

CONSTRAINT SATISFACTION AND FIXES: REVISITING SISYPHUS VT

Trevor Runcie, Peter Gray and Derek Sleeman

Abstract This paper explores the solution of the VT Sisyphus II challenge using a Constraint Satisfaction Problem (CSP) paradigm and is an extension of the ExtrAKTor work presented at EKAW 2006. ExtrAKTor takes a Protégé KB describing a propose and-revise (PnR) problem, including both constraints & fixes. Subsequently, it extracts and transforms these components so that they are directly usable by the ECLiPSe CSP toolkit to solve a range of configuration tasks. It was encouraging to note that (a) the solver coped very well with constraints involving real variables even when using a generalised propagation technique and (b) the techniques needed no “fix” information, yet successfully dealt with the “antagonistic constraints” and the associated “thrashing” problem that had been a key issue in the original Marcus, Stout & McDermott VT paper. Consequently, we believe this is a widely useable technique for automatically generating and then solving this class of constraint problems, when they are expressed as Protégé ontologies.

1 Introduction

In a previous paper presented at EKAW2006 [7] we explored the reuse of knowledge bases through semi-automated extraction of knowledge and code generation of new KBs. The approach was explored in the context of the Sisyphus II VT (Vertical Transport) challenge and the related class of problems, and the “propose and revise” (P&R) algorithm. In this paper we extend that work by investigating how Constraint Logic Programming (CLP) techniques [12] can be used to find solutions to “Parametric Design” or “Configuration Tasks” such as VT, by working directly from the

Trevor Runcie
Department of Computing Science, University of Aberdeen,
Aberdeen, AB24 3FX, Scotland, UK e-mail: t.runcie@abdn.ac.uk

Peter Gray e-mail: p.gray@abdn.ac.uk · Derek Sleeman e-mail: d.sleeman@abdn.ac.uk

acquired knowledge so that one format both documents the knowledge and is also directly executable.

With the P&R PSM, as we have noted, each constraint is associated with a set of ordered fixes. When a “proposed” solution does not satisfy all the constraints, the first violated constraint is identified, and the associated fixes are applied in the order specified to change the initial value for a specified variable until that constraint is satisfied or the list of fixes is exhausted.

However, a crucial issue with P&R is that “thrashing” can occur. We report results using a CLP package where we use the CLP solver itself to search for consistent variable values which overcomes the “thrashing” problem, without needing any explicit fixes from the domain expert. This works even with “anatomistic constraints” which were excluded from the Sisyphus challenge KB.

The structure of the rest of the paper is as follows: Section 2 describes the VT design task, the Sisyphus-II challenge, and gives an overview of constraint satisfaction techniques; Section 3 describes the form of the generated constraint knowledge suitable for a CLP solver; Section 4 describes how we investigated the solution space and how we refined the knowledge format to deal with serious performance problems; Section 5 gives the experimental results, while Section 6 discusses related work, future work, and our overall conclusions.

2 The VT problem and VT Sisyphus-II Challenge

2.1 VT Problem

The Vertical Transportation (VT) domain is a complex configuration task involving the interaction of the components required to design a lift/elevator system. Parameters such as physical dimensions, weight and choice of components are regulated by physical constraints. The VT domain [3] was initially used to solve real-world lift design by the Westinghouse Elevator Company.

The Sisyphus [10] version of the VT domain was created so that researchers would have a common KB for experimentation. It is the Protégé version of the VT system from Stanford University which has been used in this project [9].

2.2 An Overview of Constraint Satisfaction Techniques

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

- a set of variables $X = \{X_1, \dots, X_n\}$,

- for each variable X_i , a finite set D_i of possible values (its domain), and
- a set of constraints $C_j \subseteq D_{j1} \times D_{j2} \times \dots \times D_{jt}$, restricting the values that subsets of the variables can take simultaneously.

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 from the problem by reasoning about the constraints using algorithms such as node consistency and arc consistency checking. Two important features introduced in CLP are goal suspension and constraint propagation, and these concepts are described in the following paragraphs.

Goal Suspension: In brief, suspension improves on Prolog by delaying the processing of arithmetic constraints until all their variables are fully instantiated. For example the Prolog statement, $2 < Y + 1, Y = 3$ would result in an error since in the first occurrence of Y, it is not sufficiently instantiated. Using the ECLiPSe statement, $(2 < Y + 1)$, the query is suspended until Y is fully instantiated and the query is ready for evaluation. This allows statements to be written in an order which is natural to the user but is still executable.

Constraint Propagation: Constraint propagation results in the removal of all values that cannot participate in any solution to a CSP. In ECLiPSe constraint propagation is activated (triggered) as soon as a new constraint is encountered, and this mechanism attempts to reduce the domains of all related variables including variables that may have already been considered during the processing of other constraints. If while enforcing arc-consistency, the set of acceptable values for a node is reduced to zero, then the entire sub-tree of potential solutions can be pruned. This is the real power of CSP.

2.3 ECLiPSe - Constraint Logic Programming System

ECLiPSe [11] is a software system for the development and execution of constraint programming applications. It contains several constraint solver libraries, a high-level modelling and control language, interfaces to third-party solvers, an integrated development environment, and interfaces to allow embedding in host environments.

ECLiPSe Libraries (sd and ic) The ECLiPSe system is an extension of Prolog. The sd (symbolic domains) library is the ECLiPSe library used to process domains of symbols e.g. $\{x, \text{motor}, \text{current_value}, \dots\}$, which makes constraints more easily readable. This extends the utility of the ic (interval constraints) library, which is the ECLiPSe library used to process simple numeric interval data e.g. $[3, 4, 5, 6]$ or more complex ranges e.g. $[2..5, 8, 9..14]$. The sd and ic libraries implement node-consistency, arc-consistency, suspend and constraint propagation.

ECLiPSe (Bounded Reals) 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 $\{3.5 \dots 9.2\}$. The true value of the number may not be known, but it is definitely known to

lie between the two bounds. Fortunately, many of the techniques used for finite domain (lists of values) have been extended in ECLiPSe[1] 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 crucial variables are lengths or weights represented as reals. The *locate/2* predicate can also be used to direct search for real variables. *Locate* 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.

3 An Overview of Structure

3.1 Initial Structure of *ExtraKtor* Generated Code

The format for the generated constraints and declarations was deliberately structured so as to create a compact structure that was eminently readable by knowledge engineers yet at the same time solvable. This was made possible by the declarative nature of the code. The format uses widely known logical and arithmetical syntax with the usual semantics, and so is easy to use for a wide range of configuration problems encountered in engineering. There was a big gain over the thousands of lines in the original CLIPS version. In the new version, the code segments each contain around 50 lines of code which are summarised below.

Within each code segment, statements can come in any order yet still give the same results on execution. For convenience the code was sorted into the most readable form with variables ordered alphabetically. Each section is described here in detail since the structure had a major bearing on the next stage of the project.

Prolog Data Tables (Tuples)

This is a standard Prolog data table structure. Each predicate represents a different entity and each parameter represents a specific attribute, as in a record structure or relational database. The “_” (underscore) represents “anything” in terms of Prolog pattern matching.

```
hoistcable(0.5, 0.03, 14500.0).
```

There are 28 of these tables in the system, with an average of 10 rows per table.

Variable Declarations

The following example shows integer variable declarations with lower and upper bounds. Note the “#:” syntax used to declare integers and also the use of “1.0Inf” to declare no upper bound (Infinity).

```
%Integers with Constraints
[Car_buffer_blocking_height] #: 0 .. 120,
[Platform_width] \#: 60 .. 1.0Inf
```

CONSTRAINT SATISFACTION AND FIXES: REVISITING SISYPHUS VT

Real declarations have a similar syntax to integers but without the use of “#”. The following example shows real variable declarations with lower and upper bounds.

```
%Reals with Constraints
[Car_runby] :: 6.0 .. 24.0,
[Car_return_left] :: 1.0 .. 1.0Inf
```

Data Table Functions

The data predicates pattern match against the tuples declared in the “Prolog Data Tables”. The Prolog predicate unifies with the rows in the tables, and the Prolog variables are instantiated to the column values.

```
hoistcable(Hoistcable_diameter, Hoistcable_unit_weight,
Hoistcable_ultimate_strength),
```

Assignments

These are simple assignments where the value of the right hand side of the equation is assigned to the left hand side of the equation. The “ic:” syntax tells ECLiPSe to use the ic solver when processing this line of code. Since ECLiPSe is a CLP system, any of the variables on the left and right side can be unknown at this point. For example, the assignment

```
ic:(2.5 == A + B)
```

is a perfectly acceptable construct, as is:

```
ic:(Car_buffer_striking_speed_maximum ==
1.15 * Car_speed)
```

Constraints

Constraints are used to limit the values that variables can take. ECLiPSe uses this information when performing arc-consistency checks and rapidly eliminates impossible combinations of values. General arithmetic expressions (even non-linear) may be included.

```
ic:(Car_buffer_load >= Car_buffer_loadmin)
```

3.2 *ExtraKTor Structure Summary*

As stated earlier, the code structure made it very easy to visually scan for variable names during debugging. It is worth noting that variable names are identical to the slot names used in the original ontology.

4 Investigating the Sisyphus-VT Solution Space

4.1 Constraint Types for Relaxation

The constraints in VT can be classified as one of 3 types:

- Universal Constraints
- Site Specific Constraints
- Design Specific Constraints

Universal Constraints: These express physical limits on certain parameters that apply to every VT design. Examples of this class are angles which can only be between 0.0 and 360.0 (expressed in degrees), and positive real values which must be greater than 0.0. *Site Specific Constraints:* These are physical limits of certain values that apply only to a specific site. An example is “the building has 8 floors”. *Design Specific Constraints:* These are physical limits of certain values that apply only to a specific design. An example is “the lift must be able to support a load of 10 passengers”.

4.2 Early Performance Issue

Surprisingly, processing of the ExtrAKTor generated code for the tasks specified by the Sisyphus VT challenge is completed in seconds. Universal constraints apply to all real world problems and Site Specific constraints cannot physically be altered for a given site, so it was therefore decided to relax the “Design Specific Constraints” to investigate an enlarged solution space. Relaxing the constraints and expanding the search space had a massive detrimental effect on the processing time for the original problem. Execution times increased from 0.01seconds to periods in excess of 1 hour (in fact the code never ran to completion). The performance degradation was traced to significant backtracking that was required for “component” selection.

The explanation seems to be as follows: Initially the unification process selects a set of components. Next, when constraints are specified, ECLiPSe determines that some of the initial choices of components violate the constraints. The system must then *backtrack* using standard Prolog techniques to select an alternative “component”. This then forces the assignments to be recalculated. Consider the following simple code fragment that demonstrates the problem.

```
a(999), a(998), ... a(1).
...
a(X), ... ic:(Z == X * 1.34),
ic:(Y == (Z * Z) / 2.33),
...
ic:(Y>3), %Constraint C2
ic:(X<2), %Constraint C1
```

First, X is instantiated to 999, and constraint C1 fails. Then, X is instantiated to 998, and constraint C1 fails. After 1000 instantiations, X is instantiated to 1, and constraint C1 is satisfied. With this style of code, hundreds of complex calculations may be performed on the instantiated values of X before C1 and C2 are tested and fail, forcing backtracking and recalculation of all intermediate values.

4.3 Performance Enhancement - “domain” & “infers most”

It was disappointing that ECLiPSe which had performed so well in most numeric constraint solving seemed to have such poor handling of lists of values and data tables. However, this prompted a more detailed review of the more esoteric capabilities of ECLiPSe which eventually led to the discovery of the “domain” and “infers most” constructs.

The “domain” and “infers most” constructs were not documented to any great extent in the ECLiPSe manuals, and are not mentioned in the ECLiPSe reference book [1]. Fortunately, one of the authors recalled its use in a previous project [13] and this showed how to use this construct to instantiate variables in “Data Table Functions” without backtracking. Initially, it took considerable trial and error to get the code to work, but the effort paid handsome dividends. Although the use of “domain” is independent of the “infers most” declaration it greatly extends its effectiveness. The following paragraphs document how we extended our format to keep it declarative while greatly enhancing the KBS’s performance.

The “infers most” [2, 8] construct requires the ECLiPSe Propia library to be loaded. This library implements the Generalised Constraint Propagation technique developed by Le Provost and Wallace [2]. The construct was able to use the implied constraint information in Prolog Data Tables far more efficiently than by standard backtracking, as described in the following paragraphs.

4.3.1 Domain Declaration

The first part of this construct declares a domain by providing a (domain) name and (domain) values. Importantly, domain values must be unique within the ECLiPSe KB; ie declaration of two domains; domain1('A', 'B', 'C') and domain2('C', 'D', 'E'), where element 'C' appears in two domains, is not allowed. The string constants act as a kind of object identifier, whose associated values (e.g. Power of motor model) are taken from the tables.

```
%Domain Declaration
:-local domain(motor_model("motor_10HP", "motor_15HP",
"motor_20HP", "motor_25HP", "motor_30HP", "motor_40HP").
```

4.3.2 Tuple Declaration

The second part of the construct is to declare the data tuples using standard Prolog syntax. Comments starting with “%” are generated as documentation.

```
%Tuples
%motor(model,power_min,power_max,weight,current_max)
motor("motor_10HP", 0, 10, 374, 150).
motor("motor_15HP", 10, 15, 473, 250).
motor("motor_20HP", 15, 20, 539, 260).
```

4.3.3 Domain Assignment

The third part of the construct is to assign the domain to a variable.

```
%Domain Assignment
Motor_model &:: motor_model,
```

This is absolutely necessary since without it “Motor_model” in the above example would be treated as a standard Prolog variable, and backtracking of the data structure, which was the main cause of performance degradation, would be the primary method of searching for plausible components.

4.3.4 Infers Most

The fourth and final part of the construct is the “infers most” annotation.

```
motor(Motor_model, Power_min, Power_max, Motor_weight,
Motor_max_current) infers most,
```

Any *Goal* in Prolog can be turned into a constraint by using the annotation *Goal infers most*. 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 domains (as in 4.3.3). Propia needs this information in order to explore alternative domain values efficiently within the solver, without defaulting to backtracking. The new constraint accepts and rejects the same symbolic values as when using standard backtracking techniques, however the processing is significantly different. Propia extracts as much information as possible from the constraint before processing the Prolog goal. Note that the variable *Motor_model* is assigned to domain *motor_model* declared previously.

4.3.5 Final Code Structure

The code described in sections “Domain Declaration”, “Tuple Declaration”, “Domain Assignment”, and “Infers Most” was replicated for each of the symbolic values

in each of the fifteen VT component selections. This again made the generated code very readable by comparison with the CLIPS original.

4.3.6 Summary

If you consider the ECLiPSe solver to have two levels of processing namely Prolog and the CSP, then the addition of the “infers most” has the effect of pushing the Prolog tuple definitions down into the CSP level enabling the power of the CSP to be invoked. Thus the CSP is able to choose combinations of alternative values from the local domains provided, instead of repeatedly having to suspend constraint solving and backtracking to get another value in a sequence over which it has no control. A further advantage is that just adding a simple Design Specific goal as a constraint followed by an output statement:

```
ic:(Power_min =< 12),
write("Motor_model is "), write(Motor_model), nl,
```

returns the solution as ranges of symbolic or numeric values.

```
Motor_model is Motor_model{[motor_10HP, motor_15HP]}
```

Thus where Prolog would only return the first instantiation of motor_model, ECLiPSe returns all feasible values.

4.4 *ExtraKTor Upgrade*

The ExtraKTor system was upgraded to generate an enhanced form of ECLiPSe code. This entailed adding the following three programming constructs for each of the lift components. Firstly, a local domain was created for each component,

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

Secondly, a domain assignment was created for each component,

```
Motor_model &:: motor_model
```

Thirdly, “infers most” was added to the end of the Prolog database predicate for each component,

```
motor(Motor_model, Power_min, ... Motor_max_current)
infers most
```

4.4.1 Summary

By pushing the Prolog tuple definitions down into the CSP level, the performance of the loosely constrained KB returns to that of the highly constrained KB without

affecting the readability of the code. By reducing Prolog backtracking, processing time was very significantly reduced from hours to seconds, and most importantly the performance of this structure is largely independent of clause ordering. This then made it possible to explore the solution space in detail which was not achieved by previous researchers.

5 Experimentation - Exploring The VT Solution Space

A key value 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 [3]. `Car_weight` is calculated as the simple sum of several variables described in the following equation:

$$\begin{aligned} \text{Car_weight} = & \text{Car_cab_weight} + \text{Platform_weight} + \\ & \text{Sling_weight} + \text{Safetybeam_weight} + \text{Car_fixture_weight} \\ & + \text{Car_supplement_weight} + \text{Car_misc_weight} \end{aligned}$$

All of these variables have dependencies except `Car_supplement_weight(CSW)` which is defined in the VT Sisyphus documentation as taking only one of two discrete values either 0 or 500. We decided to remove this limitation and to iterate CSW over a range of values from 0 to 1000 in steps of 50 (manually running the KBS for each new value of CSW). The initial experiments did not show the expected relationship between CSW, MGP and HCTR and this led to the discovery of an error in the Protégé Sisyphus-VT ontology. In the ExtrAKTor version of the KB derived from the original Sisyphus-VT code from Stanford there was a statement for constraint C-48 as follows:

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

However, on further investigation we realised that the original full Sisyphus-VT Documentation states, “the HOIST CABLE TRACTION RATIO is constrained to be *at most* $0.007888 Q + 0.675$... (where $Q = \text{machine_angle_of_contact}$)” This would suggest that the “>” should in fact be a “<=” and that the version of the Protégé Sisyphus-VT ontology used up to this point had a significant error. After correcting the error, we then decided to iterate over a range of values 0 to 1000 in steps of 1, this time automatically running the KBS for each new value of CSW. Figure 1 shows the outcome of this test. As had been suggested in the original VT paper, as CSW increases MGP increases and HCTR decreases. With steps of 50, both variables appeared to change in linear fashion, but steps of 1 showed that HCTR actually behaved in a sawtooth fashion.

In order to experiment over such a wide range of CSW constraints known as C-31 and C-42 were disabled. Without this, at $\text{CSW} \geq 721$ the MGP exceeds the Maximum Machine_Groove_Pressure ($\text{MaxMGP}=119$) and the solver correctly identifies that there are no solutions.

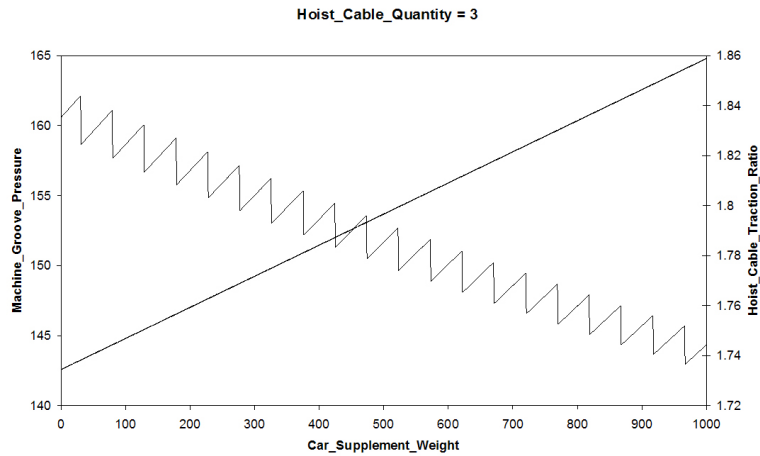


Fig. 1 CSW 0 to 1000 Step 1: sawtooth shows HCTR; straight line shows MGP

Prior to this experiment, we knew that if a task is consistently and fully represented that it can be solved by a CSP system; but of course this might take a very long time. What we were not sure about before this experiment was whether some vital information was contained in the fixes which meant that the formulation would be incomplete until the information inherent in the fix was added to the formulation of the task. The experiment showed that this perspective / hypothesis was not correct for the VT task with a particular contemporary CSP system. In retrospect, we now view the role of fixes as changing the focus of the search and hence we believe they would be useful in very large search spaces. Again empirically we have seen that sizable tasks / search spaces are handled efficiently by CLP and so to date we have not needed to use these heuristics (fixes) in conjunction with CLP searches for the VT tasks.”

6 Discussion of Related Work

6.1 Comparison with VITAL Results

Seven papers were presented at KAW94, each of which described a methodology for modelling and solving VT Sisyphus II; these are: Soar / TAQL, Protégé II [6], VITAL [5], CommonKADS, Domain-Independent Design System (DIDS), KARL, CRLM and DESIRE.

Of these seven papers, only one reported multiple runs of their implementation and that was VITAL [5]. Further, Menzies [4] reviewing this work emphasises that little testing was conducted on the various methods beyond this one example. Table

1 recreates the table presented in the VITAL paper. The most immediate observation is the apparent random nature of success and failure in finding a solution. For example, using the initial design constraints 3500lbs weight capacity and a 200ft/min lift speed, VITAL was successful in finding a solution. It is intuitively obvious that the same lift design would also be a valid solution for any lower weight i.e. the lift design which is valid (“success”) for 3500lbs would be a valid solution for 3000lbs, 2500lbs, and 2000lbs; but as can be seen in Table 1, this was not the case; further, VITAL could not find a solution for 3000lbs and 2500lbs.

Table 1 Recreation of Experimentation Table for VITAL

| | 200 ft/min | 250 ft/min | 300 ft/min | 350 ft/min | 400 ft/min |
|---------|------------|------------|------------|------------|------------|
| 2000lbs | success | success | fail | success | success |
| 2500lbs | fail | fail | success | success | success |
| 3000lbs | fail | fail | fail | fail | fail |
| 3500lbs | success | fail | fail | fail | fail |
| 4000lbs | fail | fail | fail | fail | fail |

For comparison purposes, the same set of tasks were solved by the ECLiPSe VT Sisyphus KB; these results are shown in Table 2. Motta, Zdrahal et al [5] argued that the failure of VITAL to find a solution was due to the VT Sisyphus problem specification. As such, the other six implementations should have seen similar problems, and since this was not reported by any other research group, it suggests that the other systems had not been tested extensively [4].

Table 2 Recreation of Experimentation Table for VITAL

| | 200 ft/min | 250 ft/min | 300 ft/min | 350 ft/min | 400 ft/min |
|---------|------------|------------|------------|------------|------------|
| 2000lbs | success | success | success | success | success |
| 2500lbs | success | success | success | success | success |
| 3000lbs | success | success | success | fail | fail |
| 3500lbs | success | success | fail | fail | fail |
| 4000lbs | fail | fail | fail | fail | fail |

6.2 Future Work

Our previous paper [7] focussed on the extraction of KB components from the VT-Sisyphus domain so that they can be reformulated and re-used in conjunction with different problem solvers. This led to the automated extraction of an ontology, formulae, constraints, and fixes. In this paper the extracted information has been used to solve real problems with a powerful CLP problem solver which is capable of us-

ing real number domains. Further, as far as we have ascertained, no other research has used constraint solvers with the VT domain.

There are a number of areas for further investigation. These include:

- 1) extension of the KB to support a “Sketch and Refine” interactive design process. This would accept initial design ranges, solve the KB followed by output of new acceptable ranges, and then allow user input of refined ranges.
- 2) investigation of the performance of CLP with more complex problems than VT, perhaps involving non-linear constraints.

6.3 Conclusion

We have shown that a classic AI problem, VT configuration design, is very well suited to solution by advanced CLP techniques, as provided in the ECLiPSe library. Much more significantly, we were able to code generate the CLP description of the problem direct from the Protégé knowledge base used in the Sisyphus II challenge. This means that a person with almost no experience of CLP or of programming in Prolog can now use these powerful techniques with confidence to get solutions to this class of problem.

Formulating problems in such a way that a powerful solver (in this case ECLiPSe) can get solutions has always been an important AI goal. Here we have shown a systematic way to generate the CLP automatically from a KB in a widely used representation (Protégé) which is accessible to many engineers through its graphic front end. This in turn uses a widely taught object model (ontology) based on entities, attributes, relationships and constraints. Thus the knowledge engineer has a clear way to think about and represent the problem, and a good graphic tool to capture the information. Further, there is no need to worry about how to order the information to achieve a solution, as one does with a programming language. Details of this knowledge capture method are given in our EKAW06 paper [7].

Here we have described how we used the various features of ECLiPSe to get very fast performance on this class of problem – far quicker than any other published study on VT. Some of these features were not at all obvious, but now we have refined and tested them on VT they can be code generated for future users to use on various kinds of configuration problem. Without these features we could not have done the in-depth study of the VT solution space that is presented here.

The original Protégé KB for Sisyphus II contained information on entity types, attribute names (variables), initial values, constraints and fixes for use with propose-and-revise algorithm. The CLP solver proved so powerful that we did not have to use any *fix* information. We thought we might have to use it to write a *labelling* predicate which is commonly used to guide the CLP solver about the order in which variables should be solved; and whether to increase or decrease values. It was not needed. This was so even though we re-introduced pairs of antagonistic constraints into the Sisyphus description, that had to be treated very specially in the [3] paper in

order to avoid thrashing. Instead these constraints were just entered like any others and were not specially labelled for ECLiPSe.

We have described in section 3 how the Protégé information, some of it in tables of alternative values, is represented in ECLiPSe. This is to convince the reader that it really is very straightforward to code generate for any configuration problem. Some problems might need more complex non-linear constraints, but these can also be generated. They do not affect the code generator, only the speed of finding solutions.

Acknowledgments

We acknowledge useful discussions of Research issues with Tomas Nordlander & David Corsar. Additionally, we acknowledge discussions with the Protégé team at Stanford University, including Mark Musen who also made available their version of the Sisyphus-VT code.

References

1. K. R. Apt and M. G. Wallace. *Constraint Logic Programming using ECLiPSe*. Cambridge University Press, 2007.
2. T. Le Provost and M. Wallace. Generalised Constraint Propagation Over the CLP Scheme. *JPL*, 16:319–359, 1991.
3. S. Marcus, J. Stout, and J. McDermott. VT: An Expert Designer That Uses Knowledge-Based Backtracking. *AI Magazine*, pages 95–111, 1988.
4. T. Menzies. Evaluation Issues for Problem Solving Methods. In *KAW98*, 1998.
5. E. Motta, K. O’Hara, N. Shadbolt, A. Stutt, and Z. Zdrahal. Solving VT in VITAL: a study in model construction and knowledge reuse. *IJHCS*, 44(3/4):333–371, 1996.
6. T. E. Rothenfluh, J. H. Gennari, H. Eriksson, A. R. Puerta, S. W. Tu, and M. A. Musen. Reusable ontologies, knowledge-acquisition tools, and performance systems: PROTÉGÉ-II solutions to Sisyphus-2. *IJHCS*, 44(3/4):303–332, 1996.
7. D. Sleeman, T. Runcie, and P. M. D. Gray. Reuse: Revisiting Sisyphus-VT. In *Lecture Notes in Computer Science*, pages 59–66, Podebrady, 2006. Springer.
8. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
9. Protégé VT Sisyphus Ontology. <ftp://ftp-smi.stanford.edu/pub/protege/S2-WFW.ZIP>, August 2004.
10. Sisyphus II. <http://ksi.cpsc.ucalgary.ca/KAW/Sisyphus>, December 2005.
11. ECLiPSe. <http://eclipse.crosscoreop.com/eclipse>, April 2006.
12. Online Guide to Constraint Programming. <http://ktiml.mff.cuni.cz/bartak/constraints/>, March 2006.
13. Kit ying Hui and Peter M. D. Gray. Developing finite domain constraints - a data model approach. In *CL ’00: Proceedings of the First International Conference on Computational Logic*, pages 448–462, London, UK, 2000. Springer-Verlag.