challenges of providing computer support for human problem solving more fully, we also need a wider investigation into how people solve problems with and without the support of computer programming.

4.2.2 Difficulties of supporting problem solving by programming

Traditional programming paradigms are suitable for problem solving at the abstract level. They provide a high degree of expressiveness and generality. Programming language designs usually give much support to abstract thinking and may even enforce programmers to think abstractly (cf. 'abstraction hunger' described in [Gre98a]). For example, it is generally considered to be more useful to develop a program to sort an arbitrary number of names than to sort a fixed number of names in alphabetical order, even though the problem at hand is to sort just a given number of names.

Hoc and Nguyen-Xuan [Hoc90] point out that in normal problem solving situations the goal is specific, whereas in most programming situations the goal is usually generic. Learning programming therefore has two difficulties [Hoc90]. Firstly, there is "a shift from value to variable processing". In other words, there is shift from thinking in concrete terms to thinking in abstract terms, and from dealing with specific to generic problems. Secondly, beginner programmers have to consciously transform natural procedures to machine procedures, and search for representations that they are not aware of in normal problem solving situations. The difficulty of programming therefore can be viewed as stemming from a huge gap between natural languages and programming languages [Hon90, Mye98, Pan01].

By contrast, much end-user programming research promotes programming at a concrete level. Examples are programming by demonstration [Cyp93] and spreadsheet programming [Nar93]. This is illustrated by considering the traditional spreadsheet. The advantages of spreadsheet programming are as follows [Nar93, Geh96]:

- it provides task-specific programming primitives that are already in the user's problem domain. The primitives include a set of predefined functions that are commonly useful for the task at hand.
- it supports a trial and error style of programming. The underlying automatic dependency maintenance between cells allows the user to explore different settings quickly.
- it supports concrete display of data. The user does not have to worry about abstract data types.

All the above benefits make learning spreadsheet programming a lot easier than learning traditional programming. However, the major drawback of traditional spreadsheet programming is the lack of an abstraction mechanism. This makes reuse of a spreadsheet difficult. Some recent spreadsheet programming research are attempting to introduce abstraction mechanisms into spreadsheet programming. Examples are the structured spreadsheet Basset [Tuk96] and the visual programming language Forms/3 [Bur01]. However, these mechanisms are usually difficult to use. Figure 4.1 depicts problem solving at two different levels of abstraction. The conventional spreadsheet paradigm provides facilities for the user to solve a problem at a concrete level. It usually supports the user in considering a problem instance and a specific method to solve the problem. The goal is usually to obtain a specific solution to the problem. On the other hand, a traditional programming paradigm encourages the user to think abstractly. It supports the user in considering a problem class and in finding a generic method to solve them all. The goal is to obtain potential generic solution for all similar problems.

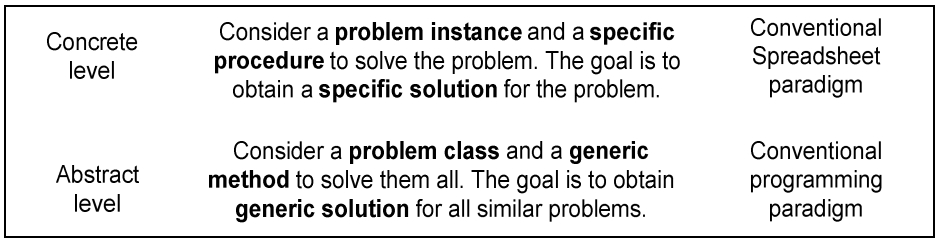

Figure 4.1: Programming for problem solving at two different levels of abstraction: concrete and abstract

However, the process of problem solving usually involves thinking in both concrete and abstract terms at the same time. Therefore, neither the spreadsheet nor a traditional programming paradigm gives comprehensive support for the natural demands of human problem solving. In the next section, building on pioneering research in problem solving [New72] and the psychology of programming [Hoc90], we shall investigate an EM approach to problem solving using the computer.

4.3 Empirical Modelling for problem solving

4.3.1 Construal of the problem solving situation (CPSS)

As we discussed in the last section, problem solving not only involves searching in multiple problem spaces but also thinking at different levels of abstraction. Figure 4.2

depicts our new understanding of human problem solving. For simplicity, the figure shows only two levels of abstraction: *concrete* and *abstract*. There are three problem spaces associated with each level. At the concrete level, they are the problem instance space, the method space and the specific solution space. At the abstract level, they are the problem class space, the generic method space and the generic solution space. Problem solving can be viewed as building up a construal of the problem solving situation as informed by the searching of all the problem spaces together with the problem solver's past knowledge and experience.

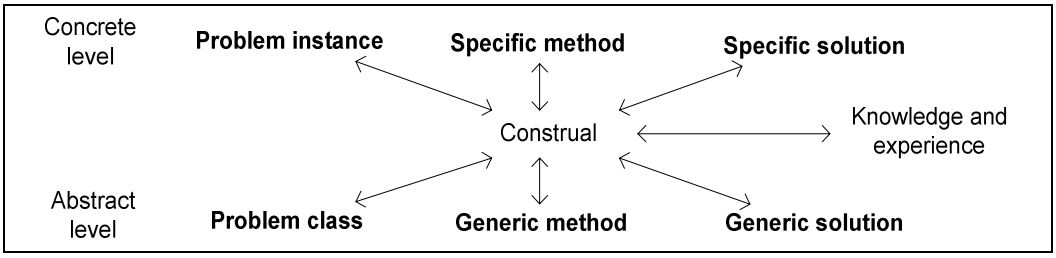

Figure 4.2: Problem solving as search in problem spaces at two different levels of abstraction

Martinez described human beings are "problem solvers who think and act within a grand complex of fuzzy and shifting goals and changing means to attain them" [Mar98b]. This implies that there is no predetermined order of search in the problem spaces. Usually problem solving starts off by a quick analysis of the problem. However, after the initial analysis, the problem solver can choose to search in any problem spaces according to their subjective judgment about what will make the best progress. A complete path from initially posing a problem to finding its solution involves arbitrary navigation between the problem spaces.

This new perspective motivates a shift from *supporting problem solving by programming* to *supporting problem solving by modelling*. With a conventional programming approach to problem solving, the role of the computer is mainly to provide means to specify and automate the procedural knowledge obtained by searching the method and generic method spaces. With a modelling approach to problem solving, the role of the computer is extended to provide means to represent the construal which consists not only of knowledge obtained from searching the method and generic method spaces but also the other problem spaces. By applying EM principles, we can construct an Interactive Situation Model (cf. chapter 2) of the problem solving situation. This model, which will be referred to as a Construal of the Problem Solving Situation (CPSS), embodies knowledge obtained by searching all the problem spaces. In a CPSS, knowledge is represented in terms of observation,

dependency and agency. Integrated development and use of a CPSS can give powerful support for problem solving. In particular, in order to obtain a comprehensive solution, the problem solver needs to be able to account for all his or her partial knowledge of potential methods and solutions and how these are related to context and previous experience. The CPSS supports this kind of problem solving activity.

We shall illustrate the use of a CPSS with reference to a class of simple recreational problems, called Crossnumbers. Figure 4.3 shows a Crossnumber problem. The task is to place the numbers given in the list into the blank cells of the given 5 by 5 grid. Crossnumbers are adopted as a case study in this section because they are a convenient vehicle for illustrating the general principles behind CPSSs even if in practice a human problem solver would be unlikely to consider semi-automatic approaches to their solution.

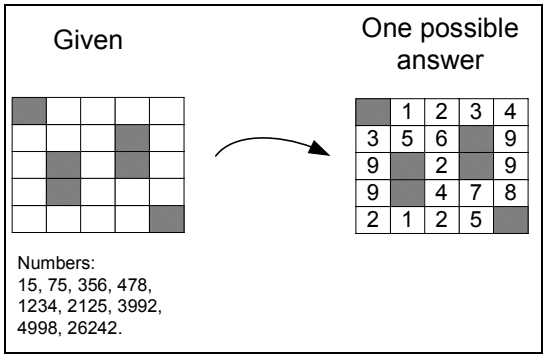

Figure 4.3: A Crossnumber problem with 5 x 5 grid (left) and its solution (right)

With reference to Figure 4.2, at the concrete level, the particular 5 by 5 Crossnumber problem is in problem instance space; any particular solution to solve the problem sits in the method space; whereas the solution shown on the right of the figure sits in the specific solution space. At the abstract level, the problem class space may contain all possible Crossnumber problems with grids of different sizes; search in generic method space represents development of generic program to solve all similar Crossnumber problems; and the generic solution space contains all possible solutions for similar Crossnumber problems.

Appendix D contains the complete listing of a CPSS for solving the Crossnumber problem. Solving problems like Crossnumbers involves free navigation of all spaces at different levels of abstraction. In putting the emphasis on semi-automatic human problem solving rather than fully automatic problem solving, we are motivated to consider well-recognised heuristics that can be used to guide

problem solving. We shall show how the construction of a CPSS gives support in applying these heuristics.

4.3.2 Supporting heuristic problem solving

The process of problem solving involves understanding the problem, exploring the methods, and designing solutions to test the understanding of the problem and methods. The process can be guided by heuristics. Martinez describes a heuristic as "a strategy that is powerful and general, but not absolutely guaranteed to work" [Mar98b]. There are two kinds of heuristic: heuristics that are specific to a particular problem situation; and heuristics that are generic enough to apply in any problem solving context. We shall consider some generic heuristics discussed in the literature [Pól45, Hew95, Mar98b] and see how the CPSS of Crossnumber supports their application. They are External Representation, Problem Reinterpretation, Suspending Evaluation, Ends-means Analysis, and Successive Approximation.

External Representation. External Representation relates to finding ways to represent what is in our mind externally. By using an external representation, the problem solver can lay out complex information that cannot be considered internally in our mind at once [Mar98b]. It also forces the problem solver to clarify his or her thinking. For example, Pólya [Pól45] encourages the solver to draw a figure and to introduce a suitable notation in order to understand mathematical problems before solving them. External representation is sharable with other people who might try to consider the problem and might offer help. Good external representation renders visible what is actually in the problem solver's mind. However, it is commonly agreed that it is not easy to achieve this in practice (e.g. [Hoc90, Pan01). The use of observables, dependencies, and agents can arguably help us to construct more natural external representations. By way of example, consider the external representation of the grid extracted from CPSS of the Crossnumber problem.

```
grid =  [['x',' ',' ',' ',' '],
         [' ',' ',' ','x',' '],
         [' ','x',' ','x',' '],
         [' ','x',' ',' ',' '],
         [' ',' ',' ',' ','x']
        ];

a1 is grid[1][1]; a2 is grid[1][2]; a3 is grid[1][3]; a4 is grid[1][4]; a5 is grid[1][5];
b1 is grid[2][1]; b2 is grid[2][2]; b3 is grid[2][3]; b4 is grid[2][4]; b5 is grid[2][5];
c1 is grid[3][1]; c2 is grid[3][2]; c3 is grid[3][3]; c4 is grid[3][4]; c5 is grid[3][5];
d1 is grid[4][1]; d2 is grid[4][2]; d3 is grid[4][3]; d4 is grid[4][4]; d5 is grid[4][5];
e1 is grid[5][1]; e2 is grid[5][2]; e3 is grid[5][3]; e4 is grid[5][4]; e5 is grid[5][5];
```

The two blocks of script above represent two different conceptualisations of the

grid in the problem solver's mind. The first block is a list of lists. The second block defines each cell of the grid as a separate observable. Which representation is closer to the problem solver's conception depends on context. But by using definitive script, the problem solver can have many different representations at the same time. The consistency between representations can be automatically maintained. For example, in this case, values of the second representation are always consistent with the corresponding values in the first representation by definitions.

Suspending Evaluation. It is sometimes useful to temporarily stop evaluating the usefulness of the observations we introduced. This heuristic facilitates the discovery of new methods to solve a problem. For example, all the following definitions were initially conceived simply by observation of the grid structure with no specific purpose or goal in mind.

```
blocks  is [a1,b4,c2,c4,d2,e5];
position1 is [a2,a3,a4,a5];
position2 is [b1,b2,b3];
position3 is [d3,d4,d5];
position4 is [e1,e2,e3,e4];
position5 is [b1,c1,d1,e1];
position6 is [a2,b2];
position7 is [a3,b3,c3,d3,e3];
position8 is [d4,e4];
position9 is [a5,b5,c5,d5];
```

In a definitive script, introducing extra observations does not affect the existing observations. Therefore, there is no harm in introducing observations which may never be used to solve the problem. Eventually, such apparently redundant observations may lead us to a reinterpretation of the problem, such as will now be discussed.

Problem Reinterpretation. The way in which a problem is interpreted may affect the difficulty of solving the problem [Hew95]. Therefore, when we have found the problem is not as easy as expected, it is useful to reinterpret the problem in other ways. This is usually accompanied by a change of problem representation. In the case of the Crossnumber problem, the problem is transformed when we choose to represent the 9 positions where the given 9 numbers are to be filled in as follows:

```
numbers is ["1234","2125","26242","3992","4998","356","478","15","75"];
position1 is [a2,a3,a4,a5];
position2 is [b1,b2,b3];
position3 is [d3,d4,d5];
position4 is [e1,e2,e3,e4];
position5 is [b1,c1,d1,e1];
position6 is [a2,b2];
position7 is [a3,b3,c3,d3,e3];
position8 is [d4,e4];
position9 is [a5,b5,c5,d5];
```

The problem becomes: assign digits to cells so that each of the observables from `position1` to `position9` contains one of the numbers in the observable `numbers`. After this reinterpretation of the problem, the criteria for the correctness of the solution can be specified as follows:

```
ok1 is containsString(numbers, digitsToString(position1));
ok2 is containsString(numbers, digitsToString(position2));
ok3 is containsString(numbers, digitsToString(position3));
ok4 is containsString(numbers, digitsToString(position4));
ok5 is containsString(numbers, digitsToString(position5));
ok6 is containsString(numbers, digitsToString(position6));
ok7 is containsString(numbers, digitsToString(position7));
ok8 is containsString(numbers, digitsToString(position8));
ok9 is containsString(numbers, digitsToString(position9));
solved is ok1 && ok2 && ok3 && ok4 && ok5 && ok6 && ok7 && ok8 && ok9;
```

The definitions `ok1` through `ok9` monitor the correctness of numbers filled in from `position1` to `position9` respectively. The overall correctness is defined by the definition `solved`. This can be regarded as a big step towards solving the problem because the goal is clearly represented by definitions. We can also monitor the status of any purported solution by querying the related definitions.

Ends-means Analysis. Ends-means Analysis involves “form[ing] a subgoal to reduce the discrepancy between your present state and your ultimate goal state” [Mar98b]. In other words, try to do something which seems to be making progress towards the ultimate goal state. This heuristic is useful when the problem is too complex for a solution to be found at once. In constructing the CPSS of Crossnumber, the introduction of the following definitions enables a semi-automatic way of solving the problem (‘achieving a subgoal’), which may also eventually lead to an automatic solution.

```
set1 is findNumberWithConstraints(numbers, digitsToString(position1));
set2 is findNumberWithConstraints(numbers, digitsToString(position2));
set3 is findNumberWithConstraints(numbers, digitsToString(position3));
set4 is findNumberWithConstraints(numbers, digitsToString(position4));
set5 is findNumberWithConstraints(numbers, digitsToString(position5));
set6 is findNumberWithConstraints(numbers, digitsToString(position6));
set7 is findNumberWithConstraints(numbers, digitsToString(position7));
set8 is findNumberWithConstraints(numbers, digitsToString(position8));
set9 is findNumberWithConstraints(numbers, digitsToString(position9));

stuck is set1#==0 || set2#==0 || set3#==0 || set4#==0 || set5#==0 || set6#==0 || set7#==0
|| set8#==0 || set9#==0;
```

The initial values of `set1` to `set9` are:

```
set1: ["1234","2125"]
set2: ["356","478"]
set3: ["356","478"]
```

```
set4: ["1234","1215","2992","4998"]
set5: ["1234","1215","2992","4998"]
set6: ["15","75"]
set7: ["26242"]
set8: ["15","75"]
set9: ["1234","1215","2992","4998"]
```

We can immediately observe that `set7` contains only one number `26242`. This means that only number `26242` can go into position 7. This is the number to be assigned into the grid first, which may further constrain the choices of numbers in other positions. One strategy may be to assign numbers to the positions where fewest choices are available. The observable `stuck` indicates when we should stop assigning numbers and backtrack to make an alternative choice.

Successive Approximation. Successive Approximation involves initially constructing a less than satisfactory solution and then making iterations to improve the solution until a satisfactory one is developed. The whole development and use of CPSS for Crossnumber problem can be viewed as Successive Approximation. Initially the support of CPSS is limited in terms of actually solving the problem. Over time, semi-automatic or even automatic methods for solving the problem may emerge from the CPSS.

4.3.3 Integrating development and use

Problem solving skills can be regarded as a form of tacit knowledge. Heuristics provide guidance for us in learning and applying these skills. Constructing a CPSS of a problem helps the problem solver to apply these skills. We have shown how the development and use of CPSS supports the application of a variety of generic problem solving heuristics. Being able to develop and use a CPSS at the same time is very important to its ability to help problem solving. There are three reasons. First, by integrating development and use, the problem solver can get immediate feedback on the progress in terms of the understanding of the problem, the reliability of proposed methods and the validity of the solution. The problem solver can directly take this information into account to improve the usefulness of CPSS. Second, the problem solver is more able to deal with the changes made to the requirement of the problem by constantly reviewing the situation of the problem. Finally, the separation of development and use is artificial. The interaction between development and use is the key to problem solving. CPSS guides a more natural way of human problem solving. CPSS is a medium to support thinking.

4.4 Case study: a timetabling problem

In this section, we describe an application of EM principles and tools to solve a real life timetabling problem. The problem involves scheduling the time and location of undergraduate final year project oral presentations in the Computer Science Department at Warwick University. A CPSS called the Temposcope was built to support the timetabling process. The name 'Temposcope' stands for 'An instrument for Timetabling with Empirical Modelling for Project Orals' [Bey00b]. A departmental administrator has been successfully using the Temposcope over the last two years as a support for solving the timetabling problem. This case study is a practical demonstration of the concepts of EM for problem solving discussed so far in previous sections of this chapter.

4.4.1 Timetabling problem for project oral presentations

The initial situations of this timetabling problem are briefly described as follows. There are about 125 students who are in the final year of two similar undergraduate courses, Computer Science (CS) and Computer and Business Studies (CBS). One of the requirements for their final year project is to have a project oral in the last week of the first semester. Each student's supervisor and an assessor from the members of staff are expected to attend the oral. The oral week lasts from Monday to Friday, 9 am to 6 pm. Each oral is allocated a 40-minute timetable slot. Each day has 13 timetable slots available. Each oral is held in one of up to 5 departmental rooms that are available. The Temposcope is used to support timetabling process which was previously being done by hand with pen and paper, and took about a whole week. The problem involves assigning projects to slots while taking account of the availability of staff, students and rooms.

Solving timetabling problems on this scale is usually non-trivial for either a human timetabler or a computer program. A huge number of choices and constraints make it difficult for a person to take all factors into account at once. Determining whether there is a feasible solution of a timetabling problem is a well-known NP-complete problem [Gar79]. Conventional techniques for computer-supported timetabling usually involve extensive searches of a huge solution space that involves evaluating an assigned weight or penalty to each decision made according to constraints [Cor94]. To construct the timetable reasonably efficiently, to achieve a good quality result and maintain flexibility, a high degree of co-operation between

human and computer is required. Research on computer-based support for solving timetabling problems are no longer solely about finding optimum computer algorithms but more about supporting human problem solving as a process that consists of a number of activities such as data capturing, data modelling, data matching, report generation and storage of timetabling results [Opt00, Sch00b].

EM principles support both the development and use of the Temposcope but do not impose the order of them. This makes the Temposcope highly adaptable to change of situations. By integrating development and use, the quality of the Temposcope is constantly improving. As the timetabler's knowledge and experience increases, better strategies may be found to produce quality timetables in a shorter period of time. In the next subsection, we shall describe how the development and use of the Temposcope helps to solve the problem.

4.4.2 Integrated development and use of the Temposcope

Construction of the timetabling EM model, Temposcope, started off from a state-based analysis of the problem in terms of observable, dependency and agency. It is an informal analysis of the particular problem at hand as perceived by the problem solver. This includes identifying key observables of the problem and finding a representation for them. For example, the basic observation might be: there are 5 rooms, 2 staff with their availability, and 2 students with their final projects. They can be represented by the following definitions:

```
room = ["104", "327", "313", "LL1", "444"];
WMB_AV = [1,2,3,7,8];
SBR_AV = [7,8,9,10,11,12];
data1 = ["Al-Khaburi", "Ali", "How secure is a secure website? ", "CBS", "DA", "AB",
"SBR"];
data2 = [" Andand", "Aradhana", "Impact & utilisation of the internet in Indian companies",
"CBS", "YM", "MCK", "AB"];
```

Choosing the right representation is very important to problem solving (as mentioned in previous section). Representing a problem in one way might make the problem easier to solve than representing in another way. One important feature of representing a problem using definitions is that the problem solver has freedom to specify different representations within the model. Different representations can exist together in the model without introducing problems of inconsistency. This is because dependencies between the data and representation are automatically maintained. For example, there are two different styles to represent staff availability in the Temposcope:

```
WMB_AV is [1,2,3,7,8];
WMB_AVBinary is makeBinary(WMB_AV); // the value would be [1,1,1,0,0,0,1,1]
```

The above two definitions demonstrate the co-existence of different representations of the same observable. The consistency between two representations of the availability of a particular staff WMB is maintained.

Also note that the observables introduced so far are at the concrete level specific to the current situation of the problem. For example, `WMB_AV`, `SBR_AV`, `DA_AV` are observations of the availability specific to three staff. This may be considered to be poor style for conventional programming paradigms, where the emphasis is on generality and abstract variables are used. However, the ability to specify observables at the concrete level (or 'concrete variables') in the Temposcope empowers the problem solver to be more engaged with the situation of the specific problem. The flexibility of representation also has a profound effect on mediating the interaction between human and computer. One example is the definition:

```
staffAV is [WMB_AV, SBR_AV, DA_AV…];
```

This definition relates to observation at a more abstract level than the previous ones. `staffAV` represents a list of availability for all staff. This supports different kinds of agency: facilitating the computer to iterate through the availabilities of staff, whilst also supplying a specific definition of the availability of each staff member that is more understandable by the human problem solver.

In the Temposcope, dependencies are specified:

```
// example definition showing explicit dependencies here!
class is makeclass(data);
ataff is makestafflist(data, [5,6,7]);
AVSTAFF is makeAVSTAFF(staff, avail);
avx is map(proj2_1, [avstud], class);
// definitions showing some joint observations here:
DJKTJA_AV is union(DJK_AV, TJA_AV);
```

When compared with a conventional programming paradigm, where dependencies are implicitly scattered around the procedural code, explicit dependency specification helps to make the model more flexible to change and comprehensible in use.

The definitions introduced so far are all intrinsic to the problem – they are not associated with any specific method for solving the timetabling problem. The emphasis on state-based rather than behavioural-based analysis enables the problem

solver to take richer observables into account with deeper insight into the dependencies amongst them. This helps the problem solver to gain more insight into the problem, with which eventually strategies or even efficient algorithms may emerge. However, even without further extending the model, the problem solver can already use the model to support decision making in timetabling. For example, the joint availability of staff can be easily obtained. Subsequent enhancement of the model will make the model more useful, but even at present it is already usable (cf. the Successive Approximation heuristic).

Recall that one of the heuristics we have discussed is External Representation. The script itself, as an interactive textual artefact, can be regarded as a form of external representation of our understanding of the problem. In addition, the Temposcope incorporates visual representations to help the problem solver to make decisions in the process of development and use. Figure 4.4 is a screen capture of the visualisation of the Temposcope.

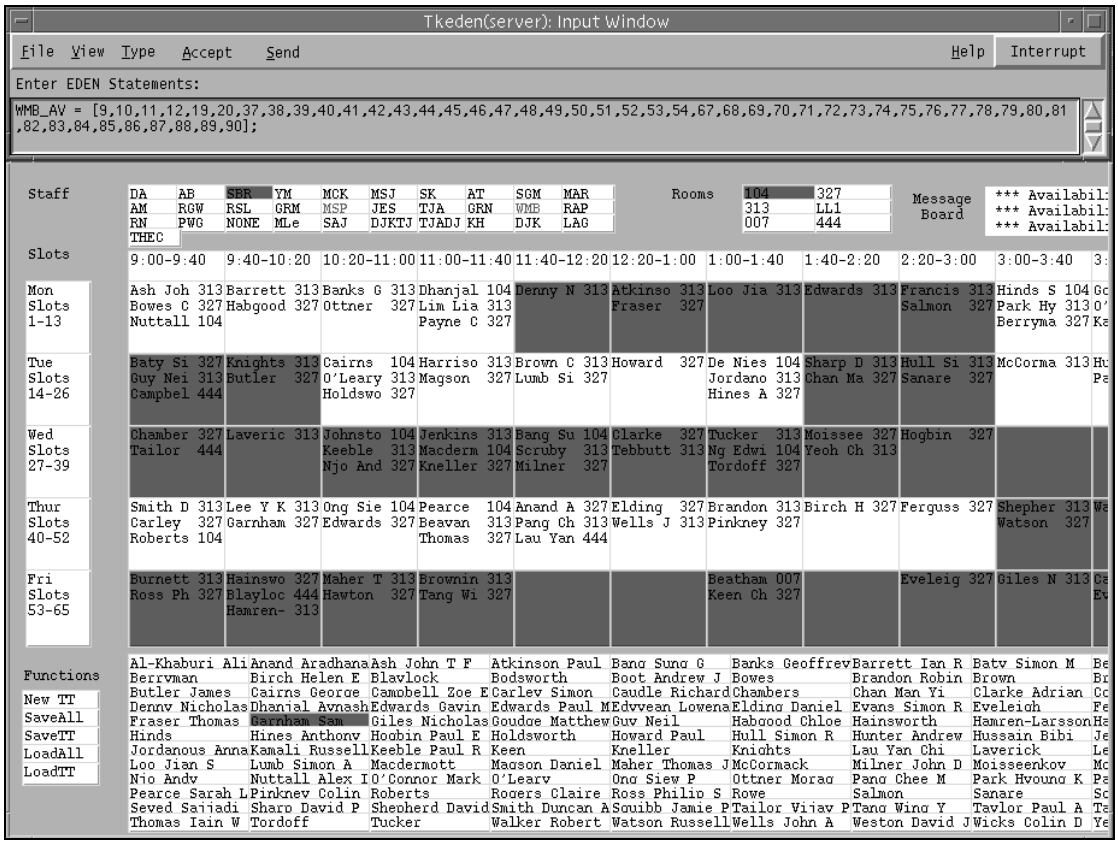

Figure 4.4: A screen capture of the Temposcope

The visualisation shown in the Figure 4.4 was developed incrementally throughout the development and use of the Temposcope. Its development involves defining hundreds of Scout windows on-the-fly, which is a tedious job if every window has to be defined manually. This task was assisted by using the mechanism of virtual agents in DTkEden (a distributed version of TkEden [Sun99a]). Scout supports building up visualisation and interface quickly without possibly losing track of the problem. The visualisation also reflects the conception of the final timetable. This enables the problem solver to 'work backwards' by modelling a virtual solution for the problem.

As the modelling of a variety of aspects of the Temposcope goes on interactively, the problem solver's knowledge about the problem is gradually increased. This knowledge enables the problem solver to develop possible methods to tackle the problem in an incremental way. Immediate feedback given by the Temposcope helps the problem solver to assess whether he or she is heading in the right direction, and enables the application of the Means-ends Analysis heuristic. The initial strategy employed was "place-and-seek" [Pae94]. This involves assigning a project to a slot and checking whether there is conflict or not. If there is a conflict, other slots will be tried until the conflict is resolved. Even with this simple strategy, the Temposcope helps the problem solver to make decisions by automatically checking for conflict on-the-fly.

As we gather experience in using the Temposcope, we can develop more advanced strategies. One strategy is to count the number of possible slots for each project after each assignment of slots. This statistic can be used to find out the most tightly constrained projects. The timetabler tries to schedule these projects first before the others. Counting the number of possible slots for more than one hundred projects is a tedious job for human but the computer can do it quickly and accurately. Human-computer co-operation makes solving the timetabling problem easier than solving by either human or computer alone.

Another important feature of the Temposcope is its openness to the problem solver: it provides free access to all its definitions at any time without restrictions. This facilitates integrated development and use that enables the quality of the Temposcope to be improved over time with possible adaptation to new situations.

4.4.3 Distinctive qualities of the Temposcope

The distinctive qualities of the Temposcope stem from its integrated development and use, based on EM concepts of observation, dependency and agency. They can be summarised as: support for a heuristic approach to problem solving; comprehensive problem analysis; adaptation and extension to a dynamic problem situation; and deeper engagement of human agents with the computer.

Support for a heuristic approach to problem solving

In the last subsection, we have shown that the Temposcope facilitates the use of a variety of human problem solving heuristics that include Means-ends Analysis, Successive Approximation and External Representation. The freedom to adopt any approach and heuristic at any time in problem solving is essential for solving problems whose solutions are uncertain.

The Temposcope also enables more flexible human-computer co-operation than conventional timetabling applications. The problem solver can conduct exploratory experiments. The computer automatically maintains dependencies between observables that are central to giving immediate feedback from the experiments. The process of conducting experiments resembles the process of making a scientific discovery [Kla88], in which people make hypotheses and test them by conducting experiments. As a result of experiment, the hypotheses are validated or invalidated. Based on the new results, new hypotheses can be made which in turn direct new experiments. In particular, the Temposcope represents a construal in which provisional knowledge about the project timetabling problem is embodied and ready to be refined. The results of experiment may eventually lead to the discovery of better methods to solve the timetabling problem.

Support for comprehensive problem analysis

In the solution of a timetabling problem, there are two aspects to be considered. On the one hand, there are generic algorithms and strategies for timetabling that are not specifically related to the particular problem. On the other hand, we need to develop a rich understanding of the specific timetabling situation. In a traditional timetabling program, the abstract algorithms and strategies are built into the program itself. This means that they are determined by the developer in isolation from the problem solving context. The timetabler’s task is to make the best possible use of the given timetabling

mechanisms to tackle the specific problem.

In using the Temposcope, it is possible to consider both the development of a timetabling method and the exploration of the specific timetabling problem in one and the same environment. In other words, timetabling using the Temposcope can be regarded as an integrated study of the timetabling problem and methods for its solution. In principle, this can lead to a deeper understanding of every aspect of the timetabling problem, out of which strategies for the best use of the power of both human and computer may emerge.

It should be noted that the integration of development and use envisaged here is more intimate than can be achieved simply by interleaving conventional development and use. By way of analogy, allowing arbitrary interleaving of conventional development and use is similar to allowing a car driver to return her car to the factory to be redesigned whenever she encounters unexpected problematic road conditions. In EM, there is no ontological distinction between the states in which development and use take place; it is as if the car driver can invoke car redesign in the context in which problematic conditions arise. This is significant for a variety of reasons: because (in the traditional redesign scenario) problematic road conditions may not be reproducible, making testing difficult; because there is a semantic issue concerning whether the redesigned car can return to ‘the same context’; because it is hard for the driver to communicate her experience in a remote situation to the designer at the factory.

As the above analogy suggests, EM involves a blurring of the roles of designer and user. This means that, when using the Temposcope, the timetabler can not only conduct problem analysis in systematic ways but also in *ad hoc* ways. In particular, the problem solver can change definitions in the Temposcope and observe the consequences. This helps the problem solver to comprehend the full implications of dependencies in the model.

The definitions introduced as the result of problem analysis are intrinsic to the problem. We can regard them together as a model of the problem from which new strategies of solving the problem can be tested.

Support for adaptation and extension to a dynamic problem situation

Conventional timetabling applications may have algorithms that are optimised to solve a particular timetabling problem. The flexibility of these applications is limited to the situations that can be conceived during the development process. Timetabling problems like the one we have discussed are dynamic in nature. The requirements and situations are changing according to the external environment in unpredictable ways. The observables of the Temposcope are direct reflections of the observables in the external environment. Changes in the external environment can be directly related to a need for change in the model. The problem solver's job is to maintain the relationships between the model and the external environment by modifying existing definitions or adding new definitions.

Since adding new observables to the Temposcope does not affect existing observables in general, new modes of observation can be introduced independently of existing observations. If new observations are based on existing observation, their dependencies are automatically maintained. Redefinition of an observable is also easy with automatic update of other observables that are dependent on it. All these features make extension to the model relatively easy. And, most importantly, new extensions can be tested quickly to allow a proper evaluation of their usefulness. The Temposcope has been already extended to allow online submission of staff availability and to estimate workload for each staff (workload weighting).

Support for deeper engagement between human and computer

Problem solving is related to subjective cognitive processes that originate from the problem solver's intuition, knowledge and experience. Thinking in terms of observables, dependencies and agents is arguably more natural than thinking in variables, procedures and functions. The gap between our natural language and machine language can be kept to a minimum. The problem solver can be more engaged with the problem at hand rather than worry about language translations.

The Temposcope allows the incorporation of subjective views of the problem solver. These views may reflect unresolved issues and unpredictable circumstances. They are provisional knowledge about the problem and its solution. Extensive interaction with the model may convert this provisional personal knowledge into more stable knowledge that can be shared with others.

4.5 Summary

In this chapter, we have explored the use of the computer to support human problem solving. Based on research on general problem solving and the psychology of programming, we have proposed a better explanation of the phenomenon of human problem solving using a computer that emphasises cooperation at two levels of abstraction in searching many different problem spaces. We have discussed the difficulties of using a conventional programming paradigm to support problem solving. We have explained how integrated development and use of a Construal of the Problem Solving Situation (CPSS) can give powerful support for problem solving. In particular, we have shown how a CPSS supports the application of well recognised problem solving heuristics with reference to a simple example: the Crossnumber problem, and to a real life application: the use of the Temposcope.