# A TOUR ADVISORY SYSTEM USING A LOGIC PROGRAMMING APPROACH

**Panagiotis Stamatopoulos** **Isambo Karali** **Constantin Halatsis**

University of Athens, Department of Informatics

**Abstract**

A PErsonalized Tourist INformation Advisor is presented, called PETINA, which is a system aiming at constructing tours that satisfy constraints specified by tourists. The system consults a database which contains information about activities, events and sites that refer to Greece. PETINA takes as input user wishes about tour generation expressed as constraints over visits' properties and its output is tours satisfying these constraints. The user wishes may be stated using either a formal language or a graphical interface. The method of computation applies to any problem domain, in case the problem involves combinatorial searching under some kinds of constraints that can be classified into some well defined categories. Although a logic programming approach is suitable and valuable for the formulation of combinatorial search problems, conventional Prolog systems fail to cope with them efficiently. PETINA has been implemented in the ElipSys language, which is a parallel logic programming system extended with various powerful mechanisms to allow efficient execution. Most of the ElipSys' features were proved to be indispensable for handling the complexity of the encountered problems.

**Keywords**

*combinatorial search, tour construction, parallel logic programming, constraint satisfaction, data driven computation*

**Introduction**

Combinatorial search problems are computationally intensive, especially if they address a significantly large search space. Unfortunately, no general and efficient algorithm exists to solve them. Many combinatorial search problems involve user defined constraints over their search space, so the straightforward and classical method to cope with them is to employ the traditional generate-and-test method. The declarative formulation of this method can be achieved in a logic programming environment, via the Prolog language [3, 19]. However, due to the inefficiency of the exhaustive search of Prolog, real-life problems cannot be solved in this basic framework.

In this paper, a specific combinatorial search problem is presented. The application that exemplifies the problem is called PETINA [6, 7, 18], which is a PErsonalized Tourist INformation Advisor about Greece. Its purpose is to construct tours that satisfy constraints specified by tourists. Although PETINA refers to Greece, the system is generic and can be used as a tourist information advisor for any country.

A parallel logic programming language, ElipSys [1], is used for the implementation of PETINA. ElipSys provides, apart from various kinds of parallelism, such as OR-parallelism, AND-parallelism and data-parallelism, other additional features which are exploited to attack the encountered problems and to make the application useful in a real-world environment. These features consist of the introduction of a data driven computation rule on top of the usual depth-first left-to-right execution strategy of Prolog and the incorporation of constraint satisfaction techniques over finite domains into the language. Another useful facility that ElipSys provides is its extension with the appropriate tools which facilitate the development of graphical user interfaces. Both PETINA and ElipSys were developed in the context of the ESPRIT II EDS (European Declarative System) project by the University of Athens and ECRC respectively.

The previous work in the tour generation problem domain comprises two prototypes, namely TIA [11] and TInA [12], which were implemented in the PEPSys parallel logic programming language [16]. Although these prototypes had to deal with similar problems to the ones of PETINA, they cannot be considered as real-life applications, since they addressed limited amount of data. Moreover, the ex-

tended database structure and the advanced functionality of PETINA, with respect to the ones of TIA and TInA, require more power by the underlying implementation framework than the one of PEPSys' parallel execution.

In the following, a brief description of the ElipSys language is given as well as PETINA's database structure, the functionality of the system's tour generation facility and the structural specification of the whole application are presented. Emphasis is put on the method employed by the Tour Generation Engine in conjunction with the exploited ElipSys features. Finally, implementation issues are discussed and performance measurements are presented.

**ElipSys Language**

ElipSys [1] is a parallel logic programming language, which has been extended to incorporate various powerful execution mechanisms. The language supports OR-parallelism, AND-parallelism, data-parallelism, data driven computation, constraint satisfaction techniques over finite domains and a facility for the development of user interfaces. ElipSys has greatly benefited by the Megalog [2], CHIP [4] and PEPSys [16] projects of ECRC.

OR-parallelism aims at the concurrent exploration of the various alternative clauses that define an ElipSys procedure. The programmer has to declare "good points" for efficient OR-parallel execution. If there are available resources, that is processing elements, a branch point is created with fertility equal to the number of alternative clauses.

AND-parallelism results from the concurrent execution of two goals in conjunction. This feature is not provided by the ElipSys execution model, neither it is supported by the run time environment of the language. It resides only at the language level and is actually compiled into OR-parallelism.

Data-parallelism [8] is a kind of parallelism arising from the concurrent treatment of the elements of a set of data. It is supported by various built-in predicates. If there are available resources, a branch point is created with fertility equal to the number of the elements in the set.

In addition, ElipSys supports a data driven computation rule [15] on top of the usual depth-first left-to-right execution strategy of Prolog. This rule modifies the reduction order of goals according to instantiations of variables by declaring that every goal that refers to a specific predicate has to be delayed, if its arguments are not adequately instantiated. A delayed goal is awakened when the desirable degree of instantiation is achieved.

Constraint satisfaction techniques over finite domains [20] lead to a priori pruning of the search space and thus, they result in more optimized execution. ElipSys provides this facility [21] by allowing the programmer to define domain variables which may range over integer intervals or enumerated sets and to use a set of built-in constraints on these variables. After stating the constraints that describe a problem, the generation of values for the domain variables must be triggered, via the appropriate built-in predicates, in order to start the constraint propagation and the pruning of the search space. In addition, the constraint satisfaction mechanism of ElipSys supports a set of higher order predicates useful for optimization problems.

Finally, another feature of ElipSys concerns an object oriented extension, named PCE [13], which was developed independently from the language. PCE allows to create X-window based user interfaces easily and quickly. It provides a set of built-in classes and, in general, a small amount of ElipSys code suffices to adapt the built-in facilities to a particular application.

**PETINA's Database**

The PETINA system consults a database, implemented as a set of ElipSys facts, that contains information about *activities*, *events* and *sites*. Activities are considered to be the tourist's visits to various spots, while events are shows that may be attended. In addition, the sites refer to the geographical entities of Greece.

Three data structures are defined in the system's database, namely the *activity tree*, the *event tree* and the *site tree*. However, the main part of the database consists of the activity and event *instances* as well as the site ones. Every instance is identified by a unique key value. The activity and event instances are linked to nodes of the corresponding trees. On the other hand, the site instances themselves compose the site tree. The activity, event and site instances are characterized by their *attributes*.

The nodes of the activity tree represent activity categories. The tree organization is based on interest hierarchy and the nodes of a part of the tree are considered as *interest nodes*, in terms of which specific interests may be expressed. Activity categories whose instances have various kinds of interest are represented by more than one tree nodes, denoted with the same keyword but with different indices. In this way, a graph idea is implemented with a tree structure. In order to refer to an activity category regardless of type of interest, a variable may be used in place of the index, e.g. $museum(X)$ represents all the "museum" nodes of the activity tree, i.e. $museum(1)$, $museum(2)$, $\ldots$, $museum(7)$.

Activity instances can be linked to more than one nodes according to the categories they belong to and according to the types of interest they present. Each type of interest corresponds to an interest node. An activity instance is linked either directly or indirectly to all, and only these, interest nodes that correspond to the types of interest it

presents.

An activity instance is characterized by its attributes, namely the *site*, the *denomination*, the *duration*, the *cost*, the *time period*, the *closed days*, the *interest* and the *detail*. The interest attribute is a special one in the sense that it collects as many values as the types of interest the instance presents.

Similar approaches have been followed for the event and the site information.

**Functionality of PETINA**

PETINA takes as input user wishes about tour generation, named *tour generation requests*, expressed as constraints over visits' properties. Its output is tours satisfying the user's constraints, as sets of activity instances or as sets of event instances. The user is also allowed to ask for information about activities, events and sites via *information retrieval queries*. Finally, management of PETINA's database is supplied by the system through *database administration commands*. The above three kinds of requests may be expressed using a formal language close, however, to natural language. This language is defined by a Definite Clause Grammar (DCG) [14] which offers the possibility to handle context sensitivity, transformation of the input and procedure calls. Moreover, a graphical interface is provided that helps the user to formulate the requests. In this case, the user does not need to know anything about PETINA's formal language.

The rest of this section is devoted to the tour generation requests giving a general description of the constraints of the language and presenting the functionality of the graphical interface, as far as the tour generation facility of the system is concerned.

There are two kinds of tours that the system produces. Consequently, there are two kinds of tour generation requests the user may express. The one concerns the activity tour generation and the other the event tour generation. In both cases, at the beginning of the request, the user has to give a *time constraint* concerning the time period when the visit is going to take place, in order to avoid visiting spots that are inactive. The other part of the request is a set of *activity constraints* or a set of *event constraints*. The answer to a tour generation request consists of the tours which satisfy all the constraints of the request.

A time constraint is satisfied by a tour, if the time period attribute of every instance that belongs to the tour has a non empty intersection with the visit period defined in the time constraint. An example of a time constraint is the following:

```
visit period is 20 Jul 92 - 10 Sep 92
```

An activity or event constraint may be either *simple* or *cross*. A simple constraint has the general form

⟨*condition*⟩ `for` ⟨*subtour*⟩

and is satisfied by a tour in case ⟨*condition*⟩ holds for the ⟨*subtour*⟩.

The ⟨*condition*⟩ may be *local*, *global*, *topological* or *complex*. The latter involves the operators "`and`", "`or`" and "`not`" applied to the first three kinds of conditions. Local conditions refer to every instance of the ⟨*subtour*⟩ individually. They involve an attribute expression, i.e. either an arithmetic expression of attributes or a single attribute. Global conditions refer to the entire ⟨*subtour*⟩ as a whole. They involve an aggregate function ("`sum`", "`avg`", "`max`", "`min`") applied to an attribute expression. Topological conditions refer to the number of instances in the ⟨*subtour*⟩. In this case, the keyword "`number`" is used.

A simple constraint may be *local*, *global* or *topological* if its ⟨*condition*⟩ is local, global or topological respectively. In case a complex condition appears in a simple constraint, the condition is transformed into *conjunctive normal form*. Then, the original constraint is substituted by one or more constraints whose conditions are the and-operands of the normal form. If an and-operand involves only local conditions, the corresponding constraint is local. The same holds for global and topological conditions. Otherwise, if no such classification can be done, the constraint is called *mixed*.

As far as the ⟨*subtour*⟩ part of a simple constraint is concerned, this is defined in terms of one or more tree nodes, possibly refined by the so called *where-properties* by using the keyword "`where`". In case of an activity constraint, an entire category (set of activity nodes) may be referenced or a single activity node by using the "`with`" specifier.

The following are examples of simple activity constraints:

1. `duration*interest >= 600`
   `for plant` (local constraint)
2. `max(religious place interest) >= 7`
   `for building with architectural place interest` (global constraint)
3. `number = 1 for picturesque spot where interest > 5` (topological constraint)
4. `min(cost) =< 300 or duration > 180`
   `for holiday resort` (mixed constraint)

In the order they appear, each of the previous constraints is satisfied by a tour if:

1. Considering the subtour of the tour that contains the instances which are finally linked to a node of the form

$plant(X)$, for every instance in the subtour, the product of its duration and its interest attributes is greater than or equal to 600.

2. Considering the subtour of the tour that contains the instances which are finally linked to the node $building(2)$, the maximum value of the religious place interest attributes of the instances in the subtour is greater than or equal to 7.

3. Considering the subtour of the tour that contains the instances which are finally linked to a node of the form $picturesque_spot(X)$ and have interest attribute greater than 5, this subtour comprises only 1 instance.

4. Considering the subtour of the tour that contains the instances which are finally linked to a node of the form $holiday_resort(X)$, either the minimum value of the cost attributes of the instances in the subtour is less than or equal to 300 or for every instance in the subtour, its duration attribute is greater than 180.

Apart from the usual comparisons operators, the "`in`" and `between`" operators may be used as well, which actually introduce complex conditions.

As already mentioned, there are cross constraints as well. A cross constraint has the general form

$\langle cum \rangle_1$ `for` $\langle subtour \rangle_1$ $\langle cmp_op \rangle$ $\langle cum \rangle_2$ `for` $\langle subtour \rangle_2$

where $\langle cum \rangle_1$ and $\langle cum \rangle_2$ are either aggregate functions applied to attribute expressions or the keyword "`number`". This constraint is satisfied by a tour if the cumulative property $\langle cum \rangle_1$ of $\langle subtour \rangle_1$ is related with the cumulative property $\langle cum \rangle_2$ of $\langle subtour \rangle_2$ via the comparison operator $\langle cmp_op \rangle$. According to the kind of the cumulative properties, a cross constraint may be *global* or *topological*.

The following are examples of cross activity constraints:

1. `sum(cost) for ancient history place =< sum(cost) for middle age history place` (global constraint)

2. `number for modern year history place > number for nature` (topological constraint)

Each of the previous constraints is satisfied by a tour if:

1. Considering the subtour of the tour that contains the instances which are finally linked to a node of the form $ancient_history_place(X)$ and the subtour of the tour that contains the instances which are finally linked to a node of the form $middle_age_history_place(Y)$, the total cost of the instances in the first subtour is less than or equal to the total cost of the instances in the second subtour.

2. Considering the subtour of the tour that contains the instances which are finally linked to a node of the form $modern_year_history_place(X)$ and the subtour of the tour that contains the instances which are finally linked to a node of the form $nature(Y)$, the first subtour comprises more instances than the second one.

Since most users are not willing to learn and use a formal language and in PETINA's case the user may be even a tourist, a graphical interface has been also developed, using the PCE extension of ElipSys, that provides a user friendly way to access the system. This interface is designed in such a way that the user composes the request via a pointing device (mouse) and with a minimum use of the keyboard. Menus and buttons are used extensively, in order to minimize the possibility of erroneous inputs.

As far as the tour generation facility of PETINA is concerned, the graphical interface firstly asks the user about the choice between activity or event tour generation. Then, a time constraint is requested in a user friendly way. Next, the interface asks the user to give either a simple or a cross constraint. In case of a simple constraint, the $\langle condition \rangle$ part of it is requested. The entry procedure of the $\langle subtour \rangle$ part of the constraint follows. Finally, the user is asked whether he/she wants to give a where-property for the $\langle subtour \rangle$. The procedure for giving an activity or event constraint accordingly is repeated until no more constraints are desired by the user. In the case of cross constraints, similar functionality is provided by the interface.

**PETINA's Structural Specification**

PETINA is a clearly modularized system. The modules it consists of are the *User Interface*, the *Language Analyzer*, the *Tour Generation Engine*, the *Information Retrieval Engine* and the *Database Administration Engine*. The User Interface module is responsible for the user-system communication. It takes as input a graphically stated request and produces the corresponding sentence of PETINA's formal language. The Language Analyzer transforms the input request expressed in PETINA's formal language into a form suitable for further processing. The Tour Generation Engine, the most important module of the system, generates activity and event tours satisfying the user's constraints. The Information Retrieval Engine supplies the information the user asked for. Finally, the Database Administration Engine manages the database contents. None of the above modules needs any change in case a different country than Greece is to be considered.

The Language Analyzer is further refined into the *Tokenizer* and the *Parser*. The Tokenizer transforms the input request

into a list of tokens. This list is recognized by the Parser, in order to produce the formal representation of the request.

This paper concentrates on the Tour Generation Engine (TGE) module of the system, as it has the most complex problem to solve and various mechanisms are required. The TGE consists of the *Domains Creator*, the *Configurator*, the *Database Filter*, the *Tour Generator* and the *Tour Evaluator*. The functionality of these submodules as well as the method employed by each one are presented in the following section.

**Method of Computation in TGE**

Some of the submodules of TGE deal with extremely cumbersome tasks. The Tour Generator involves a combinatorial search problem over a large space. The Configurator has to solve a system of equalities and inequalities. Moreover, the Database Filter has to handle a large amount of data.

Taking into consideration the above, a methodology that solves the various encountered problems in a systematic way is employed. This methodology is a general one and can be applied to all problems that involve combinatorial searching under constraints that fall into the presented general categories. The approach taken for every submodule of TGE, is presented in the following.

Firstly, the Domains Creator partitions the activity or event tree, depending on the type of the request, into *domains*. This partitioning is based on global and topological constraints, both simple and cross, as well as on the mixed ones, in such a way that no two domains have the same set of global, topological or mixed constraints applied to them. Each domain is further partitioned into *fine domains*, according to the local constraints. The partitioning is carried out by the Domains Creator in the following way. Starting from the root of the relevant tree, either activity or event, all the nodes are visited in a depth-first left-to-right fashion. During each visit, the constraints that apply directly to the node are considered as well as the ones which are inherited from the ancestor nodes. The whole set of constraints that apply to the node is the criterion for embedding the node into a fine domain and, consequently, into a domain.

The Configurator produces all possible *configurations* of the requested tours. A configuration represents acceptable numbers of instances per domain in a tour satisfying the user's constraints. This module, taking into account the simple and cross topological constraints, generates and solves a system of linear equalities and inequalities. The solution of the system is achieved by exploiting the constraint satisfaction techniques that ElipSys offers. Firstly, a set of ElipSys domain variables is generated, each one corresponding to a domain and representing the acceptable number of instances from this domain in the requested tour. Then, for every topological constraint, a linear equality or inequality is formed, which is stated as an ElipSys constraint. Finally, the generation of values for the created domain variables is triggered, which leads to the computation of the solutions of the system of the linear equalities and inequalities. Each solution corresponds to a configuration.

Next, the Database Filter selects the instances, either activity or event, according to the type of the request, that satisfy the time constraint and the relevant local constraints. Theses instances are selected for every node refined by its where-property to build the instances of a fine domain. Then, such sets are structured to form a domain. Finally, for every domain, the lists of instances corresponding to its fine domains are combined into a single list and any duplicate instances are removed, leading to the composition of the filtered database. Duplicate instances may occur in the case of an activity tour generation request due to the multiple links of the activity instances with the nodes of the activity tree. Parallelism is exploited in the Database Filter. More precisely, there are three levels of exploitation, the concurrent processing of domains, fine domains and nodes refined by their where-properties. Parallel execution is carried out during the postprocessing phase of the filtered database as well. The kind of parallelism encountered is data-parallelism.

The Tour Generator is the module where the actual tours are constructed. The method used for the construction of tours is test-and-generate implemented using the delay mechanism of ElipSys. For every configuration, each one of the simple global, cross global and mixed constraints is stated, though it is delayed until the subtours it applies to become ground. Next, the generation of instances for every domain is triggered extracting them from the filtered database. During this generation, a constraint is activated and checked as soon as all the subtours it involves become fully instantiated. The tour that is being built is rejected, if a constraint is not satisfied. Then, each tour that is computed is processed in order to flatten its subtours, check for possible duplicate elements that might appear in different domains and lexicographically sort its elements. Possible duplicate tours are removed from the whole set of tours. The main source of parallelism of the whole system exists in the Tour Generator. Firstly, there is the parallel processing of all configurations and secondly, the selection of possible instances to build a subtour for the corresponding domain in parallel. In both cases, the kind of parallelism is again data-parallelism. This kind of parallelism is also exploited in the postprocessing of the generated tours.

Finally, the Tour Evaluator sorts the tours produced by the Tour Generator in descending order according to their average interest. In addition, it replaces the key values of the instances by the corresponding denominations. The quicksort algorithm is used, which is a typical divide-and-conquer method. Thus, AND-parallelism is exploited, as it fits per-

fectly in this case.

**Performance Measurements**

The first implementation of the tour generation facility of PETINA was carried out in Sepia Prolog [17]. Sepia is an advanced sequential Prolog system. Among the various features it offers, the delay mechanism was used in the Tour Generator as well as in the Configurator, in order to solve the system of equalities and inequalities by a test-and-generate method.

The current implementation has been carried out in the ElipSys version 0.3 [5]. The submodules which involve parallelism, namely the Database Filter, the Tour Generator and the Tour Evaluator, were also implemented in the PEPSys language [16]. PEPSys is a parallel logic programming language that supports OR- and restricted AND-parallelism. The data-parallelism facility of ElipSys was simulated by the PEPSys OR-parallelism. The COKE preprocessor [9, 10], that allows to measure the theoretical performance of parallel execution of PEPSys programs, was used. The Sepia, ElipSys and PEPSys/COKE work was carried out on SUN 3/60 workstations under SunOS 4.1.1. Moreover, the ElipSys version of the implementation was tested on a Sequent Symmetry machine, the shared memory multiprocessor of ECRC. Thus, true parallel execution results were obtained as well.

The above implementations gave the opportunity for comparing various programming methodologies. The performance gain by using the delay mechanism of ElipSys in the Tour Generator ranged from 3:1 to 5:1, for typical tour generation requests. As far as the use of the constraint satisfaction techniques is concerned, a dramatical improvement was achieved by the ElipSys implementation of the Configurator with respect to the one in Sepia. In most cases, the performance gain was several orders of magnitude.

Finally, as mentioned above, parallelism was exploited in three submodules. For a complex request presented in [18], the speedup achieved by the Database Filter was 40.11 $ni/et$ (number of inferences / execution time), by the Tour Generator 825.41 $ni/et$ and by the Tour Evaluator 7.46 $ni/et$. The COKE tool was used to obtain these measurements. This tool assumes that each goal executes in one time unit and unlimited resources (processors) are available. The graphs representing the number of processes vs. execution time for the three submodules are presented in Figures 1, 2 and 3.

It is worthwhile mentioning that the quality of the graph that corresponds to the Tour Generator, which is the most computationally intensive part of the system, is very good. The shape of this graph is flat rather than peaky, which means that the exploitation of parallelism in this submodule is promising. As far as the graphs that correspond to the other two submodules are concerned, their shapes are not as good as the Tour Generator's one, but, anyway these submodules may contribute to the overall acceleration of the system.

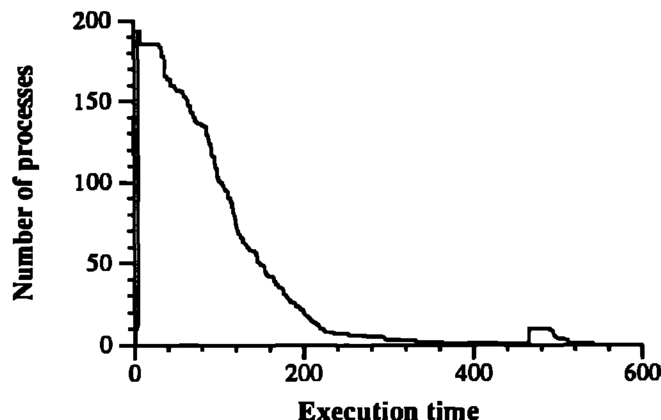


Figure 1: Database Filter graph

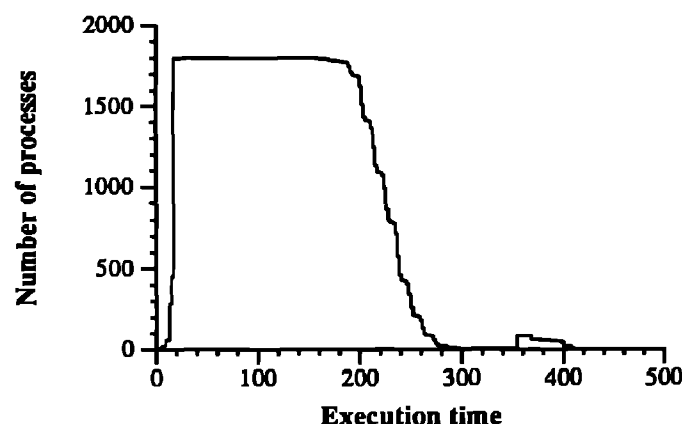


Figure 2: Tour Generator graph

Using the Symmetry as a platform, the system was tested on a true parallel machine, which provided the opportunity to check the degree of real parallelism exploitation. In order to get sequential results, ElipSys was used with one worker and the various submodules of the tour generation facility ran separately on the request into consideration. Table 1 presents the corresponding execution times in CPU seconds. In addition, the total execution time was computed. Parallel execution results were obtained for the Database Filter (DF), the Tour Generator (TG) and the Tour Evaluator (TE) submodules, where parallelism is exploited, using a number of two to six workers. The CPU times (in seconds) of the longest processes are shown in Table 2 for each of the previous submodules. The constant sum of the CPU times for the sequential submodules (Seq), i.e. Tokenizer, Parser, Domains Creator and Configurator, as well as the total exe-

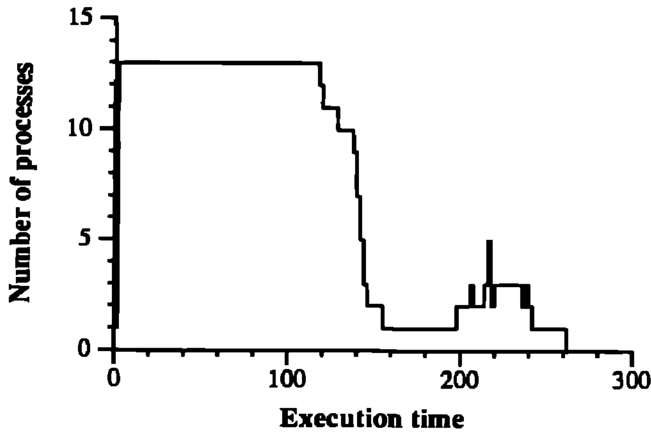


Figure 3: Tour Evaluator graph

cution results, that is the ones which are related to the whole tour generation facility, are also shown in Table 2, considering a number of two to six workers. Finally, Table 3 presents the speedups achieved for the parallel submodules as well as the total speedups of the whole execution. These results are also graphically presented in Figure 4.

| | *1 w* |
|---|---|
| *Tokenizer* | 8.29 |
| *Parser* | 4.23 |
| *Domains Creator* | 18.61 |
| *Configurator* | 0.55 |
| *Database Filter* | 23.87 |
| *Tour Generator* | 540.57 |
| *Tour Evaluator* | 1.58 |
| *Total* | 597.70 |

Table 1: Sequential execution results on the Symmetry

| | *2 w* | *3 w* | *4 w* | *5 w* | *6 w* |
|---|---|---|---|---|---|
| *Seq* | 31.68 | 31.68 | 31.68 | 31.68 | 31.68 |
| *DF* | 12.50 | 8.68 | 6.62 | 5.53 | 4.77 |
| *TG* | 277.61 | 184.33 | 138.40 | 112.39 | 94.59 |
| *TE* | 1.24 | 1.04 | 0.90 | 0.80 | 0.82 |
| *Total* | 323.03 | 225.73 | 177.60 | 150.40 | 131.86 |

Table 2: Parallel execution results on the Symmetry

To comment on the above, the theoretically good results obtained by the COKE tool were verified by the true parallel execution. The Tour Generator, where the bulk of

| | *2 w* | *3 w* | *4 w* | *5 w* | *6 w* |
|---|---|---|---|---|---|
| *DF* | 1.91 | 2.75 | 3.61 | 4.32 | 5.00 |
| *TG* | 1.95 | 2.93 | 3.91 | 4.81 | 5.71 |
| *TE* | 1.27 | 1.52 | 1.76 | 1.98 | 1.93 |
| *Total* | 1.85 | 2.65 | 3.37 | 3.97 | 4.53 |

Table 3: Speedups for parallel execution

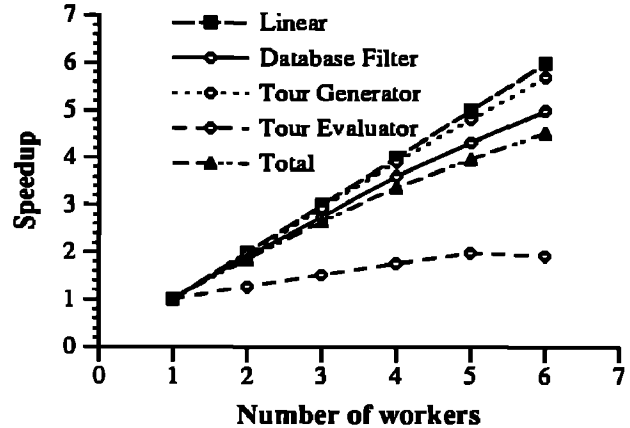


Figure 4: Speedup vs. number of workers

the computational load appears, presents a curve that approximates significantly the ideal linear curve. As it was expected, the overall speedup is mainly affected by the one of the Tour Generator, thus the corresponding curve is very close to the ideal one. The reason why the Tour Evaluator does not seem to present good speedups is that, for the specific request, this submodule has to sort just 13 tours. So, parallelism is not highly exploitable in this case.

## Conclusions and Future Work

In this paper, the most significant part of PETINA, that is the one which carries out its tour generation facility, was presented. PETINA is a PErsonalized Tourist INformation Advisor consulting a database that contains tourist data about Greece. Thus, by changing the database the system can be used for any country.

The problem of tour generation is a combinatorial search one, thus advanced mechanisms are required to cope with it efficiently. The ElipSys parallel logic programming language is a suitable vehicle in this direction, since, apart from the parallel execution, it offers the possibility of declarative formulation of the problem as well as it provides various extended features, such as data driven computation, constraint satisfaction techniques and a platform for developing

graphical interfaces.

Although PETINA's tour generation problem addresses a large search space, it was shown that ElipSys features help to attack the complexity of the algorithms needed. Parallelism was highly exploitable, as it was proved by the presented performance measurements. Moreover, the data driven computation was found useful and the constraint satisfaction facilities of ElipSys were found indispensable. Finally, PETINA, as it is a real-life application, provides a graphical and friendly way to allow causal users to access the system, exploiting the appropriate facility offered by ElipSys.

An interesting characteristic of the system is that the method employed for the tour construction is a general one and can be applied to any problem domain which involves combinatorial searching under constraints that fall into some well defined categories.

The objective of the future work in the tour generation facility of PETINA is to reduce the execution time of the Tour Generator that presents the bulk of the computational load. More precisely, a more profitable use of the delay mechanism of ElipSys is envisaged. Alternatively, the constraint satisfaction techniques may be exploited into the Tour Generator as well.

**Acknowledgements**

The authors express their thanks to ECRC GmbH (Munich, Germany), for providing access to its parallel machine and the ElipSys language. Special thanks are directed to Mike Reeve and Michael Ratcliffe from ECRC, for their encouragement, guidelines, feedback and valuable comments during the development of PETINA in the context of the EDS project.

## References

[1] U. Baron, S. Bescos, S. Delgado-Rannauro, P. Heuzé, M. Dorochevsky, M.-B. Ibáñez-Espiga, K. Schuerman, M. Ratcliffe, A. Véron, and J. Xu. The ElipSys logic programming language. Technical Report DPS-81, ECRC, December 1990.

[2] J. Bocca. Megalog — A platform for developing knowledge base management systems. Internal Report KB-75, ECRC, October 1990.

[3] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer Verlag, New York, 1981.

[4] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *International Conference on Fifth Generation Computer Systems*, pages 693–702, 1988.

[5] *ElipSys User Manual for release version* 0.3, October 1991.

[6] C. Halatsis, M. Katzouraki, M. Hatzopoulos, P. Stamatopoulos, I. Karali, C. Mourlas, M. Gergatsoulis, and E. Pelecanos. PETINA — Implementation of the tour generation. Project Deliverable EDS.WP.9E.A006, University of Athens, December 1991.

[7] C. Halatsis, M. Katzouraki, M. Hatzopoulos, P. Stamatopoulos, I. Karali, C. Mourlas, M. Gergatsoulis, and E. Pelecanos. PETINA — Performance evaluation of the tour generation. Project Deliverable EDS.WP.9E.A007, University of Athens, December 1991.

[8] P. Heuzé. Using Data-Parallelism in the ElipSys. Internal Report ElipSys-003, ECRC, June 1989.

[9] P. Heuzé and B. Ing. COKE: User manual 1.0. Internal report, ECRC, February 1989.

[10] B. Ing. COKE — An analysis tool for PEPSys programmes. Internal Report 23, ECRC, October 1987.

[11] B. Ing. Tourist information advisor: A case study of an application in PEPSys. Internal Report PEPSys/15, ECRC, April 1987.

[12] B. Ing. Tourist information advisor — A case study of an application in PEPSys — Final report. Internal Report PEPSys-32, ECRC, September 1988.

[13] *PCE Reference Manual*, October 1986.

[14] F. Pereira and D. Warren. Definite clause grammars for language analysis — A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278, 1980.

[15] M. Ratcliffe. On the use of the delay in ElipSys Prolog. Technical Report elipsys/001, ECRC, June 1989.

[16] M. Ratcliffe and J.-C. Syre. A parallel logic programming language for PEPSys. In *International Joint Conference on Artificial Intelligence*, pages 48–55, 1987.

[17] *Sepia 3.0 User Manual*, June 1990.

[18] P. Stamatopoulos, I. Karali, and C. Halatsis. PETINA — Tour generation using the ElipSys inference system. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, volume 1, pages 320–327, 1992.

[19] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, MA, 1986.

[20] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[21] J. Xu and A. Véron. Types and constraints in the parallel logic programming system ElipSys. Technical Report DPS-105, ECRC, March 1991.