Reuse of Constraint Knowledge Bases & Problem Solvers explored in Engineering Design.

PETER M. D. GRAY[1], TREVOR RUNCIE[1], and DEREK SLEEMAN[1]

[1]Department of Computing Science, University of Aberdeen, Scotland, UK

Email: pmdgray@bcs.org.uk, truncie@abdn.ac.uk, d.sleeman@abdn.ac.uk

Corresponding Author: Professor Peter Gray, Computing Science Department, Meston Building, The University of Aberdeen, ABERDEEN AB24 3FX, Scotland, UK

Telephone: +44 (0)1224 272288

Short Title: Constraint Reuse in Engineering Design

Number of pages: Main Content (Textual): 35 (including References & Authors' Bios)

Number of figures: 7 (at end of paper)

Full title: Reuse of Constraint Knowledge Bases & Problem Solvers explored in Engineering Design.

Abstract

Reuse has long been a major goal of the Knowledge Engineering community. We present a case study of the reuse of constraint knowledge acquired for one problem solver, by two further problem solvers. For our analysis we chose a well known benchmark knowledge base system written in CLIPS, which was based on the propose-and-revise (P+R) problem solving method (PSM), and which had a lift/elevator knowledge base (KB). The KB contained 4 components, including constraints and data tables, expressed in an ontology which reflects the P+R task structure. Sufficient trial data was extracted manually to demonstrate the approach on two alternative problem solvers: a spreadsheet (Excel) and a constraint logic solver (ECLiPSe). The next phase was to implement ExtrAKTor, which automated the process for the whole KB. Each KB that is processed, results in a working system that is able to solve the corresponding configuration task (and not only for elevators). This is in contrast to earlier work which produced abstract formulations of the PSMs but which were unable to perform reuse of actual knowledge bases.

Subsequently, we have used the ECLiPSe solver on some more demanding vertical transport (VT) configuration tasks. We found that we had to use a little-known propagation technique described by Le Provost and Wallace. Further, our techniques did not use any heuristic "fix"' information, yet successfully dealt with a "thrashing" problem that had been a key issue in the original VT work. Consequently, we believe we have developed a widely useable approach for solving this class of parametric design problem, by applying novel constraint-based problem solvers to data and formulae stored in existing KBs.

Keywords: Constraint Solver; Knowledge Reuse; Propose & Revise PSM; Configuration; Code Generation.

1 INTRODUCTION

A vision of Knowledge Engineering is that, having built at considerable cost a Knowledge Base (KB) which is able to design, say a lift, it is highly desirable to reuse most of the domain knowledge, when developing a further Knowledge Base System (KBS) in the same domain using a similar (perhaps more powerful) problem solver. Further, this process should be relatively straightforward and handled by a domain expert rather than by a highly specialized programmer.

In this paper, we report a case study where we have reused domain knowledge which was originally implemented for use with a Configuration PSM called Propose-and-Revise (P+R). The initial problem solver, called VT (short for Vertical Transportation) was provided with a KB which enabled it to create lift (elevator) configurations from requirements provided by the end-user (Marcus et al. 1988), and it subsequently became a benchmark problem for the KA community (Schreiber & Birmingham, 1996). We then generated from the KB, almost automatically, knowledge structures which could be directly used by two further problem solvers (PSs) - a Spreadsheet (Excel) and a Constraint Logic Solver (ECLiPSe)[1]. The tool we developed to automate this is called ExtrAKTor.

We have been able to make progress in this because our KB only contains variable declarations, constraints and equations, for use in solving a parametric configuration problem, and these can be entered in any order. Thus we are storing mathematical objects in our KB, namely well-formed algebraic expressions using standard operators (and some conditional expressions), instead of rules or program fragments whose semantics depends on the interpreter used. This makes the whole process of reuse much easier; as a result we have been able to accomplish in the constraint domain what many of the early KB pioneers hoped for (Hayes-Roth et al. 1983). This has become possible through advances in the theory and practice of constraint solving (Van Hentenryck, 1989; Apt & Wallace 2007), and particularly in using

[1] A PSM is effectively an abstract specification of an algorithm, which is associated with a number of data sources, and which is usually realized as a (generic) Problem Solver (PS). For example, the Propose-and-Revise (P+R) PSM was implemented as part of the VT (Vertical Transport) KBS which comprises a generic (P+R) PS and the VT KB, (Corsar & Sleeman, 2007), (Runcie, 2008).

constraint Propagation through tabular data structures (Le Provost & Wallace 1991), without which our automatically generated code would have run far too slowly.

Thus we took a KB developed for a production-rule PSM, extracted the essential mathematical information, and then code-generated it for use by a contemporary constraint-based PS which did not require any of the heuristics, hints or fixes that are usually used by expert systems. This PS found solutions, and what is more it gave an efficient way of finding solutions when constants, parameters or component details were changed. This is an example of practical re-use in an engineering context.

Clearly, we are not attempting to develop novel configuration algorithms or new constraint-solving techniques. Instead we have concentrated on a generator that works unaltered not just on VT but across a range of different parametric configuration problems. It enables engineers to reuse data and specifications stored in an existing KB without having to master an unfamiliar programming language or mathematical notation. Our aim is to make a powerful, but little-used, constraint-based PS available to designers and implementers of AI engineering applications.

The structure of the rest of the paper is as follows: Section 2 describes the VT (Vertical Transport) design task and the Sisyphus-VT challenge, outlines related work, and gives an overview of relevant constraint satisfaction techniques; Section 3 describes the ontological structure of a Sisyphus-VT KB built for use with a CLIPS P+R algorithm for the lift domain, and an overview of how it was reused by the tool ExtrAKTor (Sleeman et al., 2006); Section 4 describes how we automatically generated a KB for a Constraint Logic Programming (CLP) PSM which reuses tabular knowledge efficiently, so overcoming serious performance problems (Runcie et al., 2008). Section 5 describes how we explored solutions with the CLP PS and systematically investigated the solution space; Section 6 summarizes our work, reflects on it and discusses planned future work; some readers may wish to skip to this on first reading.

2 Literature Review and Background

This work combines research from four different areas which we review in the following sections: PSM (Problem Solving Methods) and the Propose-and-Revise (P+R) PSM; Configuration problems; SISYPHUS Challenges; and Constraint Satisfaction Techniques.

2.1 ***Problem Solving Methods (PSMs) & their Reuse***

Problem-solving methods (PSMs) describe the principal reasoning processes of knowledge based systems (KBs). For a useful summary of PSM related research up to 1998, see (Fensel & Motta, 1998). It was appreciated that considerable benefits would accrue from a PSM library, since the construction of KBSs using proven components should reduce development time and improve reliability. This area of research has been most notably investigated through the KADS/CommonKADS Expertise Modeling Library and also through the Protégé PSM Library. These are examined in more detail below. Part of the KADS project (1983-1994) involved the creation of a PSM library; by the mid-90s the CommonKADS library contained hundreds of PSMs, (Breuker & Van de Velde, 1994; Schreiber, Wielinga et al., 1994).

A parallel development happened in the context of Stanford's Protégé-2000 system, which includes a widely used ontology editor. Here, the PSM Librarian and associated methodology provide an ontology based KB development model that enables reuse of PSMs. There are three distinct ontologies: (1) a domain ontology, (2) a method ontology and (3) a mapping ontology. The domain ontology is self-explanatory. The method ontology is a domain-independent characterization of a PSM's inputs and outputs. The mapping ontology is a mediator that defines explicit relationships between a particular domain and a particular method without compromising the independence of these distinct components.

This journal has recently published a special issue on PSMs. In the editorial, Brown (2009) summarizes the developments in knowledge-based systems (KBSs) which led to the initial concept (namely reusable procedural "building blocks"), asks whether the PSMs identified earlier are at the right level of granularity, and then discusses the potential roles for PSMs in the world of the Semantic Web. He also discusses the difficulties posed for engineers by the formal notation used in PSMs, and pleads for *"'PSM light' versions available as well, that is, versions using less intimidating languages"*. We believe we have met this challenge by generating, from one PSM, code that is *readable and editable* for two further PSs.

The paper in the special issue on PSMs which is closest to our activity is that by O'Connor et al. (2009). This paper discusses the (software engineering) challenges of implementing systems to process complex and diverse data sets which relate to the detection of outbreaks of infectious diseases. The first system described, BioSTORM-1, focuses on taking data from many data sources and storing it in a common data repository. The various PSMs or PSM-like agents which are then run over the data, effectively extract the necessary data from the repository. That is, each PSM is associated with a mediator which converts the data in the repository to the format required by the individual PSM-agent. In this system the data transformation process is driven by an ontology which describes the data requirements of the particular PSM. The BioSTORM-1 approach has been further generalized by MAKTab (Corsar & Sleeman, 2007), which additionally acquires from the user, in a guided way, information needed by the target Problem Solver (PS) that is not available in the source KBS.

A major issue with both the CommonKADS library and the Protégé PSM Librarian is that neither supports the *execution* of a KBS. Having stated that CommonKADS has hundreds of PSMs, Fensel and Motta (1998) state “None of these methods is implemented”. A contemporary critique of CommonKADS (Menzies, 1998) makes a similar point about lack of “operationalization”. There is a similar issue with the Protégé-2000 PSM Library. The PSM librarian webpage (SMI, 2010) states: “the current version of the PSM Librarian tab does not support actual activation”. By contrast, progress has been made by a few groups in the constraint solving area, who have taken steps to generate working code from formal descriptions in Essence (Frisch et al., 2008) or Numberjack (Hebrard et al., 2010). However, their starting point is a piece of discrete mathematics, rather than a KBS.

In summary, successive enhancements of PSM libraries have led to a largely unused set of PSMs. This is disappointing since the central purpose of the exercise was to support reuse. Currently, PSM libraries state how to solve KB tasks in formal terms, but do not help with the *actual* solution of such tasks.

Figure 1. VT System Components

2.2 ***The VT Task***

The Vertical Transportation (VT) Domain is a complex configuration task involving a sizeable number of components required to design a lift (elevator) system. These components are shown in Figure 1. The parameters such as physical dimensions and weight, and also the choice of certain components, are regulated by physical constraints. The VT domain (Marcus et al., 1988) was initially used to design lifts by the Westinghouse Elevator Company. This original VT domain knowledge was simplified by removing some Antagonistic Constraints (which we restored in our own studies – Sec.2.3.1) to form the Knowledge

Acquisition Sisyphus-VT Challenge. The Sisyphus version of the VT domain was created so that researchers would have a common KB for experimentation. It thus became a valuable benchmark (Schreiber & Birmingham, 1996). In fact, it is the Protégé (v3.0) version of the VT system which has been the KB used in this project[2]. This project has enabled us to test some constraint–solving PSMs (ie ECLIPSe) which were not available at the time of the Sisyphus challenge. We have also tackled a parametric design problem in Configuration which is still a topic of industrial interest.

2.2.1 Types of Configuration Problems

The VT system falls into the category of *Configuration* research involving constraints. This is currently an active area of research; there has been a recent special issue on it in this journal (Felfernig et al., 2011). The editorial says that: "*Configuration can be defined as the composition of a complex product from instances of a set of component types, taking into account restrictions on the compatibility of those component types.*" Much of the published research is about how to support designers in formulating constraints that express these restrictions, in various styles and formats. However, we are assuming this has already been done, and the constraints are available in a KB (which is usually populated using a user-friendly tool or graphic front-end).

Furthermore, we are not concerned with assembling from parts, where the end user can add in extra assemblies at will, or fill a shopping basket with desired components which are to be assembled somehow. An example of such a system is a "*product configurator*", which is a type of expert system used to automate the creation of quote prices, sales prices, bills of materials, and other product specifications. Such systems are widely used in industry and their advantages are evaluated in a recent review by Haug et al. (2011). However, we are dealing with solvers, not configurators.

Instead, the VT design constraints describe a lift or elevator with a *fixed* number of components of given types, in a fixed relationship to one another. In fact, ours is a *parametric* design problem where the variables (which must be solved for) represent physical parameters, such as component weight and size or

[2] Protege VT Sisyphus Ontology. ftp://ftp-smi.stanford.edu/pub/protege/S2-WFW.ZIP, Version copied: August 2004.

geometric distance. Also some of the variable values must be chosen from tabular structures, as is very common in engineering (and is central to our solving methodology). Configurators, of course, also use tabular data and extract values from it, but mainly for a kind of "synthetic" computation which calculates total costs and lead times, rather than an "analytic" computation which searches for solutions. Interestingly, this synthetic computation is similar to the "spreadsheet calculation" used by one of our generated PSs; it checks a given solution against the constraints and computes certain aggregates and shows variable dependencies, but it cannot actually search for solutions.

O'Sullivan (2002) describes how systems for interactive constraint-aided conceptual design start by establishing broad solutions and proceed in phases maybe as far as detailed physical design. It is this last stage we are concerned with, because we wish to re-use or modify an existing design. Early systems for constraint design, such as IDIOM (Lottaz et al. 1998), did find solutions (for geometric parameters in floor-planning). Currently, papers on solving for such parameters in existing designs are less common, probably because the main mathematical techniques are now well known. In our case these techniques are available in solver libraries within the ECLiPSe constraint logic programming system (see below). However, many of them lie unused because most engineers learn to calculate with procedural languages, and are not familiar with logic languages or constraint-based problem solving.

There has been surprisingly little interest in code generation by the constraint-solving community. Martin et al. (2011) discuss how it can be used to compile specifications from a formal algebraic language (Rules2CP or Zinc or Essence) into procedural code but, as is common, the evaluation is mainly in terms of performance gains. We think this under-rates the value, for an engineer, of being able to read and *check* generated declarations and calculations, even though they would be unwilling to formulate such expressions themselves. Checking of both equations and constraints is surprisingly easy in the ECLiPSe notation (section 3.3).

In summary, we are concerned with end users who have already built engineering KBs of constraint-based designs that they wish to re-use, maybe by changing certain key parameters and then solving the task again. We have worked on the VT KB because it is an established benchmark, but we believe the technique is applicable to parametric design problems that involves numerical or relational constraints expressed using standard algebraic operators. Furthermore, we report the case where the constraints in the KB were originally formulated for solving by one PSM, but we extracted and restructured them for use by different more powerful PSs.

2.2.2 The P+R PSM

The Propose and Revise (P+R) method (McDermott, 1998) initially used to solve the VT configuration problem was very dependent on codified expert advice. This PSM requires 3 types of information namely:

- A list of domain variables and tables (nowadays usually provided as part of a domain ontology)
- A set of constraints between these variables which needs to be satisfied for a configuration to be acceptable, and
- A series of *fixes* associated with each constraint that might be violated

In the VT domain, the fixes, provided by the experts, were generally quite straightforward. For example:

IF the lift's load (weight of cabin & maximum passenger load) > power output of the motor THEN use a more powerful motor.

When the VT system is activated it asks the end user to interactively provide values for particular features of the lift to be designed (eg the size of the lift shaft, the size of the door aperture, the number of passengers to be carried etc). When searching for a solution, if a constraint is not satisfied then the P+R PSM considers each of the fixes in the order given in the KB. If a fix resolves a constraint violation then no other fixes are considered, (but this fix may then trigger violations to other constraints which need fixing in turn, see 2.3.1.) If none of the fixes is able to resolve the constraint violation, then the PS cannot find a solution. Note that the P+R PSM does not attempt to try alternative fixes in order to produce an improved or optimal solution.

Section 2.3.1 discusses a basic weakness of the methodology, in that a fix for one constraint can undo the fix for a second constraint, and worse still, a fix for this constraint may then undo the fix for the first one (such pairs are called "Antagonistic Constraints"); it clearly leads to looping (thrashing). This problem was recognized in the original VT paper (Marcus et al., 1988) and a pre-processor was developed to detect such cases and modify the actions (section 2.3.1). A key question in our research was whether a different (CLP) PS would suffer from the same difficulties.

There is also some discussion about the nature of the "fix" information provided by domain experts. Some suggestions include: the fix is intended to satisfy the constraint in a way that tends to reduce cost or some other metric; the fix might help guide a search in a very large search space or in a highly constrained situation; the fix might decide to ignore "soft" constraints with low priority. The way in which end users provide this information is a topic for future research (see Section 6.1).

2.3 *The SISYPHUS-VT Challenge*

In many areas of Computer Science and Artificial Intelligence, benchmarks are set to evaluate the performance of a number of systems against a common set of tasks. The results of these tests are then used to identify the strengths and weaknesses of the several systems, and this in turn helps to set the future research agenda for the sub-field. So the Knowledge Acquisition / Modelling subfield set itself a number of challenges in the 1980s/90s which it called the Sisyphus challenges.

Seven papers were presented at KAW94, each of which described a methodology for modeling and solving Sisyphus-VT; these were: Soar / TAQL (Yost, 1996) , Protégé II (Rothenfluh et al., 1996), VITAL (Motta et al., 1996), CommonKADS, Domain-Independent Design System (DIDS), KARL / CRLM (Poeck et al., 1996) and DESIRE (Brazier et al., 1996) . Of these seven papers only the VITAL team, to their credit, reported multiple runs of their implementation (Motta et al., 1996). Further, Menzies (1998), reviewing the above papers, emphasizes that little testing was conducted on the various methods beyond the one example, which we extended (section 5.2).

2.3.1 Fix Interaction - Antagonistic Constraints

The looping behaviour caused by interacting fixes is discussed in the original VT expert system paper (Marcus et al. 1988). The section called "VT's Fix Interaction and Their Special Handling" refers to 37 chains of so called interacting fixes, including 3 pairs of "Antagonistic Constraints" that might cause thrashing. One of these pairs concerns the interaction of *Maximum Machine Groove Pressure* and *Maximum Traction Ratio*; we have explored this experimentally with our system (section 5.1).

Simplification of original VT in Sisyphus

In the process of creating the Sisyphus-VT challenge, the original VT expert system was significantly simplified. We later realized that certain fixes had been removed, in particular those for the *Maximum Machine Groove Pressure* constraint (C-48 in Sisyphus-VT) considered above. The fixes would appear to have been removed in order to break the chains of Antagonistic Constraints. Strangely, the Sisyphus documentation makes no mention of this, and it was not obvious. It is also worth noting that the P+R PS in Sisyphus-VT used the smallest components as a starting point and provided *upgrade* options only. Thus for example, once a large 50HP motor was selected, there was no way to look for solutions with a *smaller* motor. In conjunction with the removal of fixes just mentioned, this probably avoided loops, but at the cost of omitting parts of the search space. We decided to explore improvements using Constraint Satisfaction.

2.4 *An Overview of Constraint Satisfaction Techniques*

Constraint Satisfaction techniques (Van Hentenryck 1989) attempt to find solutions to constrained combinatorial problems, and there are a number of efficient toolkits in a variety of programming languages. The definition of a constraint satisfaction problem (CSP) is:

- a set of variables, and for each variable X_i, a finite set D_i of possible values (its domain), and
- a set of constraints $C_j \subseteq D_{j1} \times D_{j2} \times \ldots \times D_{jt}$, restricting the values that subsets of the variables can take simultaneously. These constraints are usually written as algebraic expressions over the variables, using the usual relational and arithmetic operators.

A solution to a CSP is a set of assignments to each of the variables in such a way that all constraints are satisfied. The main CSP solution technique is *consistency enforcement*, in which infeasible values are removed by reasoning about the constraints, using algorithms such as *node consistency* and *arc consistency*. CLP (Constraint Logic Programming) systems, such as ECLiPSe (see Section 2.4.1), borrow the syntax and some constructs (e.g. unification) from the logic language Prolog, but greatly improve on its performance; they do this by using CSP techniques to re-order goals dynamically within conjunctions.

Constraint propagation aims to remove early on those values that cannot participate in any solution to a CSP/CLP. It is usually activated (triggered) as soon as a new constraint is encountered, and this mechanism attempts to reduce the domains of all related variables (including domains that have been filtered by other constraints). If the domain becomes empty, then the entire sub-tree of potential solutions can be pruned. This is the real power of both CLP and CSP, as demonstrated by the classic eight-queens problem (Van Hentenryck, 1989). There is also a *Generalised Constraint Propagation* technique (Le Provost & Wallace, 1991) for variables with values given in tabular form. This fits the VT problem well, as it uses such tables to describe components and their attributes (section 4.1). It has been central to this study, but we believe that others have missed its utility, particularly when *generating code* as described in Section 4.

2.4.1 ECLiPSe - Constraint Logic Programming System

We used ECLiPSe, which is a CLP developed at two leading academic laboratories (University of Munich and Imperial College, London) supported by industrial and European funding[3]. It contains several constraint solver libraries and an integrated development environment. Like Prolog, its variables have fixed but unknown values and rely on a solver to find them. It also has much less dependence on statement ordering than programming languages such as C and Java, which makes it easier to use for code generation (section 3.3).

ECLiPSe Libraries as PSs: The ECLiPSe system is an extension of Prolog. The libraries are computational methods made available as compiled code accessed through named predicates, such as "*locate()".* They also introduce specialized data types. Effectively, they make available some of the latest research in CLP, as a kind of executable PSM. The **propia** library (ICPARC, 2003) provides an effective implementation of Generalised Constraint Propagation which is important to this approach.

The **ic** (interval constraints) library, is a crucial ECLiPSe library used to process constraints over simple numeric domains e.g. [3, 4, 5, 6] or more complex ranges e.g. [2..5, 8, 9..14]. The **sd** (symbolic domains) library conveniently extends this to the domains of symbols e.g. {x, motor, current}, which makes constraints much more readable. There is also a **branch-and-bound** library which allows one to repeatedly call a complex Prolog goal (maybe including constraints) so as to search systematically for solutions which improve the value of some metric (given as an expression)..

[3] ECLiPSe is now marketed as open source by Cisco under a Mozilla-style Public Licence for applications such as "planning, scheduling, resource allocation, timetabling, transport, ...". The current official website (ECLiPSe, 2010) provides links to ongoing developments of the source, and updated tutorials.

2.4.2 Bounded Reals in ECLiPSe

In addition to the basic numeric variable data types (integers, floats, and rationals), ECLiPSe also supports the numeric data type *bounded real*. Each bounded real is represented by a pair of floating point numbers. For example, the statements X > 3.5 and X < 9.2, assign X the value {3.5 .. 9.2}. The actual value of the number may not be known, but it definitely lies between the two bounds. Fortunately, many of the techniques used by the **ic** library for finite domains (lists of values) have been extended in ECLiPSe (ICPARC, 2003) to apply to bounded reals, even though these represent potentially infinite sets of reals. Without this extension CLP techniques would be too weak to solve the VT design problem, because many of the important variables are lengths or weights represented as reals. Any attempt to digitize the search space by replacing each real by a discrete integer variable, e.g. XInt = INT(X*1000.0), would dramatically slow performance and could miss solutions at finer granularity.

Note that ECLiPSe does not coerce integer values into bounded real values. Instead it acts as a *hybrid solver* by using the appropriate method for each type, according to whether the domain (whose values are being filtered) is represented by a list of integers or a pair of real number bounds.

The *locate(,)* predicate is used to direct search for precise values of Bounded Real variables. The predicate works by non-deterministically splitting the domains of the variables until they are narrower than a specified precision. For example, *locate([Cable length], 0.01),* can be used to split the domain of Cable length into a set of discrete values to find a value to satisfy the constraints. However, where restricting a value for one variable leads to a large set of small ranges for another variable, then the technique could give rise to a combinatorial explosion that could dramatically affect performance.

3 EXTRACTION & REUSE BY ExtrAKTor

Our aim was to see how far we could create a tool that would automate the process of extracting constraints and variable definitions from a KBS (based on one PSM) and then outputting them as knowledge structures which could be used by further PSs - namely, Excel Spreadsheet and the ECLiPSe Constraint Solver. The starting point for the process was the VT domain KB represented in Protégé, as indicated in Figure 2. This KB was part of the Stanford solution to the VT-Sisyphus challenge based on the P+R PSM (Yost, 1994).

The tool developed is known as ExtrAKTor. A scientist or engineer with a good understanding of the domain should be able to use ExtrAKTor to solve parametric design problems like VT. In contrast with existing research, the user should require neither a high level of computing science expertise nor detailed knowledge of the Problem Solver(s). In the following sections we consider the design of ExtrAKTor and how it extracts different KBs to satisfy the various PSs (ie Excel and ECLiPSe); once the appropriate KB has been extracted, ExtrAKTor then creates, and subsequently launches the corresponding, Excel-based, or ECLiPSe-based, KBS.

Figure 2. Overview of ExtrAKTor and the stages needed to create both ECLiPSe and Excel KBs
Figure 3. Elvis - VT Domain Ontology in Protégé

3.1 *Ontological issues for Knowledge Extraction*

Firstly, the various knowledge components have to be extracted from the original KB. Fortunately it uses a well-designed generic ontology, developed at Stanford using Protégé (SMI, 2003). This is the "*elvis*" ontology that reflects the inherent structure of parametric design tasks, including propose-and-revise tasks being considered in this study. Specifically, it describes: the actual domain ontology, the data tables for component values, the domain constraints, and the fixes, (see Figure 3). It represents this by four main classes: (1) *elvis-components* which lists the problem variable names used in the constraints (or as table column names), with their types and other metadata; (2) *elvis-constraints* which are subdivided into *assign-constraints*, *range-constraints* and *fix-constraints*; (3) *elvis-fixes* which are actions to resolve constraint violations; (4) *elvis-models* for the tables, with *elvis-model-slot* for their slot metadata. Instance data from this ontology is shown in section 4.2.2 and in Figure 3.

The division of constraints into three types is interesting. An *assign-constraint* looks like an equality constraint between a variable and an algebraic expression, but it is simply used to calculate a value for the variable by evaluating the expression. The expression involves other variables, and thus expands into a directed acyclic graph showing variable dependency independent of the order of the constraints; a spreadsheet PS obviously computes this graph, as does a CLP PS. The *range-constraints* are vital to the workings of a CLP PS; they give numeric upper and lower bounds for integers and reals. The *fix-constraints* look like *assign-constraints* but are usually inequality constraints for which one or more fix actions are provided. Note that the CLP PS needs the *fix-constraints* but our study shows that it does not need the fix actions (which are held for the P+R PS as *elvis-fixes)*.

This ontology enables us to describe parametric design problems for various domains. Thus ExtrAKTor works without alteration with very different sets of variables, tables and constraints. We have in fact tested it with the U-HAUL vehicle assignment domain, (Runcie, 2008). However, it does assume the KB is in the *elvis* constraint ontology.

Besides using *elvis*, the Protégé system used for the VT KB made extraction very easy, by providing alternative Tabs which behave like Application Programmer Interfaces (API). We used the PrologTab to execute a small piece of Prolog code we had written to export knowledge from the Protégé environment into a knowledge interchange format (KIF) suitable for use in an external Prolog environment such as GNU Prolog. Thus ExtrAKTor assumes the abstract knowledge structures of *elvis*; and its inputs must be formatted as Prolog lists. Hence, in order to use a KB not in Protégé, one would need to implement a Mediator to output the relevant parts of the KB in this KIF. Unfortunately, this is a well known problem with KB reuse. The long-term solution is for constraint researchers to adopt a common Ontology for their problems, which should be largely independent of the PSM / PS, as discussed in Section 6.

3.2 Extraction and creation of KBS for a Spreadsheet Solver

Initially we started with the Sisyphus-VT code for the Propose and Revise (P+R) PSM, which consisted of 20,000 lines (421 pages) of CLIPS code, including both the domain KB and a version of the (P+R) algorithm. We found that this original “KB” covered only one VT test case and was very hard to change. Instead, we wanted a system that would check whether any changes to the configuration would still satisfy the constraints (or maybe updated ones). So it was decided to implement a constraint checker which would check constraints for various datasets. Microsoft Excel was the spreadsheet tool chosen because it is widely available, and because it provides excellent interactive facilities for data presentation. In fact, spreadsheets are a very widely used and industrially important PS, though not recognized as such.

Figure 4. VT-Excel Emulator calculating dependent values (244 rows in spreadsheet)

The entire set of variables, their associated values, and constraints in the VT CLIPS code were extracted and an Excel spreadsheet for the task was created, (Sleeman et al., 2006). Initially this was done as an

experiment, using various text editing tools. Later the process was automated, as an extension to ExtrAKTor, by taking data from a Protégé Tab (as described above) and processing it with Excel Macros. All the variables in the CLIPS code were assigned to cells in column 1, and the corresponding Excel formula was placed in the corresponding cell in column 2 (see Figure 4). To make the formulae more readable, Excel's *define name* function was used to name the cells in column 1 with descriptive names such as "car.misc.weight", instead of the usual 'X99'. These names were also used in the formulae in the second column, for example: = door.operator.weight+(4*carguideshoes_weight) + …
Note that this has been transformed from CLIPS's prefix notation into the infix notation used by Excel. This was easily done with Excel macros. For further details see Runcie (2008) and Sleeman et al. (2006).

Figure 5. Component Table accessed as Excel worksheet

Tables such as Figure 5, containing component information in the original VT-Sisyphus code, were implemented as separate named Excel worksheets; these could be searched via the "*LOOKUP"* function. Note that the spreadsheet columns have to be named with the appropriate variable names (e.g. max.current), as discussed for the CLP version (section 4.1.3).

The spreadsheet algorithm was then able to take the values of *independent variables* given to it, and to use the formulae (in any order) to calculate all the other dependent variables, and also check that the constraint formulae all evaluated to 'True'. In the course of this we discovered that there were comparatively few independent variables in the VT problem (only about 12 out of over 300). Note that Excel cannot find *which* independent variable values would lead to dependent variables satisfying the constraints (this is why we need ECLiPSe). Nevertheless (section 2.2.1), a spreadsheet PSM is still a very useful way to investigate such dependencies, to check constraints, and to calculate and check aggregate values such as costs and other metrics. This shows the clear advantage of generating PSs for spreadsheets as well as CLP (Sleeman et al., 2006).

3.3 Code Generation of a KB for use with ECLiPSe

Having shown that we could reuse the VT KB with a spreadsheet PS, we wanted to use a much more powerful PS based on the ECLiPSe solver (described in Section 2.4). Once again, we used the *elvis* ontology (section 3.1), which describes the problem in terms of variables and various types of constraints. This fits well with the way problems are presented to CLP solvers (section 2.4), and most importantly, makes our solution very general and not specific to VT. So we carried out an experiment to manually transform a subset of the VT KB into ECLiPSe syntax and to solve it with the ECLiPSe toolkit. This was very promising, so the next stage was to take the *elvis* data from the intermediate KIF file imported into ExtrAKTor and to transform it automatically into the ECLiPSe notation. We used GNU Prolog to do this, because it has very general pattern matching facilities and is excellent for transforming and generating program text.

A major advantage of ECLiPSe notation is that someone with an engineering training can *read* it and *check* that individual constraints are what is wanted. They just need familiarity with basic algebraic equations and boolean expressions. This provides the vital reassurance that is often missing with large integrated packages. Remember that, without code generation, it is tedious to *write* out constraints and avoid errors, as the slightest error in a single variable name or an operator amongst thousands, can render the constraint set insoluble. Our goal was to reassure anyone with an understanding of the domain ontology that they could use our tool to define a configuration problem precisely enough for a computer to generate good solutions; but they would NOT have to labour long and hard with unfamiliar mathematical notation. This is a crucial point, often overlooked by those devising PSMs, who forget how easily users are deterred by unfamiliar notation or concepts, so that the PSMs remain published but unused.

3.3.1 Example Constraints

Our tool generates constraints that are syntactically well-formed formulae; they are algebraic expressions with the usual infix operators:

```
ic:(Hoist_cable_traction_ratio >  (Groove_multiplier * Machine_angle_of_contact) + Groove_offset).
```

The "ic:" indicates to ECLiPSe that a hybrid integer/real arithmetic constraint solver, is to be used (section 2.4.1). The identifiers have underscores as separators, as is normal in Prolog, instead of the dots used in CLIPS. More importantly, the names have been meaningfully constructed in this domain ontology. Table components start with a class name, such as Hoist_cable, which greatly aids readability.

Sometimes we need to generate a *conditional constraint*:

```
(Compensation_cable_quantity > 0 ->
        ic:(Counterweight_to_platform_rear >= 1);
        ic:(Counterweight_to_platform_rear >= 0.75 + 1.5)),
```

This is used as a kind of "case" construct to select a constraint. It should not be read as a kind of production rule. These are the most complicated expressions in the entire ECLiPSe KB, as can be seen in section 4.2.1. Note that this KB is just a sequence of declarations in a simple grammar; thus the usual text processing tools for finding names work very well both for searching and cross-checking.

The constraint information from the VT KB has been transformed but this is not the usual *syntax-directed* translation done by many compiler packages; this transformation software understands the semantics of the constraint solver and re-orders and restructures information to make good use of it. The major transformations involved 18 or so stored tables of component values, that needed to be integrated with the constraint solver in an efficient fashion, as detailed in section 4. Once this has been done, the KB is ready for the execution phase (section 4.2.6 and Figure 6).

4 GENERATING A CLP FOR THE EFFICIENT SOLUTION OF CONFIGURATION PROBLEMS

4.1 *Role of Tables in the Solving Process*

A significant innovation of our approach, as will become clear, is the use of the *elvis-models* component of the Protégé KB; these are structured *tables* of data values as in Figure 5. In a relational database these are usually seen as tuples that group attribute values for a given entity or that represent relationships between linked entities. However, in constraint solving they have another purpose, which is to *propagate constraints*. This is because the values in a column of the table implicitly define a restricted *finite domain* for one of the problem variables. Indeed, in a simple one-column table it does nothing else, but in a multi-column table there may be implicit restrictions on values in related columns that propagate to other problem variables. When the constraint solver makes good use of this information, it can speed up processing by an order of magnitude or more, which can save literally hours of processing time. The theory of this "Generalised Constraint Propagation" technique was developed by Le Provost and Wallace (1991), but without a sophisticated code generator to make use of it, in practice it seems to have been little used. We give below more details on its use than may be considered normal, but this is to enable others to repeat our experiments. This is partly because it is not described in the ECLiPSe reference book (Apt & Wallace, 2007) and there are very few examples online. Also, one needs to understand the form of the declarations in order to see that they can easily be extracted from any KB using *elvis*, and code systematically generated.

4.1.1 Declarations referring to variables in tables

Consider the *motor* table below in Prolog notation corresponding to the Excel worksheet in Figure 5, which we shall use as a repeated example. It is one of 18 such tables in the VT KB. The rows contain data for the following descriptors: *Motor_model, Motor_max_power, Motor_weight,* and *Motor_max_current*.

```
motor("motor_10HP",  10, 374, 150).
motor("motor_15HP",  15, 473, 250).
motor("motor_20HP",  20, 539, 250).
```

The first column contains strings which are possible values of Motor_model, identifying a type of engine with a certain horsepower. We know this is a key or identifying attribute, and the other values in the same row refer to this specific engine type. However, the solver needs to be told this, as explained below.

The solver is told to associate each column with a particular Prolog variable (actually by an "*infers most*" statement as in section 4.1.3). Thus the variable "*Motor_max_current*" has to take one of the values in the fourth and last column, in this case 150 or 250. Now the solver can use this for *constraint Propagation* in one of two ways. If it has found that the variable *Motor_model* has the value motor_20HP then it knows that "*Motor_max_current*" depends on this and must take the value in the same row, namely 250. However suppose instead it has found that "*Motor_max_current*" has value 250, but doesn't know *Motor_model.* It can only deduce that its value is "motor_15HP" or "motor_20HP", since they both have 250 in the fourth column. This fits very well with the *finite domain* reduction technique --- an alternative value or range of values is carried forward and used to eliminate possibilities; for example it may only match one item in a compatible column in another table, thus eliminating other alternatives. This style of reasoning is fairly easy to grasp, but it has been implemented remarkably well in ECLiPSe Propia library (ICPARC, 2003), leading to great practical benefit.

In the course of this analysis we realized that tables need not just contain attributes of physical objects; they may instead contain numerical values for coefficients and constants used in a complex conditional formula. As long as the formula is a conjunction of repetitive disjunctions (or vice versa) the technique will work as shown in section 4.2.5. Once again it leads to significant speedups.

4.1.2 The *local domain* declaration for column types

A *local domain* declaration for our motor table example looks as follows:

```
:-local domain(motor_model("motor_10HP", "motor_15HP", "motor_20HP")).
```

This tells the solver that motor_model (the domain of the first column in the *motor* table) can only contain some or all of certain disjoint values, which can be strings or integers (but not a mixture).

One restriction is that different local domain sets cannot overlap, thus "motor_15HP" cannot also appear in another local domain declaration. In consequence it is useful to specify all the anticipated values together, including some that are not currently being used, such as "motor_90HP".

We can associate the same local domain with columns in other tables. In a relational database, these columns would usually be key columns which are then matched by a relational join operator, maybe in order to pick up values of extra attributes. It is clearly useful. We do this with the *assignments*:

```
Motor_model &:: motor_model,    Motor_modelA &:: motor_model
```

Here Motor_model and Motor_modelA are two Prolog variables matched to separate columns in different tables that share the same local domain. Their names start with a capital letter to fit Prolog conventions.

4.1.3 Table Declarations making use of Propagation (Propia).

The Generalised Constraint Propagation technique (Le Provost & Wallace, 1991) introduced the "*infers most*" declaration. In ECLiPSe it is implemented remarkably efficiently by the Propia library (ICPARC, 2003). As noted above, this important construct tells the solver which Prolog variables (each with a local domain) to associate with which column. In our motor example this would be:

motor(Motor_model, Motor_max_Power, Motor_weight, Motor_max_current) *infers most*, The variables are, of course, listed in the same order as the columns, and so where a column has no solver variable, one uses the nameless Prolog variable "_". One *infers most* declaration is generated for each table, in a separate division following those for the local domains and their assignments.

The annotation *infers most* controls the extent of constraint propagation. An alternative with less Propagation is *infers unique*. Technically (ICPARC, 2003) these turn any goal into a constraint. In fact we only use it for a goal in the form of a term structure such as motor(,..,) where some of the variables have assigned local domains. Propia is told to extract the *most* information it can from the constraint before processing the Prolog goal. This information is then passed to the solver (by a kind of incremental compilation) so that it can explore alternative domain values efficiently in conjunction with other solver techniques without having to break off and do inefficient backtracking. In fact, the theory ensures that the new constraint accepts and rejects the same symbolic values as when using standard backtracking techniques, but with much faster processing (in our case, reducing hours to minutes).

An early use of this construct was in a previous project (Hui & Gray, 2000) to instantiate variables in "Data Table Functions" without backtracking. We hope our results will encourage others to appreciate its real value, especially when used in combination with code-generation.

4.2 ExtrAKTor Upgrade – Automatic Generation

We now consider the details of the process for systematically generating ECLiPSe code, and how we have been able to automate it almost completely, in a fashion that others can adapt or emulate.

4.2.1 Structure of Generated CLP

The text of the generated code has to be acceptable to the ECLiPSe parser. Thus there need to be: Type definitions, Table type definitions, Named constants, Variable definitions giving domain ranges, Equations defining some Derived variables, then Constraint formulae, and finally various Prolog clauses listing various Goals to be satisfied and variables to be printed.

Divisions: We grouped declarations of each type into separate "divisions"; the order of divisions is significant since each potentially makes use of items declared in previous divisions (section 4.2.3). The divisions are not marked specially for the Prolog parser, but they usually start with an underlined comment.

Where a division is empty, just the comment is left in. The partial ordering is designed so that the order of declarations or definitions within a division is immaterial. This is allowed by the declarative style of CLP and it makes code generation much easier. It avoids the problems of generating procedural code where re-ordering can produce different results when executed. The declarative form is also much easier for an engineer to read and check, as each definition can usually be checked independently of others.

Likewise our tool can ensure that all the variables are defined somewhere and that only such variables are referenced. This overcomes a major problem in early Prolog systems where a mis-spelt variable name was often not detected and assumed to be just another working variable.

Initial Code Generation: The ExtrAKTor system was developed and tested in two stages. Firstly we generated code for *elvis-components, elvis-constraints* and *elvis-models*. Execution of this code was completed in seconds, much faster than Sisyphus times, even allowing for current faster hardware. So we decided to relax some variable ranges and expand the search space, but this had a massive detrimental effect as execution times increased from seconds to hours. The performance degradation was traced to excessive backtracking through combinations of table values.

Final Code Structure: We then discovered how to use the Propia library (section 2.4.1), which reduced execution times back to seconds again. To do this we only needed to generate extra code corresponding to the "Local Domain Declaration" (section 4.1.2), and "Infers Most" (section 4.1.3) for each of the 15 or so tables. The details are given below and the declarations are easily generated from the Protégé KB, except where some information is missing (section 4.2.4). These declarations must be in separate divisions (ordered as in section 4.2.3) and must come before the constraints division.

4.2.2 The Knowledge Interchange Format

ExtrAKTor works on files of objects, exported from the Protégé KB in a *Knowledge Interchange Format* (KIF); this preserves in text form the complex directed graph connecting the objects in the knowledge base. Sample extracts are given below for our motor system example. If one were using a KB not built with Protégé then, in order to use ExtrAKTor, one would need to write a Mediator to output the relevant parts of the KB in this KIF. (This implementation would be easy or hard depending on the differences in the knowledge representations involved.)

Below we provide an extract of an example data table from such a KIF. Basically, wherever there is an object class which is a subclass of *elvis_models*, with a slotname ending *"_specs"*, and which has one or more instances each of which has a value for the slot "*model-name*", then these become a table of Prolog tuples named according to the class, with model-name values stored for convenience in the first column. Thus the *elvis* ontology uses *model_name* as a kind of reserved word for a column of key values.

```
([elvis_INSTANCE_00059] of motor-system
        (has-fixconstraints …)  (has-rangeconstraints …)
        (motor-specs
                [elvis_INSTANCE_00060]
                [elvis_INSTANCE_00061] )
                …….                           )
([elvis_INSTANCE_00060] of  motors
        (max.current 150)  (weight 374.0)
        (model-name  "motor_10HP")
        (max.power 10))
([elvis_INSTANCE_00061] of  motors
        (max.current 250)  (weight 473.0)
        (model-name  "motor_15HP")
        (max.power 15))
```

These instances are generated as Prolog tuples (as in Section 4.1.1).

```
motor("motor_10HP",  10, 374, 150).
motor("motor_15HP",  15, 473, 250).
```

For each table we generate an *infers most* statement (section 4.1.3) giving the variable names holding each column value; these are: Motor_model, Motor_max_power, Motor_weight, and Motor_max_current. Note that the keyword *model_name* referred to above is mapped onto variable Motor_model, holding an instance identifier of the motor type. For each variable such as Motor_model we generate (once only) a local_domain declaration (section 4.1.2) and a domain assignment. In all, 18 tables were generated. Once again, Prolog pattern matching makes this very straightforward, working on the generic Prolog term structure output from Protégé's PrologTab.

4.2.3 Declaring variable names and types

The types and constraints are in 11 *divisions* (section 4.2.1), which *must* be in this order:

- **Local domains** such as :- Motor_model(…,).
- **Tables of data tuples** such as motor("motor_10HP", 10, 374, 150).

- **Assignments** such as Motor_model &:: motor_model,
- **Integers with enumerated values** such as Compensation_cable_quantity :: [0, 2],
- **Integers with range constraints**, such as [Sling_underbeam]::108..190, or [Platform_width] :: 60 .. 1.0Inf, (with no upper bound),
- **Reals with range constraints** such as [Car_supplement_weight] :: 0.0 .. 800.0, (Remember that each real is manipulated as a [lower bound, upper bound] pair; the closer the bounds, the more precise the number.)
- **Non-negative reals** such as [Groove_offset, Groove_pressuremax] :: 0.0 .. 1.0Inf,
- **Real, Integer or String Constants** such as Door_opening_type = "side",
- **Assign-Constraints**: equality constraints that derive fixed values such as ic:(Car_return_left =:= Platform_width - Opening_width - 3.0), or conditional expressions as (Platform_width=<128 and Platform_depth=<108) -> ZZ=1 ; ZZ=5,
- **Infers Most** statements for each table such as motor(Motor_model,…) infers most,
- **Constraints** such as ic:(Car_buffer_load =< Car_buffer_loadmax), or ic:(Car_overtravel=<(Counterweight_runby + 1.5) * (Counterweight_buffer_stroke + 24)), {Note that they could even be non-linear}.

Note how the divisions used for Generalised Propogation (Local Domains, Assignments and Infers Most) interleave neatly with the other divisions. This concept of *divisions* with freedom to reorder declarations only within divisions, is implied by online documents but not spelt out or named as such.

As noted earlier, having a systematic way to generate the Prolog names from their original CLIPS form is important, in order to match variable names in declarations with those in constraints. Firstly, dots acting as separators are replaced by underscores and the first letter is capitalized to fit Prolog conventions. Also prefixes from Class names such as “motors-” become capitalized without the plural, as “Motor_”. Thus "motors-max.current" translates to “Motor_max_current”. However "machines-model-name" translates to “Machine_model” because of the special role of “model-name” noted above.

4.2.4 Missing Type information

Unfortunately there are some situations where the *elvis* VT ontology does not store enough type information. For example, it may declare car.speed as type INTEGER, where we need exact values:

```
    Car_speed :: [200, 250, 300, 350, 400],
```

The values could be taken from the key column, but this needs confirmation from the designer or user.

A similar problem arises with a table relating pairs of objects e.g. motormachine("motor_10HP", "machine_18"). Only one of the attributes can be called model-name but we need to associate models with both columns, as in "motormachine(Motor_model, Machine_model) infers most". Unfortunately the domain ontology just records the Machine column as "(type STRING)" without referencing its object class. Clearly, the Protégé *elvis-models* ontology needs to evolve to capture this additional type information; KB designers will also need to be aware of these subtleties.

4.2.5 Representing some constraints by extra tables

In a number of cases we played the role of a skilled knowledge engineer by replacing a repetitive or awkward constraint expression by an extra table type (or by adding columns to an existing table).

Consider this long repetitive constraint expression relating to machine groove pressure:

```
(or  (and (= ?car.speed 200)(eq ?machinegrooves-model-name machine.groove_K3269)
(> ?machine.groove.pressure (* 264 ?hoistcables-diameter)))
      (and (= ?car.speed 400)(eq ?machinegrooves-model-name machine.groove_K3269)
(> ?machine.groove.pressure (* 194 ?hoistcables-diameter)))
      (and (= ?car.speed 200)(eq ?machinegrooves-model-name machine.groove_K3140)
(> ?machine.groove.pressure (* 196 ?hoistcables-diameter)))   … 14 more lines ")
```

We significantly improved upon this representation by replacing the expression with a single formula taking its parameters from the extra table as shown below.

```
machinegroovepressure(Groove_model, Car_speed, Groove_pressure_factor)

machinegroovepressure("machine_groove_K3269", 200, 264).
machinegroovepressure("machine_groove_K3269", 400, 194).
machinegroovepressure("machine_groove_K3140", 200, 196).
```

Replacing such expressions, speeds up the computation by putting it in a form that suits the ECLiPSe solver. This also makes it more readable, and some five tables were added in this way. The transformation can be applied where there is a disjunction of conjunctions of repeated expressions of the same type and form, differing only in the values of some constants which are then tabulated. However, the transformation does require the analyst to have some basic competence in both CLIPS and Prolog. In order to avoid this, one could either try to spot the repetitions by using pattern matching in Prolog or else query the end user through an elaborate visual interface. This is a direction for future work (section 6.1).

4.2.6 Printing and returning results

The generated code is analogous to a collection of procedures to be called from a main program. Here, at the top level, the user may want to add certain extra goals as constraints. These might restrict an overall cost, or number of cables or some other important quantity. One could further restrict the design by giving a constant value to a variable, but within the range specified in the generated code.

Next one needs to call the special *locate()* predicate with a list of key bounded real variables to be solved for, together with the desired precision (see section 2.4.2). Finally one needs to specify the names of the output variables. An illustrative example of such a main program is:

```
 ic:(Power_min =< 12),            locate([Car_supplement_weight], 0.01),
write("CSW is"), write(Car_supplement_weight) ,nl, write("Motor_model is "), write(Motor_model), nl.
```

This would print the solution as a range of symbolic or numeric values, as below:

CSW is 500.00__500.01

Motor_model is Motor_model{[motor_10HP, motor_15HP]}

Note that, if a real variable cannot be precisely determined, a range will be reported, as for *CSW* above, and throughout the solver output, as shown in Figure 6 below. This is a major benefit, because it gives the user feedback about the precision of the solution; the user can then adjust parameters and rerun.

Figure 6. ECLiPSe Solution to the VT Problem with Bounded Reals

A tool in regular use would need the usual kind of graphic front end with pull-down menus giving lists of variables with hyperlinks to their descriptions and so on. Conveniently, the generated Prolog code can call out to such a GUI, (even to one written in C or Java).

Currently we print the first solution that is found, but a future direction is to call a Branch and Bound package to explore a range of solutions, looking for a variable value below some desired bound. The ECLiPSe *branch_and_bound* library (section 2.4.1) actually provides a predicate *minimize(,)* for doing this, which takes an expression for a utility function as a cost parameter. For example, one could replace the call to *locate* by: `Utility = Car_supplement_weight, minimize(locate([Utility], 0.01), Utility).`

5 EXPERIMENTATION - Exploring the Solution Space

5.1 *Key Parameters*

A key parameter in the Sisyphus-VT KB is the Car weight, as this affects the two most important values in the solution space, namely Machine groove pressure (MGP) and Hoist cable traction ratio (HCTR), as described in the original paper (Marcus et al., 1988) and in section 2.3.1. Car weight is calculated as the sum of several variables, as described in the following equation:

```
 Car_weight = Car_cab_weight + Platform_weight +Sling_weight + Safetybeam_weight
+ Car_fixture_weight + Car_supplement_weight + Car_misc_weight
```

All these variables have dependencies, except Car_supplement_weight (CSW) which is defined in the Sisyphus-VT documentation as being either 0 or 500.

We decided to iterate CSW over a larger range of values but our initial experiments did not show the expected relationship between CSW, MGP and HCTR; this led to the discovery of a small but crucial error in a constraint stored in the *elvis* ontology. Constraint C-48 was initially generated as:

```
ic: (Hoist_cable_traction_ratio > (Groove_multiplier * Machine_angle_of_contact) + Groove_offset)
```

However, on further investigation we found that the full Sisyphus-VT Documentation states, “the HOIST CABLE TRACTION RATIO is constrained to be at most 0.007888 Q + 0.675 {where Q = machine angle of contact}” This suggests that the ”>” should in fact be a ”<=” . Once corrected we then observed the expected behavior. We report this error because it shows that the modeler should never take information on trust, even if it is in an ontology and copied digitally. One needs to carry out experiments to see if the modeled behavior matches one's intuition and, if not, to explore why. This was also a practical test of the intelligibility and searchability of our generated code, when looking for constraints on variables such as HCTR and comparing them with a specification.

5.2 *Comparison with published Sisyphus VT results.*

We provide a comparison of the experimental results of this study with the earlier results obtained by the Sisyphus-VT study reviewed in section 2.3. We made the tasks harder by restoring the "Antagonistic Constraints". Our results confirmed that the constant *Car_supplement_weight* (CSW) was critical and had to be within certain bounds, which we determined more precisely. The speed of our system allowed us to search for solutions for a wide range of values of CSW, from 0 to 1000 in steps of 1, automatically running the KBS for each new value. Figure 7 shows the outcome of this test. We verified that, as in the original VT paper, when CSW increases MGP increases and HCTR decreases. With steps of 50, both variables appear to change in linear fashion, but using steps of 1 we see that HCTR actually behaves in a sawtooth fashion.

Figure 7. VT Solutions for *supplement weight* CSW from 0 to 1000 Step 1 (the decreasing sawtooth values are for *traction ratio* HCTR; increasing straight line values are for *groove pressure* MGP)

Prior to this experiment, we did not know whether some vital information was contained in the fixes, without which the computation might not terminate. However, even after including "*Antagonistic Constraints*", the computations all terminated. Even when some constraints disallowed all the solutions outside certain ranges of CSW, the system still terminated correctly, reporting there were no solutions.

On reflection, we realized that antagonistic fixes are an artifact of the P+R PSM and not inherent in the VT constraint problem. Modern constraint solvers have their own generic mechanism for deciding which goals to try, and in what sequence, and so they do not rely on ad-hoc fixes from domain experts. Furthermore, they work by incrementally removing values from domains, which is a one-way process.

They do not add and remove values for variables in the way that a fix can; the latter may lead to "thrashing". The successive pruning of domains may get steadily slower (especially for bounded reals, see section 6.1) but it will not create a loop. Hence constraints which had antagonistic fixes gave us no special difficulties.

Note that there is always provision for a programmer to direct the solver's behaviour by use of a "labelling()" predicate, which chooses the goals to try first and whether to try smaller variable values first (if values can be ordered). This does not alter what solutions are found, only the time to find them. We did not need to use this, which made generation simpler (section 6).

We also repeated the tasks tried by VITAL (section 2.3). These showed that for each of five given car speeds, we agreed on the value of car weight above which there were no solutions. However, VITAL could sometimes fail at lower weights when CLP did not. This showed the reliability of the CLP technique.

6 CONCLUSIONS, DISCUSSION AND FUTURE WORK

In reviewing this project we have arrived at the general conclusions listed below, and we hope this discussion will make it easier for others to apply these ideas in different contexts.

• Clear Conceptual Model and Constraint Ontology

It is now clear that Constraint Logic fits well with the *elvis* ontology used in our Protégé KB, since they both view the world in terms of entities, attributes, relationships and constraints. Similar kinds of ontology appear in tools used widely for object-oriented program design and database schema design. Thus the knowledge engineer has a well understood way to conceptualize and then represent the problem, and a wide choice of graphic tools to capture the information (including Protégé graphic editors).

The *elvis* ontology (section 3.1) lists all the variables used in the constraints, together with their types and in many cases their numerical ranges. The constraint formulae may use LISP syntax, but it is the syntax of well-formed algebraic expressions. Their semantics is not dependent on the working of a production rule interpreter. Thus we have returned to some of the fundamental tenets of early knowledge base pioneers, that a KB should be capable of reuse by a wide variety of different applications, even using different programming languages and running on different machines.

- **Transforming Components for Different Solvers**

As Figure 2 indicates, we have developed an approach which has enabled the several components of the *elvis* ontology for the P+R (propose-and-revise) PSM to be transformed, so that they can be executed by alternative PSs. To date we have generated and tested methods based on a spreadsheet (Excel) and a constraint solver (ECLiPSe). In hindsight, the *elvis* ontology did two crucial things to help us: it kept the constraints and tables independent of the fix information used by P+R; it also kept them independent of the specific VT lift domain. In consequence we were able to build a tool (ExtrAKTor) that works unchanged across a range of parametric design problems, including the U-HAUL transport problem.

- **Using a Spreadsheet as a Problem Solver**

ExtrAKTor was easily adapted (section 3.2) to generate a KBS for use with a spreadsheet PS. This PS is a very useful way to investigate dependencies. It showed that there were comparatively few independent variables in the VT problem (about 12 out of over 300). It was also useful to check constraints, and to calculate and check aggregate values such as costs and other metrics. Such calculations are very useful at early stages of configuration design, and have widespread industrial application.

- **CLP for Non-Experts**

All that the CLP solver needs to determine solutions is the readable ECLiPSe specification of the constraints, variables and tables, as generated, together with the top-level goals. Unfortunately many PSs rely on significant human expertise to read the formal description, often in unfamiliar symbols, and to create the input the solver needs (section 3.3). Thus there is a gap between formal specification and getting actual results, but the generation of an executable CLP (or a spreadsheet) bridges that gap. Although NumberJack (Hebrard et al., 2010) also generates executable code for a range of constraint solvers, it is only in a machine-readable representation (like object code), which helps constraint programming experts but not domain experts wishing to cross-check it.

In the case of VT, we did not even have to ask the design engineer for additional control information, such as “fixes” needed for (P+R), or heuristics, or hints to the constraint solver. This made the knowledge acquisition in *elvis* very straightforward. In consequence an engineer can now use these powerful theoretical techniques to solve constraint problems, without having to master the theory or an unfamiliar language. Indeed, we hope to encourage the capture of more constraint-based KB.

The generated code is also much easier to maintain, because one can usually just edit the KB through a tool and then regenerate it; this also eases the difficulty of an industrial shortage of maintenance programmers for logic programming languages.

• Integration of Structured Tables with CLP Variables

Tables of tuples (section 4.3) provide a familiar way to show sets of alternative structured values. Table column names are related to variables in the mathematical formulae for constraints. This, in turn, makes the problem description easier to read by an engineer, which increases confidence.

The application of Generalised Constraint Propagation to the values in tables has been crucial to finding solutions speedily, and so making our approach viable. The ECLiPSe toolkit with the Propia library made it very easy to do, by generating certain additional declarations automatically (section 4.2.2). Importantly, it is applicable across the whole range of problems describable in *elvis*. The theory may not be novel, but its practical value for efficient code generation seems to have been overlooked.

• Completely Automating Generation

We believe we have given enough information to enable our approach to be applied straightforwardly to KBs for parametric design problems. We are very close to making the code generation process completely automatic, so let us consider the remaining obstacles. Foremost is the need for a standard *ontology* for constraints and a shared *constraint interchange format* (even in RDF) to be used across many different applications. Obviously, the Protégé *elvis* ontology is a good starting point for parametric design problems but it has some shortcomings such as the restrictive use of *model_name* (section 4.2.4). Another rich way of specifying constraints is to capture both constraints and executable methods in an object model using the

Colan language (Ajit et al., 2008; Embury & Gray, 1995). Others argue for a more web-based kind of knowledge representation (Felfernig et. al., 2003).

There is the ongoing problem of educating KB designers in data modeling issues and systematic naming of variables and classes (section 4.2.4). Often, we have to clean up this information in the *elvis* ontology before we can generate the CLP. Finally, conditional expressions with a repetitive pattern may need to be converted automatically, as proposed in section 4.2.5.

6.1 *Future Work*

- **Interactive "sketch-and-refine" improvement process:** This approach has been tried in the earlier stages of constraint-based design, in conjunction with an expert human designer (Felfernig et al., 2011; O'Sullivan, 2002). Instead, we wish to use a *Branch-and-Bound* solver to improve on the initial solution values returned (section 4.2.6). It needs a *utility function* to measure the improvement, and this could itself be adapted interactively by a designer, by changing weights or altering a readable formula. This very much suits the flexibility of our readable and editable code-generation approach.
- **Recognizing repetitive patterns in if-then-else rules**: This would replace a group of if-then-else rules by a simpler parametrised rule which takes its values from a table (section 4.2.5). It would enable the ECLiPSe solver to work much faster, and also make the generated code more readable. In order to automate these transformations one could try spotting the repetitions by sophisticated pattern matching in Prolog. However, it might be better to involve the end user more directly in designing the tables through a more elaborate visual interface, such as that used for capturing integrity rules for scientific databases (Gray & Kemp, 2006). This approach uses visual cues to generate a combination of AND and OR operators. Thus the engineer would not have to compose Boolean expressions in a language such as CLIPS or Prolog.
- **A web service for ExtrAKTor**: This would support the extraction of constraint knowledge from web sources. Tools like MUSKRAT (Graner & Sleeman, 1993) would allow one to select knowledge sources which suit the pre-selected Problem Solver (PS), in our case CLP. However, it still requires agreement on a common Ontology and a KIF as discussed above, which would then make it possible to write Mediators so that ExtrAKTor could extract knowledge from such KBs.

- **Issues with more Complex search Spaces:** Because of its linear geometry and hence mostly linear constraints, VT is not a computationally "hard" problem. As described in section 5, we deliberately made it harder by re-introducing antagonistic constraints and by exploring extreme ranges of the parameter CSW, but met no obstacles Nevertheless other problems might include more complex non-linear constraints, which could be generated by our code generator. This could well slow down finding solutions, or even cause non-termination, since the CSP is NP-complete.

  A related issue is the use of Bounded Reals (section 2.4.2) to represent numerical values in ECLiPSe. Constraint Propagation can be used to reduce these bounds as usual, but when precise values are sought (e.g. by the ECLiPSe *locate* function) the propagation mechanism can reduce the bounds for variables to large sets of sub-ranges, that might fragment the search space and so become very slow to compute. One of the reviewers has suggested that "problem-specific fixes (or hints to the solver) derived from expert knowledge could be used to guide the search into areas of the space where solutions are expected, or they could be used to limit the search (and so sacrifice completeness in favor of efficiency)." These are challenges to CLP theorists and Configuration problem researchers. Moreover, by making it possible for more engineers to use the CLP technique in a disciplined fashion, we hope that our approach will stimulate another set of benchmark problems like VT.

ACKNOWLEDGEMENTS

This project has benefited from an association with the Advanced Knowledge Technologies (AKT) Interdisciplinary Research Collaboration (IRC) which was sponsored by the UK's Engineering and Physical Sciences Research Council (EPSRC) under grant GR/N15764/01. We acknowledge useful discussions of Research issues with Kenneth Brown, David Corsar & Stuart Chalmers. Additionally, we acknowledge discussions with various members of the Protégé team at Stanford University, including Mark Musen who made available their version of the Sisyphus-VT code. The figures are original.

REFERENCES

Ajit, S., Sleeman, D., Fowler, D & Knott, D. (2008). Constraint capture and maintenance in engineering design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 22*, 325-343.

Apt, K. & Wallace, M. (2007). *Constraint Logic Programming using ECLiPSe*, Cambridge Univ. Press.

Brazier, F.M.T., Van Langen, P.H.G., Treur, J., Wijngaards, N. J. E. & Willems, M. (1996). Modelling an elevator design task in DESIRE: the VT example. *IJHCS 44(3-4)*, 469-520.

Breuker, J. & Van de Velde, W. (1994). *The CommonKADS Library for Expertise Modeling*. Amsterdam: IOS Press.

Brown,D.C. (2009) "Editorial: Problem Solving Methods: Past, Present, and Future" *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 23*, 327–329

Corsar, D. & Sleeman, D. (2007). KBS development through ontology mapping and ontology driven acquisition. *4th Int'l Conference on Knowledge Capture*, Whistler, Canada, ACM, New York.

ECLiPSe (2010). *The ECLiPSe Constraint Programming System Home Page*. Accessed at *http://eclipseclp.org/* and *http://www.sourceforge.net/projects/eclipse-clp* on December 20, 2013.

Embury, S.M. & Gray, P.M.D. (1995). Planning Complex Updates to Satisfy Constraint Rules Using a Constraint Logic Search Engine. *RIDS '95 Second International Workshop on Rules in Database Systems*, LNCS 985:pp. 230-244. Springer-Verlag.

Felfernig, A, Friedrich, G, Jannach, D, Stumptner, M & Zanker, M. (2003). Configuration knowledge representations for Semantic Web applications. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 17(1)*, 31-50.

Felfernig,A., Stumptner,M. & Tiihonen,J. (2011). Special Issue Editorial: Configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 25*, 113-114.

Fensel, D. & Motta, E. (1998). Structured Development of Problem Solving Methods. *11th Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada.

Frisch, A.M., Harvey, W., Jefferson, C., Hernandez, B.M. & Miguel, I. (2008). Essence - A constraint language for specifying combinatorial problems. *Constraints 13*, 268-306

Graner, N. & Sleeman, D. (1993). MUSKRAT: A Multistrategy Knowledge Refinement and Acquisition Toolbox. *Second International Workshop on Multistrategy Learning*.

Gray, P.M.D. & Kemp, G.J.L. (2006). Capturing Quantified Constraints in FOL, Through Interaction with a Relationship Graph. *EKAW 2006, Podebrady*, LNAI 4248, pp. 19-26. Springer-Verlag.

Haug,A., Hvam, L. & Mortensen, N.H. (2011). The impact of product configurators on lead times in engineering-oriented companies. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 25*, 197-206.

Hayes-Roth, F., Waterman, D. & Lenat, D. (1983). *Building Expert Systems*. London, Addison-Wesley.

Hebrard, E., O'Mahony,E., & O'Sullivan,B. (2010). Constraint Programming and Combinatorial Optimisation in Numberjack. *CPAIOR 2010*, LNCS 6140: 181-185

Hui, K. & Gray, P.M.D. (2000). Developing Finite Domain Constraints - A Data Model Approach. *Computational Logic - CL 2000*, London, Springer, pp. 448-462.

ICPARC (2003). ECLIPSE Constraint Library Manual, Imperial College London**:**. Accessed at http://eclipseclp.org/doc/libman/index.html on December 20, 2013.

Le Provost, T. & Wallace, M. (1991). Generalised Constraint Propagation Over the CLP Scheme. *Journal of Logic Programming 16*, 319-359.

Lottaz, C., Stalker, R. & Smith, I. (1998). Constraint solving and preference activation for interactive design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12*, 13-27.

Marcus, S, Stout, J & McDermott, J (1988). VT: An Expert Designer That Uses Knowledge-Based Backtracking. *AI Magazine. Spring***:** pp. 95-111.

Martin, J., Martinez, T. & Fages, F. (2011). Procedural Code Generation vs Static Expansion in Modelling Languages for Constraint Programming. In *Recent Advances in Constraints*, LNAI 6384, 38-58. Springer-Verlag

McDermott, J (1998). Preliminary Steps Toward a Taxonomy of Problem-Solving Methods. *Automatic Knowledge Acquisition for Expert Systems*: 225-254.

Menzies, T. (1998). Evaluation Issues for Problem Solving Methods. *Knowledge Acquisition Workshop98*, Banff.

Motta, E., Stutt, A., Zdráhal, Z., O'Hara, K. & Shadbolt, N. (1996). Solving VT in VITAL: a study in model construction and knowledge reuse. *IJHCS 44(3-4)*, 333-371.

O'Connor, M.J., Nyulas, C., Tu, S., Buckeridge, D.L., Okhmatovskaia, A. & Musen, M.A. (2009). Software-engineering challenges of building and deploying reusable problem solvers. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 23*, 339-356

O'Sullivan, B. (2002). Interactive constraint-aided conceptual design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 16, 303-328.

Poeck, K., Fensel, D., Landes, D. & J., Angele. (1996). Combining KARL and CRLM for designing vertical transportation systems. *IJHCS 44(3-4)*, 435-467.

Rothenfluh, T.E., Gennari, J.H., Eriksson, H., Puerta, A.R., Tu, S.W. & Musen, M. A. (1996). Reusable ontologies, knowledge-acquisition tools, and performance systems: PROTÉGÉ-II solutions to Sisyphus-2. *IJHCS 44(3-4)*, 303-332.

Runcie, T. (2008). *Reuse of Knowledge Bases & Problem Solvers Explored in the VT Domain*. Ph.D Thesis, Dept. of Computing Science, University of Aberdeen.

Runcie, T., Gray, P.M.D. & Sleeman, D. (2008). Constraint Satisfaction and Fixes: Revisiting Sisyphus VT. *Research and Development in Intelligent Systems XXV, Proc. AI-2008* (Bramer, Coenen & Petridis Eds.). London, Springer**:** 105-118.

Schreiber, A.T. & Birmingham, W.P. (1996). Editorial:the Sisyphus-VT initiative. *IJHCS 44(3-4)*, 275-280.

Schreiber, G., Wielinga, B., De Hoog, R., Van de Velde, W. & Anjewierden, A. (1994). CML: the CommonKADS conceptual modeling language. *EKAW94*.

Sleeman, D., Runcie, T. & Gray, P.M.D. (2006). Reuse: Revisiting Sisyphus-VT. *Managing Knowledge in a World of Networks, EKAW 2006, Podebrady*, LNAI 4248**:** 59-66. Springer.

SMI (2003). ELVIS. Accessed at http://protege.stanford.edu/plugins/psmtab/psmtab_download.html.

SMI (2010). PSMTab. Accessed at http://protege.stanford.edu/plugins/psmtab/PSMTab.html.

Van Hentenryck , P. (1989). *Constraint Satisfaction in Logic Programming,* MIT Press.

Yost, G.R. (1994). *Sisyphus 1993 - Configuring Elevator Systems,* SMI.

Yost, G.R. (1996). Implementing the Sisyphus-93 task using Soar/TAQL. *IJHCS 44(3-4)*, 281-301.

AUTHOR BIOGRAPHIES:

Peter Gray is well known in the International Database Research community. After a physics PhD at Oxford, and working on Production Control Systems with Plessey, he has been at Aberdeen University since 1968 and was awarded a personal chair in 1989. He started there with a pioneering group (1968-72) on both Equational and Logic Programming and then applied this to the integration of heterogeneous data models, developing the P/FDM testbed using Prolog. He has contributed articles to Springer's on-line *Encyclopedia of Database Systems* (2010). From 1996-2000 he headed the EPSRC-funded KRAFT consortium which built an evolving network for knowledge fusion using CLP and mediators.

Trevor Runcie is an Honorary Research Fellow at Aberdeen University, having completed his PhD in Computing Science in 2008. He also has a B.Sc (Engineering) and an MSc (Artificial Intelligence) from Aberdeen University. Trevor has over 25 years of commercial IT experience and is the owner of a specialist software company "Agile Knowledge Management Limited". His main research interests are in knowledge management, constraints, engineering design and ontologies. He is currently investigating the use of constraint solving techniques to optimize oil well drilling and manpower scheduling.

Derek Sleeman was a Lecturer at the University of Leeds and Associate Professor at Stanford before joining Aberdeen as Professor of Computing Science in 1986. His Research activities are at the intersection of AI and Cognitive Science, and include systems for Intelligent Tutoring, Knowledge Refinement, Knowledge Reuse, and Ontology Management. He has been involved in all the KCAP series of meetings; and was the Conference Chair for KCAP-2007. He has also served on various Editorial boards including the Machine Learning Journal and the IJHCS. Latterly, he was a PI on the EPSRC-sponsored IRC in *Advanced Knowledge Technologies,* funded 2000 – 2007. Sleeman was elected a Fellow of Royal Society of Edinburgh in 1992 and a Fellow of ECCAI in 2004.

FIGURE CAPTIONS

Figure 1: VT System Components.

Figure 2: Overview of ExtrAKTor and the stages needed to create both ECLiPSe and Excel KBs.

Figure 3: ELVIS - VT Domain Ontology in Protégé

Figure 4: VT-Excel Emulator calculating dependent values (244 rows in spreadsheet)

Figure 5: Component Table accessed as Excel worksheet

Figure 6: ECLiPSe Solution to the VT Problem with Bounded Reals

Figure 7: VT Solutions for *supplement weight* CSW from 0 to 1000 Step 1 (the decreasing sawtooth values are for *traction ratio* HCTR; increasing straight line values are for *groove pressure* MGP)

Figure 1: VT System Components.

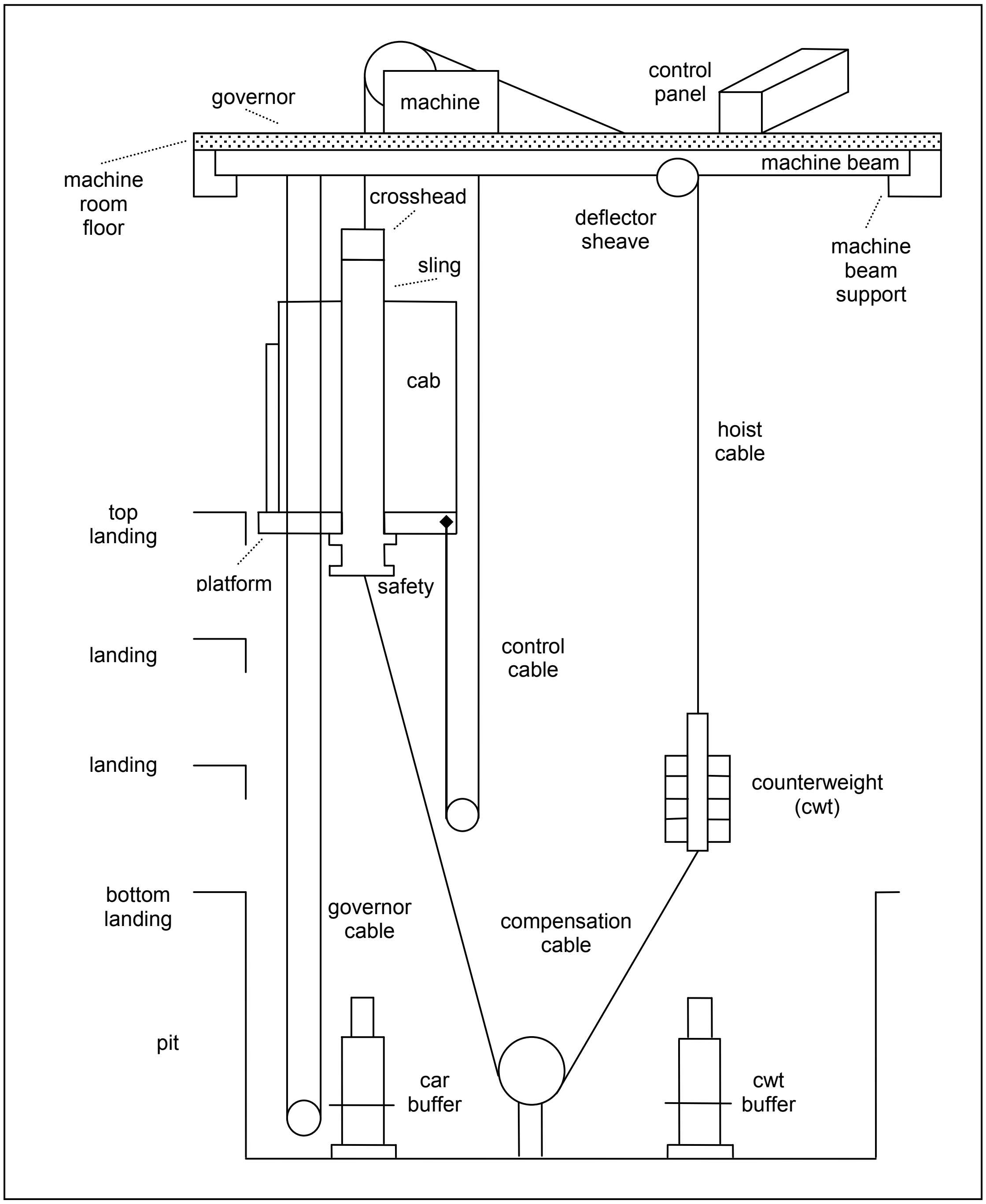


Figure 2: Overview of ExtrAKTor and the stages needed to create both ECLiPSe and Excel KBs.

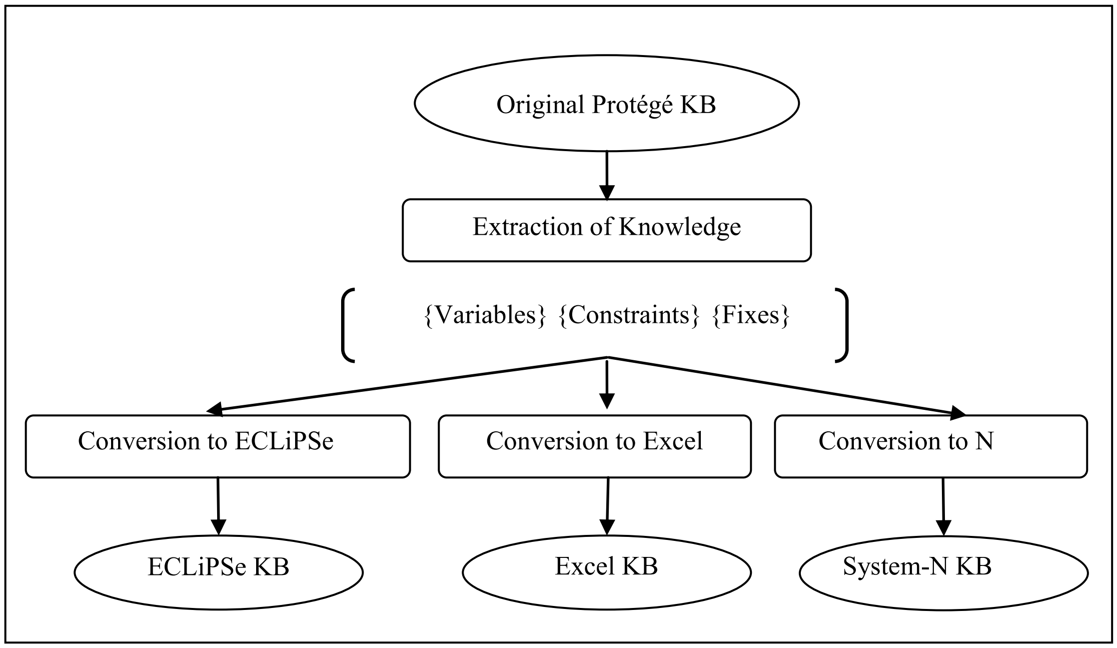
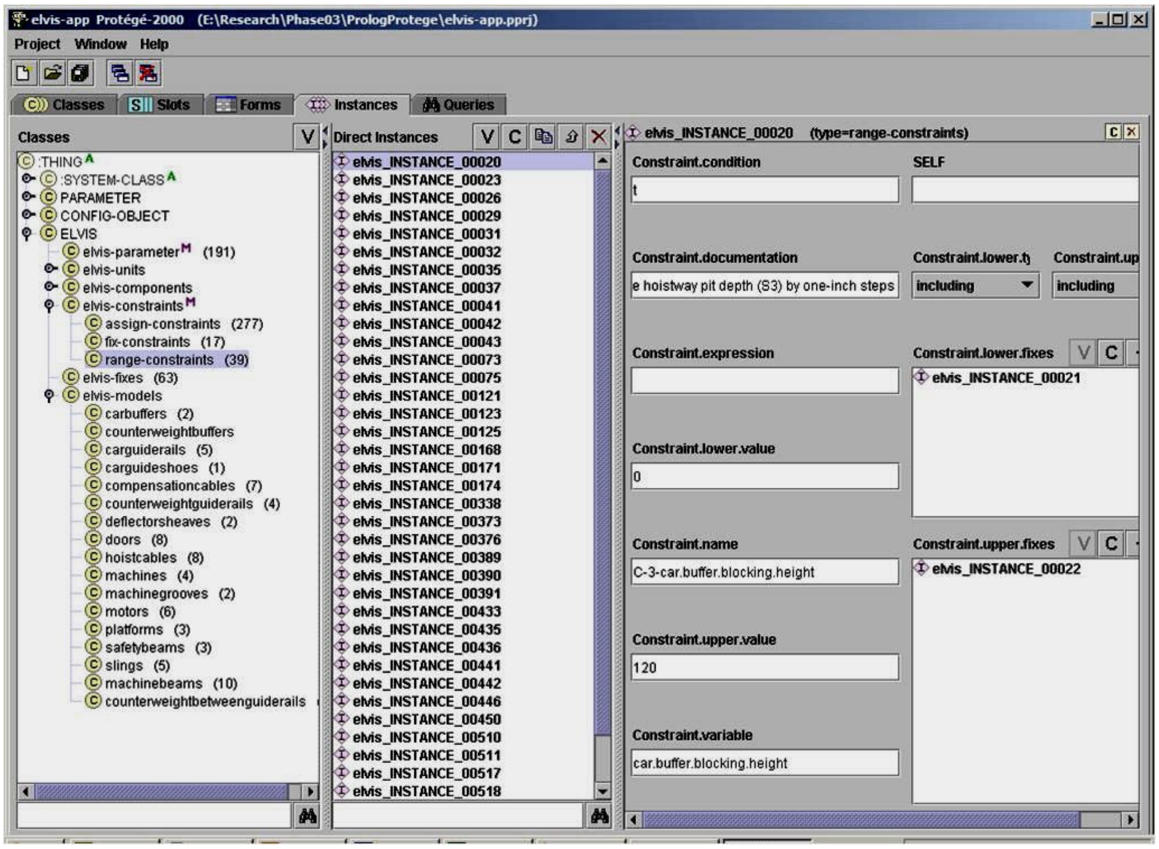


Figure 3: ELVIS - VT Domain Ontology in Protégé


elvis-app Protégé-2000 (E:\Research\Phase03\PrologProtege\elvis-app.pprj)

Project Window Help

Classes | Slots | Forms | Instances | Queries

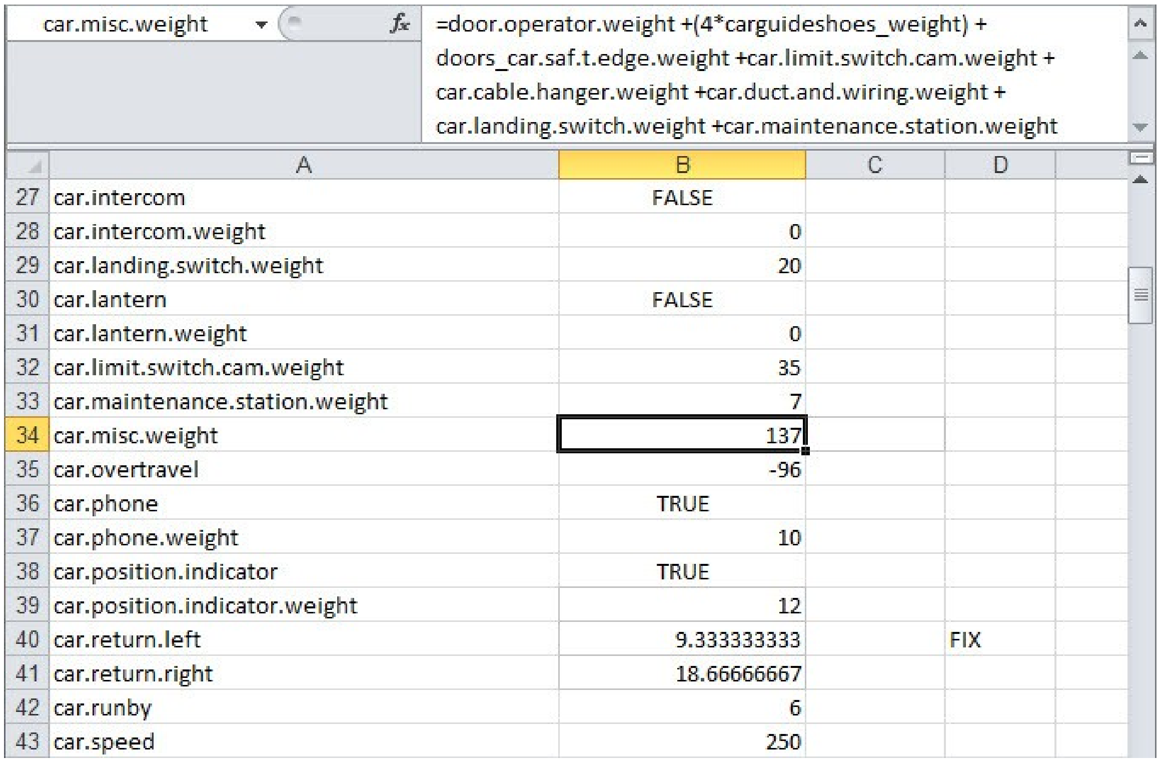
```
Classes
:THING
  :SYSTEM-CLASS
  PARAMETER
  CONFIG-OBJECT
  ELVIS
    elvis-parameter (191)
    elvis-units
    elvis-components
    elvis-constraints
      assign-constraints (277)
      fix-constraints (17)
      range-constraints (39)
    elvis-fixes (63)
    elvis-models
      carbuffers (2)
      counterweightbuffers
      carguiderails (5)
      carguideshoes (1)
      compensationcables (7)
      counterweightguiderails (4)
      deflectorsheaves (2)
      doors (8)
      hoistcables (8)
      machines (4)
      machinegrooves (2)
      motors (6)
      platforms (3)
      safetybeams (3)
      slings (5)
      machinebeams (10)
      counterweightbetweenguiderails
```

```
Direct Instances
elvis_INSTANCE_00020
elvis_INSTANCE_00023
elvis_INSTANCE_00026
elvis_INSTANCE_00029
elvis_INSTANCE_00031
elvis_INSTANCE_00032
elvis_INSTANCE_00035
elvis_INSTANCE_00037
elvis_INSTANCE_00041
elvis_INSTANCE_00042
elvis_INSTANCE_00043
elvis_INSTANCE_00073
elvis_INSTANCE_00075
elvis_INSTANCE_00121
elvis_INSTANCE_00123
elvis_INSTANCE_00125
elvis_INSTANCE_00168
elvis_INSTANCE_00171
elvis_INSTANCE_00174
elvis_INSTANCE_00338
elvis_INSTANCE_00373
elvis_INSTANCE_00376
elvis_INSTANCE_00389
elvis_INSTANCE_00390
elvis_INSTANCE_00391
elvis_INSTANCE_00433
elvis_INSTANCE_00435
elvis_INSTANCE_00436
elvis_INSTANCE_00441
elvis_INSTANCE_00442
elvis_INSTANCE_00446
elvis_INSTANCE_00450
elvis_INSTANCE_00510
elvis_INSTANCE_00511
elvis_INSTANCE_00517
elvis_INSTANCE_00518
```

elvis_INSTANCE_00020 (type=range-constraints)

Constraint.condition	t
SELF	
Constraint.documentation	e hoistway pit depth (S3) by one-inch steps
Constraint.lower.ty	including
Constraint.up	including
Constraint.expression	
Constraint.lower.fixes	elvis_INSTANCE_00021
Constraint.lower.value	0
Constraint.name	C-3-car.buffer.blocking.height
Constraint.upper.fixes	elvis_INSTANCE_00022
Constraint.upper.value	120
Constraint.variable	car.buffer.blocking.height

Figure 4: VT-Excel Emulator calculating dependent values (244 rows in spreadsheet)

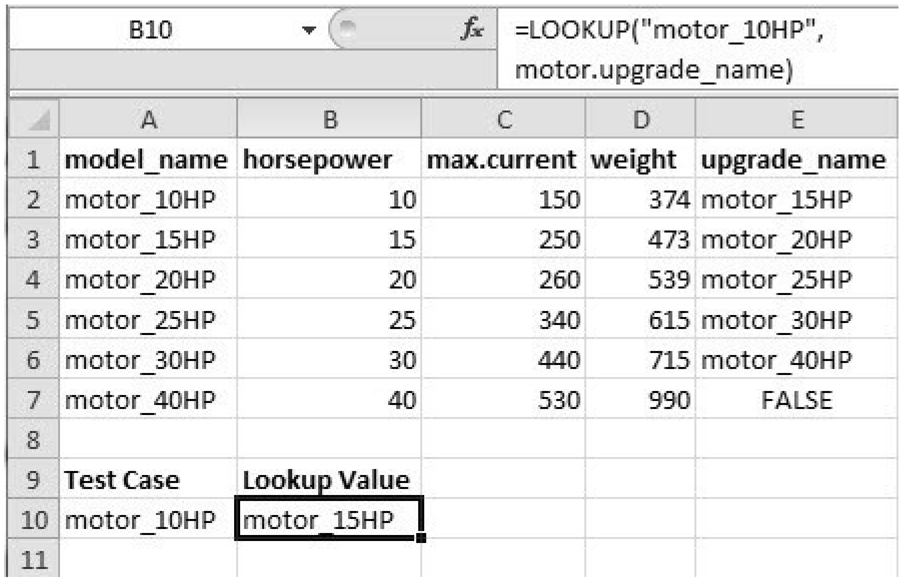


car.misc.weight

=door.operator.weight +(4*carguideshoes_weight) + doors_car.saf.t.edge.weight +car.limit.switch.cam.weight + car.cable.hanger.weight +car.duct.and.wiring.weight + car.landing.switch.weight +car.maintenance.station.weight

	A	B	C	D
27	car.intercom	FALSE		
28	car.intercom.weight	0		
29	car.landing.switch.weight	20		
30	car.lantern	FALSE		
31	car.lantern.weight	0		
32	car.limit.switch.cam.weight	35		
33	car.maintenance.station.weight	7		
34	car.misc.weight	137		
35	car.overtravel	-96		
36	car.phone	TRUE		
37	car.phone.weight	10		
38	car.position.indicator	TRUE		
39	car.position.indicator.weight	12		
40	car.return.left	9.333333333		FIX
41	car.return.right	18.66666667		
42	car.runby	6		
43	car.speed	250		

Figure 5: Component Table accessed as Excel worksheet

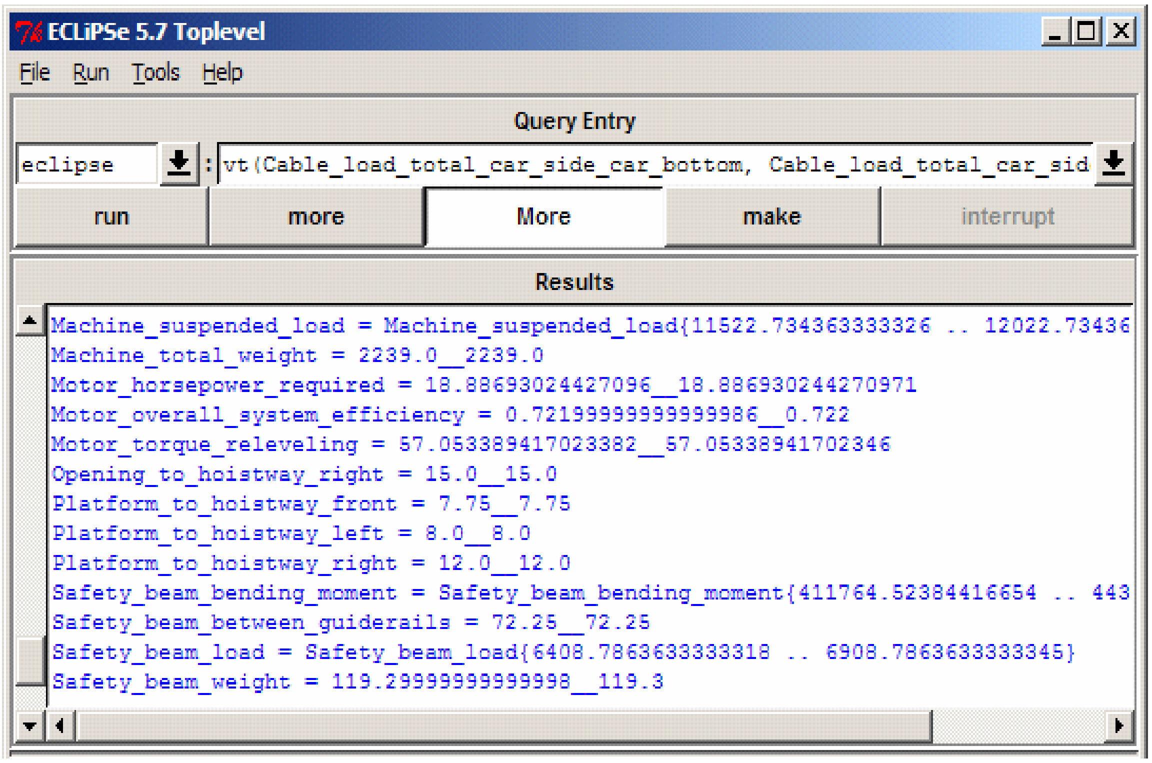


B10

=LOOKUP("motor_10HP", motor.upgrade_name)

	A	B	C	D	E
1	**model_name**	**horsepower**	**max.current**	**weight**	**upgrade_name**
2	motor_10HP	10	150	374	motor_15HP
3	motor_15HP	15	250	473	motor_20HP
4	motor_20HP	20	260	539	motor_25HP
5	motor_25HP	25	340	615	motor_30HP
6	motor_30HP	30	440	715	motor_40HP
7	motor_40HP	40	530	990	FALSE
8					
9	**Test Case**	**Lookup Value**			
10	motor_10HP	motor_15HP			
11					

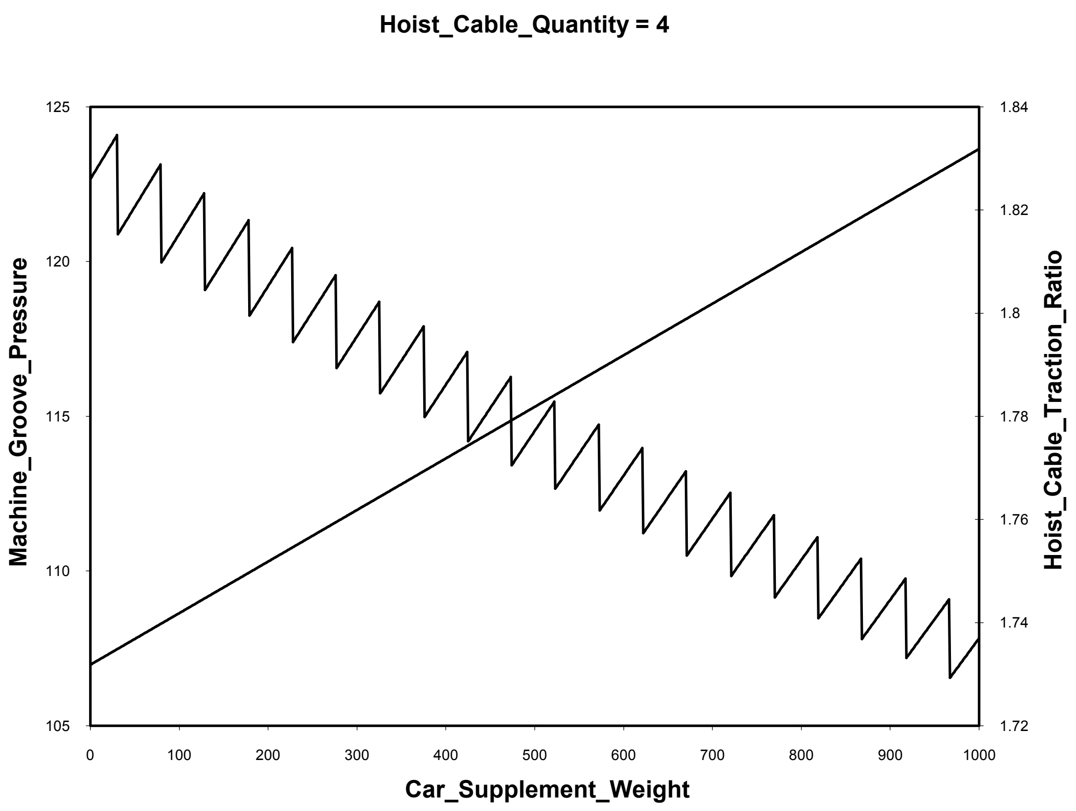
Figure 6: ECLiPSe Solution to the VT Problem with Bounded Reals



```
vt(Cable_load_total_car_side_car_bottom, Cable_load_total_car_sid
```

```
Machine_suspended_load = Machine_suspended_load{11522.734363333326 .. 12022.73436
Machine_total_weight = 2239.0__2239.0
Motor_horsepower_required = 18.88693024427096__18.886930244270971
Motor_overall_system_efficiency = 0.72199999999999986__0.722
Motor_torque_releveling = 57.053389417023382__57.05338941702346
Opening_to_hoistway_right = 15.0__15.0
Platform_to_hoistway_front = 7.75__7.75
Platform_to_hoistway_left = 8.0__8.0
Platform_to_hoistway_right = 12.0__12.0
Safety_beam_bending_moment = Safety_beam_bending_moment{411764.52384416654 .. 443
Safety_beam_between_guiderails = 72.25__72.25
Safety_beam_load = Safety_beam_load{6408.7863633333318 .. 6908.7863633333345}
Safety_beam_weight = 119.29999999999998__119.3
```

Figure 7: VT Solutions for *supplement weight* CSW from 0 to 1000 Step 1 (the decreasing sawtooth values are for *traction ratio* HCTR; increasing straight line values are for *groove pressure* MGP)


Hoist_Cable_Quantity = 4
Machine_Groove_Pressure
Hoist_Cable_Traction_Ratio
Car_Supplement_Weight