SICS Technical Report T2000:13
SICS-T--2000/13-SE

ISSN: 1100-3154

The Use of Abstractions to Solve Large Scheduling Problems

by

Per Holmberg

Swedish Institute of Computer Science
Box 1263, S-16429 Kista, SWEDEN

M. Sc. Thesis

The use of abstractions to solve large scheduling problems

Per Holmberg

Advisor: Per Kreuger, piak@sics.se
Examiner: Seif Haridi, seif@sics.se

Abstract

This thesis investigates how abstractions can be used to improve performance in a railroad scheduling system that uses constraint programming. The idea behind abstractions is to solve a large problem in smaller parts and extract information from these parts. That information can then be used when solving the entire problem.

Two different types of abstractions are introduced: Relations and Net abstractions. The use of relations builds orders between trips or parts of trips. These orders can be used to reduce the search necessary to find a solution to the scheduling problem. When using net abstraction, the problem is solved in an abstract search space, where it is easier to solve. The solution computed in the abstract search space is then used to reduce search when solving the problem in the original space.

It is shown that these two types of abstraction can improve performance in problems with various settings. Relations can successfully be used in problems that have few solutions and are hard to solve. Net abstraction on the other hand works best for problems with many valid solutions.

Table of contents

1 Introduction

This section gives a brief introduction to the report. The goals of the thesis and an outline of the following chapters are presented.

The aim of this thesis is to investigate and implement abstraction methods in a railroad scheduling system. The system, TUFF (TågplaneUtveckling För Framtiden – Train planning development for the future) features the following functionality:

- Train scheduling
- Vehicle routing
- Personnel scheduling

Train scheduling is the process of determining departure and arrival times for each train. For the convenience of passengers, passenger trains often have periodic timetables, whereas freight transports are schedule according to the customers' demands. TUFF uses technology from the field of Constraint Programming to perform train scheduling.

Vehicle routing determines how locomotives and cars are assigned to trains, that is, which locomotive and which cars should form a certain train on the trip from a station to another.

Personnel scheduling determines how to assign drivers and other personnel to a train. In this process, it is necessary to take into account things like breaks, costs for overtime, etc. The mechanisms for personnel scheduling are currently being developed in TUFF.

TUFF is more thoroughly described in Section 5 The TUFF system.

1.1 Goals

More precisely, the goals of this thesis were to:

- Investigate how relations between trips and parts of trips (slots) may be used in the train scheduler of TUFF.
- Implement relations and evaluate if the use of relations can improve performance and/or reduce memory consumption.
- Design, implement and evaluate functionality for net abstraction in the train scheduler. A net abstraction is to take a set of tracks, paths and trips, and transform them to a more abstract level, with fewer details.

1.2 Outline of the report

Section 2 Mathematical foundations presents the mathematics necessary for this work.

Section 3 Constraint programming gives an introduction to constraint programming. It has no intention of being a complete description. Instead, its purpose is to give the basic understanding of constraint programming necessary for this work.

In Section 4 Fundamentals of abstraction, a background to the concept of abstraction is presented. Abstractions have been studied in AI for some time, and some of the results and definitions of that research are presented in this section.

Section 5 The TUFF system describes the techniques used in TUFF more thoroughly.

In Section 6 Abstractions in TUFF, the abstractions that have been designed and implemented are presented, and in Section 7 Tests and results, they are evaluated.

Section 8 Conclusions and future work, summarizes the thesis, and presents future work that is not covered by this thesis.

In the appendixes test examples and a definition of a script language, TUFFScript can be found. The script language is a tool for combining abstraction methods. However, implementation of TUFFScript is beyond the scope of this thesis.

2 Mathematical foundations

This section introduces the mathematical definitions needed for this work. These are standard definitions, which can be found in any textbook on discrete mathematics. See for instance [PY73].

Let **S** be a set with N elements. Let the elements be T_1, T_2, … T_N, which can be thought of as departure or arrival times. If there is no order between any of the times, i.e. there is no order between T_i and T_j for any i and j, then **S** can be represented by

$$\mathbf{S} = \{T_1, T_2, \ldots T_N\}$$

To be able to state properties of a set, a few further definitions are needed.

Definition 2.1: Reflexivity. A relation α *on a set* ***S*** *is said to have the reflexive property (to be reflexive) if, for every* $s \in S$, $s \alpha s$.

Definition 2.2: Antisymmetry. A relation α *on a set* ***S*** *is said to have the antisymmetric property (to be antisymmetric) if, for* ***a*** *and* ***b*** *in* ***S****,* $\boldsymbol{a} \alpha \boldsymbol{b}$ *and* $\boldsymbol{b} \alpha \boldsymbol{a}$ *imply* $\boldsymbol{a}=\boldsymbol{b}$.

Definition 2.3: Transitivity. A relation α *on a set* ***S*** *is said to have the transitive property (to be transitive) if, for* ***a****,* ***b****, and* ***c*** *in* ***S****,* $\boldsymbol{a} \alpha \boldsymbol{b}$ *and* $\boldsymbol{b} \alpha \boldsymbol{c}$ *imply* $\boldsymbol{a} \alpha \boldsymbol{c}$.

Definition 2.4: Partial ordering. A partial ordering is a reflexive, antisymmetric and transitive relation on a set.

This work deals with sets where some of the elements – but not all – are ordered. It is therefor necessary to introduce the partly ordered set (poset):

Definition 2.5: Partly ordered set. A partly ordered set (poset) consists of a set ***S*** *and a partial ordering relation* $\leq$ *on* ***S****. A poset is usually denoted by the pair* $[\boldsymbol{S}, \leq]$*, which also denotes the graph of the relation* $\leq$ *on* ***S****.*

Most interesting when considering this work is a special case of posets, chains.

Definition 2.6: Chain. A chain is a poset where any pair of elements is comparable. A chain ***C*** *with the elements* $T_1, T_2, \ldots T_N$ *is represented by* $\boldsymbol{C} = < T_1, T_2, \ldots T_N >$

A notation for sets of chains will also be needed. Let **S** be a set containing the chains $\mathbf{C_1}$ and $\mathbf{C_2}$. $\mathbf{C_1}$ is $< T_1, T_2, T_3 >$ and $\mathbf{C_2}$ is $< T_4, T_5 >$. The notation

$$\mathbf{S} = \{\mathbf{C_1}, \mathbf{C_2}\} = \{< T_1, T_2, T_3 >, < T_4, T_5 >\}$$

states that in the set **S**, there is an order between T_1, T_2 and T_3. There is also an order between T_4 and T_5, but there is no order between any of the elements in $\mathbf{C_1}$ and the elements in $\mathbf{C_2}$.

In some cases, all elements in a set will be ordered. The set is then called a *totally ordered set*. The representation of such a set **S** with three elements T_A, T_B and T_C will be **S** = {<T_A, T_B, T_C>}. The definition of totally ordered set is from [Col99].

*Definition 2.7: Totally ordered set. Given a reflective relation **R** for a set **S**, such that for every pair of elements **a**, **b** in **S** either **a R b** or **b R a**, there is a one-to-one function **f** that preserves the relations on **S** in both ways, i.e. a* **R** *b* $\Leftrightarrow$ *f*(**a**) < *f*(**b**).

Furthermore, there is a need to investigate the structure of a set. An interesting structure is the *lattice*.

*Definition 2.8: Join. For **a** and **b** in a poset [**S**, ≤], a join of **a** and **b** is an element **c** of **S** which satisfies the relationships **a**≤**c** and **b**≤**c**, and there is no other **x** in **S** for which **a**≤**x**≤**c** and **b**≤**x**≤**c**. If two elements have a unique join, the latter is denoted by (**a**∨**b**).*

*Definition 2.9: Meet. For **a** and **b** in a poset [**S**, ≤], a meet of **a** and **b** is an element **d** of **S**, which satisfies the relationship **d**≤**a** and **d**≤**b** and there is no other x in **S** for which **d**≤**x**≤**a** and **d**≤**x**≤**b**. If two elements **a** and **b** in **S** have a unique meet, the latter is denoted by **a**∧**b**.*

*Definition 2.10: Lattice. A lattice is a poset [**S**, ≤], any two elements of which have a unique join and meet. The symbol for a lattice is [**S**,∨, ∧].*

3 Constraint programming

Since one of the goals of this work is to speed up the scheduling process, some knowledge of how this process works is necessary. The train scheduler uses techniques from the domain of Constraint Programming (CP). This section gives an introduction to the concept of CP. It is by no means intended to cover all issues dealt with in CP, but merely to give the basic understanding of CP necessary for this work. It relies heavily on [Tsa93] and [SS99].

3.1 Basic concepts

A Constraint Satisfaction Problem (CSP) consists of three parts:

1. A finite set of variables
2. A domain associated with each variable
3. A set of constraints restricting the values that the variables can take simultaneously.

When solving a CSP, the task is to assign values to all variables in such a way that no constraint is violated, and the value for each variable is within the domain of the variable.

Example 3.1: Send more money
This is a classical example of a CSP. The story goes that a boy was out on a journey, and ran out of money. He then sent his parents a postcard, with the equation

```
      S    E    N    D
+     M    O    R    E
---------------------------------
M     O    N    E    Y
```

The parents now had to solve this equation to find out how much more money the boy needed. Each of the variables (D, E, M, N, O, R, S, Y) should have a value between one and nine, and no two variables should have the same value.

It is easily shown that this problem is a CSP. The three parts are:

1. The finite set of variables is {D E M N O R S Y}.
2. The domain is the same for all variables: $0 \leq x \leq 9$, where x is the variable.
3. The constraint is that no two variables can take the same value simultaneously.

In Section 3.6 Constraint programming in Mozart/Oz, it is shown how this problem is solved with CP in Mozart/Oz.

3.2 Formal description of a CSP

This section gives a more formal description of a CSP. It does by no means try to cover the entire field of descriptions in CSPs. Instead, its purpose is to present the definitions necessary for this brief introduction.

Definition 3.1: Domain. The domain of a variable is the set of all values that may be assigned to the variable. D_x denotes the domain for the variable x.

The values in a domain do not have to be restricted to numbers. In some problems, it may be convenient to use other types of values. Consider for instance a coloring problem, where a number of countries on a map should be colored. No two neighboring countries should have the same color. In such a problem, it is natural to use colors instead of numbers in the domains.

Definition 3.2: Label. A label is a pair of a variable and a value. It represents the assignment of that value to the variable. A label that assigns the value v to the variable x is denoted by <x, v>. <x, v> is only meaningful if v is in the domain of x, that is, $v \in D_X$.

Definition 3.3: Compound label. A compound label is the simultaneous assignment of values to a set of variables. The compound label of assigning $v_1, v_2, \ldots, v_n$ to $x_1, x_2, \ldots, x_n$ is denoted by ($<x_1, v_1>, <x_2, v_2>, \ldots, <x_n, v_n>$).

A constraint can be seen as a set that contains all legal compound labels for the variables in the CSP. However, in practice constraints can be equalities, inequalities, or other types of relations. Regardless of how the constraint is represented, the constraint represents a number of legal compound labels for the problem variables.

Definition 3.4: Constraint. A constraint on a set of variables is conceptually a set of compound labels for the variables in the problem. A constraint on the set of variables S is denoted C_s.

Definition 3.5: Satisfaction. Assume there is a compound label $X=(<x_1, v_1>, <x_2, v_2>, \ldots <x_n, v_n>)$ *and a constraint C.*
X is said to satisfy the constraint C if the variables in X and C are the same, and X is an element of C, that is, $(<x_1, v_1>,<x_2, v_2>, \ldots, <x_n, v_n>) \in C_{x1, x2, \ldots, xn}$.

Definition 3.6: Constraint satisfaction problem (CSP). A CSP is a triple (Z, D, C), where

Z is a finite set of variables $\{x_1,x_2, \ldots, x_n\}$

D is a function, which maps every object in Z to a set of objects of arbitrary type, $D:Z \rightarrow$ *finite set of objects (of arbitrary type).* D_{x_i} *is the set of objects obtained by mapping from* x_i *using D. These objects are the possible values of* x_i*, and* D_{x_i} *is the domain of* x_i.

C is a finite set of constraints on a subset of the variables in Z, that is, C is a set of compound labels.

3.3 Solving a CSP

When solving a CSP there are two categories of methods: *Problem reduction* and *Search.* The type of problem at hand determines which category and method that is most feasible. The best solving method is usually not a single one of these methods. Instead, the most efficient way of solving a CSP is often to combine the two techniques. Which the optimal solution method is also depends on whether one or all solutions are required.

3.4 Problem reduction

The idea of problem reduction is to make the problem smaller by reducing the domains of the variables. The CSP is then hopefully easier to solve or recognize as insolvable. Problem reduction alone does not normally solve a CSP. However, when it is combined with Search, it can reduce the effort necessary to solve the CSP.

Problem reduction is made by constraint propagation, which is a way to limit the domains of the variables of the CSP. There are two types of propagation: *domain propagation* and *interval propagation.* Consider an example:

Example 3.2: Domain and interval propagation.
Assume that there is a CSP with two variables X and Y. Let the domains of X and Y be

$D_x = \{1, 2, \ldots, 10\}, \; D_y = \{1, 2, \ldots, 7\}$

The constraint of X and Y is C_{xy}*:* $2X=Y$.
If domain propagation is used, then the domains are narrowed as much as possible. The new domains becomes

$D_x = \{1, 2, 3\}, D_y = \{2, 4, 6\}$

However, if interval propagation is used, only the bounds of the intervals are narrowed. Thus, the new domains become

$$D_x = \{1, 2, 3\}, D_y = \{2, 3, 4, 5, 6\}.$$

Normally interval propagation is used, because of its lower computational cost.

In most problems, it is not optimal to perform a complete problem reduction, since it is expensive. Instead, it is usually preferable to do some reduction, and then use Search methods to solve the CSP.

3.5 Search

A lot of research effort has been made to find good search strategies for various problems, and there are various techniques in the literature. This section tries to give some insight and basic understanding in concept of search in CSPs. For a more extensive overview of search techniques, see [Tsa93].

3.5.1 Backtracking

A simple but widely used technique for search is *simple backtracking.* The idea is to consider one variable at a time, and try to assign a value to that variable. The algorithm is simply as follows:

1. Pick one of the variables in the CSP.

2. Choose a value, which is in the domain for this variable. Check if this value satisfies the constraints. If it does, go to step 1 and pick another variable. If it does not, choose another value.

3. If there are no values that satisfies the constraints for a variable, go back to the variable that was assigned a value last. Change the value of that variable, and try again to assign the current variable a value.

4. This continues until a solution has been found or all the combinations of labels have been examined and failed. Figure 3.1 shows the idea of backtracking.

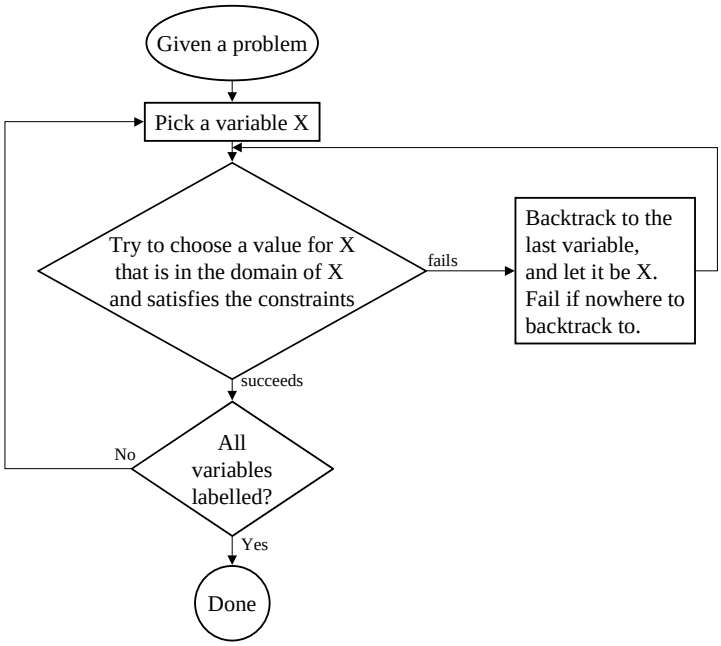

Figure 3.1: Backtracking. The backtracking algorithm tries to label all variables, and backtracks if there is no possible value to assign a variable.

The basic algorithm for backtracking above is normally combined with problem reduction to improve efficiency. This means that in step 2, after a value has been chosen for a variable and it has been checked that the value satisfies the constraints, problem reduction is performed. An attempt is made to reduce the domains of all variables. If a domain of a variable is reduced, some other variable may be affected by this reduction and the domain of that variable may also be reduced.

3.5.2 Labeling

A question that arises is how to choose the variable that should be labeled and how to label the variables in such a way that backtracking is avoided? There are a number of approaches to this problem. Some possible strategies for choosing which variable to label are:

- **The naive strategy:** Choose the first variable in an arbitrary ordering of the variables. This strategy performs very poorly as a general method.

- **First fail strategy:** Choose the variable that is most likely to fail. This strategy aims at recognizing dead-ends as soon as possible, and thereby reduce the amount of computation. One simple way of determining which variable is most likely to fail is to compare the size of the domains of the variables. This approach is used in the constraint programming language CHIP with impressive results [Tsa93].

- **Minimal width ordering:** This strategy can be used in CSPs where some variables are constrained by more variables than others. The idea is to first choose the variable that is constrained by the largest number of other variables. Since the last variables to be labeled are those that affect the smallest number of other variables, the chances of avoiding backtracking are good.

Once a variable has been chosen, it must be determined what value to assign to that variable. There are also in this case a number of standard techniques. A simple approach is to label the variable with the lowest, highest or middle value in its domain. Another technique is to decrease the size of the variable's domain instead of assigning it a value. In this case, the new domain of the variable is set to the lower or upper half of the original domain. This may reduce the number of backtracks necessary.

3.5.3 Search spaces

The search space is the space of all combinations of compound labels and unlabeled variables. Different search strategies give different search spaces. The backtracking algorithm searches the space of all compound labels. This makes the search space look like in Figure 3.2.

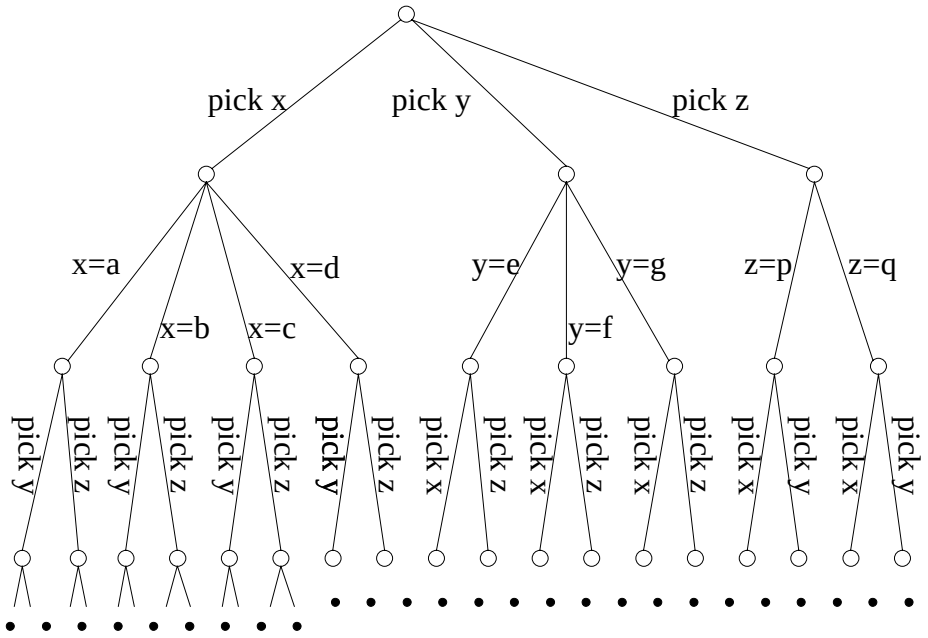

Figure 3.2a: Search space of the backtracking algorithm. The variables in the CSP are Z={x, y, z}, and the domains D_x={a,b,c,d}, D_y={e,f,g} and D_z={p,q}. Note that the entire tree is not shown – the bottom row is left out.

The search space is highly dependent on the order in which the variables are labeled. If the CSP from Figure 3.2a is taken as example, then either x, y or z can be labeled first. If the variables are labeled in the order (x,y,z), the search space will look like in Figure 3.2b. However, if they are labeled in the order (z,y,x), the search space looks different. This search space can be seen in Figure 3.2c.

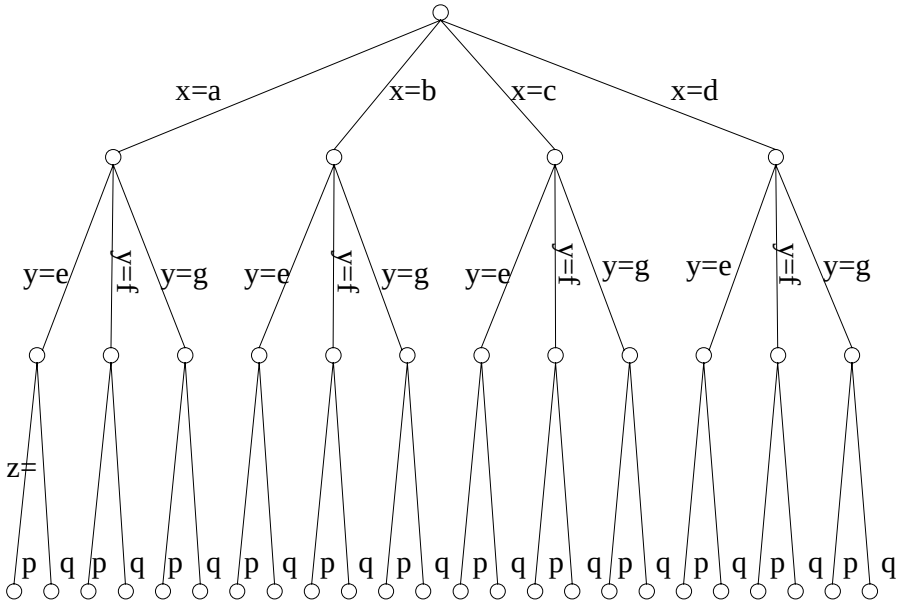

Figure 3.2b. Search space of the backtracking algorithm when the variables are labeled in the order (x,y,z).

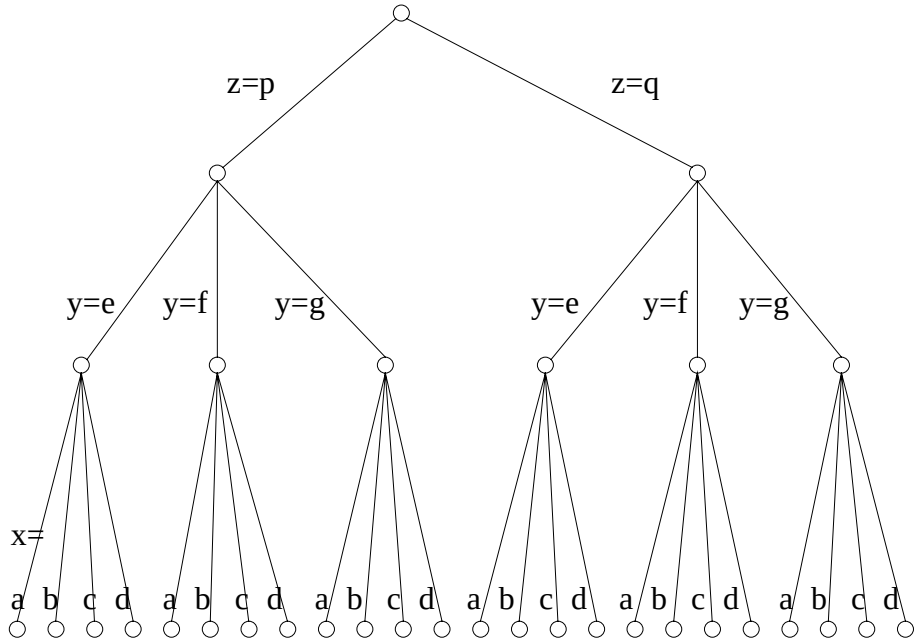

Figure 3.2c. Search space of the backtracking algorithm when the variables are labeled in the order (z,y,x).

3.5.4 Characteristics of search spaces

When determining how to approach a certain CSP, it is necessary to get some structured overview of the problem. The properties listed below provide a great help

in this process. Based on these properties, specialized search techniques have been developed that solve CSPs more efficiently.

The properties that have to be considered are:

- **The size of the search space is finite**.
  As can be seen when comparing Figure 3.2.b and 3.2.c, the number of nodes in the search tree depends on the order in which the variables are labeled. However, the number of leaves in the tree, i.e. the number of nodes at the bottom of the tree, is always the same. The number of leaves for a CSP with the variables Z={X_1, X_2, ..., X_N} is $L = \left|D_{X_1}\right| * \left|D_{x_2}\right| * \ldots * \left|D_{x_N}\right|$.
  This is also the term that dominates the size of the entire search tree. The size of the entire tree, counted in number of nodes, is [Tsa93]:
  $$N = 1 + \sum_{i=1}^{N} (\left|D_{X_1}\right| * \ldots * \left|D_{X_i}\right|)$$

- **The depth of the tree is fixed**.
  When the order in which the variables are labeled is fixed, as in Figure 3.2b and 3.2c, the depth of the search tree is equal to the number of variables. If no order is specified, as in Figure 3.2a, the depth is two times the number of variables.

- **Subtrees are similar**
  As can be seen when comparing Figure 3.2a, 3.2b and 3.2c, the subtrees are similar regardless of how variables are ordered. This fact can be exploited to construct more efficient search algorithms. For instance, learning algorithms can be created.

3.5.5 Combining search and problem reduction

In most cases, it is more effective to combine search and problem reduction than to try to use only one of these methods. The more compound labels that are removed using problem reduction, the fewer backtracks will be done during the search. However, it is normally not effective to perform a complete problem reduction. The computational cost increases as the number of values that are possible to remove decreases. An approximate relation between search and problem reduction can be seen in Figure 3.3. As can be seen in the figure, it is often important to find a balance between the two methods to be able to minimize the computational cost.

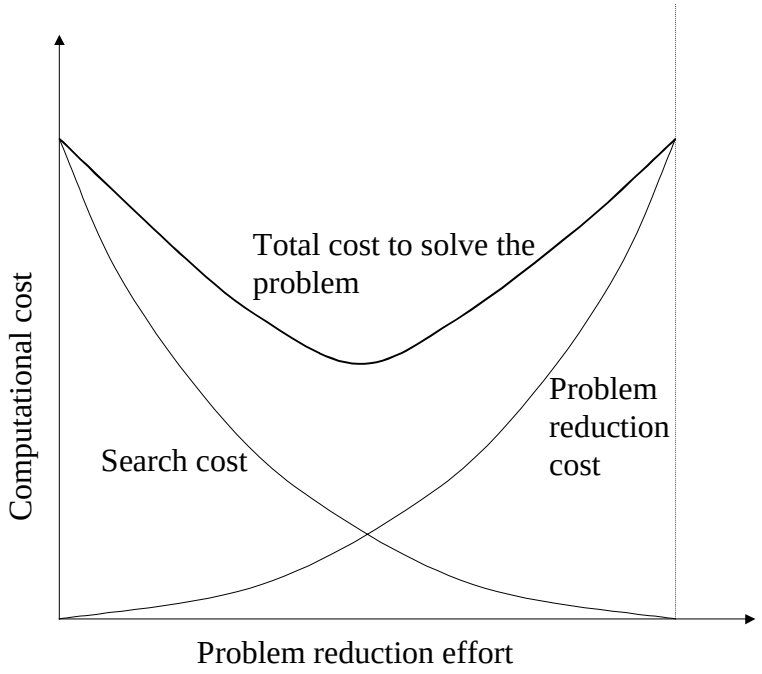

Figure 3.3: Search and problem reduction costs. The optimal way to solve a CSP is often to combine the two methods.

3.6 Constraint programming in Mozart/Oz

This section presents a program written in Mozart/Oz solving the CSP introduced in Example 3.1, “Send More Money”.

The program uses one variable for each letter. This means that there are eight variables. The search mechanism uses the first-fail method described above. For this problem, there are three constraints created:

1. There may be no leading zeros, i.e. S and M must not be zero.

2. Two variables must not have the same value.

3. The following sum must be correct:

```
    S   E   N   D
+   M   O   R   E
-------------------
M   O   N   E   Y
```

The program listing (from [Ss99]) is:

```
proc {Money Root}
S E N D M O R Y
in

      Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y)  % 1
      Root ::: 0#9                                 % 2
      {FD.distinct Root}                           % 3
      S \=: 0                                      % 4
      M \=: 0
      1000*S + 100*E + 10*N + D                    % 5
      +    1000*M + 100*O + 10*R + E
      =: 10000*M + 1000*O + 100*N + 10*E + Y
      {FD.distribute ff Root}                      % 6

end
```

The commented lines are:

%1: These are the labels and variables in the problem.

%2: Specifies that all values must be between zero and nine.

%3: All values must be distinct. This means that no two values may be equal. The line implements constraint 2 above.

%4: *S* and *M* must not be equal to zero. These lines implement constraint 1.

%5: The sum must be correct. This is constraint 3.

%6: This line tells the system to start solving the problem, using the first-fail method described above.

Since there are 8 variables and each variable has a domain of 10 values, there are 10^8 possible leaves in the search tree. This means that a completely naive algorithm, that investigates all possible combinations of labels, would have to do a lot of work to find the solution. Problem reduction combined with the first-fail principle, on the other hand, manages to solve the problem creating only the small search tree showed below in Figure 3.4.

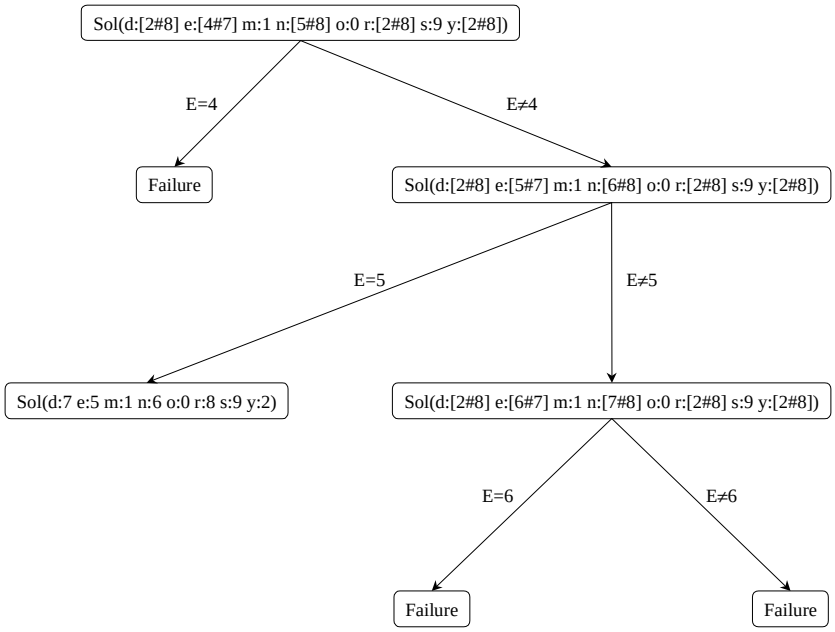

Figure 3.4: Search tree for "Send more money" using the first-fail principle. The root node is obtained by performing problem reduction on the specified domains. For each new node in the tree, problem reduction is performed.
The notation ***x****:[y#****z****] specifies that the variable* ***x*** *has a domain ranging from* ***y*** *to* ***z****.*

4 Fundamentals of abstraction

Abstraction is the process of taking a problem from its original problem space to some simpler, abstract space. The idea is that it should be easier to solve the problem in the abstract space, and that the abstract solution should be used when solving the original problem. This often reduces the effort to solve the original problem.

In the area of artificial intelligence, AI, abstraction has been thoroughly studied. A large number of problem-solving systems that use the idea of abstraction have been implemented and studied. Examples of such systems, presented in [Yan97], are GPS, ABSTRIPS, LAWLY, NOAH, NONLIN, MOLGEN, SOAR, SIPE and ABTWEAK.

In this work, an attempt is made to use the ideas of abstraction for scheduling problems instead of AI problems, *planning problems*. The scheduling problem and the planning problem are somewhat different.

In scheduling, the task is to allocate the resources necessary for some action. This must be done in such a way that the result is a valid schedule. Additionally, it might be desirable to achieve a schedule for which some cost function is minimized.

In planning, there are normally specifications for the effect that an action has to the outside world. This outside world is modeled by the use of states, and each action changes the state of the outside world. There may be preconditions specifying that a certain action may only be performed from certain states.

This section gives a brief description of what has been done in the field of abstraction in AI. In AI, the problems to be solved are often referred to as *plans*. Therefor, the term *plan* will be in this section.

4.1 Abstraction hierarchies

There are several approaches when creating an abstraction. A common way is to use an *abstraction hierarchy*. Such a hierarchy reaches from the original *concrete* plan, possibly through several intermediate levels of abstract plans, to the most abstract plan. This creates an abstraction hierarchy, which may look as in Figure 4.1 (from [Yan97]). The figure shows that on the highest, most abstract level, the plan to solve is relatively small, whereas on the concrete level, the plan is large.

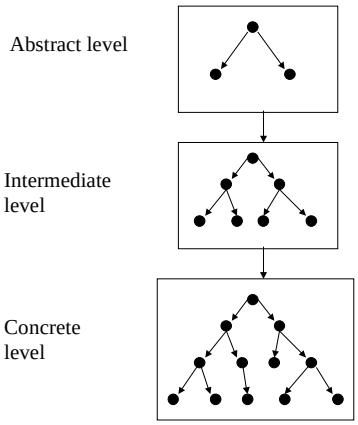

Figure 4.1. An abstraction hierarchy, reaching from an abstract level down to the concrete level where initial problem resides.

The process of obtaining a solution is to

- Solve the plan on the highest, most abstract level
- *Refine* it to account for the missing components when taking it to the next, lower level (for a further discussion of refinement, see below), and
- Iterate until the concrete level is reached.

An example of the use of abstraction hierarchies is a simple routing problem:

Example 4.1: Total order. Suppose one wants to drive from a street S_X in city X to the street S_Y in city Y. It might then be necessary to drive through other cities on the way, and thus find ways to get out of these cities on the right exit.

An abstract plan would then correspond to an inter-city route, that is $<City_x => City_z => City_u => City_y>$. *The concrete plan also features all intra-city information. It would mean that the abstract plan could be expanded to* $<City_x => City_xStreet_{x1} => City_xStreet_{x2} => ... => City_xStreet_{xN} => City_z => City_zStreet_{z1} => ... => City_uStreet_{uN} => City_y>$.

Example 4.1 shows an example of a situation where all steps are ordered. Each step has to be either before or after every other step. Situations like this, where all steps are ordered with respect to every other step, are called *total orders*.

An example of a situation where there is no total order is found in example 4.2. Here, there is an order between a step and some, but not all, of the other states. Such a situation is called a *partial order*.

Example 4.2: Partial order. When getting ready to go to work in the morning, several steps must be performed. Suppose you want to take a shower, get dressed and have breakfast. Each of these steps can be subdivided into more concrete steps, which means that abstractions can be used when planning your morning activities. However, in this example there is no total order between each step. It is impractical to get dressed before taking a shower, so getting dressed must be done after taking a shower. But there is no reason for an order between taking a shower and having breakfast. Therefor only some of the steps are ordered, and the plan is not totally, but partially ordered.

4.2 Abstraction methods

When the decision has been made to use a hierarchical abstraction model, there are several approaches to how to build this model. This section presents a short overview of various abstraction methods. Further details can be found in [Kno94].

4.2.1 State abstraction

If State abstraction is used, a hierarchy of abstraction spaces is introduced. For each abstract level, the plan is somewhat simpler than on the previous, more concrete level, as in Figure 4.1 above. The problem is first solved in the most abstract space. It is then refined at successively more detailed levels. The solution from a more abstract level can be used when solving a plan on a more concrete level.

4.2.2 Abstract operators

An alternative approach which, is quite similar to state abstraction, is to use abstract operators. As before, an abstraction hierarchy is introduced. On each level there are a number of operators – rather few on the most abstract level, and considerably more on the concrete level. For each hierarchical step, the number of operators and the detail of each operator are increased.

It should be noted that the difference between abstract operators and state abstraction is small. In fact, as shown in [Kno94], abstract operators can be used to implements state abstraction.

The routing situation in Example 4.1 above shows the use of abstract operators.

4.2.3 Macro operators

When the concepts of state abstraction and abstract operators are combined, the result is macro operators. The idea is to combine several operators into a macro operator. A number of such macro operators together form a *macro problem space*, and the abstraction hierarchy is built of a number of such spaces. Once the problem has been solved in a macro problem space, the concrete problem is solved, since a macro operator encapsulates the effect of several operators.

4.3 Properties of abstractions

Does the existence of a solution to a plan on some level guarantee the existence of a solution at some other hierarchical level? [Yan97] defines two properties that deal with this issue: Upward solution property and Downward solution property.

4.3.1 Upward solution property

Consider an abstraction hierarchy, where there are one or more abstract plans P_A and one concrete plan P_C. If the existence of a solution to a concrete plan P_C implies the existence of a solution to an abstract plan P_A, then the hierarchy satisfies the *upward solution property*.

In the definition of the upward solution property, the levels are numbered from 0, as the concrete level, up to N, which is the most abstract level, as shown in Figure 4.2.

Definition 4.1: Upward solution property. Whenever any i-th-level solution P_i exists, there exists an abstract solution P_{i+1} at level i+1

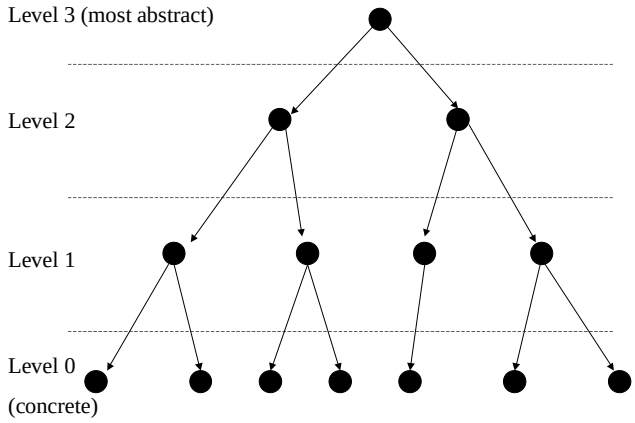

Figure 4.2. The abstraction levels for the definition of upward solution property.

If an abstraction hierarchy fulfills the upward solution property, and there is no solution for a problem at an abstract level, there is no solution at any lower level either. In such cases, a top-bottom approach is a good way to traverse the hierarchy. If there is no solution at a certain abstract level, then there is no need to continue the search at lower levels. It can be stated that there is no solution simply by solving an abstract level plan.

4.3.2 Downward solution property

If a hierarchical structure fulfills the condition that whenever there is a solution on an abstract level, there exists a solution at the concrete level, then the hierarchical structure fulfills the *downward solution property*.

Definition 4.2: Downward solution property: If there exists a solution to a plan P_i on any level i higher then 0, then there is a solution to a plan P_0 on the concrete level.

If this property is fulfilled, and there exists a solution at any abstract level, then there exists a solution at the concrete level. That is, as soon as a solution is found on an abstract level, the downward solution property guarantees that there is a solution to the concrete plan. In addition it follows from this property that if there is no solution at the concrete level, there are no solutions at any higher level either.

4.4 Refinement methods

The upward- and downward solution properties deal with the existence of solutions in a useful way. But they do not say anything about the exact relationship between two solutions at different levels. No matter which abstraction method is used; when an abstract plan is taken to a lower level, it must be extended by additional steps.
This extension is called *refinement* and can be done in several different ways.

Below two different types of refinement are discussed: *Forward-chaining, total-order refinement* and *backward-chaining, partial order refinement.* A property of the refinement, *monotonic refinement*, is then presented.

4.4.1 Forward-chaining, total order refinement

The idea behind forward-chaining, total order refinement is to introduce gaps when taking a plan from a higher level down to a lower, more concrete level. When the plan is refined at the lower level, every gap is filled with new steps.
Figure 4.2 (from [Yan97]) shows an example of how gaps are introduced and filled with additional steps as a plan is taken to a lower level.

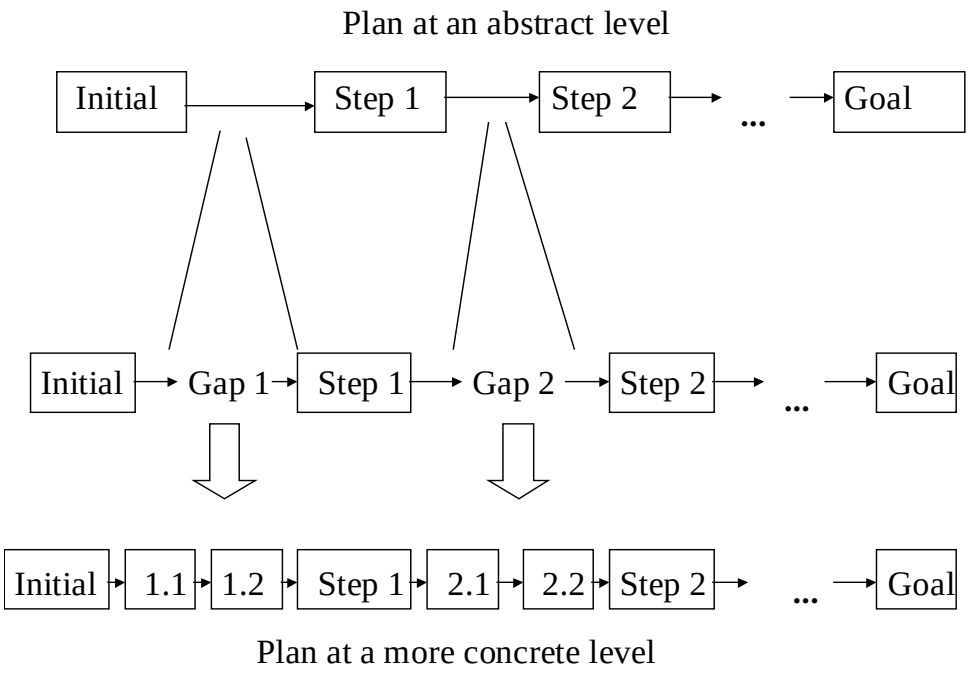

Figure 4.2. In forward-chaining, total order refinement a gap is introduced between every pair of steps when a plan is taken to a lower level. During the refinement process, these gaps are filled with new, additional steps.

The gaps are filled using a length-first method starting from the Initial-state at the left and advancing towards the Goal-state at the right. In the construction of state 1.1, the preconditions that must be considered are specified in the Initial-state.
When advancing to state 1.2, the preconditions are obtained by applying the plan <Initial => 1.1>. For step j, the preconditions are obtained by applying <Initial => 1.1 => ... => j-1>.

This process continues until all gaps have been filled with new steps, and the Goal-state is reached.

4.4.2 Backward-chaining, partial order refinement

For partially ordered plans, the approach is somewhat different. Assume there is a partially ordered abstract plan P. When taking P to a lower abstraction level, each step is replaced by its corresponding step on the lower level. An abstract step may have several corresponding steps on the next lower level. If this is the case, one of these steps is chosen, and the others are saved. If the plan is impossible to solve, it is then possible to go back and check if one of the saved steps work.

The refinement process is then to plan the lower level in such a way that the preconditions of each step are fulfilled.

Figure 4.3 shows the process of backward-chaining, partial order refinement.

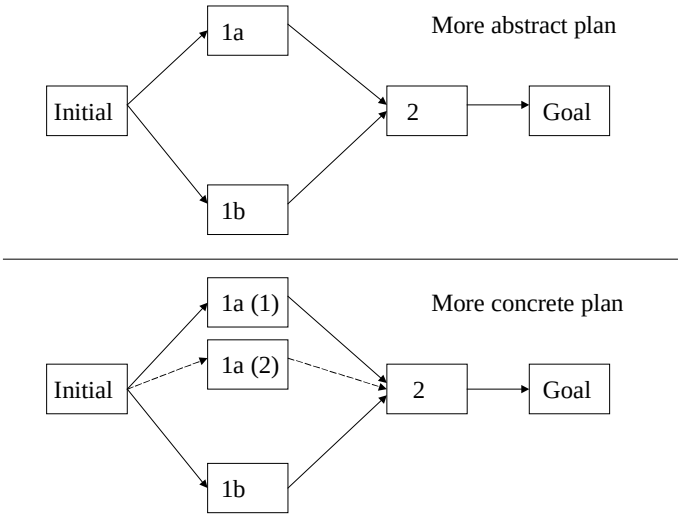

Figure 4.3: Backward chaining, partial order refinement. Both step 1a and 1b must be executed before step 2 can be executed. Each step in the abstract plan is replaced by a corresponding step in the more concrete plan. If more than one step correspond to an abstract step, one of these steps is chosen. In the figure, 1a (1) has been chosen to replace 1a, and 1a (2) is saved if solving the more concrete plan would fail.

4.4.3 Monotonic refinement

A refinement that preserves the order between all steps calculated on a higher abstraction level, and where no steps from the abstract plan are deleted or moved is called a *monotonic refinement*. When using a monotonic refinement, a plan will never decrease in size when it is taken from one level to a lower, more concrete level.

Example 4.1 above (on page 17) fulfills the monotonic refinement property.

5 The TUFF system

5.1 Definitions

In order to make the TUFF system familiar to railroad planners, most of the terms are originally railroad terms. However, sometimes a term has been used to denote several different things. In these cases, to avoid ambiguity, a new term has been introduced in the system.

The terms used in the system are explained in text, and also presented graphically below.

Station A station is a location where trains may be turned or shifted to a new track. In addition, stations are the only places where a train may overtake another train.

Headway Headway is the minimum time period between two trains going in the same direction on a track, i.e. the safety distance in time.

Track A track is the physical connection between two locations. Once a train has started on a track, there is no way of turning back until the end of the track is reached.
Tracks may be double tracks or single tracks. If the track is a double track, the different directions of the track have individual ids (and can be considered two different tracks). If there for instance are two double track between A and B with ids T_1 and T_2, then T_1 will go from A to B and T_2 from B to A.

Path A path is an ordered series of tracks connecting a start and an end station. For instance, the path between Stockholm and Gothenburg may be
(T_1 -> T_2 -> T_3 … T_N), where T_i are tracks.

Trip A trip is a train traversing a path with pre-defined arrival and departure times. These times may also be intervals.

Slot A slot is the minimal part of a trip. Several slots, placed in order, form a trip. Each slot represents the traversal of a track and the stop at a station. For instance, a trip departing from Stockholm at 12.04 towards Gothenburg may be (S_1 -> S_2 -> S_3 … S_N), where S_i are slots.

Tripset A tripset is a set of trips and corresponding slots. It is used as problem specification to the schedulers of the TUFF system. The result of a scheduling is also stored in a tripset.
Tripsets are the basic building blocks in the TUFF system. They are modeled as single assignment structures, i.e. once a tripset has been instantiated, it can not be changed. All operations in the system are done on tripsets.

Net A net is built from locations, tracks and paths. It determines how these elements are connected.

These definitions are perhaps easiest to understand through examples. Figure 5.1 illustrates the concepts of locations, tracks and paths, and Figure 5.2 the relations between slots, trips and tracks.

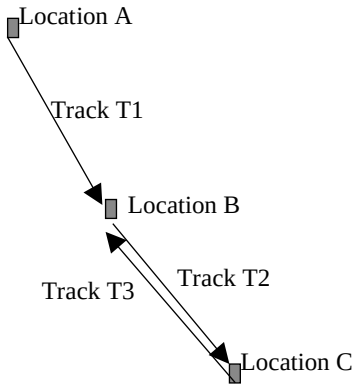

Figure 5.1: Tracks, locations and paths. Track T1 and T2 form a path between location A and location C.

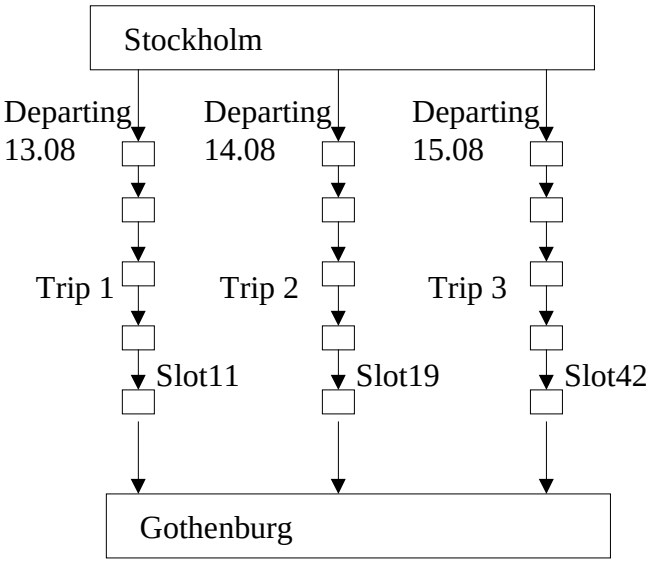

Figure 5.2: Trips, slot and tracks. Several trips traverse the same track between two locations. Each trip consists of several slots. The slot gets a new id for each trip, although the traversed track is the same.

5.2 The architecture of TUFF

TUFF has an agent-based architecture, which consists of a number of independent agents: A coordinator, a train scheduler, a vehicle router and (in a near future) a personnel scheduler. In addition to these vital parts, a Graphical User Interface (GUI) is used for communication between the TUFF system and the user. Data necessary for the system is stored in Nets and Tripsets. The relations between the components can be seen in Figure 5.3.

A *train scheduler* creates timetables, a *vehicle router* assigns cars and locomotives to trains, and a *personnel scheduler* assigns personnel to trains. See Section 1 Introduction for a brief description of the various types of planning mechanisms necessary in a railroad planning system.

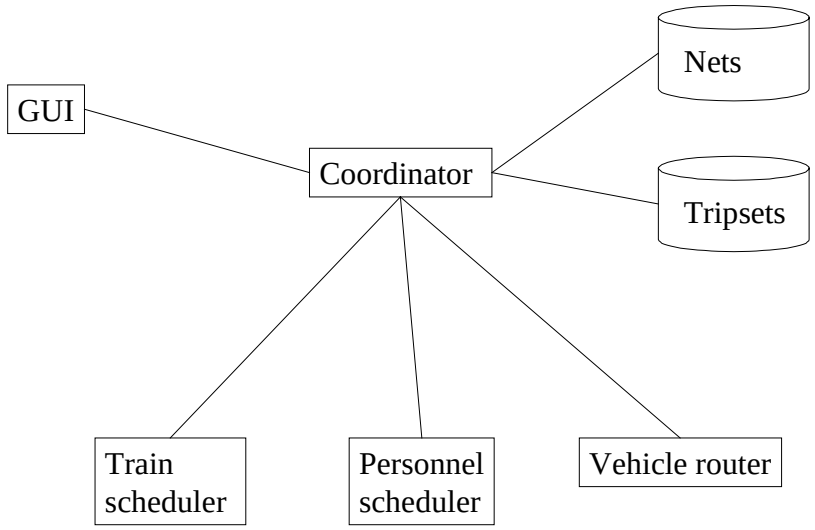

Figure 5.3: The architecture of TUFF.

The application uses two different languages: Prolog, a constraint logic programming language, which is used for calculation and constraint programming, and Mozart/Oz, a high-level language used for graphics and communication between agents.

The entire application is controlled through the GUI, which is connected to the coordinator. A user can give orders to the coordinator via the GUI. These orders are then propagated to the agent involved in the operation. If the agent is in need of further information, it asks the coordinator, using one of the pre-defined protocols for questions. When the agent has finished its task, it returns the solution to the coordinator, which may present the result graphically to the user.

Example 5.1. The user wishes to make a timetable for the trips between Stockholm and Gothenburg. He/she selects the desired specification for the trips, adds it to the tasks that are supposed to be computed, and chooses to make a schedule. The coordinator tells the train scheduler to make such a schedule.

However, the train scheduler does not know anything about the railroad net that connects the involved location. Nor does it know the specifications for the arrival- and departure times for each trip. Therefor, it has to ask the coordinator to supply this information. When all needed information is present at the train scheduler, the schedule is calculated, and the result is sent back to the coordinator.

5.2.1 The coordinator

The objective of the coordinator is – as the name implies – to coordinate all activities in the system. The responsibilities for the coordinator are:

- Handling input and output to/from the user. The coordinator is the only component connected to the GUI, and is therefor responsible for the management of all user requests.
- Propagating user orders to the train scheduler, the vehicle router and the personnel scheduler.
- Keeping a database of all Tripsets generated by the system.
- Responding to requests from the train scheduler agent, the vehicle router agent and the personnel scheduler agent. These requests may be to get a specific Tripset, or to get information about the Net.

5.2.2 The train scheduler

The train scheduler is divided in two parts:

One part written in Mozart/Oz. This part handles the communication with the coordinator and with the Prolog part, but does very little about the scheduling. Hardly anything is stored in the scheduler itself – all necessary information must be retrieved from the coordinator by using various method calls. Some of the data must then be converted before it can be passed on to the Prolog part of the scheduler. Sending all information as text strings solves the differences in representation in Mozart/Oz compared to Prolog.

The Prolog part is the one that does the actual work. It uses constraint programming techniques to calculate the schedule. See Section 3 Constraint programming for further information about constraint programming.

Based on the information it gets from the Mozart/Oz scheduler, the Prolog scheduler tries to calculate a valid schedule. The data sent is:

- A number of specifications for **trips.** In each specification, a departure time and an arrival time are specified. The paths for the trips are specified as well as all tracks the trip traverses on its way. For each trip a number of locomotive types are allowed. The total time for a trip and the total waiting time at stations are also limited.

- The necessary resources. This includes all **stations** involved in the trips. Each station has an id, a minimum turn time - which determines the time it takes to turn a train - and a number of tracks.
  In addition, all **tracks** traversed by the trips are sent together with a specification of the time period when a trip traverses the track, and the direction the track is traversed in. Each track has a headway, which determines the minimum safety time distance.

- Optionally, **relations** can be sent. These relations state that there should be a specific order between the arrival/departure of two trips or two slots, possibly with some offset. In the case of trips, the following may be specified: 'The train from Stockholm to Gothenburg (with trip id X) should arrive 5 minutes before the train from Gothenburg to Karlskrona departs'. The vehicle router may produce relations when creating a vehicle route. These relations can then be used in the train scheduler, and thus be used to create a complete schedule, involving both a train schedule and a vehicle route. See Section 6.6 Creating relations for further information about relations.

All these restrictions are used to create constraints. The resulting problem is then a constraint satisfaction problem (CSP). CSPs can be solved by using subroutines in Prolog. The generated solution is guaranteed to be valid. Alternatively, no solution is found.

When a train schedule has been calculated, a textual representation of the schedule is sent back to the Mozart/Oz part of the train scheduler. There a new Tripset containing the solution is created. The Tripset is sent back to the coordinator and stored in a database.

5.2.3 The vehicle router

The structure of the vehicle router is similar to that of the train scheduler. It consists of a part written in Mozart/Oz, and a part written in Prolog.

The Mozart/Oz part handles communication between the coordinator and the Prolog part. Data is retrieved from the coordinator and sent in a text string representation to Prolog. When the result is returned from the Prolog part, it is transformed to a format meaningful to the coordinator. Some effort is also made to create relations between the trips involved in the solution. The handling of these relations is presented in Section 5.2.2 The train scheduler above.

The Prolog part does the actual calculation of the vehicle router. In the current version of the system, a variation of the insertion heuristic is used. A heuristic is an attempt to use trial-and-error in an intelligent way for solving large problems. The insertion heuristic is the best known heuristic for this type of problems [Sol87]. It produces a valid but not optimal result. However, the worst-case performance is very poor [Sol86]. Nevertheless, the system currently used by the Swedish Railway (SJ), uses similar techniques [DHKK97].

5.2.4 The GUI

A Graphical User Interface is used to handle all communication with the user. The GUI can be seen in Figure 5.5 (from [Mar00]). Through the GUI, the user may choose a tripset to schedule. He or she may also choose whether data from previous scheduling should be used.

When a scheduling is finished, it is possible to have the result presented graphically. For a vehicle route, the entire result is presented in the same window. For a train

schedule, it is possible to choose a part of the path that should be presented. It is then shown in a separate window.

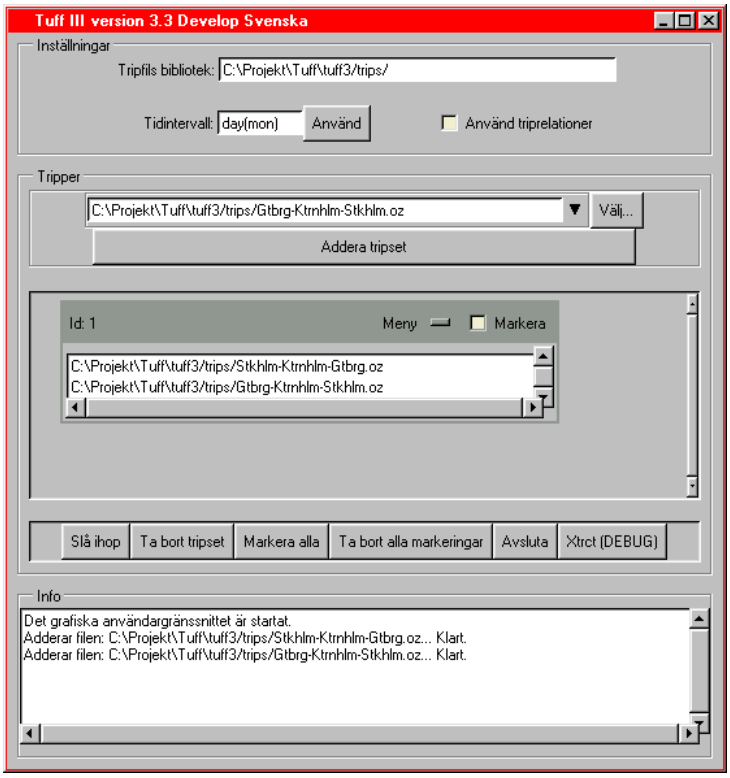

Figure 5.5: The Graphical User Interface of TUFF.

The GUI also does some fault handling to prevent illegal requests to reach the coordinator. When an error occurs, a message containing the problem is presented to the user, who may adjust the input to get a proper result.

6 Abstractions in TUFF

The main goal of this work is to investigate how abstractions can be used in TUFF. There are several reasons to introduce abstractions. AI research has shown that abstractions may drastically speed up the scheduling process [HMZM96] [HPZM96]. In addition, the notion of abstractions provides a more flexible way of representing data. Abstractions can for instance be used when combining a solution and a problem. The important parts of the solution can be extracted, and then the solution and the problem combined.

In this section, the various abstraction methods that have been implemented are discussed. The testing process of these abstractions and the testing results can be found in Section 7 Tests and results.

6.1 Abstraction methods

Given a scheduling problem in the railroad domain, there are several ways of abstracting it. Whether a method yields a good result or not is highly dependent of the nature of the problem. In most cases, it is necessary to examine several methods before the optimal method is found.

Some of the abstraction and concretion methods that may be usable are:

- **Relaxation:** Assume there is a scheduling problem with departure times and arrival times. Let these times be fixed times or time intervals. If a departure or arrival time is an interval, enlarge the interval. If the time is fixed let it become an interval. The times have then been relaxed.

- **Strengthening:** The opposite of relaxation, i.e. intervals are narrowed and fixed times are left unchanged.

- **Net abstraction:** Given a number of slots and trips in a tripset, create new slots such that each new slot corresponds to one or more of the previous slots. The result is a tripset with a reduced number of slots, and less details than the original.

- **Net refinement:** When net abstraction has been used on a tripset, net refinement can be used to take the tripset back to its original state. Each slot in the abstract tripset is replace by the corresponding original slots. Information from the abstract slot is transferred to the original slots.

- **Creating relations:** From a solution to a scheduling problem, relations between trips and/or slots can be computed. These relations can then be added to a relaxed version of the solution. To create and add relations is actually to make a problem more concrete rather than more abstract, but it may be used for abstraction if it is combined with relaxation.

These methods will be defined and more thoroughly described in the following sections. It should be noted that the presented abstractions are local for a tripset. This means that when an abstraction is performed on one tripset, no other tripsets will be affected. The methods that have been implemented and tested in this work are Relaxation, Net abstraction, Net refinement and Creating relations.

6.2 Relaxation

Relaxation is to enlarge the domains for departure and arrival times. The domain of the traversal time as well as the waiting time at a station may also be enlarged.

A formal definition of relaxation (from [Kre00]) is:

Definition 6.1. If ρ is an relaxation operator it must satisfy:

1. $\rho(\text{Tripset}_i) = < \rho(\text{Trips}(\text{Tripset}_i)), \rho(\text{Rels}(\text{Tripset}_i)) >$
   $\rho(\text{Trips}(\text{Tripset}_i)) = \{\rho(\text{Trip}_{ik})\}$, $1 \le k \le n$ where n is the number of trips in Tripset_i and
   $\text{Trips}(\text{Tripset}_i)$ is an operator which extracts all trips from Tripset_i,
   $\text{Rels}(\text{Tripset}_i)$ is an operator which extracts all relations from Tripset_i.

2. $\rho(\text{Rels}(\text{Tripset}_i)) \subseteq \text{Rels}(\text{Tripset}_i)$

3. $\rho(\text{Trip}_j) = < \text{Id}_j, \rho(\text{Dep}_j), \rho(\text{Trav}_j), \rho(\text{Arr}_j), \rho(\text{Turn}_j),$
   $< \rho(\text{Slot}_{jk}) \ldots \rho(\text{Slot}_{jn}) >>$, $1 \le k \le n$, where n is the number of slots in Trip_j.
   Dep, Trav, Arr and Turn represent the intervals of departure, traversal, arrival and turn time respecitvly.

4. For each trip j in Tripset_i: $\text{Dep}_{ij} \subseteq \rho(\text{Dep}_{ij})$, $\text{Trav}_{ij} \subseteq \rho(\text{Trav}_{ij})$, $\text{Arr}_{ij} \subseteq \rho(\text{Arr}_{ij})$, $\text{Turn}_{ij} \subseteq \rho(\text{Turn}_{ij})$.

5. For each slot k in each trip j in Tripset_i: $\text{Dep}_{ijk} \subseteq \rho(\text{Dep}_{ijk})$, $\text{Trav}_{ijk} \subseteq \rho(\text{Trav}_{ijk})$, $\text{Arr}_{ijk} \subseteq \rho(\text{Arr}_{ijk})$, $\text{Turn}_{ijk} \subseteq \rho(\text{Turn}_{ijk})$

When a tripset in TUFF is relaxed, the domains of the departure time, arrival time, traversal time and turn time of all or some trips and slots of that tripset may be enlarged. Example 6.1 presents the idea of relaxation.

Example 6.1: Relaxation. Assume there is a tripset with one trip, ***T****. Let* ***T*** *have two slots,* $\boldsymbol{S_1}$ *and* $\boldsymbol{S_2}$*. The departure and arrival times of the slots are* $Dep(S_1)=2..3$, $Arr(S_1)=3...4$, $Dep(S_2)=4..5$, $Arr(S_2)=7..8$.

A possible relaxation is then to enlarge the intervals to $Dep(S_1)=1..3$, $Arr(S_1)=3...6$, $Dep(S_2)=3..8$, $Arr(S_2)=5..11$.

6.3 Strengthening

Let Trips($Tripset_i$), Rels($Tripset_i$) be defined as in Section 6.2 above, and Dep, Trav, Arr and Turn represent the intervals of departure, traversal, arrival and turn time respecitvly.

A strengthening operator σ can then be defined.

Definition 6.2 Strengthening operator.
σ is a strengthening operator if it satisfies

1. $\sigma\ (Tripset_i) = <\sigma\ (Trips(Tripset_i)), \sigma\ (Rels(Tripset_i))>$

2. $\sigma\ (Rels(Tripset_i)) \supseteq Rels(Tripset_i)$

3. $\sigma\ (Trip_j) = < Id_j, \sigma\ (Dep_j), \sigma\ (Trav_j), \sigma\ (Arr_j), \sigma\ (Turn_j),$
   $<\sigma\ (Slot_{jk}) ... \sigma\ (Slot_{jn})>>$, $1 \leq k \leq n$, where n is the number of slots in $Trip_j$.
   Dep, Trav, Arr and Turn represent the intervals of departure, traversal, arrival and turn time respecitvly.

4. For each trip *j* in $Tripset_i$: $Dep_{ij} \supseteq \sigma\ (Dep_{ij})$, $Trav_{ij} \supseteq \sigma\ (Trav_{ij})$, $Arr_{ij} \supseteq \sigma\ (Arr_{ij})$, $Turn_{ij} \supseteq \sigma\ (Turn_{ij})$.

5. For each slot *k* in each trip *j* in $Tripset_i$: $Dep_{ijk} \supseteq \sigma\ (Dep_{ijk})$, $Trav_{ijk} \supseteq \sigma\ (Trav_{ijk})$, $Arr_{ijk} \supseteq \sigma\ (Arr_{ijk})$, $Turn_{ijk} \supseteq \sigma\ (Turn_{ijk})$

6.4 Net abstraction

Net abstraction is a way to reduce the number of slots and tracks in a tripset by merging several slots into one abstract slot, and several tracks into one abstract track. The abstraction can be used in two ways:

1. Net abstraction can be used to produce an abstract schedule.
   In this case, a tripset is abstracted and then scheduled, and the result is considered the complete schedule, although arrival and departure times have not been calculated for all slots.

2. If a more detailed schedule is desired, scheduling can be done in two steps.

   First, an abstract schedule is calculated (as in 1). Then net refinement is used to replace the abstract slots with the original, concrete slots, and the abstract tracks with the original tracks. Information is transferred from each abstract slot to the corresponding concrete slots. This means that some of the arrival times, traversal times, etc. in the concrete slots will be fixed, and some will still be intervals.

   Second, another scheduling of the concrete slots determines the times that have not yet been fixed. The result is a complete, concrete schedule.

To calculate a schedule in two steps, as in 2, may seem complicated and time consuming. However, AI research has shown that this type of hierarchical scheduling may reduce the computational cost of solving the problem [Yan97]. This is also the case here. As can be seen by the evaluation in Section 7 Tests and results, both the computational cost and the memory usage is reduced when compared to ordinary scheduling in one step.

6.4.1 Abstract slots

Define the ordinary slots in the TUFF system as *concrete slots.* Abstract slots are then created by merging one or more concrete slots. The technique used is comparable to State abstraction, which is presented in Section 4.2 Abstraction methods.

Assume that an abstract slot **AS** should be created based on the concrete slots $\mathbf{S_1}$, $\mathbf{S_2}$, …, $\mathbf{S_n}$. Then the following properties should hold:

- Departure time(**AS**)=Departure time($\mathbf{S_1}$)
- Arrival time(**AS**)=Arrival time($\mathbf{S_n}$)
- Traversal time(**AS**)= $\sum_{S_1}^{S_n} Traversal\ \ time(\mathbf{S_i}) + \sum_{S_1}^{S_{n-1}} Waiting\ \ time\ \ at\ \ location(\mathbf{S_i})$
- Waiting time at station(**AS**)=Waiting time at station($\mathbf{S_n}$)
- Headway(**AS**)=max(Headway($\mathbf{S_1}$), Headway($\mathbf{S_2}$), …, Headway($\mathbf{S_n}$))
- Origin(**AS**)=Origin($\mathbf{S_1}$)
- Destination(**AS**)=Destination($\mathbf{S_n}$)

Example 6.2 shows how abstract slots can be created from concrete slots.

Example 6.2: Net abstraction
*Assume there is a trip **T** with four slots, $\mathbf{S_1}$, $\mathbf{S_2}$, $\mathbf{S_3}$, $\mathbf{S_4}$. Let these slots have specifications as shown in the table below:*

Slot	*Departure time*	*Traversal time*	*Arrival time*	*Waiting time*	*Head-way*	*Origin*	*Destination*
S_1	*3…5*	*1…2*	*4…6*	*1…2*	*2*	*A*	*B*
S_2	*4…6*	*1…2*	*5…7*	*0…1*	*3*	*B*	*C*
S_3	*5…7*	*2…3*	*7…9*	*0…1*	*2*	*C*	*D*
S_4	*7…9*	*2…3*	*10…11*	*0…1*	*4*	*D*	*E*

A possible net abstraction is then to merge S_1 *and* S_2 *to the abstract slot* AS_1*, and* S_3 *and* S_4 *to the abstract slot* AS_2*. Following the above rules,* AS_1 *and* AS_2 *will become as in the below table:*

Slot	*Departure time*	*Traversal time*	*Arrival time*	*Waiting time*	*Head-way*	*Origin*	*Destination*
AS_1	*3…5*	*2…6*	*5…7*	*0…1*	*3*	*A*	*C*
AS_2	*5…7*	*4…7*	*10…11*	*0…1*	*4*	*C*	*E*

6.4.2 Abstract tracks

As previously mentioned in the description of the TUFF system, Section 5.1 Definitions, each slot is associated with a track. This should be the case also for abstract slots. Therefor it is necessary to introduce the concept of *abstract tracks*.

The idea is that each abstract slot is associated with an abstract track. An abstract track is created based on several concrete tracks. To be useful, the abstract tracks must fulfill the following conditions:

- The concrete tracks covered by the abstract track form a route from the origin to the destination of the abstract track.
- Each concrete track may appear in only one abstract track.
- An abstract track can be considered a double track only if all concrete tracks covered by this abstract track are double tracks, and go in the same direction.

An example will clarify how abstract tracks are created.

Example 6.3: *Abstract tracks. Assume two abstract tracks should be created. One track going from location A to locations B, and one going from B to A.*
1 *shows a situation where all concrete tracks between A and B are double tracks. Then the abstract tracks can also be double tracks.*
In ***2****, there is a single track between C and B, and therefor, the abstract track will be a single track.*

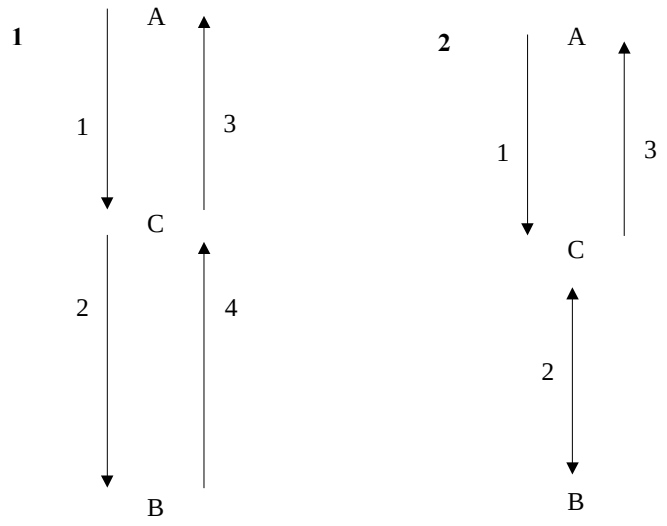

Figure 6.1: Creation of abstract tracks

6.4.3 Properties of the implemented net abstraction

The implementation of net abstraction is intended to be as safe as possible. Ideally, the net abstraction should fulfill the downward solution property. However, this is not the case. In fact, examples can be created where problems with both tracks and stations prevent a refined schedule from being solved.

6.5 Net refinement

When an abstract tripset has been scheduled, it must be refined to get back to the concrete level. The implementation of net refinement is rather straightforward. The abstract slots are simply replaced with the concrete slots they cover. Then the arrival time for each abstract slot is assigned to the first of the covered concrete slots. Similarly, the departure times are assigned to the last covered slots. The resulting concrete tripset can then be scheduled.

Referring to Section 4.4 Refinement methods, the method used for refinement is forward-chaining, total order refinement. This is a natural choice since each trip can be seen as a totally ordered plan, with slots being the elements of the plan. The refinement method fulfills the monotonic refinement property. As can be recalled from Section 4.4.3 Monotonic refinement, this property states that the plan will not decrease in size during refinement. Furthermore, the order calculated in the abstract plan remains unchanged in the refined plan.

6.6 Creating relations

The idea behind relations is to solve a large scheduling problem in smaller parts. When these parts have been solved, relations between trips and/or slots in each part can be created. The entire problem can then be solved with these relations added. When solving the entire problem, the relations will provide guidance for the constraint programming system. Big parts of the search tree can then be ignored, and the search can be focused where it is likely to find a solution.

Relations can be created in several different ways; between slots or between trips, between arrival times or departure times, and based on locations or tracks. To create relations between trips is to specify that one trip should arrive or depart before some other trip. Relations between slots specify that a slot should arrive or depart to a location or on a track before some other slot. That is, relations between trips specify orders at end locations or end tracks, whereas relations between slots specify orders at any location or track.

The way relations are created can be specified by a list with three elements:

[trip/slot, track/location, departure time/arrival time/both].

It is then easily verified that there are (2*2*3=) 12 possible ways of creating relations.

Example 6.4: A location L is connected to three tracks: T_A, T_B and T_C. There are three trains A, D and E heading for L, and two trains, B and C, leaving L. The situation is shown in Figure 6.2.

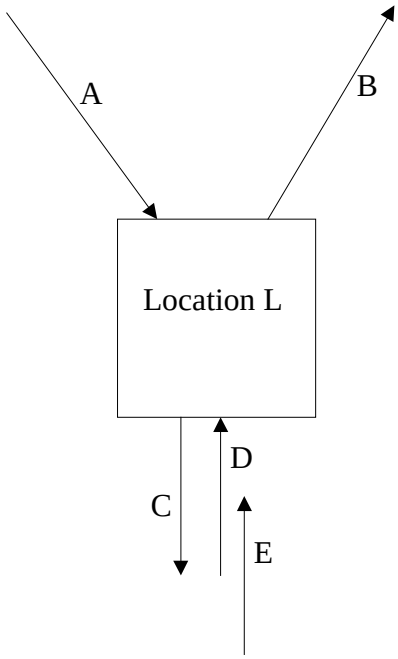

Figure 6.2. A location L is connected to three tracks. Two trips - B and C - leave L, and three trips - A, D and E - arrive at L.

6.6.1 Creating relations on location basis for both arrival and departure time

Let trip A arrive on track T_A, trip B leave on track T_B, trip C leave on track T_C, and trip D and E arrive on track T_C. Order all trips at this location by examining the arrival times for the arriving trains A, D and E, and the departure times for the departing trains B and C. There will then be an order between all five trips.

If a set S consisting of the five trips is created, then S will be a totally ordered set. Using the notation from Section 2 Mathematical foundations, **S** can be represented as **S** = {<A, B, C, D, E>}.

Each departure and arrival is related to every other departure and arrival. The ordering may for instance be (A<C<B<D<E). This would mean that the first thing that happens is that A arrives at L. Then, C leaves, B leaves and D and E arrive.

6.6.2 Creating relations on location basis for arrival and departure time individually

A similar approach to the one presented above is to maintain the order among the arriving trips and the order among the departing trips, but ignore the relations between arriving trips and departing trips. In this way, no departure will be related to any arrival, but to all other departures. Similarly, each arrival will be related to every other arrival, but not to any of the departures. Referring to Example 6.4 above, this will create relations between trip A, D and E, and between trip B and C. The set of relations will contain two chains, that is
S = {< A, D, E >, **<** B, C >}.

6.6.3 Creating relations on a location basis for arrival or departure time

In some situations, it may be interesting to create relations that only deal with arrival times. Once again referring to Example 6.4, the set of relations will contain only one chain – the one that involves the arriving trips. That is, **S** = {<A, D, E>, B, C}.

Ordering by departure time is similar to ordering by arrival time. The only difference is that different trips will be involved in the resulting relations. In this case, still referring to Example 6.4, the set of relations will be **S** = {<B, C>, A, D, E}.

6.6.4 Creating relations on track basis

When relations are created on a track basis, the situation is slightly different. All trips or slots that depart on a track must also arrive on that same track. Therefor, the set of relations is independent of whether relations are created on arrival time, departure time or both.

As mentioned in Section 5.1 Definitions, there are two different types of tracks: single tracks and double tracks. For a single track, there is only one track between two locations. There can only be trips going in one direction on that track at the same time.

However, if there are double tracks between two locations, two trips may go in opposite directions simultaneously.

Consider Example 6.4. The situation is once again shown in Figure 6.1 below.

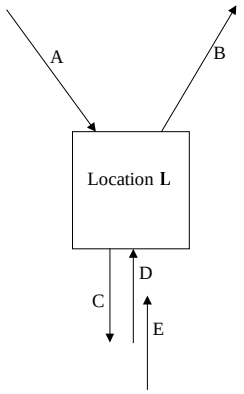

Figure 6.1. A location L is connected to three tracks. Two trips - B and C - leave L, and three trips - A, D and E - arrive at L.

Suppose the trips C, D and E run on a double track. C will then run on one track, going away from L. D and E will run on another track, going towards L. Since A and B are going in different directions, they will have separate tracks. However, C, D and E may all run on a single track. If this is the case, then C cannot be on that track simultaneously with D or E, but D and E may be on the track as long as the headway time – that ensure the safety distance – is respected. Thus on a single track, all trips or slots arriving or departing to a location on that track will be related.

If relations are created for all tracks connected to location L, and C, D and E run on a single track, the set of relations will be **S** = {<A>, <B>, <C, D, E>}. However, if C, D and E run on a double track, D and E will be considered to go on a different track from C, and the set of relations becomes less restricted. The set then becomes **S** = {<A>, <B>, <C>, <D, E>}.

In fact, a single track will always give a more restricted set than a double track. This is because on a single track, trips or slots in both directions must be considered, but on a double track, only trips or slots in one direction can be considered.

6.6.5 Structure of methods to create relations

The methods to create relations described above give various restrictions for a location or a track. The order of the set S from Example 6.4 ranges from total order to almost no order at all. In this section, the grade of restriction for the methods that create relations is investigated. A few definitions are necessary for this task.

Definition 6.3: Basic element. Let a basic element be an identifier of a slot or a trip.

Definition 6.4: Set of relations. A set S is called a set of relations if each of its elements is either a basic element or a chain of basic elements.

Definition 6.5: Basic element extraction. Let BaseEl(S) be a flattening operator on the set S that extracts all the basic elements.

Let there be a location L. Let the tracks connected to this location be T_1, T_2, ..., T_N. To reach the station L, one of the tracks T_i must be traversed. Create two sets of relations: One set S_1 for arrival and departure time for all slots or trips at L, and one set S_2 for all slots or trips traversing a track T_i, $1 \le i \le N$. Since all slots or trips in S_1 are related, S_1 will be a totally ordered set. Furthermore, all trips or slots that reaches L must have traversed one of the tracks connected to L. This means that the relation $BaseEl(S_1) = BaseEl(S_2)$ must hold. S_1 is totally ordered (see above). Since there is no stronger ordering restriction that a total order, S_1 is as least as restricted as S_2. The statement S_2 LessRes S_1 with the operator lessRes defined as below will therefor hold.

Definition 6.6: Less restricted operator. For two sets of relations, S_1 and S_2, where $BaseEl(S_1) = BaseEl(S_2)$, define a less restricted operator $\le$. Let there be N chains in S_1 and M chains in S_2. Then the statement $S_1 \le S_2$ holds if every chain in S_1 is a subset of a chain in S_2, i.e. $\forall i : C_{S1_i} \subseteq C_{S2_j}, \ 1 \le i \le N, \ 1 \le j \le M$

A set of relations obtained by creating relations on departure time and arrival time individually is always more restrictive than a set obtained when creating relations only on departure time or only on arrival time. The reason is of course that the former set considers both arrivals and departures, but the later set only considers either arrivals or departures. As described above in Section 6.6.4 Creating relations on track basis, the situation is similar when dealing with double or single tracks. Creating relations for a single track always gives a more restrictive result than for a double track.

The structure of the methods of creating relations is shown in Figure 6.2. This structure holds for all possible cases where tracks are connected to a location. The observant reader may note that the presented structure is almost a lattice. If the single track case was removed, each pair of elements would have a unique join and meet, and the structure would thus be a lattice.
(The set S in the figure refers to the Example 6.4).

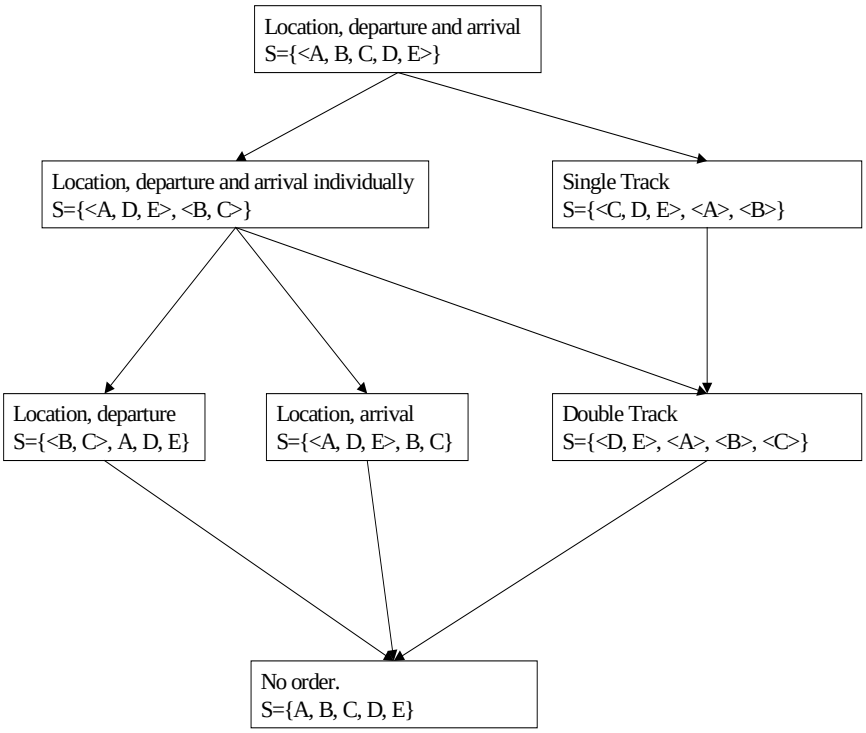

*Figure 6.2. The result of the various ways to create relations ranges from total order to almost no order at all. The arrows can be considered to be Less restricted operators, ≤. The case that no relations are created ('No order') is added to obtain a complete structure. The set **S** is the set that is used in Example 6.1.*

Which type of relation should then be used in practice? The answer to that question is that there is probably no single 'best' type. The type of relations that suits one problem best is likely not optimal for some other problem. In Section 7.3 Creating relations, experimental results of using various methods for creating relations can be found.

7 Tests and results

This section presents the results of using abstraction, and creating and adding relations. In the first subsection, the result of using abstractions is shown. Then the effects of creating and adding relations are shown. Finally, a number of strategies are presented. These strategies involve both abstractions and creation of relations.

7.1 Test settings

The tests were performed on a network of tracks based on real data. The main paths used were the two different routes between Stockholm and Gothenburg. The first route goes south of Mälaren, via Katrineholm. This is the main route for personnel traffic, and consists almost exclusively of double tracks. The second route goes north of Mälaren, via Örebro. This route is used mainly for freight transports. It consists largely of single tracks. The routes can be seen in Figure 7.1.

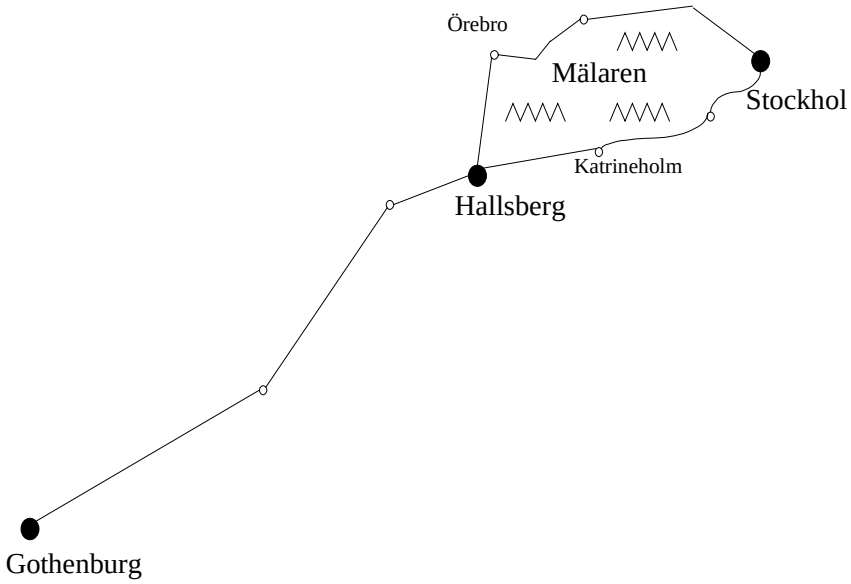

Figure 7.1: Network of tracks.

In the representation of the network, the path from Stockholm to Gothenburg via Katrineholm consists of 35 tracks connecting 36 stations. The second path, from Stockholm to Gothenburg via Örebro, consists of 63 tracks connecting 64 stations.

7.2 Net abstractions

The net abstraction mechanism was evaluated on the route from Gothenburg to Stockholm via Katrineholm. The number of trips was varied between 19 and 104 trips, all scheduled on one single day. The abstract scheduling was performed in two steps:

1. First, the tripset was abstracted. The abstraction made is of various degrees. 'Light abstraction' reduces the number of slots by 50 percent. 'Hard abstraction' reduces the number of slots by 80 percent. 'Really hard abstraction' reduces the number of slots as much as possible, that is, it removes all slots not connected to end stations.
2. Then the abstracted tripset was scheduled.
3. The abstract schedule was refined, and scheduled once more to get a concrete schedule.

The total processing time for all steps was measured, and compared to the processing time of the concrete scheduling. As can be seen in Figure 7.2, the time required for scheduling is up to 4 times shorter for the best type of abstraction than for the concrete scheduling. Furthermore, the abstract scheduling requires less memory than the concrete scheduling. In fact, the concrete scheduling was unable to handle more than about 2500 slots due to lack of memory. The best abstraction used less memory, and managed 5500 slots.

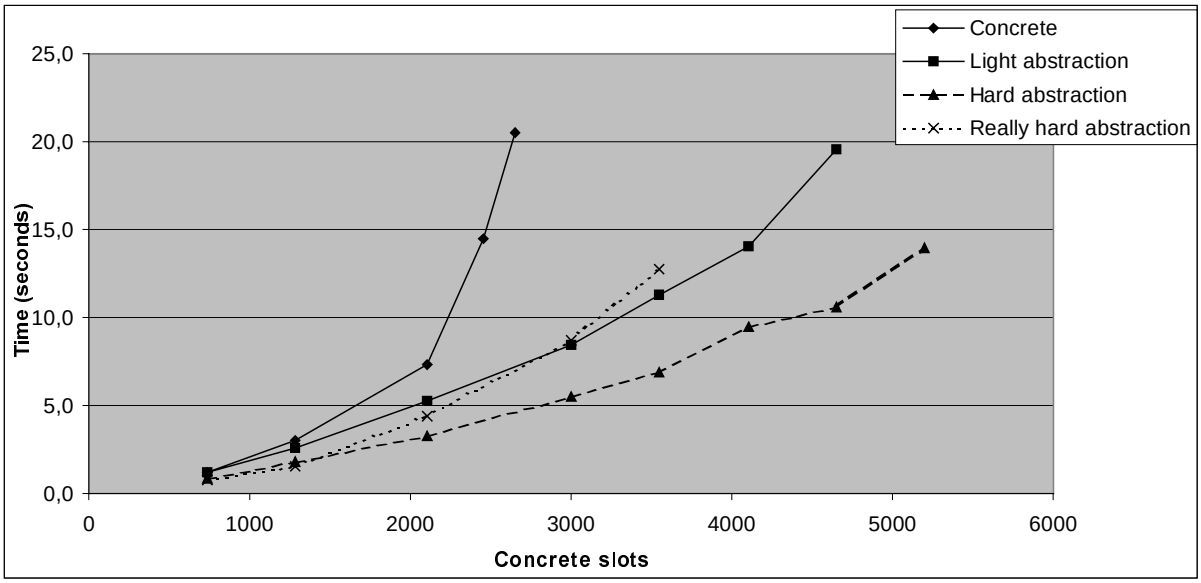

Figure 7.2. Total processing time for scheduling using various degrees of abstraction, and for concrete scheduling.

The test revealed that the different degrees of abstraction gave very different results. An interesting question is how far the abstraction should ideally go – how many concrete tracks should ideally be replaced with one abstract track? This was examined in a second experiment. As before, the route used was the one from Stockholm via Katrineholm to Gothenburg. 48 trips and 2712 slots were scheduled with different amount of abstraction, and the processing time was measured. As can be seen in

Figure 7.3, the time for scheduling the refined tripset is about constant. Scheduling the abstract tripset, on the other hand, is an almost linear function of the number of abstract slots, that is, the amount of abstraction. Only maximum abstraction differs from the pattern. Apparently, at such a high level of abstraction, it becomes harder to find solutions to the problem.

As earlier mentioned the route Stockholm-Katrineholm-Gothenburg consists mainly of double tracks. This makes scheduling this route relatively easy. The conclusion is that for such 'easy' problems, net abstractions works well, especially if a high degree of abstraction is used.

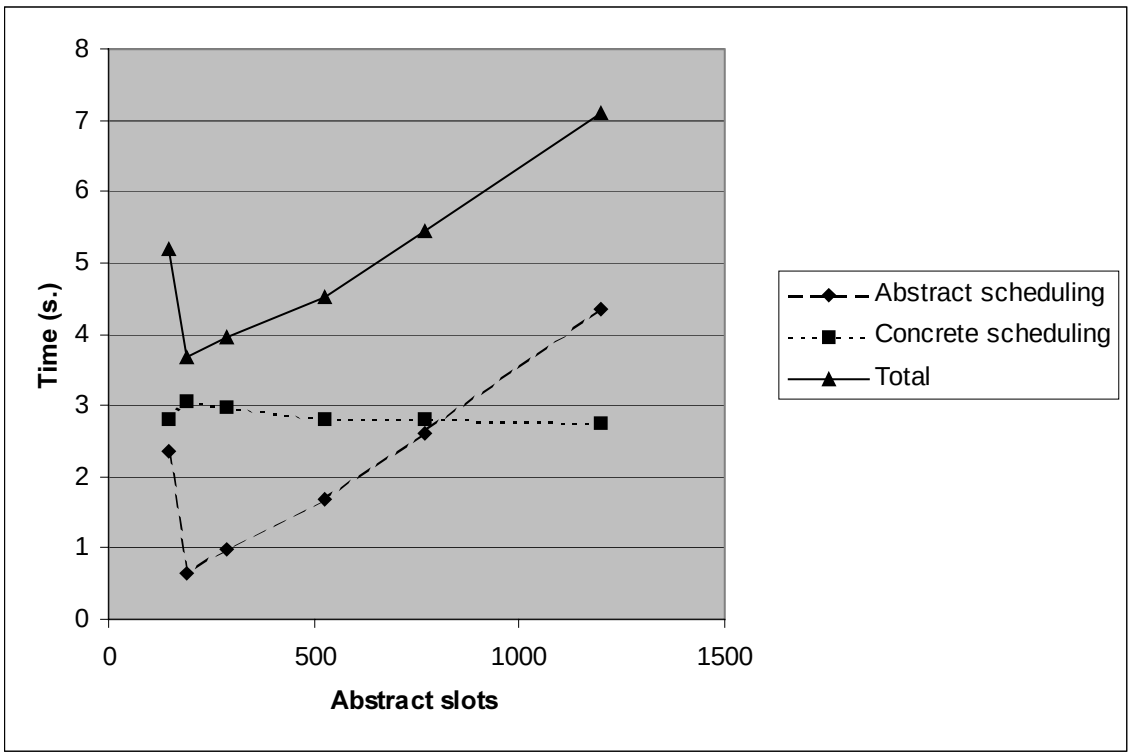

Figure 7.3. Scheduling time as a function of the amount of abstraction. The first scheduling is made on the abstract tripset. The tripset is then refined, and a second scheduling is done to achieve a concrete schedule.

7.3 Creating relations

This section presents an evaluation of using relations between trips or slots to improve scheduling performance. The idea is that the relations should guide the search by cutting out undesired parts of the search tree during the scheduling.

Two different tests have been made. In both tests, the same method has been used. The scheduling problem was split in two parts of about equal size. Each part was scheduled individually, and relations were extracted. Then the entire problem was scheduled, with the relations added.

7.3.1 Small test

The settings for the first test are similar to those of the abstraction tests. 48 trips are scheduled, all of them going from Stockholm to Gothenburg. 24 of them go via Katrineholm, and 24 of them via Örebro. The trips going via Katrineholm where scheduled as one set, and the trips via Örebro as another. From these sets, relations were created. The relations were used when all of the 48 trips were scheduled simultaneously.

As can be seen in Table 7.1, performance depends both on the number of relations and whether relations are created on a slot or a trip basis. Generally, relations created on trips basis are faster than those on slot basis are. In fact, for these settings the total scheduling time is only reduced if relations are created on trip basis.

Type of relations	*Number of relations*	*Scheduling time (seconds)*
No relations, ordinary scheduling	0	11,4
Locations, departing trips	30	6,9
Locations, arriving and departing trips	60	7,1
Locations, arriving trips	30	7,8
Locations, departing slots at [CST]	30	11,0
Locations, departing slots at [HPBG]	30	11,2
Locations, arriving slots at [G]	30	11,9
Locations, arriving and departing slots at [CST HPBG G]	90	19,2

Table 7.1. The table shows scheduling times for 48 trips going from Stockholm to Gothenburg, using various types of relations. [CST] represents Stockholm, [G] Gothenburg and [HPBG] Hallsberg.

The conclusion of this test is that relations created on slot basis give too much overhead, and therefor the performance improvement is small – in same cases performance is even worse than without relations. Relations created on trip basis, on the other hand, may reduce the scheduling time. It can also be noted that the scheduling times when using relations based on arrival times is somewhat longer than when using relations based on departure times.

7.3.2 Larger test

The second test is larger, and harder to solve. 86 trips are scheduled, mainly on the route from Stockholm via Örebro to Gothenburg. 39 of the trips are freight trips, and 47 personnel trips. As can be seen in Table 8.2, relations created on trip basis give better performance than those created on slot basis. Note that when relations are created on the slots passing [HPBG] (Hallsberg) – which include most trips – the restrictions posed by the relations were too hard, and it was impossible to calculate a valid schedule. For relations at [HSA], where fewer trips pass, it was possible to calculate a schedule, but also in this case the restrictions were hard, and the scheduling time-consuming.

Type of relations	*Number of relations*	*Scheduling time (seconds)*
No relations, ordinary scheduling	0	68,9
Locations, departing trips	68	24,3
Tracks, departing trips	67	24,7
Locations, arriving and departing trips	151	26,4
Locations, arriving trips	67	26,8
Tracks, arriving trips	65	26,8
Locations, arriving and departing slots at [HSA]	45	49,9
Locations, arriving and departing slots at [HPBG]	56	No solution

Table 7.2. Tests of relations on a harder example.

It is also in this test evident that relations created on location basis give a better performance improvement than those created on slot basis. However, as long as relations are created on location basis, it only makes marginal difference if they are based on locations or tracks, and on arriving or departing trips.

7.4 Strategies

In the last sections, the effects of using abstraction and the effect of using relations have been presented. This section shows how abstraction and relations can be combined. Such a combination will be referred to as a *strategy.* A strategy can be formulated by using a script language such as TUFFScript, which is presented in Appendix B - TUFFScript. The idea behind TUFFScript is similar to that of [CLS99].

There an ‘algebra’ for a constraint programming system is presented. The ‘algebra’ is used as a script language, and it is shown that a learning algorithm, which uses the ‘algebra’, can improve performance of the system. However, in TUFFScript there is no learning. All strategies have to be manually generated.

Four different strategies have been developed and tested. The testing has been made on the two routes from Stockholm to Gothenburg – one of them via Örebro, and one via Katrineholm. The routes have common tracks from Hallsberg to Gothenburg. The routes can be seen in Figure 7.1. The number of trips and slots is varied from 16 trips/904 slots to 48 trips and 2712 slots. Note that this is relatively simple problem, since all trips go in the same direction.

The ideas behind the different abstractions are:

1. Concrete scheduling – no abstraction or relations used.
2. The route Stockholm-Katrineholm-Gothenburg, where half the trips go, is scheduled. Then the route Stockholm-Örebro-Gothenburg – with the other half of the trips – is scheduled. Relations are extracted from the resulting schedules and added to the original tripset, which is then scheduled.
3. The route Stockholm-Katrineholm-Gothenburg is abstracted in such a way that Stockholm-Hallsberg becomes one single slot. It is then scheduled. The same thing is done with the route Stockholm-Örebro-Gothenburg.
   Relations are extracted from the two schedules, and added to the original tripset, which is then scheduled.
4. Both routes are abstracted in such a way that Stockholm-Hallsberg becomes a single slot. The tripset is refined. Relations from the tripset are added to itself, and the tripset is scheduled.
5. As 3, but without adding relations, that is: Abstract Stockholm-Hallsberg to a single slot. Refine the tripset. Schedule the tripset.

The performance of the strategies can be seen in Figure 7.4. For these settings, the only strategy that outperforms a simple, concrete scheduling is the one that only uses net abstraction, and no relations. The conclusion that can be drawn is that this test is so simple that the extra overhead of propagating the relations makes scheduling slower than if no relations were used. When using net abstraction the number of solutions is decreased. Net abstraction works best for problems with many valid solutions, and therefor losing some solutions is acceptable. The overhead when using Net abstraction is small, and therefor performance is improved.

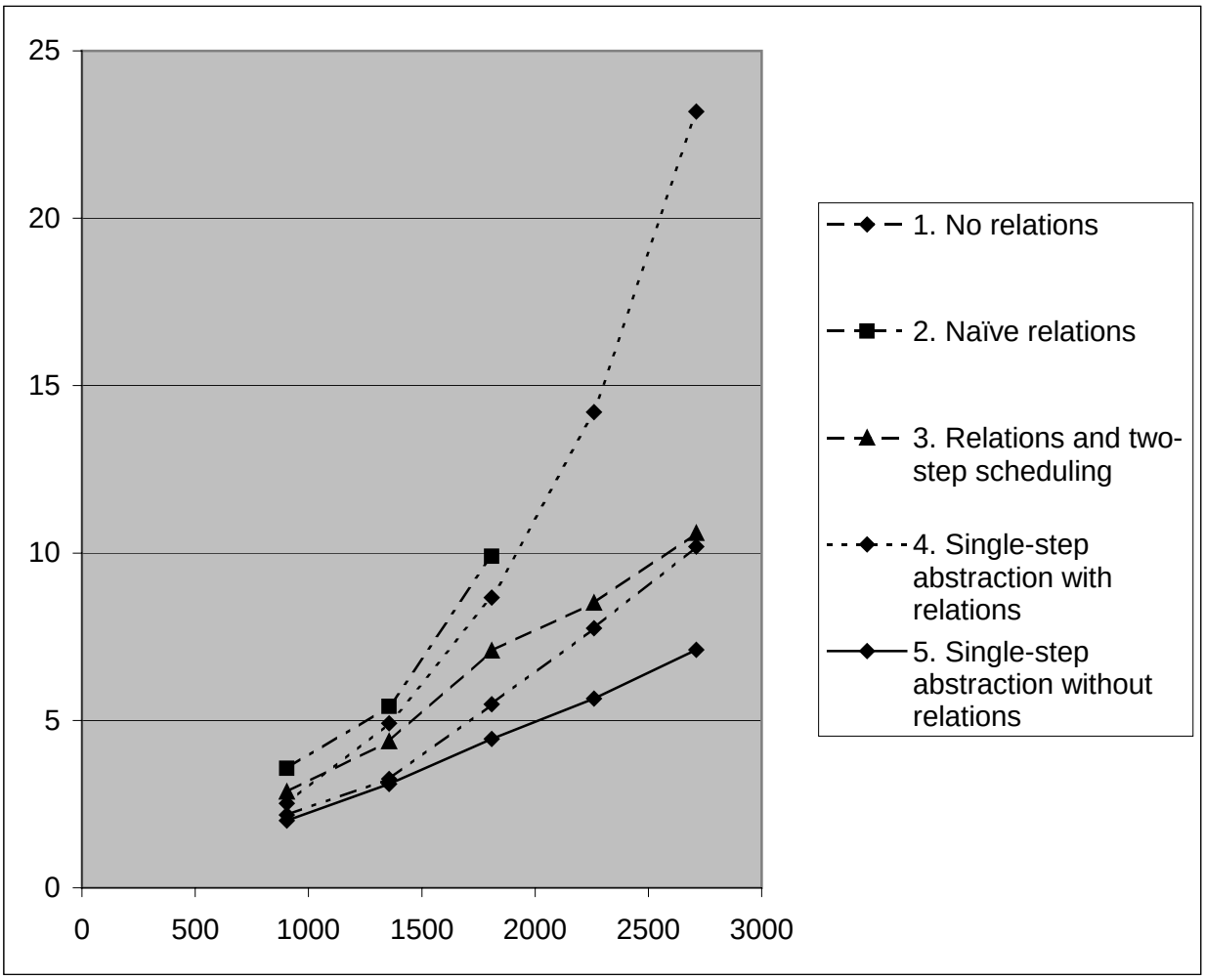

Figure 7.4. Performance of strategies.

8 Conclusions and future work

From the testing and experiments in Section 7 Tests and results, it is evident that abstractions and relations must be handled with care. If properly used, performance is improved. However, if handled incorrectly, there will be no improvement or – in some cases – system performance may deteriorate.

The trend that can be observed is that abstraction is useful for ‘easy’ problems with many valid solutions. Relations, on the other hand, should be used for hard problems. In those cases it is a good idea to solve the problem in smaller parts, extract relations from these solutions, and use the relations to guide the search when the entire problem is solved.

To be useful in a practical system, a script language, which allows the user of the system to experiment and find the most suitable solving method for a certain problem is necessary. A definition of such a script language, TUFFScript, has been made. It has also been implemented and integrated with the system. The definition of TUFFScript can be found in Appendix B – TUFFScript.

Additionally, other types of abstraction methods than those implemented in this work may also be useful. Examples of possible operations are time abstraction, location abstraction and trip abstraction. Furthermore a system, which automatically generates abstractions, could make the use of abstractions easier to handle and would certainly be of great scientific interest.

References

[Col99] E. Cooper. The primordial mathematics of Virtualist. Web page, 1999. http://www.ec3.com/upperized/totally.htm

[CLS99] Y. Caseau, F. Laburthe, G. Silverstein. A meta-heuristic factory for vehicle routing problems. Accepted paper at the Fifth International Conference on Principles and Practice of Constraint Programming, Alexandra, Virginia, 1999.

[DHKK97] J. Drott, E. Hasselberg, N. Kohl, M. Kremer. A planning system for locomotive scheduling. Technical report, Carmen Systems AB, 1997. http://www.carmen.se.

[HMZM96] R. Holte, T.Mkadmi, R.Zimmer, A. McDonald. Speeding up problem solving by abstraction: a graph oriented approach. Artificial Intelligence Journal, vol. 85, pages 321-361, 1996.

[HPZM96] R.Holte, M. Perez, R.Zimmer, A.MacDonald. Hierarchical A*: Searching abstraction hierarchies effeciently. In 'Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)'. ISBN 0-262-51091-x. AAAI Press, California, 1996.

[KCSA98] P. Kreuger, M. Carlsson, T. Sjoland and E. Astrom. Sequence dependent task extensions for trip scheduling on mixed single and double track networks. Technical report, SICS, February 1998.

[Kno94] C. Knoblock. Automatically generating abstractions for planning. Artificial Intelligence Journal, vol. 68(2), pages 243-302, 1994.

[Kre00] P. Kreuger. Task structure abstraction. Technical report, SICS, 2000.

[Mar00] M. Aronsson. TUFF – Systemöversikt och arkitektur. Technical report, SICS, 2000.

[PY73] F. Prepata, R. Yeh. Introduction to discrete structures. Addison-Wesley Publishing Company, 1973.

[Sol86] M. Solomon. On the worst-case performance of some heuristics for the vehicle routing and scheduling problem with time windows. *Networks*, volume 16, pages 161-174, 1986.

[Sol87] M. Solomon. Algorithms for the vehicle routing and scheduling problem with time window constraints. *Operations research*, volume 35, pages 254-265, March-April 1987.

[SS99] C. Schulte and G. Smolka. Finite domain constraint programming in Oz – a tutorial. Mozart/Oz online documentation, 1999. http://www.mozart-oz.org/documentation/fdt/index.html.

[Tsa93] E. Tsang. Foundations of constraint satisfaction. Academic press limited 1993.

[Yan97] Qiang Yang. Intelligent planning – A decomposition and abstraction base approach. ISBN 3-540-61901-1. Springer-Verlag, Berlin, 1997.

Appendix A – Test examples

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Testfile for relations and abstract slots
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Start the TUFF system.
declare SCH CIR
[Agent Coord GUI SAgent CAgent]={Module.link ['agents/agent.ozf'
'coordinate/coordinatorAgent.ozf' 'graphics/gui.ozf'
'schedule/scheduleAgent.ozf' 'circuit/circuitAgent.ozf']}
Show=System.show Coordinator=Coord.coordinatorAgent
NewAgent=Agent.newAgent
ScheduleAgent=SAgent.scheduleAgent CircuitAgent=CAgent.circuitAgent
Ko={NewAgent Coordinator init}
{Ko setTimeRng(day(mon))}
{Ko loadNet(file:'net/new-net.ozf')}
thread
   SCH={NewAgent ScheduleAgent init(coordinator:Ko pdebug:true)}
   {SCH add}
end
thread
   CIR={NewAgent CircuitAgent init(coordinator:Ko pdebug:true)}
   {CIR add}
end

%% No GUI used for these tests
%declare Gui={New GUI.coordinatorGui init(coordinator:Ko _)}

/*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Tests of mixed goods and personell traffic.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Test 0: Solve everything in one single shot.
declare Trips={Compiler.virtualStringToValue "\\insert
'"#'trips/PHO_Stkhlm-Ktrnhlm-Gtbrg6.oz'#"'\n" $}
declare Trips={Compiler.virtualStringToValue "\\insert
'"#'trips/PHO_Goods_and_P.oz'#"'\n" $}
declare T={Ko addTrips(trips:Trips $)}
declare Tsch={Ko schedule(tripSet:T $)}

% Test 1: Solve the problem in two steps, using relations.
declare Trips={Compiler.virtualStringToValue "\\insert
'"#'trips/PHO_Goods_and_Pa.oz'#"'\n" $}
declare Trips2={Compiler.virtualStringToValue "\\insert
'"#'trips/PHO_Goods_and_Pb.oz'#"'\n" $}
declare T1={Ko addTrips(trips:Trips $)}
declare T2={Ko addTrips(trips:Trips2 $)}
declare Tsch={Ko schedule(tripSet:T1 useRels:false $)} %Snabb!
declare Tsch2={Ko schedule(tripSet:T2 useRels:false $)}
declare TSsch={Ko getTripset(id:Tsch $)}
declare TSsch2={Ko getTripset(id:Tsch2 $)}
{Ko saveTripset(id:Tsch file:tsch guiData:g failure:_)}
{Ko saveTripset(id:Tsch2 file:tsch2 guiData:g failure:_)}
declare TS={Ko getTripset(id:Tsch $)}
declare TS2={Ko getTripset(id:Tsch2 $)}
declare Rels1={TS makeRels(extractMeth:[locs depTime trips] $)}
declare Rels2={TS2 makeRels(extractMeth:[locs depTime trips] $)}
declare T3={Ko merge(T1 T2 $)}
declare TS3={Ko getTripset(id:T3 $)}
{TS3 clearRels}
{TS3 addRels(rels:Rels1)}
{TS3 addRels(rels:Rels2)}
{Ko saveTripset(id:T3 file:t3 guiData:g failure:_)}
```

```
declare Tsch={Ko loadTripset(file:tsch guiData:_ $)}
declare Tsch2={Ko loadTripset(file:tsch2 guiData:_ $)}
declare T3={Ko loadTripset(file:t3 guiData:_ $)}
declare TS3={Ko getTripset(id:T3 $)}
declare TS={Ko getTripset(id:Tsch $)}
declare TS2={Ko getTripset(id:Tsch2 $)}
{TS3 clearRels}
declare Rels1={TS makeRels(extractMeth:[locs depTime slots]
locList:['HSA'] $)}
declare Rels2={TS2 makeRels(extractMeth:[locs depTime slots]
locList:['HSA'] $)}
{TS3 addRels(rels:Rels1)}
{TS3 addRels(rels:Rels2)}
declare T3sch={Ko schedule(tripSet:T3 useRels:true $)}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Tests of abstract slots on single tracks.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

declare Trips={Compiler.virtualStringToValue "\\insert
'"#'trips/PHO_Stkhlm_Orbr_Hllsbrg.oz'#"'\n" $}
declare T1={Ko addTrips(trips:Trips $)}
declare Tripset={Ko getTripset(id:T1 $)}
{Tripset makeAbsSlots(locs:['CST' 'HUV' 'SPÅ' 'JKB' 'KHÄ' 'BRO'
'EKO1' 'GIB' 'VÅV' 'KBÄ' 'MORP' 'ARB' 'ÖA' 'HSA' 'ÖB' 'HPBG' ])}
declare Tsch={Ko schedule(tripSet:T1 useAbstr:true useRels:false $)}
declare Tripset2={Ko getTripset(id:Tsch $)}
{Tripset2 concretize}
declare Tsch2={Ko schedule(tripSet:Tsch useRels:false $)}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Tests of strategies
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Test 0: Solve CST-K-G + CST-ORB-G in one single shot.
declare Trips={Compiler.virtualStringToValue "\\insert
'"#'trips/PHO_Goods_and_P.oz'#"'\n" $}
declare T={Ko addTrips(trips:Trips $)}
declare Tsch={Ko schedule(tripSet:T useRels:true $)}

% Test 1: Solve CST-K-G + CST-ORB-G with relations added
%
% Solve CST-K-G. Extract relations. Solve CST-ORB-G. Extract
% relations.
% Solve CST-K-G + CST-ORB-G with both these relations.
declare Trips1={Compiler.virtualStringToValue "\\insert
'"#'trips/PHO_Stkhlm-Ktrnhlm-Gtbrg-Stkhlm1a.oz'#"'\n" $}
declare Trips2={Compiler.virtualStringToValue "\\insert
'"#'trips/PHO_Stkhlm-Ktrnhlm-Gtbrg-Stkhlm1b.oz'#"'\n" $}
declare T1={Ko addTrips(trips:Trips1 $)}
declare T2={Ko addTrips(trips:Trips2 $)}
declare T1sch={Ko schedule(tripSet:T1 useRels:false $)}
declare T2sch={Ko schedule(tripSet:T2 useRels:false $)}
declare Tripset1={Ko getTripset(id:T1sch $)}
declare Tripset2={Ko getTripset(id:T2sch $)}
declare Rels1={Tripset1 makeRels(extractMeth:[locs arrDepTime trips]
$)}
declare Rels2={Tripset2 makeRels(extractMeth:[locs arrDepTime trips]
$)}
declare TJoined={Ko merge(T1 T2 $)}
declare TripsetJoined={Ko getTripset(id:TJoined $)}
declare {TripsetJoined addRels(rels:Rels1)}
declare {TripsetJoined addRels(rels:Rels2)}
declare Tsch={Ko schedule(tripSet:TJoined useRels:true $)}
```

```
% Test 2: Solve CST-K-G + CST-ORB-G in two steps, with relations
% added.
%
% 1.Abstract CST-K-G in such a way that CST-HPBG becomes one single
% slot.
% 2.Abstract CST-ORB-G in such a way that CST-HPBG becomes one single
% slot.
% Solve 1. Solve 2. Concretize the two schedules.
% Extract relations from these schedules. Merge the two original
% tripsets,
% and solve the resulting plan using the relations from above.
declare Trips1={Compiler.virtualStringToValue "\\insert
'"#'trips/PHO_Stkhlm-Ktrnhlm-Gtbrg-Stkhlm1a.oz'#"'\n" $}
declare Trips2={Compiler.virtualStringToValue "\\insert
'"#'trips/PHO_Stkhlm-Ktrnhlm-Gtbrg-Stkhlm1b.oz'#"'\n" $}
declare T1={Ko addTrips(trips:Trips1 $)}
declare T2={Ko addTrips(trips:Trips2 $)}
declare Tripset1={Ko getTripset(id:T1 $)}
declare Tripset2={Ko getTripset(id:T2 $)}
{Tripset1 makeAbsSlots(locs:['CST' 'HPBG' 'TÄL' 'LÅ2' 'LÅ' 'GDÖ'
'SLE' 'T' 'MH' 'VÄ' 'SK' 'RMTP' 'SS' 'F' 'FBY' 'HR' 'VGÅ' 'A' 'VBD'
'NS' 'FD' 'SN' 'LR' 'ASD' 'APN' 'P' 'SEL' 'SÄV' 'GSV' 'OR1' 'OR'
'GRO' 'G'])}
{Tripset2 makeAbsSlots(locs:['CST' 'SUB' 'SPÅ' 'KHÄ' 'STT' 'TOT'
'EKO1' 'EP' 'VÅ' 'VÅV' 'KBÄ2' 'MORP' 'VSG' 'ARB' 'ÖA' 'ÖR' 'KLA'
'HPBG' 'TÄL' 'LÅ2' 'LÅ' 'GDÖ' 'SLE' 'T' 'MH' 'VÄ' 'SK' 'RMTP' 'SS'
'F' 'FBY' 'HR' 'VGÅ' 'A' 'VBD' 'NS' 'FD' 'SN' 'LR' 'ASD' 'APN' 'P'
'SEL' 'SÄV' 'GSV' 'OR1' 'OR' 'GRO' 'G'])}
declare T1sch={Ko schedule(tripSet:T1 useRels:false useAbstr:true $)}
declare T2sch={Ko schedule(tripSet:T2 useRels:false useAbstr:true $)}
declare Tripset1sch={Ko getTripset(id:T1sch $)}
declare Tripset2sch={Ko getTripset(id:T2sch $)}
{Tripset1 concretize}
{Tripset2 concretize}
{Tripset1sch concretize}
{Tripset2sch concretize}
declare Rels1={Tripset1sch makeRels(extractMeth:[locs depTime slots]
locList:['HPBG'] $)}
declare Rels2={Tripset2sch makeRels(extractMeth:[locs arrTime slots]
locList:['HPBG'] $)}
declare T3={Ko merge(T1 T2 $)}
declare Tripset3={Ko getTripset(id:T3 $)}
{Tripset3 addRels(rels:Rels1)}
{Tripset3 addRels(rels:Rels2)}
declare T3sch={Ko schedule(tripSet:T3 useRels:true $)}

% Test 3.
% Solve CST-K-G + CST-ORB-G in two steps, with relations added.
%
% 1.Abstract CST-K-G + CST-ORB-G in such a way that CST-HPBG becomes
% one single slot. Solve 1. Concretize. Extract relations at HPBG.
% Solve the remaining part of the plan with relations added.
declare Trips1={Compiler.virtualStringToValue "\\insert
'"#'trips/PHO_Stkhlm-Ktrnhlm-Gtbrg-Stkhlm1.oz'#"'\n" $}
declare T1={Ko addTrips(trips:Trips1 $)}
declare Tripset1={Ko getTripset(id:T1 $)}
{Tripset1 makeAbsSlots(locs:['CST' 'SUB' 'SPÅ' 'KHÄ' 'STT' 'TOT'
'EKO1' 'EP' 'VÅ' 'VÅV' 'KBÄ2' 'MORP' 'VSG' 'ARB' 'ÖA' 'ÖR' 'KLA'
'HPBG' 'TÄL' 'LÅ2' 'LÅ' 'GDÖ' 'SLE' 'T' 'MH' 'VÄ' 'SK' 'RMTP' 'SS'
'F' 'FBY' 'HR' 'VGÅ' 'A' 'VBD' 'NS' 'FD' 'SN' 'LR' 'ASD' 'APN' 'P'
'SEL' 'SÄV' 'GSV' 'OR1' 'OR' 'GRO' 'G'])}
declare T1sch={Ko schedule(tripSet:T1 useRels:false useAbstr:true $)}
declare Tripset1sch={Ko getTripset(id:T1sch $)}
{Tripset1sch concretize}
declare Rels={Tripset1sch makeRels(extractMeth:[locs depTime trips]
$)}
{Tripset1sch clearRels}
{Tripset1sch addRels(rels:Rels)}
declare T2sch={Ko schedule(tripSet:T1sch useRels:false $)}
```

```
% Test 4 - as Test 3, but without relations.
% Solve CST-K-G + CST-ORB-G in two steps, without relations added.
%
% 1.Abstract CST-K-G + CST-ORB-G in such a way that CST-HPBG becomes
% one single slot. Solve 1. Concretize. Solve the remaining part of
% the plan.
declare Trips1={Compiler.virtualStringToValue "\\insert
'"#'trips/PHO_Stkhlm-Ktrnhlm-Gtbrg-Stkhlm1.oz'#"'\n" $}
declare T1={Ko addTrips(trips:Trips1 $)}
declare Tripset1={Ko getTripset(id:T1 $)}
{Tripset1 makeAbsSlots(locs:['CST' 'SUB' 'SPÅ' 'KHÄ' 'STT' 'TOT'
'EKO1' 'EP' 'VÅ' 'VÅV' 'KBÄ2' 'MORP' 'VSG' 'ARB' 'ÖA' 'ÖR' 'KLA'
'HPBG' 'TÄL' 'LÅ2' 'LÅ' 'GDÖ' 'SLE' 'T' 'MH' 'VÄ' 'SK' 'RMTP' 'SS'
'F' 'FBY' 'HR' 'VGÅ' 'A' 'VBD' 'NS' 'FD' 'SN' 'LR' 'ASD' 'APN' 'P'
'SEL' 'SÄV' 'GSV' 'OR1' 'OR' 'GRO' 'G'])}
declare T1sch={Ko schedule(tripSet:T1 useRels:false useAbstr:true $)}
declare Tripset1sch={Ko getTripset(id:T1sch $)}
{Tripset1sch concretize}
declare T2sch={Ko schedule(tripSet:T1sch useRels:false $)}
*/
```

Appendix B – TUFFScript

Abstract

This document describes a script language for TUFF. The language is only intended to be used together with TUFF, and is therefor rather limited. It handles variables, assignments, procedure and function calls, and provides a conditional statement as well as equality and inequality test operators. Variables in this script language are similar to variables in Mozart, and most of the types are present in the form of classes in the TUFF system.

The language provides the possibility of storing long series of commands to TUFF, and enables a more fine-grained control of the TUFF system. The many similarities between TUFFScript and the structures in TUFF are intended to make it easy for a user who is familiar with the GUI of TUFF to understand and create a TUFFScript. The similarities with Mozart will simplify the execution of the TUFFScript.

Types

The types in TUFFScript are of various kinds, but limited to the elements necessary for TUFF. All present types are described below.

Tripset

A Tripset can be described by a triple <Tasks, Rels, Res>. The Tripset may contain one or more tasks, which in turn may contain one or more subtasks. Rels specifies relations between tasks, and Res the resources necessary to perform the tasks. The tasks must not necessarily traverse the same tracks, or even the same geographical area.

A Tripset can be obtained by using the function GetTripset, which reads a specification from file and creates a new Tripset. A Tripset may exist on various levels of abstraction. This is further described in a separate document[1]. An example of this is that the departure, traversal and arrival times for a tripset are either intervals or fixed points in time. This means that a Tripset may be the argument to a scheduling operation as well as the result of the same operation.

Abstract Tripset

An Abstract Tripset is a Tripset that has been abstracted. It works like a Tripset, except that no pathwindow can be shown, i.e. it is not possible to call procedure

[1] [Kre00]

ShowPathWindow with an Abstract Tripset as argument. However, if the Abstract Tripset is first concretized, a pathwindow can be shown.

Relations

Relations are used to add conditions to a Tripset. Relations may exist for any level of tasks or subtasks, and specify that there must be a certain order between two tasks or subtasks.

For instance, if a trip traverses the slots [1 2 3], then it is necessary to traverse slot 1 before slot 2. Relations can also be used to specify orders between trips. It is possible to restrict the order between a departing trip and another departing trip, between an arriving trip and another arriving trip, and between an arriving and a departing trip. The two possible ways of creating relations are on tracks and on locations. If Relations are created on locations, then the trips and slots arriving and departing at a certain location are considered.

If they are created on tracks, then only the trips or slots that traverses that that track are considered. Relations are treated the same way regardless of the basis on which they are created.

Extraction method

The Extraction method determines how to extract Relations from a Tripset. The extraction method is a list with three elements: [Element1 Element2 Element3]. The elements have the following objectives:

1. The first element is a either a list of resources, describing the resources for which relations should be created, or the atom tracks or locations. In the case of an atom, it determines whether relations should be determined based on locations or tracks.

2. The second element is arrTime, depTime or arrDepTime, and specifies if relations should be extracted based on arrival time, departure time or both arrival and departure time.

3. The third element can be slots or trips, and tells the system to generate relations between slots or between trips.

Example: A valid Extraction method is [tracks depTime slots]. This example shows how to specify an Extraction method for the departure time of all slots on common tracks.

Filename

A Filename is a text-string that specifies the location of a file. The text-string should be enclosed with quotation signs.

Example: ‘‘/home/me/my_plans.oz”

Path description

A Path description is either a text-string or a record, which describes the path that should be shown by the procedure DrawSchedule. It must specify a valid path. If the path is specified by a text-string, it must be one of the predefined paths. If the path description is a record, it should have the features orig, dest and locs. orig should be the location that the path originates from, dest the location that is the destination of the trip, and locs a list of locations that the trip should pass on its way from orig to dest.

Examples:

'CST-K- 0' (this is a text-string),
path(orig:'CST' dest:'G' locs:['K']) (and this is a record).

List

A list may contain one or more elements of one of the types listed above. All elements in a list must be of the same type. The list is enclosed with brackets.

Example: [E1 E2 E3].

Operators

The assignment operator =

Assignments are made with the operator =. A variable can be assigned the return value of a function call, or the value of another variable. Since all variables are single assignment, the assignment operator can operate only once on each variable.

Example: Variable1 = Variable2

The equality test operator ==

Tests can be made to check the equality of all combinations of return value and variables. That is, equality can be checked between two variables, the return values of two function calls or a variable and a the return value of a function call. The test operator returns the Boolean value true if the structures of the two compared items are equal. If the structures are unequal, the Boolean value false is returned.

Example: Variable1==Variable2 may return true or false.

The inequality test operator \=

This operator works similarly to the equality test operator, except that it tests inequality. It returns one of the Boolean values true or false.

Variables

Variables are dynamically typed, and do not have to be declared. All variables are single-assignment, and thus have a lot in common with variables in Mozart. All names of variables must start with a capital letter. The name of a variable may contain letters, numbers and underscores.

Function and procedure calls

Function calls should follow one of the patterns

Variable_name = Function_name (Argument1) or

Variable_name = Function_name (Argument1, Argument2)

A function always returns a Tripset or the reserved word FAILURE if the function failed. The return value must be stored in a variable, and since all variables are single assignment, no variable may be assigned a value more than once.

The functions available are listed below in Table 1. For instance, Schedule takes as argument a Tripset, and returns a new Tripset, where the intervals of the departure and arrival times have been reduced to fixed points in time.

The names of all functions reflect the terms used in the GUI of TUFF. This is a deliberate attempt to make it easier for someone familiar with TUFF to understand a TUFFScript.

Procedure calls are similar to function calls, except that they do not give any return value. A procedure call should follow one of the patterns

Procedure_name (Argument1) or

Procedure_name (Argument1, Argument2)

The two available procedures in TUFFScript and their arguments are shown in Table 2.

Function	Arguments	Return value
Schedule	Tripset	Tripset
Circuit	Tripset	Tripset
GetTripset	Filename	Tripset
Merge	Tripset1, Tripset2	Tripset
LoadTripset	Filename	Tripset
CreateRelations	Tripset, Extraction method	Relations
NetAbstract	Tripset, strength	Abstract tripset
Concretize	Abstract tripset	Tripset
AddRelations	Relations, Tripset	Tripset
Future functions for abstraction		

Table 1: The functions available in TUFFScript, their arguments and return values.

Procedure	Arguments
ShowPathWindow	Tripset, Path description
ShowCircuitWindow	Tripset
SaveTripset	Tripset, Filename
Extract	Tripset

Table 2: The procedures available in TUFFScript.

Conditional statement

TUFFScript provides a conditional statement with the syntax

```
if B then S1 else S2 end
```

where B is a Boolean, S1 and S2 are statements. If B is evaluated to true then statement S1 is executed. Otherwise, S2 is executed. The conditional statement is the only statement that may contain other statements. A conditional statement may contain another conditional statement. This means, that a nested structure of

statements is allowed when using conditional statements.

Comments

All comments are started with the marker /* and closed with */. A comment can be placed on a line of its own as well as before or after a statement on a line. It can be one or more lines long, and may contain any characters.

Example: /* This is a comment*/