PETINA — Tour Generation Using the ElipSys Inference System

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

University of Athens, Department of Informatics

Abstract

PETINA is a PErsonalized Tourist INformation Advisor system aiming at helping tourists to construct tours satisfying specified constraints. The system consults a large database that contains tourist data. PETINA has been implemented in the ElipSys language, which is a pure parallel logic programming system extended with various powerful mechanisms and features to allow efficient parallel execution. Although the expressive power of logic programming is profitable for the development of PETINA, standard Prolog systems lack of facilities that are vital for the PETINA application to work. ElipSys has proved to be very suitable tool for the implementation of PETINA, as most of the former's features are indispensable for handling the complexity of the encountered problems.

Introduction

Many combinatorial search problems are NP-complete [5] and no general and efficient algorithm exists to solve them. A big subset of these problems involves 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 [2, 16]. 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 combinatorially hard search problem is presented. The application that exemplifies the problem is called PETINA [6], which is a PErsonalized Tourist INformation Advisor about Greece. Its purpose is to help tourists to construct tours satisfying specified constraints. A parallel logic programming language, ElipSys [1], is used for the implementation. ElipSys provides, apart from parallelism, other additional features which are exploited to attack the encountered problems. Both PETINA and ElipSys are being 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 specific problem domain comprises two prototypes, namely TIA [10] and TInA [11], which were implemented in the PEPSys parallel logic programming language [14]. 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 extended 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 Engine and the structural specification of the whole application is 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 is a parallel logic programming language, which has been extended to incorporate various powerful execution mechanisms. The language supports pure OR-parallelism, data-parallelism, data driven computation,

constraint satisfaction techniques over finite domains and an interface to relational databases.

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. This is achieved by using the *parallel*/1 directive, which declares that all clauses defined by a specific predicate might be explored in parallel, if there are available resources, that is processing elements. In this case, a branch point is created with fertility equal to the number of alternative clauses.

Data-parallelism [7] 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, e.g. *par_member*/2, *par_delete*/3, *par_select*/3 etc. In all cases, if there are available resources, a branch point is created with fertility equal to the number of the elements in the set of data.

In addition, ElipSys supports a data driven computation rule [13] 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. The *mode*/1 directive is used to declare 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 degree of instantiation declared by the *mode*/1 directive is achieved.

Constraint satisfaction techniques over finite domains [17] may tackle real-world problems efficiently, since they lead to a priori pruning of the search space and thus, they result in more optimized execution. ElipSys provides this facility [18] by allowing the programmer to define domain variables which may range over integer intervals or enumerated sets, using the :: /2 and ::: /2 built-in predicates respectively. A set of arithmetic and symbolic built-in constraints on domain variables is supported, either forward or lookahead checkable. The programmer has to define his/her problem by stating the constraints that describe it. Then, the generation of values for the domain variables must be triggered, via the *indomain*/1 predicate, in order to start the constraint propagation and the pruning of the search space. Moreover, the constraint satisfaction mechanism of ElipSys supports additional predicates that are used to make choices, that is to generate values in a specific way, as well as a set of higher-order predicates useful for optimization problems.

The last, but not least, major feature of ElipSys concerns the connection of the language with conventional relational databases. More precisely, an interface to the ESQL language [4] of the EDS database server is supported. Two levels are provided in the interface. At the lower level, the built-in predicates *sql*/2 and *par_sql*/2 are used to embed the full power of ESQL into the ElipSys language. At the higher level, the directive *relation*/1 is used to define that database relations are treated as predicates and can be manipulated in a similar way as normal predicates.

PETINA's Database

The PETINA system consults a database 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 he/she may attend. 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 activities' and events' *instances* as well as the sites' 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*. Currently, PETINA's database is implemented as a set of ElipSys facts. Once the actual ElipSys to ESQL connection is available, the above information will be implemented using a relational database to be handled by ESQL.

The nodes of the activity tree represent activity categories. The tree organization is based on a tourist interest hierarchy and the nodes of a part of the tree are considered as *interest nodes*. These nodes are located higher than other nodes in the tree. The ElipSys implementation of the activity tree is carried out by the `activity_ako/2` predicate. For example, the fact:

```
activity_ako(church(1),historical_site(2)).
```

denotes that the activity category `church(1)` is a kind of `historical_site(2)`, which is also an activity category.

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. 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. The links be-

tween the activity instances and the activity tree nodes are implemented by the `activity_isa/2` predicate, e.g.:

```
activity_isa(ainst00001,museum(1)).
activity_isa(ainst00001,building(1)).
activity_isa(ainst00001,museum(4)).
```

The above example denotes that the activity instance with key value `ainst00001` is an instance of the categories `museum(1)`, `building(1)` and `museum(4)`.

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. The activity attributes are implemented by ElipSys facts with predicates of arity 2 which relate the key value of the instance with the attributes' values.

The event tree nodes correspond to event categories. The tree's organization is based on event type hierarchy. The event tree implementation is carried out in a similar way to the one of the activity tree. However, the structure of the event tree is simpler. A pure tree idea is reflected, so there are no indexed nodes. In contrast to the activity instances, event instances are not linked to more than one tree nodes. The implementation of the links, however, is carried out in a similar way. An event instance is characterized by its attributes, namely the *site*, the *denomination*, the *duration*, the *cost*, the *time period*, the *interest*, the *takes place*, the *series* and the *detail*. The interest attribute does not play any special role in the case of the event instances. The event attributes implementation is carried out in a similar way to the activity ones.

Finally, the nodes of the site tree are site instances. The tree hierarchy reflects a site inclusion relation, needed to refer to villages, cities, islands, island groups, provinces, districts etc. The site tree implementation is carried out in a similar way to the ones of the other two trees. In the case of the site tree, however, the instances themselves compose the site tree, so no other kind of link is introduced. A site is characterized by its attributes, namely the *type*, the *accommodation*, the *approachability*, the *entertainment*, the *tour availability*, the *eating facilities* and the *detail*. Every site is identified by its name which is a unique key value. As far as the attributes' implementation is concerned, ElipSys facts are again used.

Functionality of the System

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*. Currently, the above three kinds of requests are expressed using a formal language close, however, to natural language. This language is defined by a Definite Clause Grammar (DCG) [12] which offers the possibility to handle context sensitivity, transformation of the input and procedure calls. However, in the future, a graphical interface will be developed. The rest of this section concentrates on the tour generation requests giving a general description of the constraints of the language.

There are two kinds of tours 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 his/her 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*⟩. 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 has to be 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. 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*. An entire category (set of nodes) may be referenced or a single 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)

Apart from the usual comparison 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

⟨*cum*⟩$_1$ **for** ⟨*subtour*⟩$_1$ ⟨*cmp_op*⟩ ⟨*cum*⟩$_2$ **for** ⟨*subtour*⟩$_2$

where ⟨*cum*⟩$_1$ and ⟨*cum*⟩$_2$ are either aggregate functions applied to attribute expressions or the keyword "**number**". This constraint is satisfied by a tour if the cumulative property ⟨*cum*⟩$_1$ of ⟨*subtour*⟩$_1$ is related with the cumulative property ⟨*cum*⟩$_2$ of ⟨*subtour*⟩$_2$ via the comparison operator ⟨*cmp_op*⟩. 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)

A complete activity tour generation request is the following:

```
visit period is 20 Jul 92 - 10 Sep 92 &
cost < 500 for animal &
duration*interest >= 600 for plant &
site in ["Athens","Crete","Cyclades"]
    for general activity &
not closed days in ["Saturday","Sunday"]
    for geology &
avg(interest) > 8 for general activity &
sum(duration) > 120 and avg(cost) < 300
    for ancient history place &
max(religious place interest) >= 7
    for building with architectural place
    interest &
number >= 1 for middle age history place &
number = 1 for picturesque spot
    where interest > 5 &
number < 2 for architectural place &
number =< 1 for religious place, culture &
number \= 2 for holiday resort &
number between [2,4] for general activity &
min(cost) =< 300 or duration > 180
    for holiday resort &
sum(cost) for ancient history place =<
    sum(cost) for middle age history place &
number for modern year history place >
    number for nature.
```

The syntax and the semantics of the event tour generation requests are similar.

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. 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.

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 *Config-*

urator, the *Database Filter*, the *Tour Generator* and the *Tour Evaluator* submodules.

The Domains Creator partitions the appropriate tree, either activity or event, into *domains*. The Configurator produces all possible *configurations* of the solutions, taking into account the topological constraints. The Database Filter determines the candidate instances for the solution by rejecting ones that do not satisfy the local constraints. The Tour Generator produces all tours, according to the configurations, that satisfy the global and mixed constraints. Finally, the Tour Evaluator arranges the generated tours, taking into account some criteria, currently their average interest.

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 real-world database. 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 has to be employed. This methodology, as followed in 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 in a similar way to the partitioning into domains.

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 global, mixed and topological constraints that apply directly to the node are considered as well as the ones of the same types which are inherited from the ancestor nodes. As far as the cross constraints are concerned, actually their left and right hand sides are considered separately. The node into consideration is embedded into a domain among those that have been currently established, if the set of constraints that characterizes this domain is the same to the one that applies to the node. Otherwise, a new domain is created that contains this node. At the same time, the local constraints are also considered, both the ones that apply directly to the node and the ones that are inherited from ancestor nodes. The whole set of the local constraints that apply to the node is the criterion for embedding the node into a fine domain in its domain. An already generated fine domain may be used or a new one has to be created. The decision is based upon the comparison of the set of the local constraints that apply to the node with the sets of the local constraints that characterize the fine domains in the domain into consideration.

For simplicity reasons, the presented method does not take into account the where-properties that may refer to nodes. Actually, the method is correct, if instead of referring to nodes of the tree, partitioned nodes are considered, that is nodes refined by where-properties. For example, if a node nd is associated with the where-properties $wh1$ and $wh2$, then four parts of this node, that is four sets of instances, have to be considered. These parts refer to the node nd refined by the where-properties $wh1 \wedge wh2$, $\overline{wh1} \wedge wh2$, $wh1 \wedge \overline{wh2}$ and $\overline{wh1} \wedge \overline{wh2}$. This partitioning of the node nd is inherited to all its successor nodes, where it is possible to be combined with other where-properties which are directly associated with these nodes and to produce more fine partitions.

In order to demonstrate how the Domains Creator partitions the activity or event tree, consider the tree in Figure 1. Each one of the symbols $c1$, $c2$, $c3$, $c4$ and $c5$ represents either a simple non-local constraint (global, topological, mixed) or one of the left and right hand sides of a cross constraint (global, topological). The symbols $wh1$ and $wh2$ represent where-properties referring to the nodes $nd12$ and $nd13$ in association with $c4$ and $c5$ respectively. Only the case of partitioning this tree into domains is considered here, since the partitioning of domains into fine domains is similar.

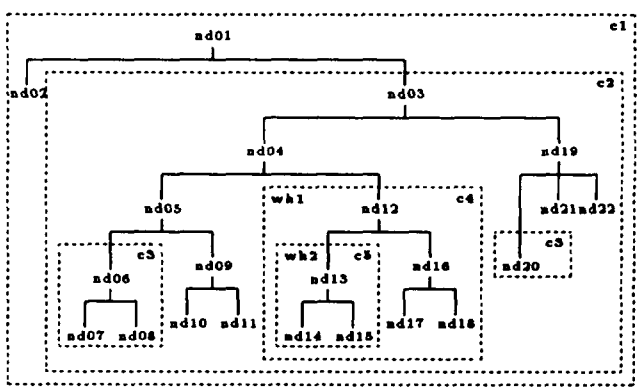

Figure 1: The constraints' application

Six domains are created by the Domains Creator, in this example. Each one of these domains contains a number of nodes, possibly refined by where-properties, and is related to a subset of the set $\{c1, c2, c3, c4, c5\}$. This

subset identifies the domain. The results are presented in Table 1.

doms	constrs	wh-props	nodes
$d1$	$c1$	—	$nd01, nd02$
$d2$	$c1, c2$	—	$nd03, nd04, nd05, nd09, nd10, nd11, nd19, nd21, nd22$
		$\overline{wh1}$	$nd12, nd16, nd17, nd18$
		$\overline{wh1} \wedge \overline{wh2}$	$nd13, nd14, nd15$
$d3$	$c1, c2, c3$	—	$nd06, nd07, nd08, nd20$
$d4$	$c1, c2, c4$	$wh1$	$nd12, nd16, nd17, nd18$
		$wh1 \wedge \overline{wh2}$	$nd13, nd14, nd15$
$d5$	$c1, c2, c4, c5$	$wh1 \wedge wh2$	$nd13, nd14, nd15$
$d6$	$c1, c2, c5$	$\overline{wh1} \wedge wh2$	$nd13, nd14, nd15$

Table 1: The partition of a tree into domains

The Configurator produces all possible configurations of the requested tours, either activity or event. 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, leading to the composition of the filtered database. 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. Such duplicates 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.

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. Firstly, a configuration is selected and 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. This procedure is repeated for every configuration. 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.

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.

ElipSys Exploitation

PETINA application is a complex one and involves a computationally intensive problem. Thus, although it profits from the expressive power of Prolog, it requires additional mechanisms that standard Prolog systems lack of. More precisely, as it processes a large amount of instances, grouping them together and exploiting the facility to process the groups in parallel can be very useful. What is more, is that in case alternative candidate tours are constructed and tested as soon as they are created, the execution time would decrease considerably. In addition, it is extremely hard for a standard Prolog system to solve any arbitrary system of equalities and inequalities on the domain of integers. Finally, the size of the database the system will actually consult exceeds the amount of information Prolog can handle efficiently. In this way, there is the need for a powerful system, such as ElipSys, to meet the above requirements. Most of the ElipSys features are exploited and are considered vital to make PETINA work.

More precisely, in the various submodules of TGE, the following exploitation is made. The Configurator generates and solves a system of linear equalities and inequalities. The system is solved by the ElipSys constraint satisfaction mechanism, which has proved to be indispensable. The topological constraints of the request are directly mapped into ElipSys built-in arithmetic constraints, thus assigning the whole computation to the internal ElipSys constraint solver.

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, in the sense that all elements of a set are processed in parallel. This is achieved by the *par_member/2* ElipSys predicate.

However, the main source of parallelism of the whole system exists in the Tour Generator. Initially, there are two levels of parallelism exploitation. 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, expressed in terms of the *par_member/2* and *par_select/3* ElipSys predicates. Data-parallelism is also exploited in the postprocessing of the generated tours. In addition, in this submodule, data driven computation is carried out using the delay mechanism of ElipSys. The employed test-and-generate method forces specific constraints to be tested as soon as the relevant subtour is created, although the whole tour is still under construction.

Finally, the ElipSys to ESQL interface will be exploited, when the relational database approach is followed. The Database Filter, the Tour Generator as well as the Tour Evaluator consult the database, so they'll exploit the interface.

Implementation & Performance Measurements

The first implementation of TGE was carried out in Sepia Prolog [15]. 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.

Currently, TGE is implemented in the ElipSys version 0.2 [3]. This version considers only one worker, so parallelism cannot be actually exploited. The submodules of TGE which involve parallelism, namely the Database Filter and the Tour Generator, were also implemented in the PEPSys language. PEPSys is a parallel logic programming language that supports OR- and restricted AND- parallelism. The data-parallelism facility of ElipSys, required by TGE, was simulated by the PEPSys OR-parallelism. The COKE preprocessor [9, 8], that allows to measure the 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.

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 TGE with respect to the one in Sepia. In most cases, the performance gain was several orders of magnitude.

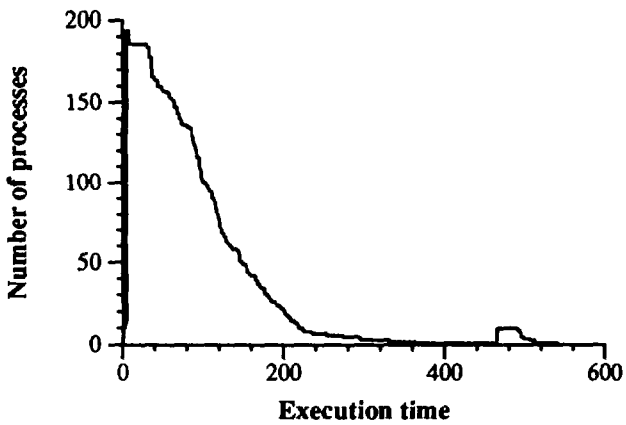

Figure 2: Database Filter graph

Finally, as mentioned above, parallelism was exploited in two submodules. For the presented complete request, the speedup achieved by the Database Filter was 40.11 *ni/et* (number of inferences / execution time) and the one by the Tour Generator was 825.41 *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 both submodules are presented in Figures 2 and 3.

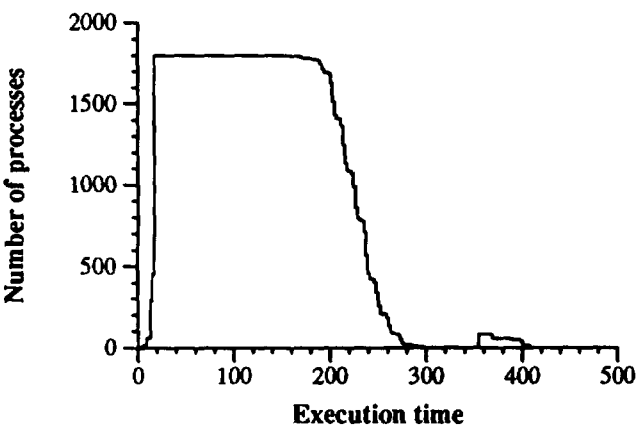

Figure 3: Tour Generator graph

Commenting on the performance of these submodules, where parallelism is used, the following might be mentioned. The main source of parallelism exists in the Tour Generator. As far as the Database Filter is concerned, the more balanced the partitioning into domains is, the more worthwhile the parallelism exploitation is.

Conclusions

In this paper, the most significant part of PETINA, that is its Tour Generation Engine, was presented. PETINA is a PErsonalized Tourist INformation Advisor about Greece consulting a large database that contains tourist data.

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 an interface to relational databases.

Although a tourist database for a whole country is addressed by PETINA and the search space of the tour generation problem is extremely large, 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 and the constraint satisfaction facilities of ElipSys, especially the latter one, were found to be indispensable. Finally, PETINA could not be considered as a real-life application, if an interface between ElipSys and relational databases did not exist. The amount of data that needs to be handled, exceeds the capabilities of standard logic programming systems.

References

[1] U. Baron, S. Bescos, S. A. 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] W. Clocksin and C. Mellish. *Programming in Prolog.* Springer Verlag, New York, 1981.

[3] *ElipSys User Manual for release version* 0.2, May 1991.

[4] G. Gardarin, P. Valduriez, M. Bèrard, L. Chen, O. Gerbe, M. Lopez, and J. Mondelli. ESQL: An Extended SQL with Object Oriented and Deductive Capabilities. Project Deliverable EDS.DD. 11B.0910, INFOSYS, December 1989.

[5] M. Garey and D. Johnson. *Computers and Intractability.* Freeman, New York, 1979.

[6] C. Halatsis, M. Katzouraki, M. Hatzopoulos, P. Stamatopoulos, I. Karali, C. Mourlas, M. Gergatsoulis, and E. Pelecanos. PETINA: PErsonalized Tourist INformation Advisor. Project Deliverable EDS.WP.9E.A005, University of Athens, December 1990.

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

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

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

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

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

[12] 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.

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

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

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

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

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

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