Paper submitted to IJHCS, Special Issue on Sisyphus: The VT Design Problem.

Solving VT in VITAL: A Study in Model Construction and Knowledge Reuse

Enrico Motta*, Kieron O'Hara, Nigel Shadbolt**, Arthur Stutt*, and Zdenek Zdrahal***

*Knowledge Media Institute,
The Open University,
Walton Hall,
Milton Keynes,
MK7 6AA.
U.K.
E.Motta@open.ac.uk

**Artificial Intelligence Group,
Dept. of Psychology,
University of Nottingham
University Park,
Nottingham,
NG7 2RD.
U.K.
nrs@psychology.nottingham.ac.uk

Abstract. In this paper we discuss a solution to the Sisyphus II elevator design problem developed using the VITAL[1] approach to structured knowledge-based system development. This involves a process of top-down refinement of models at different levels of abstraction which enables us to handle the complexity of a problem and produce an efficient and maintainable solution. In the paper we discuss in detail the knowledge-level architecture of a particular class of Propose&Revise problem solvers, called 'Complete-Model-then-Revise', and we show that they compare favourably in terms of competence with alternative Propose&Revise architectures. We also analyse in detail the VT domain, we critically examine the issues affecting the development of reusable ontologies, and we show that different assumptions about the completeness of the available domain expertise lead to alternative modelling choices. Finally, we discuss the performance of our problem solver and we show how we can use machine learning techniques to uncover additional strategic knowledge not present in the VT domain.

1. KNOWLEDGE MODELLING APPROACH

1.1 A Methodology and Workbench for Knowledge-Based System Development

The VITAL methodology (Shadbolt et al., 1993) enables complexity in an application to be expressed and mastered by the development of *multiple models*, an approach first advocated in the KADS methodology (Wielinga et al 1992). In the VITAL approach, the model development is structured by the idea of a *process product*, a notion which situates each important model in the life cycle of the application development. The four main process products are: the *requirement specification*, a document providing a description of the expected functionalities of the application, and eventual constraints which need to be obeyed; the *conceptual model*, a model of the expertise required to perform the task; the *design models*, a series of increasingly more specific models of the target system; and the *executable* code, which is the the actual target application. There is no assumption

---

[1] The VITAL project is a 4.5 year research and development enterprise involving seven organisations drawn from four countries. The total effort invested is about 80 man-years. VITAL is partially funded by the ESPRIT Program of the Commision of the European Communities, as project number 5365. The partners in the VITAL project are the following: Syseca Temps Reel (F), Bull Cediag (F), Onera (F), The Open University (UK), University of Nottingham (UK), University of Helsinki (SF), and Andersen Consulting (E).

that the production of these process products is a linear process as this depends on the chosen *life-cycle configuration* (Kontio and Rouge, 1991).

The VITAL methodology is supported by an integrated set of tools, the VITAL workbench (Domingue et al., 1993), which provides software support for the various activities, models, and languages which are defined in the VITAL approach.

In this paper we discuss a solution to the VT elevator design problem (Yost, 1994), which has been developed using the VITAL approach to structured knowledge-based system (KBS) development and implemented in the VITAL workbench. Before discussing our solution to the VT problem we provide a brief overview of the VITAL methodology and workbench.

1.2 Overview of the VITAL Methodology

A requirement specification in VITAL is a structured document which outlines and analyses the various functional, technical, and organizational requirements that a target application has to satisfy. The VITAL approach to requirement analysis is largely based on standard software engineering techniques and is discussed in detail elsewhere (Fassert et al., 1994).

The conceptual model is the output of the *knowledge acquisition* (KA) activity which involves three phases: i) the *initial information capture*, ii) the *selection/construction of generalised directive models (GDMs)*, and iii) *detailed conceptual modelling*.

A conceptual model is meant to characterise precisely the *application task(s)* in hand, the *problem solving method(s)* used by domain experts, and the relevant domain knowledge. A conceptual model is characterised at the *knowledge-level*, in the knowledge engineering sense that it provides an implementation-independent description of the application domain in hand (Van de Velde, 1994).

During the initial information capture activity a number of issues which affect the organization of the knowledge acquisition process are tackled. These typically comprise ‘contextual’ activities such as - for instance - identifying the number of experts and their areas of expertise, and assessing the quality and nature of the available expertise.

The selection/construction of GDMs[2] is performed through a process of model *refinement*, in which an initial, highly coarse-grained problem solving model is made increasingly fine-grained during various cycles of KA, using a *generative grammar* of model fragments (Leroux et al., 1993). At each cycle of this model refinement process stable classes of domain knowledge emerge, which become the focus for domain KA. This domain KA then in turn suggests further rewrites of the current GDM. This cyclic structure for KA is the reason for the name ‘directive models’ - the structure of the models *directs* domain KA. These models are said to be generalized since they generalize from the models to which they can be rewritten; each rewrite replaces a model with a more specific description of the problem-solving.

The main advantage of this approach is that - in contrast with model-library-based approaches to knowledge engineering - it allows the knowledge engineer to 'customize' and refine his/her model to fit a domain. Moreover, because GDMs are not selected from a library but constructed by applying a number of rewrite rules from the generative grammar, a comparatively small number of rewrite rules can provide an extensive library

---

2 GDMs were first developed as part of an earlier ESPRIT project, ACKnowledge (P2576) (Anjewierden et al. 1992), to model KADS structures (van Heijst et al 1992; Terpstra et al, 1993). They have since been used to model other knowledge level approaches, for example, Generic Tasks (Allemang and van Heijst, 1993).

of problem-solving types. The development of a GDM for the VT domain will be discussed in section 2.3.

During detailed conceptual modelling a complete knowledge-level model of the problem is devised, taking as starting point the outcome of the GDM analysis. In particular, the domain knowledge acquired during the GDM analysis is completed and formalised, and the final GDM model is refined into a precise specification of the problem solving architecture which we plan to emulate with our target application. The conceptual model can be formalised by means of a suitable modelling language such as K_{bs}SF (Jonker & Spee, 1992), OCML (Motta, 1995), or Ontolingua (Gruber, 1993a).

Design in VITAL consists of *functional* and *technical* design. During functional design an initial, coarse-grained architecture of the system is produced, taking into account the relevant requirements elicited during requirement analysis as well as the conceptual model. This coarse-grained architecture identifies the main components of the system as well as the mapping between these and the main tasks characterising the application (as identified during conceptual modelling). Moreover, the problem solving model produced during conceptual modelling is revised with the aim of producing an implementation independent description of the problem solving behaviour of the system. In some cases - for instance when we want to keep a close similarity between expert and system problem solving behaviour - this refinement might be minor and will typically consist of augmenting the model to include tasks which do not need to be considered during conceptual modelling. These include communication tasks as well as those which typically result from the application of non-functional requirements (e.g. those concerning the provision of explanation capabilities). Normally the same formalism is used for conceptual and functional design models.

Knowledge-level models - i.e. conceptual or functional design models - specified in OCML can be operationalised and run, thus allowing the KBS developer to validate and test conceptual and functional design models and to prototype the application. This is the approach we have used to produce our solution to the VT problem. Alternatively, at the technical design stage a suitable implementation platform is chosen, an implementation-dependent specification of the system is produced, and the mapping between the entities in the functional design model and the corresponding implementation structures is recorded. In (Stutt and Motta, 1995) we illustrate how argumentation-based techniques can be used to produce a structured record of design decisions.

1.3 The VITAL Workbench

The VITAL workbench mirrors the decomposition of the VITAL methodology, as it is functionally organised into four *Assistants*, each supporting a different process product. This decomposition is functional rather than structural, as typically there is overlap between tools in different assistants.

Knowledge acquisition (KA) support for constructing and selecting GDMs is afforded by a GDM manipulation tool, which contains the rules of the GDM grammar and a set of acquisition heuristics. These drive an interview tool providing the interface between the GDM grammar and the knowledge engineer. The purpose of the interview is to isolate features of the problem-solving that help to reduce the set of applicable rewrites. A number of additional knowledge acquisition and machine learning (ML) tools in the VITAL workbench provide software support for specific KA/ML techniques such as laddered grids, card sorts, repertory grids, and rule induction.

The workbench also comprises a set of tools supporting the development of knowledge-level models in OCML. These tools include style checkers, an interpreter which allows the user to perform rapid prototyping in OCML, browsers, and an integrated graphical and text editor.

System implementation is supported by means of i) VITAL-KR, an architecture for developing hybrid KBSs (Gaspari et al., 1993); ii) compilers from OCML models to external commercial shells, such as KnowledgeWorks™ (Willoughby et al., 1991); and iii) Viz (Domingue et al., 1992) - a tool providing a high-level programming language for building domain visualizations and acquisition interfaces.

Finally, the VITAL workbench also comprises a desktop management tool - the *VITAL Tower* - which enables the user to customise the workbench in response to alternative life-cycle configurations, or user preferences. (Domingue et al., 1993).

2. INITIAL PROBLEM ANALYSIS

2.1. Requirement Specification

The VT domain has been chosen in the context of the Sisyphus II initiative (Schreiber & Birmingham, 1994) to provide a common data set on which to compare different approaches to KBS development. In particular, the Sisyphus II call for papers specifies a number of requirements and 'quality criteria' which solutions to the VT problem should obey. The three main requirements we need to consider as part of our requirement specification are the following:

- The solution has to be a model of the expert's behaviour.
- The solution has to be based on reusable technology.
- The problem solver should be able to explain its behaviour.

2.2. Knowledge Acquisition in the VT Domain

Because the KA activity - in particular the knowledge elicitation process - has largely been assumed in the Sisyphus II problem statement, the account that we give here of the KA process in the VT domain is to an extent 'reconstructive'. Having no access to the original domain expert, we can skip the initial information capture activity and move on directly to the GDM selection/construction process.

2.3. GDM Selection and Construction

Our aim in this section is to describe the development of a GDM for the VT problem. The GDM is to be developed using the VITAL GDM tool, which is shown in figure 1 (behind the help window). The idea of this tool is that it enables the development of a model for an application to take place while allowing KA to be performed simultaneously. The development of the GDM is constrained by a set of rewrite rules (known as the *GDM grammar*) that allow fragments of the model to be replaced by more specific fragments; in this way the GDM gets increasingly fine-grained throughout the model development process. A GDM is a data-flow model, where static domain knowledge is indexed by the *role* it plays in problem-solving (symbolized, in the graphic representation, by the rectangles — cf. KADS interpretation models (Wielinga et al, 1992)). The roles are linked by *inference steps* (symbolized by the ovals), which represent problem solving steps.

The rewrites of the GDM are rewrites of the inference steps; intuitively, a rewrite is a process of replacing an inference, coarsely described, with a finer-grained description, which may itself involve more than one 'sub-inference', and may introduce further roles. If the user clicks on an inference step in the current GDM, the left hand window will display a list of available rewrites by name - see figure 1. These names are not very mnemonic, and so a third window, at the right, can show how each rule will rewrite the selected fragment of the current GDM. The window at the bottom explains how the rewrite is to be understood in the context of the current GDM (i.e. how to interpret the variables used in the right hand window). A hypertext help system is also available for all rules, inference steps and roles, as well as for all major concepts utilized by the GDM theory. To rewrite the current GDM, the desired rewrite rule is simply selected and applied. The current GDM can also be rewritten upwards, by restoring the previous GDM after the application of a rewrite[3]. Selection of a rule to be applied is governed by the help on offer; the form that the problem-solving must take for a rule to be applied is described on the help card, and it is then the responsibility of the knowledge engineer to ascertain whether or not the problem-solving in the domain fits that description. The knowledge engineer must acquire sufficient knowledge about the problem-solving to discriminate between all the applicable rules.

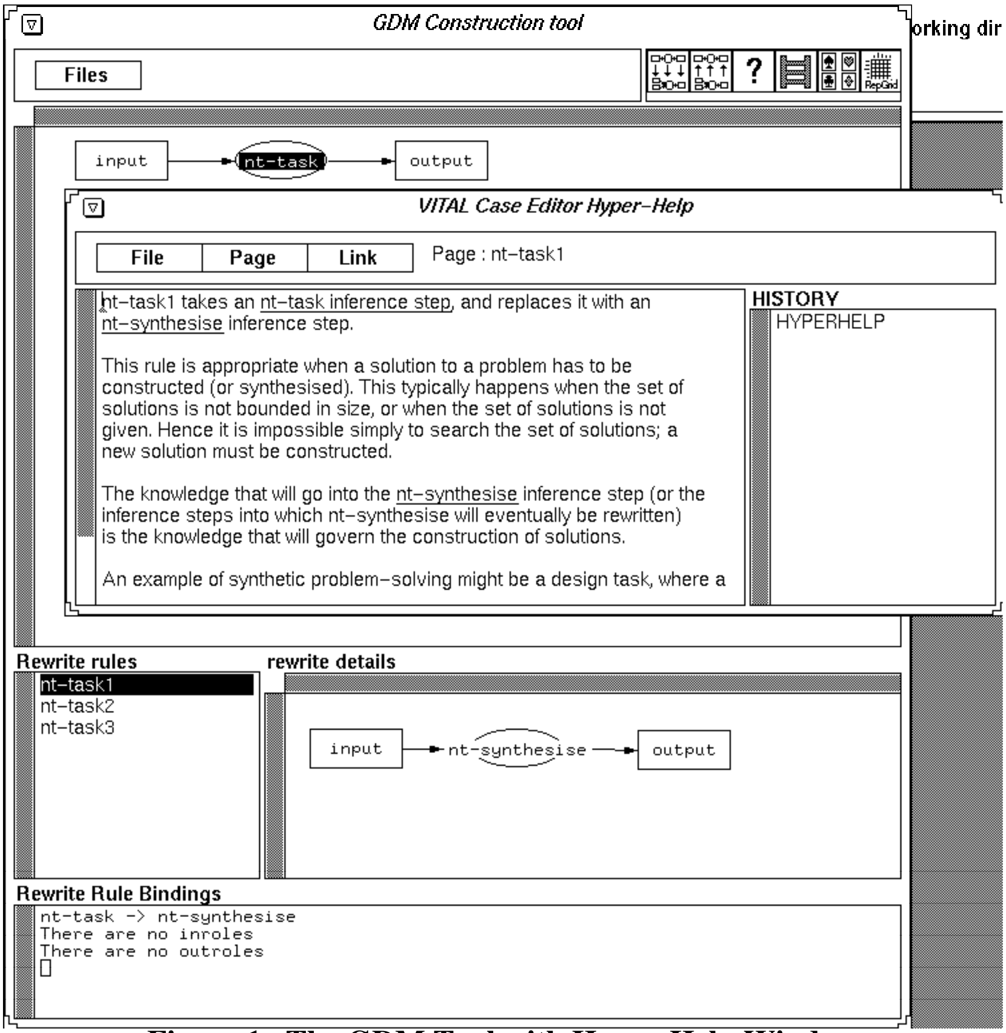

Figure 1. The GDM Tool with Hyper-Help Window

Steps of acquisition of static domain knowledge can be interleaved with steps of model development, by invoking one of the VITAL KA tools from within the GDM tool; the knowledge acquired from these sessions is then associated with one of the roles or inference steps of the current GDM. The principles of the GDM grammar mean that

---

3 The process of backtracking is problematic for GDM analysis, and we consider that an effective model refinement strategy should avoid backtracking where possible. One should seek to minimize structural re-presentation of acquired domain and inference knowledge, since this re-presentation process is expensive. However, it is harmless to apply and retract rewrite rules if no knowledge is acquired during this process (e.g. if the knowledge engineer simply wants to look ahead to see what the current GDM can be rewritten as).

these roles and steps will remain in place throughout the development of the model (although roles may get renamed, and inference steps may be rewritten).

The initial model shown at the top of figure 1 merely denotes that problem-solving goes on. An input and an output role are connected by an nt-task[4] (i.e. completely general) inference step. The first choice we are called upon to make is to decide whether the problem-solving is *analytic* (problem-solving where the set of solutions is given, such as in diagnosis), *synthetic* (problem-solving where the solution is constructed, such as in design), or consisting of an analytic phase followed by a synthetic phase (such as in medical treatment, where a disease is diagnosed, and then a treatment developed). The VT elevator design problem is clearly synthetic as, although the general set of components is relatively small and given in advance, a new, unique configuration of those components is going to be given in the general case. Hence the first GDM links the input role with the output role using an nt-synthesise inference step. The input consists of an initial assignment of user-specified values to a subset of the VT design parameters. The output is a complete and consistent design model, represented as an assignment of values to the VT design parameters.

Now we come to rewrite nt-synthesise. This is a pretty straightforward rewrite; we can rewrite the nt-synthesise inference step as an nt-design inference step, with an additional input role, components. Clearly the VT application uses a given set of components, and the design process involves configuring these components into the final design. The choice of this configuration model was made over the alternatives in the GDM library, which are a planning model, a modelling model, and a straightforward design model - see figure 2.

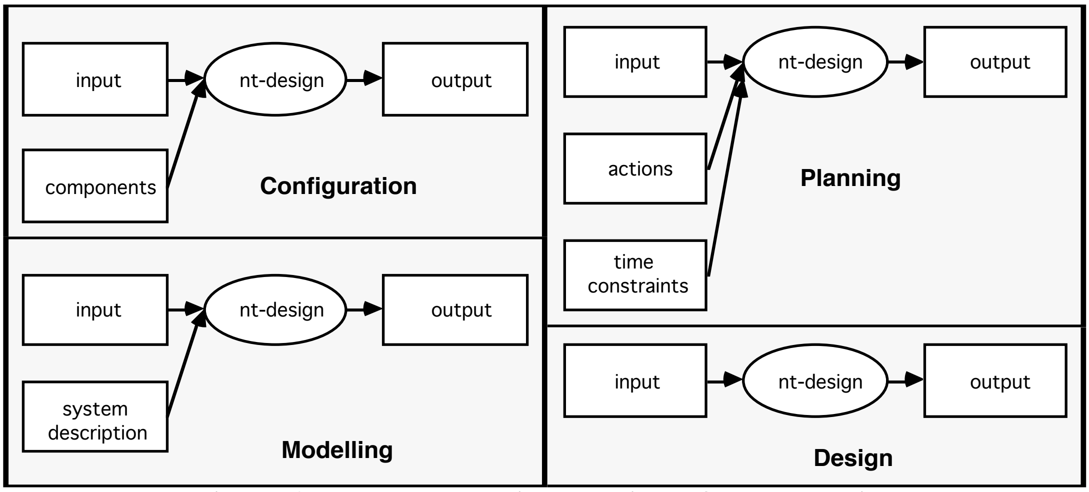

Figure 2. Four alternative rewrites of nt-synthesise

It is important to point out that *in the context of this particular GDM development* the design model is strictly on the same level of abstraction as the other three models (since it appears as a possible rewrite at the same stage as the other three). In particular it is not more generic than the configuration one. When models such as these are taken at the same level of abstraction, then the absence of a role is intended to indicate that there is *no* knowledge that plays that particular role in problem-solving. So, in this context, the design model states (by omission) that, among other things, there is no set of components

---

[4]The 'nt-' prefix stands for *non-terminal* (van Heijst et al 1992). This signifies that there exist rewrites in the grammar for the inference step. Note, though, that the applicability conditions for rewrite rules also make reference to the *context* of an inference step; hence it does not follow from the fact that a rewrite exists for an inference step, that that rewrite is applicable in the current context.

which are used to build the artefact to be designed. This is clearly false in the VT domain.

Moreover, the names *design*, *configuration*, *modelling* and *planning* are used here solely as a convenience for the reader. The user of the GDM tool will not see these names — indeed, a large number of GDMs can be drawn up which are not typical, and do not have common names. The important matter in the development of a GDM is the *structure* of the problem-solving as described in the GDM. *Identification* of that structure as, say, configuration or modelling can lead to problems with model development (O'Hara & Shadbolt, 1993a; 1993b). A major assumption of the GDM approach is that models should be built without the use of global assumptions that problem-solving is of a certain *type* which can be identified independently of, and in advance of, any information about the individual inferential steps used (O'Hara 1993). This is one of the major differences between the VITAL decompositional approach, and the CommonKADS decompositional approach (Breuker & Van de Velde, 1993), where problem-solving still has to be identified as belonging to a certain type before access is given to the decompositional libraries.

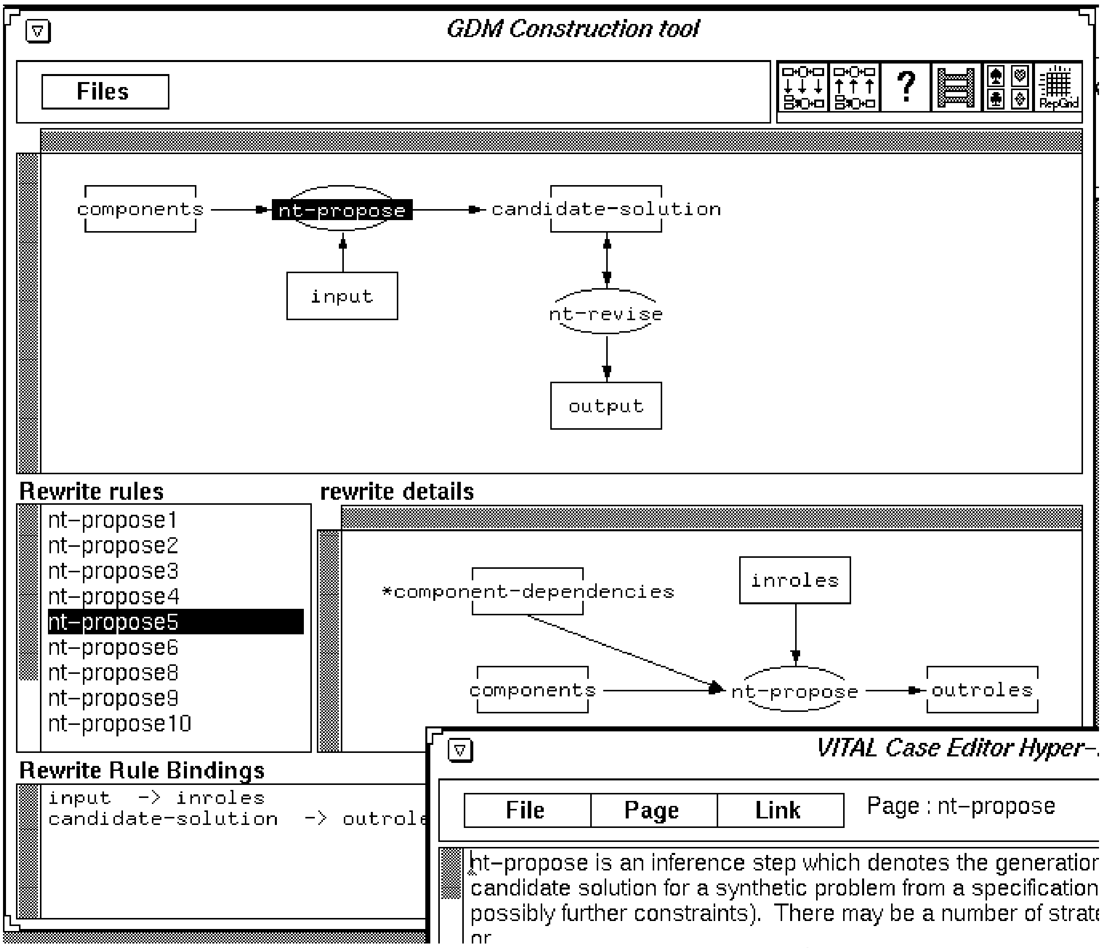

Figure 3. Rewriting the proposal step of a Propose&Revise model

Again, we only have one inference step to rewrite, nt-design. The six rewrites for nt-design available in the current GDM grammar are all variants on the Propose/Critique/Modify approach to design problem solving outlined in (Chandrasekaran, 1990). We select one of them and rewrite the current GDM as the one in the main window of figure 3, which reflects the Propose & Revise approach described in (Yost, 1994), where an initial, possibly partial solution is proposed, which (typically) will fail to meet a number of design constraints. In such an approach, the input (the initial specification) and the components are used to propose a candidate solution, which itself is revised to produce the output (the complete design). The double-headed arrow between candidate-solution and nt-revise symbolizes the possibility of loops here; there may be a number of revision steps, with the candidate solution being iteratively improved each time.

We now have a GDM with two non-terminal inference steps, so the rewriting can continue. We choose to rewrite the nt-propose step first; nothing much hangs on that decision. One thing to note at this stage of the analysis is that this is the first stage where the user encounters recursive rewrites. A recursive rewrite is one where the non-terminal inference step which is being rewritten appears in the rewrite. This means that (apart from changes in context) the same rules applicable before the rewrite will be applicable after the rewrite. Compare the rewrites of nt-synthesise discussed above; the nt-synthesise inference step is rewritten to nt-design each time, and therefore does not itself appear in any of the rewrites.

This has the effect of making the rewrites mutually exclusive; the choice of one route effectively rules out the other routes. However, with a recursive rewrite, other routes do not necessarily get closed off. Recursive rewrites can either take the form of the simple addition of a role to the model (acting either as an input to or an output from the inference step being rewritten), or involve the addition of an extra step of inference (figure 4). It is generally advised in the GDM tool help facility to investigate recursive rewrites before non-recursive ones, in order to maximize the rewriting possibilities available to the knowledge engineer at each stage.

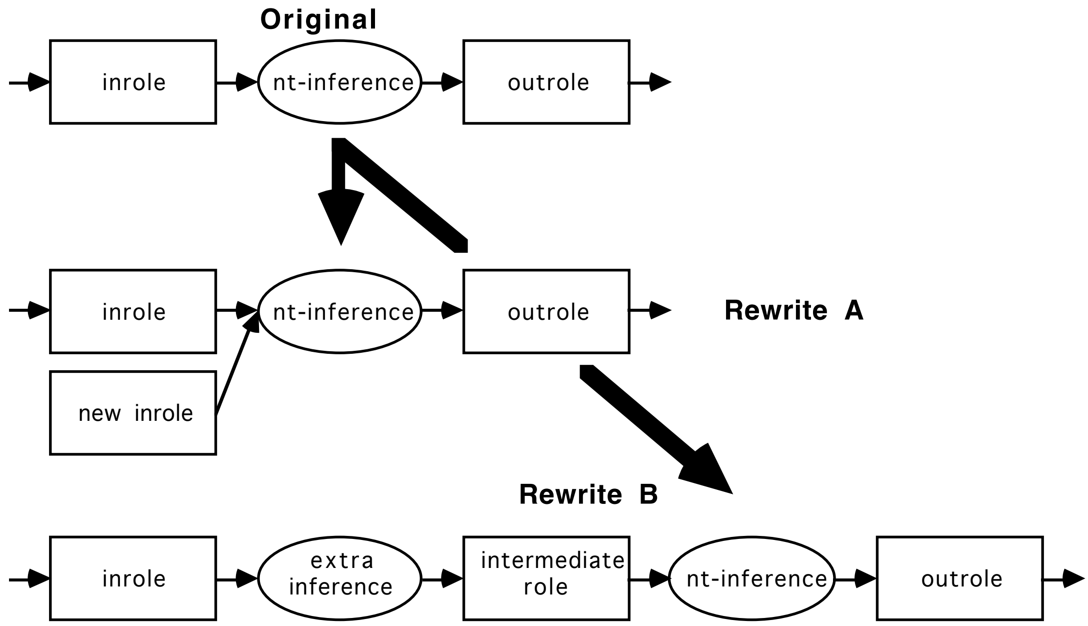

Figure 4. Two types of recursive rewrite

Only one of the rewrites available for nt-propose seems appropriate for VT, and that is a recursive one which adds a role to the inference (and therefore has a form similar to that of Rewrite A in figure 4). This role added is called 'component dependencies'. The rewrite is only available when the context of the nt-propose includes a role called 'components'; which is the case here. Specifically, the nt-propose inference step must include a components role as input, and we can see this clearly in figure 3. The rule adding a component dependency role has been selected in the minor windows of the tool. Component dependencies are the relations between components which can be used to generate candidate solutions and therefore can be assimilated to the model extension procedures described in (Yost, 1994). Such dependencies can be ‘genuine’ dependencies between components - for instance the counterweight stack weight is calculated as the counterweight plate weight times the number of plates in the stack - or they can simply be *preferred strategies* for computing candidate solutions. For example, the preferred strategy for the position of the counterweight is to place it such that “the distance between the counterweight and the rear of the platform is the same as the distance between the counterweight and the U-bracket” (Yost, 1994). As discussed in detail in section 3.4, our characterization of VT procedures as ‘preferred design extension strategies’ is a major difference between our ontology and the Ontolingua-based one (Gruber et al., 1994).

Having dealt with the proposal stage of the configuration, we now move onto the revision phase. Again, recursive rewrites are available, and these are investigated first. To be brief, four are applied. Firstly, a constraints role is added before the nt-revise (another rewrite of the form of Rewrite A in figure 4). Then a step of constraint evaluation is added before the nt-revise (hence this rewrite is of the form of Rewrite B). This takes a model fragment of the form

{[candidate solution] & [constraints]} -> (nt-revise)

and replaces it with a finer-grained fragment of the form

{[candidate solution] & [constraints]} -> (evaluate) -> [constraint violations]
-> (nt-revise)

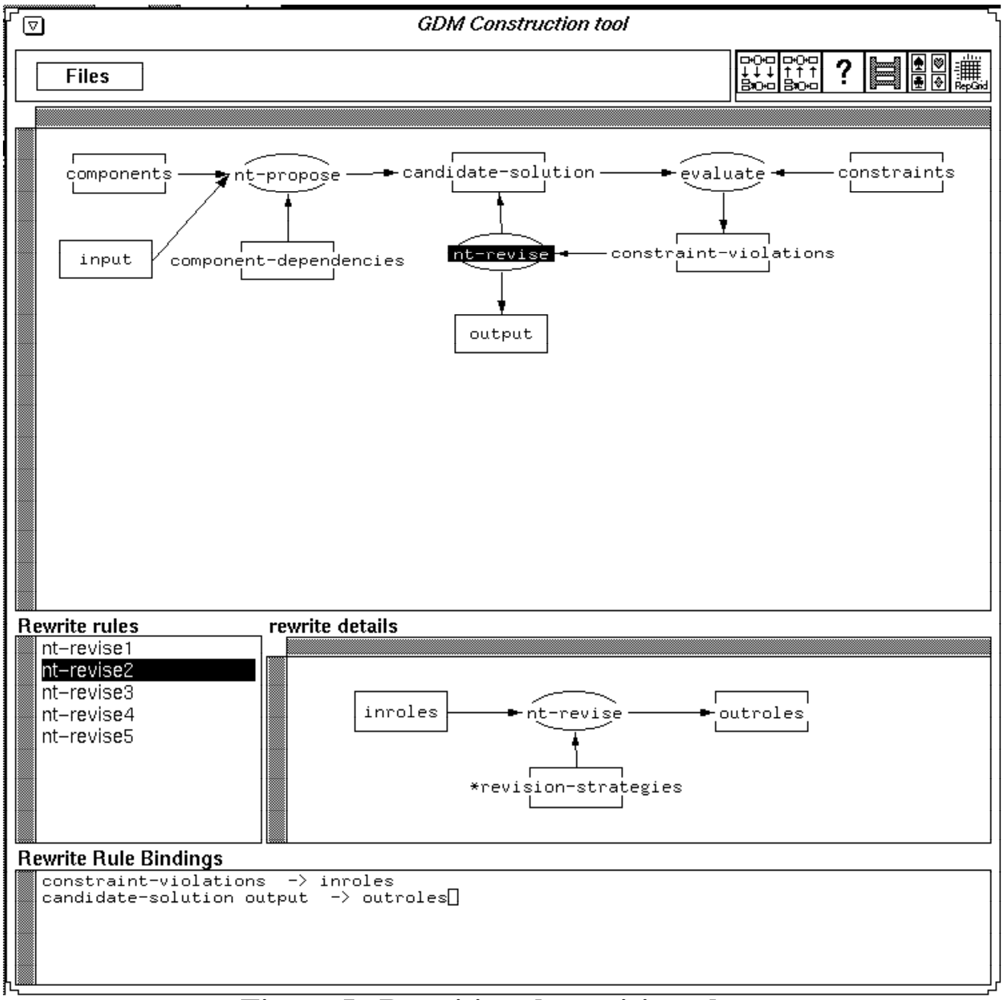

Figure 5. Rewriting the revision phase

This rewrite has been applied before figure 5. Note how the double headed arrow between candidate solution and nt-revise (cf. figure 3) has been replaced with a looping structure, where the candidate solution is evaluated against a set of constraints to produce constraint violations, which provide the raw material for the revision phase, which in turn gives us a new candidate solution. The double headed arrow symbolized the fact that candidate-solution was acting both as an input and an output to the nt-revise inference step; after this rewrite, candidate solution is no longer an input to nt-revise.

The third recursive rewrite of the nt-revise inference step involves the inclusion of revision strategies into the revision phase. A revision strategy is a method for performing the revision, and this rewrite should be chosen when there is a suite of such strategies that the designer can call upon when his proposal fails to meet all the relevant constraints. In VT, this is clearly the case and the Yost document refers to these as *fixes*. For example, the VT domain specifies that the underbeam space must be at least 12 inches (hence this information would be stored in the constraints role of the GDM). If this constraint is violated, the appropriate fix decreases the cab height by the amount of the violation. We can see this rewrite rule being selected ready to apply in figure 5.

The fourth rewrite of nt-revise can only be chosen when revision strategies already appear as an input to nt-revise (as is now the case). This gives us a step of selection; a strategy (or strategies) is (are) selected from the set of strategies, and used in the revision. This rewrite gives us the GDM shown in figure 6.

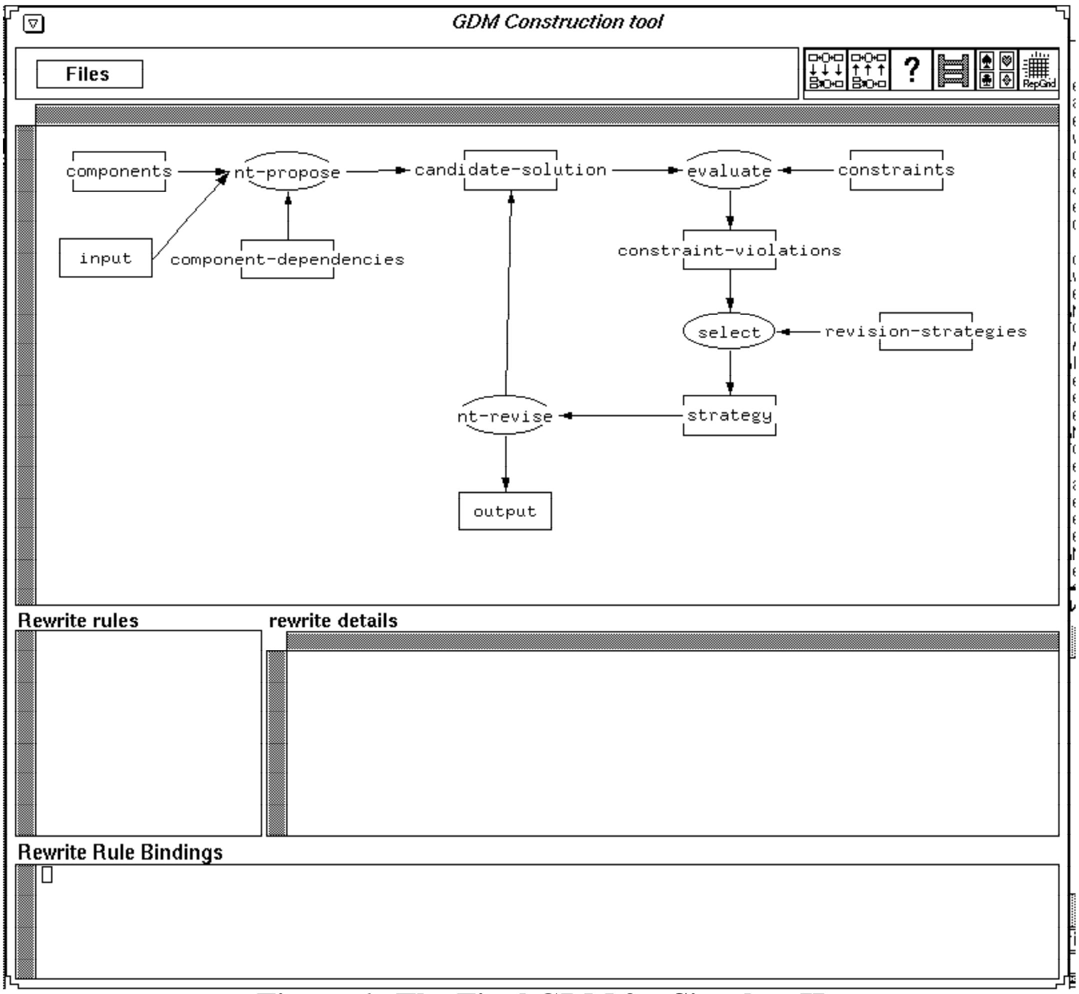

Figure 6. The Final GDM for Sisyphus II

One final point we should make concerns the completion of GDM analysis. GDM analysis is intended to be relatively open in form, and the idea is that a knowledge engineer, using a GDM tool, can call a halt to GDM analysis whenever he/she feels that the model he has is sufficient for his task (van Heijst et al 1992; O'Hara 1993). In this way, the knowledge engineer is freed from the constraints of a library of models, where he/she is restricted not only to a limited number of models, but also to a set of models whose grain size may not be appropriate. In the GDM analysis described in this section, the model has evolved from the coarse-grained (figure 1), to the relatively fine-grained model shown in figure 6. Because the model contains non-terminal inference steps, it could in theory be rewritten at a finer grain. However, because both the data-flow structure of a problem solving model for VT and the main classes of data in the VT domain (as described in the Yost document) have been produced, we can stop here. In the general case it is up to the knowledge engineer to determine when the goals of a GDM-based analysis have been achieved.

Note that the only difference between a non-terminal inference such as nt-revise and a terminal one, such as evaluate, is that the former can be rewritten using the GDM grammar, while the latter cannot. There is no relevant 'semantic' difference. A non-terminal step such as nt-revise still indicates a type of problem-solving inference (i.e. the revision of a candidate solution to a synthetic problem into a form more likely to evade constraint violations), and can still be used as the basis for KA as described above.

3. DETAILED CONCEPTUAL MODELLING

In the previous section we illustrated in detail the process of producing a GDM for the VT task. As already pointed out this process was to a large extent 'reconstructive', given that - in contrast to what normally happens with KBS projects - the important products of the knowledge analysis activity were already given to us in the form of i) a formal domain ontology (Gruber et al., 1994) and ii) a document providing a detailed domain analysis and problem solving model (Yost, 1994). Nevertheless, the previous section was needed to show how model-driven knowledge acquisition is carried out in the VITAL workbench.

The next stage in the VITAL methodology is to produce a detailed conceptual model of the problem. The outcome of the GDM activity comprises a data-flow characterization of expert problem solving in the domain and an initial method-oriented ontology (Gennari et al., 1994). The latter is acquired by doing KA on the knowledge roles identified during the GDM construction process. To produce the conceptual model we need to provide in addition i) a precise characterization of the application task(s), ii) a complete description of our problem solving model, in particular modelling control as well as data-flow aspects; and iii) a detailed, formalised, and complete domain ontology.

The VITAL workbench provides a knowledge-level modelling language - OCML - which i) supports both formal and informal approaches to knowledge-level modelling; ii) is suitable for different styles of KBS development - e.g. rapid prototyping and 'conventional' KBS development; iii) is compatible with emerging knowledge representation standards for ontology development (Gruber, 1993a); iv) provides a modelling framework which facilitates the reuse of KBS models; and v) enables the recording of design decisions (Stutt & Motta, 1995). In the following sections we show how we have used OCML for developing a conceptual model of the VT problem in VITAL.

3.1 Characterizing the VT task

3.1.1. VT as a parametric design task

The available VT domain ontology (Gruber et al., 1994) formally characterizes the VT task as a *parametric design* task. This is a specialised form of configuration design in which the design can be described in terms of a finite set of parameters. This characterisation is more fine-grained than the one we considered during the GDM construction process, where no distinction is made with respect to the form of the output, and it is the one we will use in the rest of the paper.

3.1.2. Representing tasks in OCML

A task in OCML is characterized in terms of its *goal*, *input* and *output roles*, *constraints*, and *modelling requirements*. A goal is an expression which specifies whether a task has been carried out successfully. Constraints are expressions which must be satisfied by the (domain-level) knowledge structures instantiating the task roles. A specification of the goals and constraints associated with a task normally imposes some 'modelling requirements' on the 'intended' application domain. In OCML it is possible to express these modelling requirements by explicitly listing in a task specification the required domain functions and relations. A role in OCML is the same as a role in GDM: it is a label denoting domain level structures which are input or output to a task. A *problem solving architecture* consists of tasks and roles connected by data flow and control relations. Roles in a problem solving architecture can be further characterized as *solutions*, *intermediate roles*, *case data*, *problem data*, and *user data.*

Tasks can be *primitive* or *structured*. Structured tasks decompose a task into a number of subtasks[5]. Primitive tasks do not introduce (new) subtasks and can therefore be seen as the leaves of the task-subtask hierarchy. An example of a task definition is given below.

```
(def-task-type parametric-design (design-task) ?task
   "A parametric design task is defined as the task of assigning
    values to parameters so that each parameter has a value and each
    constraint is satisfied.
    Parameters can be organized around components.
    The specification of a parametric design task requires that the
    domain relation HAS-VALUE (?parameter ?value) is specified in the
    theory"
   ((has-input-role constraints components parameters)
    (has-output-role design-model)
    (has-goal (and
               (forall (?p)
                      (=>
                         (parameter ?p ?task)
                         (exists ?x (has-value ?p ?x)))
               (forall (?c)
                      (=>
                         (constraint ?c ?task)
                         (satisfied ?c)))))
    (requires-domain-relation has-value)))
```

Figure 7. Sample definition of task parametric-design

The definition in figure 7 makes use of two relations, parameter and constraint, linking a parametric design task to the domain-level knowledge structures denoted by the input roles labelled 'parameters' and 'constraints'. This separation of task and domain level structures is common to a number of approaches to KBS modelling - e.g. Components of Expertise (Steels, 1990) - and has the advantage of introducing a degree of modularity which facilitates the reuse of domain and method ontologies (Gennari et al., 1994).

OCML provides three mechanisms supporting the definition of level-bridging mappings from problem solving architectures to domain-level knowledge structures.

---

5 The task-subtask decomposition can also be mediated by the selection of a problem solving method.

- A special relation, domain-reference, enables KBS modellers to map roles to domain structures. For instance, figure 8 shows how the role ‘parameters’ in the VT parametric design task can be linked to the domain-level parameters specified in the VT ontology.
- Primitive tasks in a problem solving architecture can be defined by means of functions and rules in a domain.
- Tasks and abstract problem solving methods - see next section - can be realised in a particular domain by instantiating their modelling requirements. For instance, the definition of task parametric design shown in figure 7 assumes that a specific parametric-design domain will provide the definition of a relation, has-value, modelling the assignment of values to parameters. A definition instantiating has-value in the VT domain is shown in figure 9.

```
(def-axiom reference-for-parameters-role
    "This axiom specifies that the role `parameters'
     in the definition of the VT parametric design task maps
     to the set of all parameter slots in the vt domain ontology.”
   := (domain-reference
               parameters
               vt-design-problem
               (setofall ?p
                        (exists ?c
                           (and
                              (vt-component ?c)
                              (has-parameter ?c ?p))))))
```

Figure 8. Level-bridging definition of VT domain parameters

```
(def-rule map-has-value-to-vt-domain
    "This rule instantiates the relation has-value, which is
     required by the parametric design task definition, in the VT
     domain”
   (has-value ?p ?value)
   if
   (and (vt-component ?c)
        (holds (?p ?c ?value))))
```

Figure 9. Instantiating has-value in the VT domain.

3.2 Modelling problem solving methods in OCML

Tasks are tackled by means of problem solving methods (PSMs). We use the term ‘problem solving methods’ both to denote general-purpose classes of problem solving architectures, such as Propose&Revise, and to refer to specific, special-purpose solutions. Like tasks, PSMs can be either structured or primitive. A structured PSM is basically a task-subtask decomposition with an (optional) control structure. For instance, a Propose&Revise problem solver can be characterised at a very high level of granularity by means of the following definition.

```
(def-psm Propose&Revise ?method
   "The Propose & Revise class of PSMs takes a set of parameters, a
   set of model extension procedures, a set of constraints
   on the values of the parameters, and a set of revision strategies
   associated with constraints and returns a target model
   defined as a consistent assignment of values to parameters"
   ((introduces-subtask Propose Evaluate Revise)
    (requires-domain-relation model-complete?)
    (subtask-control-regime
                :default-value
                       (loop
                          (do Propose Evaluate Revise)
                              until (model-complete?
                                       (domain-reference target-model
                                                         ?method))))
    (has-input-role Model-Extension-Procedures Parameters
                       Constraints Revision-Strategies)
    (has-output-role Target-Model)))
```

Figure 10. Generic Propose&Revise problem solving

OCML allows PSMs to be defined at various levels of abstraction, ranging from simple functions (primitive PSMs) to generic classes of PSMs. For instance, figure 11 shows an example of a primitive problem solving method, which provides a default way for selecting a parameter during the ‘propose’ step.

```
(def-psm default-select-parameter-method ?method
   "The default method for task select-parameter retrieves an
    unassigned parameter, whose value can be calculated from
    parameters which have already been assigned"
   ((requires-domain-relation depends-on)
    (has-input-role Parameters)
    (has-output-role Selected-parameter)
    (body  (lambda (?task)
                  (if (exists ?param
                          (and (unassigned ?param ?task)
                               (forall ?dparam
                                    (=> (depends-on ?param ?dparam)
                                        (assigned ?dparam ?task)))))
                          ?param))))))
```

Figure 11. Default parameter selection in Propose&Revise

The application of a PSM to a task is defined by instantiating the roles of the PSM in the target domain and by associating the chosen PSM with the task. For instance, let’s assume we have decided to solve VT by means of Propose&Revise. This PSM will introduce new subtasks for which we will also need to select PSMs. This process is repeated until no more ‘open’ subtasks are left. The resulting task-subtask decomposition is then augmented with control knowledge to produce a detailed problem solving architecture.

Finally, OCML also provides a graphical language for representing problem solving architectures. This will be introduced in the next section, which discusses our problem solving architecture for the VT task.

3.3 Solving VT by Propose&Revise

3.3.1. Complete-model-then-revise vs Extend-model-then-revise

Both our GDM analysis and the Yost document indicate that problem solving in the VT domain follows a Propose&Revise approach. More specifically, the Yost document informally describes a specific instantiation of Propose&Revise problem solving, *Extend-Model-then-Revise* (EMR), in which revision steps are intertwined with proposal ones. An alternative Propose&Revise architecture, *Complete-Model-then-Revise* (CMR), is one in which a complete, albeit possibly inconsistent solution is produced first, all constraints are processed, and only then revisions are carried out.

The EMR method was used in the original implementation of the VT problem and provided the organization of the SALT knowledge acquisition tool (Marcus et al., 1989). A diagrammatic representation of the EMR method in the graphical OCML language is shown in figure 12. The ovals represent tasks, while the boxes represent knowledge roles. Dashed lines represent data-flow relations, while continuous arrows show control flow. The hexagons indicate choice points. Static roles are represented as boxes with no fill pattern and dynamic roles as boxes with a dark fill pattern. Italics are used to show repeated occurrences of roles or tasks.

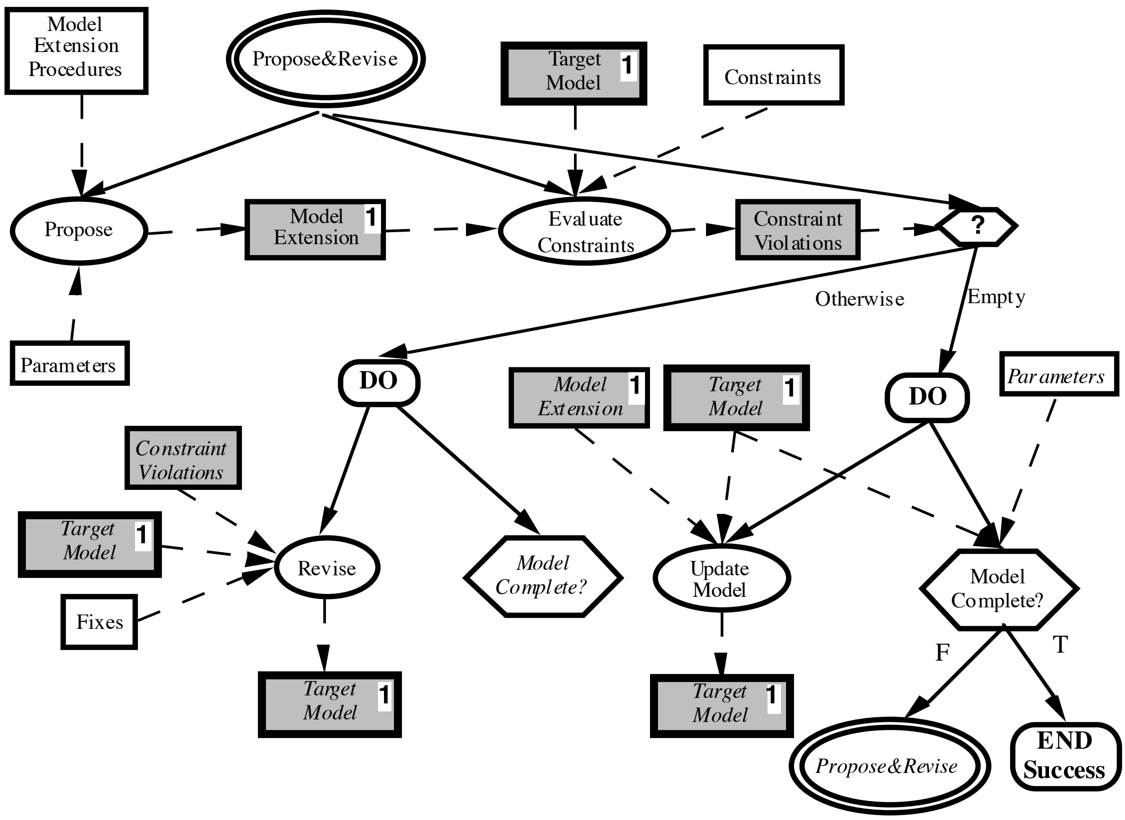

Figure 12. Extend-model-then-revise problem solving

As shown in the figure, the EMR method employs the following fixed problem solving strategy: it works by proposing one model extension at a time, and then checking that no constraint violations are violated (the control arrows from a task to its subtasks denote a left-to-right sequence of task execution). If constraints are violated, the current solution (represented as the box labelled 'Target Model') is revised using heuristic fixes[6], and then the Propose&Revise process starts again (the 'double border' associated with the Propose&Revise task marks this as the top-level task). Otherwise, if no constraints are violated, then the model is updated and new extensions are proposed, until a complete,

---

[6] For the sake of convenience from now on we will use the VT-specific term 'fix', rather than the more generic one, 'revision strategy', which is used in the GDM library.

consistent solution is reached (the number 1 in the Target Model role denotes the fact that this role has cardinality 1).

Figure 13 shows the architecture of a CMR problem solver, in which revision only starts after all model extension procedures have been applied to produce a complete, albeit possibly inconsistent model. The CMR approach has one important advantage over EMR due to the fact that, during the revision stage, the CMR method can exploit the interdependencies between constraints and between constraints and fixes, in order to select and apply those fixes which minimize the number of remaining constraint violations. As discussed in detail in (Minton et al., 1992) this strategy can produce significant improvement in the performance of the problem-solver, especially when the number of interdependencies between constraints is high.

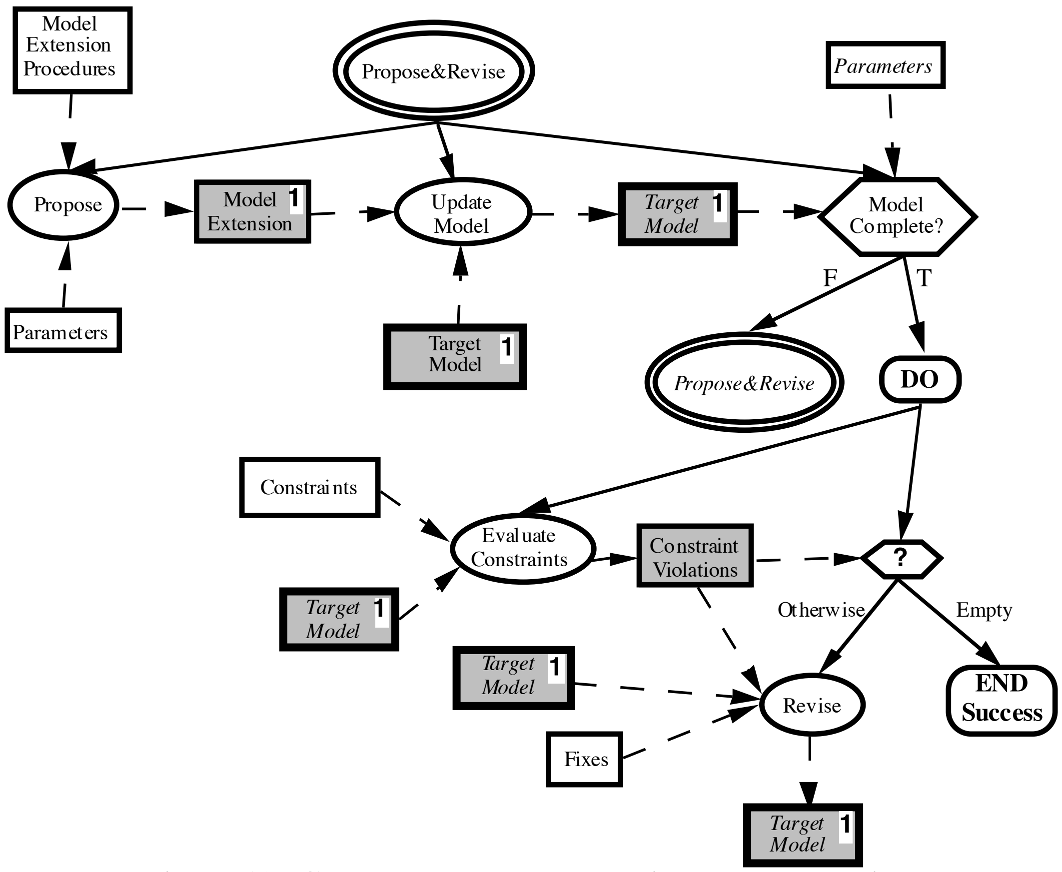

Figure 13. Complete-model-then-revise problem solving

Another advantage of the CMR model is that it increases the competence of the problem solver in the sense that it can find solutions which cannot be produced by means of the EMR approach. The reason for this is that the EMR approach can only find the solutions which can be reached by following those sequences of revision steps which are consistent with the order of constraint violations generated during the various cycles of model extensions. In constrast, the CMR approach can reach any solution which can be found through a sequence of revision steps associated with distinct constraint violations generated during the propose phase. Obviously the latter is a superset of the former given that it is not necessarily the case that the dependencies between parameters provide the degrees of freedom needed to generate all possible sequences which can be produced from a set of constraint violations. The following example clarifies this point and also shows how the CMR method can be more efficient than the EMR in exploiting interdependencies between constraints and fixes. Let’s suppose we have the following sets of parameters (P) constraints (C), fixes (F), and procedures (D).

P: {a, b, c, d}; C: {c_1, c_2}; F: {f_1, f_2}; D: {d_a, d_b, d_c, d_d}

d_a: a:= 1	c1: c > 4	f1: b:= b + 1
d_b: b:= a + 1	c2: d > 2	f2: c:= c + 1
d_c: c:= b * 2		
d_d: d:= c - 2(b + a)		

If we use the EMR method, we can solve the problem as follows:

a is assigned 1, b 2, and c 4 in the first three propose steps. This third extension violates c_1. We fix this constraint by applying f_1, so that b is now 3 and c becomes 6. We go back to the propose step which sets d to 2; c_2 is now violated. We apply fix f_2 which sets c to 7. d becomes now 3 and the problem is solved. The final solution is {a = 1; b = 3; c = 7; d = 3}.

If we use the CMR method, the propose phase will produce the initial solution {a = 1; b = 2; c = 4; d = 2} and the set of constraint violations {c_1, c_2}. If we fix c_1 first by applying f_1, we get {a = 1; b = 3; c = 6; d = 2}. c_2 is still violated, so we apply f_2 and produce the same solution as the EMR method, {a = 1; b = 3; c = 7; d = 3}. If instead we analyse the interdependecies between constraints, parameters, and fixes, we discover that f_2 can be used to fix both c_1 and c_2. So, instead of applying f_1 first, we apply f_2 directly, which produces the following, alternative solution, {a = 1; b = 2; c = 5; d = 3}.

This simple example illustrates the advantages of using the CMR strategy: it makes possible i) to exploit the interdependencies between constraints and fixes, and ii) to generate solutions which cannot be produced by the EMR strategy. In addition, the CMR is also more suitable than the EMR for finding optimal solutions - see (Zdrahal & Motta, 1995).

During the course of our VT experiment we have produced knowledge-level and implementation models for both the CMR and EMR problem solving architectures. In the rest of this paper we will discuss in detail the former, because of the aforementioned advantages and because it enables us to produce some insights about the interplay of procedures, constraints, and fixes in the VT domain.

3.3.2. The Propose task in CMR and EMR

Our method assumes that each parameter is associated with one and only one model extension procedure. This assumption is reasonable given that less preferred procedures can be represented as fixes in a Propose&Revise modelling framework. This is in fact the case with the VT domain - i.e. some fixes are actually less preferred strategies - see section 3.4.

Given this unique procedure assumption, the proposal phase in CMR is quite trivial, as we just need to take all components and all procedures and derive values for every single design parameter in the model. However, we should point out that, even with the unique procedure assumption, the order in which model extensions are carried out in the EMR method is actually very important. Because the EMR method cannot exploit the interdependencies between constraints and parameters during the revision phase, it is crucial that a 'good' sequence of model extensions is chosen during the proposal stage. In (Zdrahal & Motta, 1995) we suggest a number of strategies for improving the choice of parameters during the proposal phase.

Back to CMR, we note that the Evaluate task is also pretty simple, as it consists of verifying all relevant constraint expressions. In contrast, the Revise task needs to be analysed in detail, as we still need to specify a number of aspects, such as the conditions under which the revision process can fail, and how to control the application of revision strategies.

An important point concerning problem solving models is that their correctness is only relative to the current level of analysis. For instance, in the models in figures 12 and 13 there is no account of the conditions under which the design process will fail. This is because it can only fail if the revision process fails. Because we have not yet expanded the revise phase of our problem solving model, we can (for the moment) ignore the problem, which will have to be tackled when a more fine-grained model is generated.

3.3.3. An In-depth Look at the Revise Task

As shown in figure 13, the main input to the revise phase is given by the pool of constraint violations and the relevant fixes. However, an analysis of the VT data shows that not all constraints have associated fixes. Therefore, we can initially have a simple expansion of the revise task which filters the ‘fixable’ constraint violations. To this purpose we need to introduce a new role, called ‘FCV Pool’, to store all constraint violations which have fixes associated. So, a first decomposition of the Revise task looks like the diagram in figure 14.

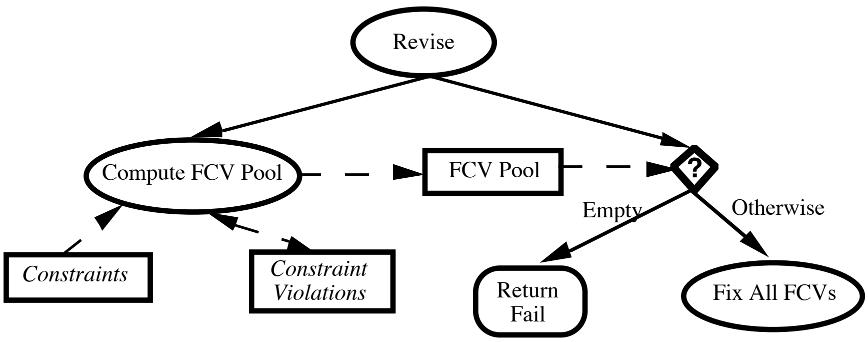

Figure 14 Initial decomposition of task Revise

As shown in figure 14, we first compute the set of fixable constraint violations (FCV Pool). If this is empty, then it means that none of the current constraint violations is fixable, and therefore we fail. Otherwise, we proceed to fix all FCVs. The latter can be decomposed into two subtasks: selecting a FCV and fixing it.

3.3.3.1. FCV Selection

FCV selection is a very important task, as, depending on the order in which FCVs are fixed, different designs can be generated, some of them more ‘expensive’[7] than others. However, the only information provided in the problem description tells us that if the machine groove pressure and the traction ratio are violated at the same time, then the former should be fixed before the latter. Hence, for the time being, we assume that in all other cases, selection is done at random. In section 6 we will come back to this point and show how we were able to induce some heuristics which can help us to generate optimal sequences of constraint violations.

3.3.3.2. Fixing Constraint Violations.

Fixing a constraint violation (CV) involves generating an ordered list of fix combinations and then applying them in turn, until one succeeds. The order in which fix combinations should be tried is given by Yost informally, by showing how to order a four element combination comprising two fixes at level 1, and two at level 2. We have formalized this informal criterion, by assigning a ‘cost’ to each fix combination. This is defined as follows.

[7] Because fixes have a cost - see section 3.3.3.2 - we can compare the cost of two designs in terms of the total costs of the fixes applied.

Let's suppose we have a fix combination Fc, with elements El_1, ..., El_n. Each El_i has a cost associated, given by its 'desirability'. The cost of Fc is then defined as a 10 element vector $<x_{10},....., x_1>$, where x_i is the number of individual fixes in Fc, with desirability i. The rule for ranking two fix combinations Fcx, and Fcy, is then expressed as follows. Let $Cx = <x_{10},....., x_1>$ be the cost of Fcx, and $Cy = <y_{10},....., y_1>$ be the cost of Fcy. Let j be the first position in the two vectors, Fcx and Fcy, such that, scanning them from left to right, $x_j \neq y_j$. If $x_j < y_j$, then Fcx should be tried before Fcy. If $y_j > x_j$, then Fcy should be tried before Fcx. If no such j exists, then Fcx and Fcy have the same cost and they can be tried in either order.

Let's assume we have two fix combinations, Fcx with cost $<x_{10},....., x_1>$, and Fcy with cost $<y_{10},....., y_1>$. The combined cost of Fcx and Fcy is then defined as $<x_{10} + y_{10},....., x_1 + y_1>$.

An individual fix can be *incremental* - i.e. it steps "a value along some dimension" (Yost, 1994). We generalize this notion to fix combinations by saying that a fix combination is incremental if any of the component fixes is incremental.

When a fix combination, say Fc, is tried, there are three possible cases. Either it succeeds (i.e. it fixes the selected CV without producing any new CV), or it fails (i.e. it produces a CV which is not already in the CV pool), or it does neither (i.e. it does not fix the current CV and does not produce a new one). In the first case, we make the tentative changes permanent. In the second case, we backtrack and try the next cheapest fix combination in the ranked list. In the third case there are two possibilities: if Fc is incremental we can try it again. Otherwise we try the next one. A data-flow view of task Fix FCV is shown in figure 15, while figure 16 shows a task-subtask hierarchy with the main tasks encountered so far.

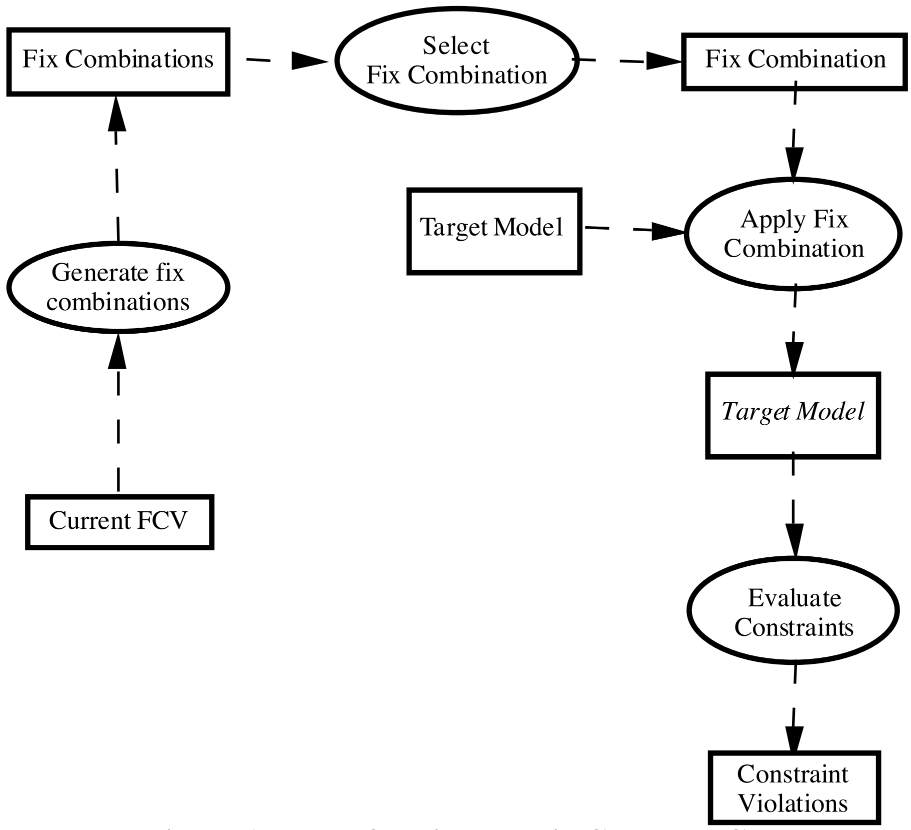

Figure 15. Data flow in task Fix Current FCV

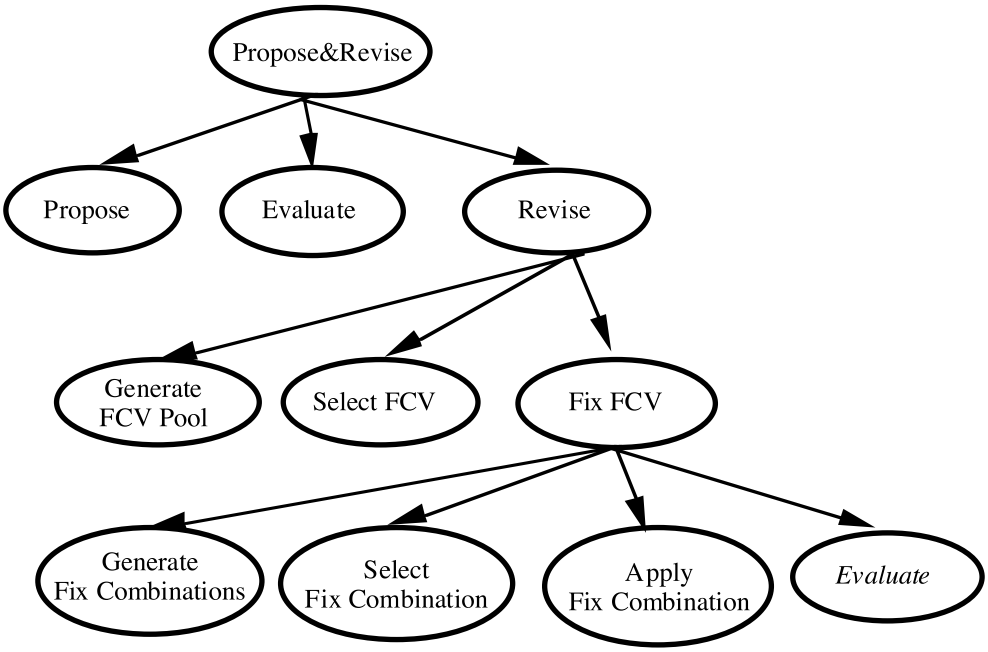

Figure 16. Task decomposition in Propose&Revise

If there are no applicable fix combinations, then it is not clear what to do, as this case is not discussed in the Yost document. The easy solution is of course to fail. However, because different sequences of successful fix applications can produce different designs, we could reshuffle the FCV pool and try an alternative sequence of FCV fixing. This solution guarantees that - sooner or later - a solution is found if it exists. Other, more efficient solutions can also be devised. Unfortunately this situation is not satisfactory from a knowledge acquisition point of view. In KA terms, the importance of FCV ordering means that crucial strategic knowledge, which is needed to drive the ordering of fixable constraint violations in CMR, is missing from the VT domain specification. Ideally we should go back to the domain expert and attempt to elicit it from him/her. Because we did not have access to a VT domain expert, we tried an alternative strategy and induced a number of FCV ordering heuristics by means of machine learning techniques - see section 6.

Our analysis is now complete, except for one minor aspect, which has to do with incremental fix combinations.

3.3.3.3. Managing Incremental Fix Combinations

When an incremental fix combination is applied, “all possible combinations of steps for that fix combination should be tried before moving on to the next basic fix combination” (Yost, 1994). This appears to be non-problematic until one looks in detail at what to do when the application of a particular sequence of steps for an incremental fix combination results in a new constraint violation. According to the problem description, the answer is to undo the changes and move on to the next fix combination. However, when the constraint violation has been produced by an incremental fix combination, we can’t just move on to the next one, as it is possible that a different combination of steps might fix the current constraint violation. On the other hand, we cannot just apply combinations *ad infinitum*, hoping eventually to fix the current CV. This is because it might be the case that the current combination cannot fix the current CV. Unless we specify a halting criterion for incremental fix combinations, our design program will never stop!

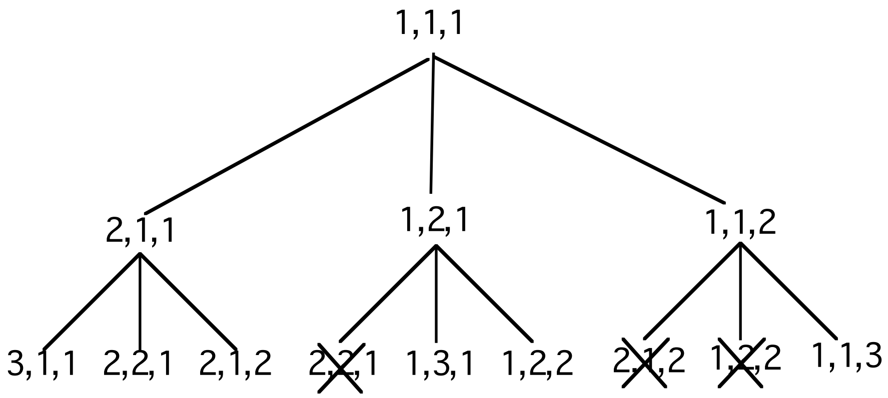

Figure 17. Search space for a sample incremental fix

The solution we have adopted is to view the combination of steps within a particular incremental fix combination as a search space. As shown in figure 17, let's suppose we have a fix combination comprising three incremental fixes. We will first try applying each fix by one step (represented as the triple '1,1,1' in the figure). If this doesn't work, then we generate three combination of steps, by increasing by one the number of steps of each fix, in turn. This process goes on recursively to produce a search tree. This is searched breadth-first, as intuitively a combination at level n is 'cheaper' than one at level n+1. Of course, duplicate branches are skipped. If a particular combination of steps, say '2,1,1', fails, then, instead of moving on to the next fix combination, we just prune the part of the search tree under '2,1,1'[8]. However, we can still explore the other branches of the tree. This process goes on until either we fix the constraint violation, or we run out of alternatives. The main assumption here is that constraint violations are generated in a 'monotonic' fashion. This means that if a constraint, say Ci, which is not violated before the application of an incremental fix combination, say Fci, is violated after n steps of Fci, we assume that further incrementing of Fci won't restore the consistency of Ci. This assumption allows us to prune the search tree as shown in figure 17.

With this we have concluded our analysis of a CMR problem solving model for the VT domain. The model we have described is very general and domain-independent, although - for the sake of convenience - we also added to our analysis some modelling aspects which are VT-specific (e.g. the VT-specific notion of the cost of a fix combination). In addition to the 'obvious' assumptions concerning the existence of the required knowledge structures (in particular parameters, model extension procedures, constraints, and fixes), in our analysis we only had to consider two aspects which can potentially limit the applicability of CMR. The first one concerns the application of incremental fixes and the point is that, in order to make use of incremental fixes, we need to be sure that their associated search spaces are bounded. This is the case if two conditions are satisfied: i) incremental fixes are 'monotonic', and ii) additional constraints exist in the domain, to circumscribe the application of an incremental fix. Both these conditions apply to the VT domain.

The second important aspect which we need to consider when characterizing the applicability of CMR is the unique procedure assumption. As already pointed out, this is not a limiting factor as long as it is the case that alternative model extension procedures in the domain can be ranked according to a preference order. Thus, less preferred procedures can be modelled as revision strategies.

---

8 In practice, by using dependency-directed backtracking one can also prune other parts of the search space.

The CMR model described here has been completely formalised in OCML and has provided the basis for the design of our VT design system, of an electronic circuit design application, and the Sisyphus I room allocation problem (Linster, 1994).

3.4 Modelling the VT domain ontology in OCML

The final component of our conceptual model is a domain ontology for the VT domain. When developing a solution for the KAW '94 workshop (Motta et al., 1994) we constructed our domain ontology from scratch, using OCML to model the domain specification given in (Yost, 1994). Afterwards, as an experiment in knowledge reuse, we decided to solve the VT problem by applying our CMR problem solver to the Ontolingua-based VT domain ontology (VT-Onto). In order to do this, we wrote a simple program to translate VT-Onto to OCML, with the idea of mapping CMR to the resulting OCML-based ontology. Achieving this goal turned out to be less straightforward than we expected; as a result of some of the modelling choices informing VT-Onto, this diverged in significant aspects from VT-Yost (the VT specification as informally given in Yost) and could not be reused without modifications. In a separate paper (Motta & Zdrahal, 1995) we discuss in detail VT-Onto and we critically examine the ideas of *knowledge separation* (Runkel & Birmingham, 1994) and *minimal ontological commitments* (Gruber, 1993b), which inform its organization. In the next section we briefly outline some of the main points which emerged from our analysis.

3.4.1. VT-Onto: A critique

VT-Onto is a formal, declarative domain ontology which attempts to be independent of any particular problem solving method and therefore doesn't include any *search-control knowledge* (Runkel & Birmingham, 1994). This is defined as knowledge expressing "commitment to a problem solving approach" (Runkel & Birmingham, 1994 - pp 2). In particular, the designers of VT-Onto have characterised both fixes and default initial values as search-control knowledge and therefore have not included them in VT-Onto. The remaining model extension procedures (i.e. those which are not initial default parameter values) have been modelled as declarative constraints. Although we fully subscribe to the idea of knowledge separation - for instance our tripartite organization of the conceptual model aims to maximise reuse of method ontologies as well as domain ontologies - it is not always the case that knowledge can be neatly separated into different categories. For instance, in some cases a knowledge structure used by a particular problem solving method - and as such to be classified as search-control knowledge - can also play a role in a domain ontology or task specification. This is particularly the case when a problem is described *procedurally*, by providing strategies to attain a solution, rather than *declaratively*, in terms of a complete set of criteria specifying the range of acceptable solutions.

Let's consider the VT domain, as described in (Yost, 1994). An in-depth look at VT model extension procedures shows that not all of them express (hard) constraints on the solution. Some of them only specify *preferred strategies* for calculating parameter values. For example, the preferred strategy for the position of the counterweight is:

> ***Strategy 1:*** *Place the counterweight so that the distance between the U bracket protrusion and the counterweight is the same as the distance between the counterweight and the rear of the platform.*

If this strategy cannot be applied, in particular because it produces a constraint violation, the following alternative is suggested:

> ***Strategy 2:*** *Place the counterweight so that its distance from the rear of the hoistway is 0.75 inches.*

These two strategies for placing the counterweight are expressed procedurally in the Yost specification. The procedural representation is convenient, because it helps the domain expert to cope with the large computational complexity of the design task by describing the design as a process defined by ‘legal moves’ in the solution space. Because the design engineer uses a Propose&Revise approach to elevator design, he/she would characterise the preferred strategy as a procedure and the other one as a fix.

In constrast with Yost’s procedural characterization of the two above strategies, VT-Onto takes a different approach. The preferred strategy (strategy 1) is modelled as a constraint, while the other (strategy 2), which is represented in VT-Yost as a fix, is considered search-control knowledge and therefore not included in VT-Onto. These two modelling decisions cause a number of problems.

- Modelling strategy 1 as a constraint on the space of legal solutions over-constrains the theory. As a result, VT-Onto specifies a design problem which is not the same as the one informally described in VT-Yost. In particular a Propose&Revise engine which makes use of the fixes listed in VT-Yost will not be able to apply the fix corresponding to strategy 2 without violating the VT-Onto constraint which models strategy 1[9].
- One possible interpretation of strategies 1 and 2 above is that they specify the ‘legal rules’ for positioning the counterweight, in the sense that - as far as we know - experts only use these two strategies for positioning the counterweight. More in general, this view maintains that the space of possible solutions to VT is expressed not only declaratively, in terms of the given constraints, but also procedurally, in terms of a number of legal moves in the space of possible designs. If we take this approach then it follows that a constraint expressing the requirement that the position of the counterweight should be consistent with one and only one of strategies 1 and 2 should be added to the VT task specification. In practice, because we do not have access to a domain expert, this kind of ‘closed world assumption’ cannot be validated nor rejected. However it allows us to make two points about domain modelling in general: i) search-control knowledge is not necessarily disjoint from *task knowledge* (i.e. knowledge about the task in hand); ii) different assumptions about the completeness and role of the available domain knowledge lead to alternative modelling decisions. In the context of the VT ontology we can therefore conclude that i) fixes do not necessarily play only a search-control role; ii) procedures and fixes are strongly interrelated - e.g. they together specify procedurally the space of possible designs and express alternative design strategies - and therefore a domain ontology cannot contain only one of these two categories; and iii) the underlying assumptions about the availability and completeness of domain expertise should be made explicit in the VT domain ontology. In (Motta & Zdrahal, 1995) we discuss three alternative ontologies for VT which are based on different assumptions on the available domain knowledge.
- There is no explicit information in VT-Onto which tells us which constraints in VT-Onto were derived from constraints in VT-Yost and which were derived from procedures. As a result, when trying to apply a Propose&Revise problem solver to VT-Onto, we can only identify procedures in terms of their syntactic form. The problem with this approach is that it is really a sort of a kludge. Relying on the syntactic format of an expression to decide whether or not it should be used as a procedure is not a very ‘principled’ approach. It can be done, but it relies on a number of implicit assumptions including: i) the designers of VT-Onto have used a consistent syntactic style in encoding constraints, and ii) there is no other constraint in VT-Yost which exhibits this structure - i.e. only procedures do. It seems to us that this approach goes against the idea of a reusable ontology providing a “content-

---

9 The sample solution specified in both VT-Yost and VT-Onto can only be reached by applying strategy 2. This means that it too is inconsistent with VT-Onto.

specific agreement" (Gruber, 1993a). We believe it would be much better to make explicit in VT-Onto which constraints are used by the VT designers as procedures during Propose&Revise problem solving and which are not. In this way, the ontology itself could provide a semantic rather than syntactic basis for reuse.

The bottom line here is that the original goal of applying our CMR problem solver to VT-Onto cannot be achieved without modifying it and without resorting to the additional knowledge contained in VT-Yost. Hence, rather than simply translating VT-Onto to OCML, we carried out the following tasks: a) we translated VT-Onto to OCML, thus producing VT-OCML; b) we augmented VT-OCML with the procedures and fixes listed in the Yost specification; c) we removed all the constraints in VT-OCML which had been 'inherited' from VT-Onto, but were not listed as constraints in VT-Yost; and d) we added to the set of constraints in VT-OCML all those procedures in VT-Yost which express relations which cannot be modified. For instance, the procedure for calculating the weight of the counterweight stack says that "The counterweight stack weight is calculated as the counterweight plate weight times the number of plates". This procedure models both a way of computing the stack weight as well as a physical constraint which cannot be relaxed. This kind of procedures are described procedurally in Yost only for convenience and they are represented both as procedures and as constraints in VT-OCML.

3.4.2. Modelling VT in OCML

OCML provides an operational subset of Ontolingua and therefore writing a program to translate from VT-Onto to VT-OCML was straightforward. The only exception concerned parameter slots which are modelled as unary functions in VT-Onto, while they need to be represented as binary relations in VT-OCML. This is because slots denote relations in OCML. As a result some additional transformations had to be carried out when translating expressions which referred to parameter slots.

For example, figure 18 shows how constraint C-18 is represented in VT-Onto (first definition in the figure) and VT-OCML (second definition in the figure). Apart from minor syntactic differences, the main difference between the two definitions is that the constraint expression in the VT-Onto is written in a 'functional' style, while in VT-OCML is written in a 'relational' style. In particular the term (counterweight_frame_height ?x) is translated into the sentence (counterweight_frame_height ?x ?y).

```
(define-frame C-18
 :own-slots (
  (INSTANCE-OF constraint)
  (CONSTRAINT-ON elevator)
  (CONSTRAINT.EXPRESSION
    '(and (>= (counterweight_frame_height ?x) 90.0)
          (=< (counterweight_frame_height ?x) 174.0)))))

(def-domain-instance C-18 constraint
   ((CONSTRAINT-ON elevator)
    (expression (and (elevator ?x)
                     (counterweight_frame_height ?x ?y)
                     (>= ?y 90.0)
                     (=< ?y 174.0)))))
```

Figure 18. Sample constraint translation from VT-Onto to VT-OCML

Finally figure 19 shows how procedures and fixes are represented in VT-OCML. The representation of an incremental fix assumes that the range of the parameter to be modified is ordered and selects the successor of the current value as the new parameter value.

```
(def-domain-instance COUNTERWEIGHT_TO_PLATFORM_REAR_C procedure
  ((PROCEDURE-ON elevator)
   (concludes-what counterweight_to_platform_rear)
    (expression
        (lambda (?x)
           (if (and (counterweight_space ?x ?s)
                    (counterweight_plate_depth ?x ?d)
                    (counterweight_u_bracket_protrusion ?x ?p))
                (* 0.5 (- ?s (+ ?d ?p))))))))

(def-domain-instance inc-Car-return-left fix
   ((fix-on elevator)
    (concludes-what Car_return_left)
    (desirability 8)
    (violated-constraints PLATFORM_TO_HOISTWAY_RIGHT_C)
    (expression (lambda (?x)
                  (if (and (Car_return_left ?x ?c)
                           (Platform_to_hoistway_right ?x ?p))
                      (+ ?c (- 8 ?p)))))))

(def-domain-instance inc-Sling-model incremental-fix
   ((fix-on elevator)
    (concludes-what elevator.sling)
    (desirability 4)
    (violated-constraints
               CROSSHEAD_BENDING_MOMENT_C
               CROSSHEAD_DEFLECTION_INDEX_C
               SLING_UNDERBEAM_C)
    (expression
       (lambda (?x)
           (if (and (elevator.sling ?x ?y)
                    (successor ?y ?w))
              ?w)))))
```

Figure 19. Representing procedures and fixes in VT-OCML

4. DESIGNING THE APPLICATION

As discussed in section 1.1, the VITAL workbench provides a number of pathways to implementation. In this case - given that our requirement specification did not include any delivery-related constraints - we decided to use a rapid prototyping approach and we fine-tuned and ran the detailed conceptual model specified in OCML. In addition we also developed a more efficient solution, by encoding the primitive tasks and roles of our model directly in Lisp. It is important to point out that the two alternatives are not actually 'true alternatives', as it is possible to describe at the knowledge-level only a subset of the primitive tasks and roles, and realize others by attaching lisp code to them.

A nice feature of the resulting prototype system - even the one with lisp attachments - is that there is a close correspondence between design and implementation. In particular, because the performance system can be executed from the task structure in the OCML model, it automatically produces knowledge-level traces of the program, corresponding to the invocation of the tasks in the design model.

5. VT IN THE VITAL WORKBENCH

Figure 20 shows a screen snapshot of some of the tools we have used to design and implement the VT application in the VITAL workbench. The main window shows part of the control layer of the functional design model for the VT domain, drawn using the graphical OCML editor. The window in the bottom right corner is a browser which enables the knowledge engineer to select different parts of the current design model. The window at the bottom of the figure is a domain visualization window showing part of the parameter dependency network. Finally, the window on the top right corner shows the interface to the Vital Tower, a tool management tool which enables the user to browse, select and load tools in VITAL.

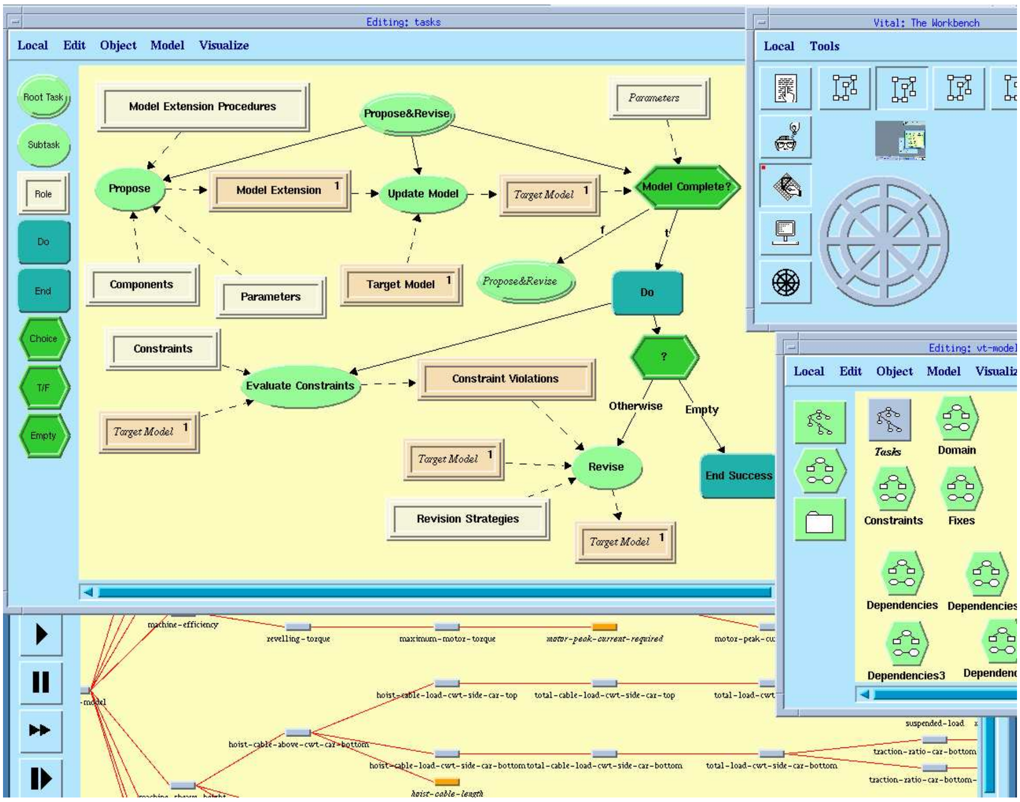

Figure 20. Designing and implementing VT in the VITAL workbench

6. ANALYSIS OF EXPERIMENTAL RESULTS

The test case in (Yost, 1994) gives a 'deterministic' relationship between a particular input specification and a design model. We found however that, depending on the order in which constraint violations are fixed, different designs can be produced. In particular, while we were able to reproduce the solution given by Yost, we were also able to generate two alternative, but more expensive designs. This suggested to us that expert VT designers were able to use some additional domain knowledge which allowed them to generate near-optimal solutions. In this section we look at the classes of solutions we generated for the VT problem and we show how, by means of machine learning techniques, we were able to derive some heuristics about the ordering of constraint violations.

6.1 Solution vs Non-solution paths

Since there is only one procedure associated with a parameter, the Propose task in the CMR architecture results in the same initial parameter values regardless of the order of their calculation. In the given test case the initial design model produced at the end of the propose phase violates the 7 constraints shown in Table 1.

Constraint	Abbreviation	Yost's notation
PLATFORM-TO-HOISTWAY-LEFT	MIN-PHL	C-39
MACHINE-BEAM-SECTION-MODULUS	MIN-MBSM	C-30
MOTOR-MODEL	LV-MM	C-34
HOIST-CABLE-TRACTION-RATIO	MAX-TR	C-48
CAR-GUIDERAIL-VERTICAL-FORCE	MAX-VF	C-50
HOIST-CABLE-SAFETY-FACTOR	MIN-HCSF	C-29
MACHINE-GROOVE-PRESSURE	MAX-MGP	C-31

Table 1. Constraints violated in CMR after the Propose task

If we analyse the interdependencies between constraints it is possible to see that the fixes applicable to the first constraint in the table, MIN-PHL, do not affect any of the other constraints in the table. Therefore this constraint violation can be resolved independently from the others and we do not need to include it in our analysis.

If we confine our analysis to the remaining six constraint violations, there are in theory 720 different ways of ordering them when carrying out the Revise task. However, we know from the Yost document that MAX-MGP must be always resolved before MAX-TR (Yost, 1994 p.34). This means that we only have 360 sequences to consider. Each of them either produces a solution, whose cost can be measured in terms of the cost of the applied fixes, or fails to result in a consistent design. The ordering of constraint violations is therefore crucial both if we want to produce a consistent design and if we want to produce a 'cheap' solution.

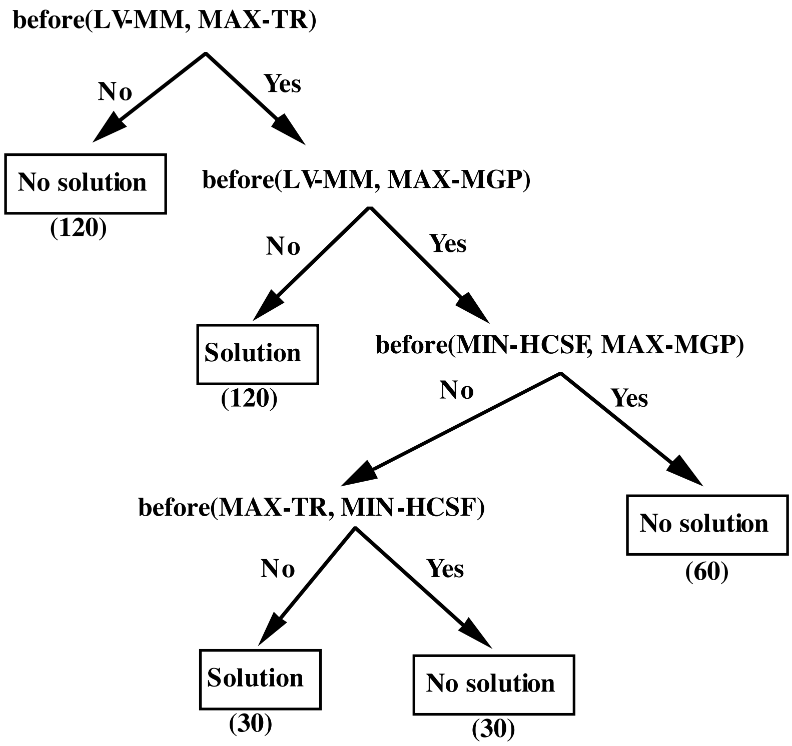

Figure 21. A characterization of solution paths in terms of relations between CVs

Because very little domain knowledge was available about effectively ordering constraint violations, we decided to evaluate all 360 revision sequences with the aim of getting some insights into the interdependencies between constraint violations.

The evaluation of all 360 revision sequences produced the following results: 150 sequences produced solutions and 210 sequences did not. In order to characterize the 'winning' sequences we defined a binary relation, 'before', which is defined over constraints. The expression 'before(x, y)' means that constraint x has been fixed before constraint y in the course of the revision task. Since we consider only 6 constraints and we know that before(MAX-MGP, MAX-TR) must always be satisfied, only 15 instances of the relation are relevant here. This means that any particular sequence of constraint violation fixing can be characterised in terms of a 16-element boolean vector, whose first 15 elements denote the boolean value of the 'before' relations over all possible pairs of constraints. The 16th element specifies whether or not the sequence produced a solution to the design problem. All sequences form a set of 360 examples, 150 positive (producing a solution) and 210 negative. We analysed this data set using the ID3 algorithm (Quinlan, 1986) and we derived the decision tree shown in figure 21.

The numbers in parentheses show how many examples belong to each class and path in the tree. From the tree in figure 21 we can derive the following heuristic rules, which express possible ordering relations between constraint violations which guarantee that our CMR problem solver can reach a solution.

before(LV-MM, MAX-TR) & not before(LV-MM, MAX-MGP) --> Solution

before(LV-MM, MAX-TR) & before(LV-MM, MAX-MGP) &
& not before(MAX-TR,MIN-HCSF) & not before(MAX-TR, MIN-HCSF)
--> Solution

6.2 Ranking solutions in terms of their cost

The classification of sequences into just two classes, {Not solution, Solution}, is only a coarse grained characterization. Individual solutions can be characterised in terms of their overall cost, which in the VT domain can be specified in terms of the combined desirability of the applied fixes - see section 3.3.3.2. We have classified the 150 solutions in terms of their cost and found that there are only three groups of solutions:

(a) one D3, five D4, two D6 and one D8	Cheap
(b) two D4, one D6 and two D8	Medium
(c) three D4, one D6 and two D8	Expensive

The order Cheap, Medium, Expensive corresponds to the order induced by the definition of our cost function.

Class	**Number of examples**
No solution	210
Cheap	36
Medium	48
Expensive	66
Total	360

Table 2. Cost-based grouping of solutions

Table 2 shows that only 10% of the possible sequences of constraint violations produce a cheap solution. This means that a program which has no knowledge about ordering constraint violations during the revision phase has only a 10% chance of generating a good design. However, it is very likely that a design engineeer almost always produces cost-effective designs. For instance, the solution specified by the VT test case is optimal. It seems likely that design engineers make use of additional domain dependent heuristic knowledge which is not part of the VT data set. If an engineer solves the design problem by applying the EMR method instead of CMR, these heuristics will be related to the selection of the sequence of propose steps so that the constraints are violated in the correct order. Using machine learning techniques we have been able to induce some possible heuristics that can be used by expert designers. Ideally, we should go back to domain experts and validate these heuristics. Figure 22 shows the decision tree derived from running the ID3 algorithm on the more fine-grained, cost-based characterization of the set of solutions to the VT problem.

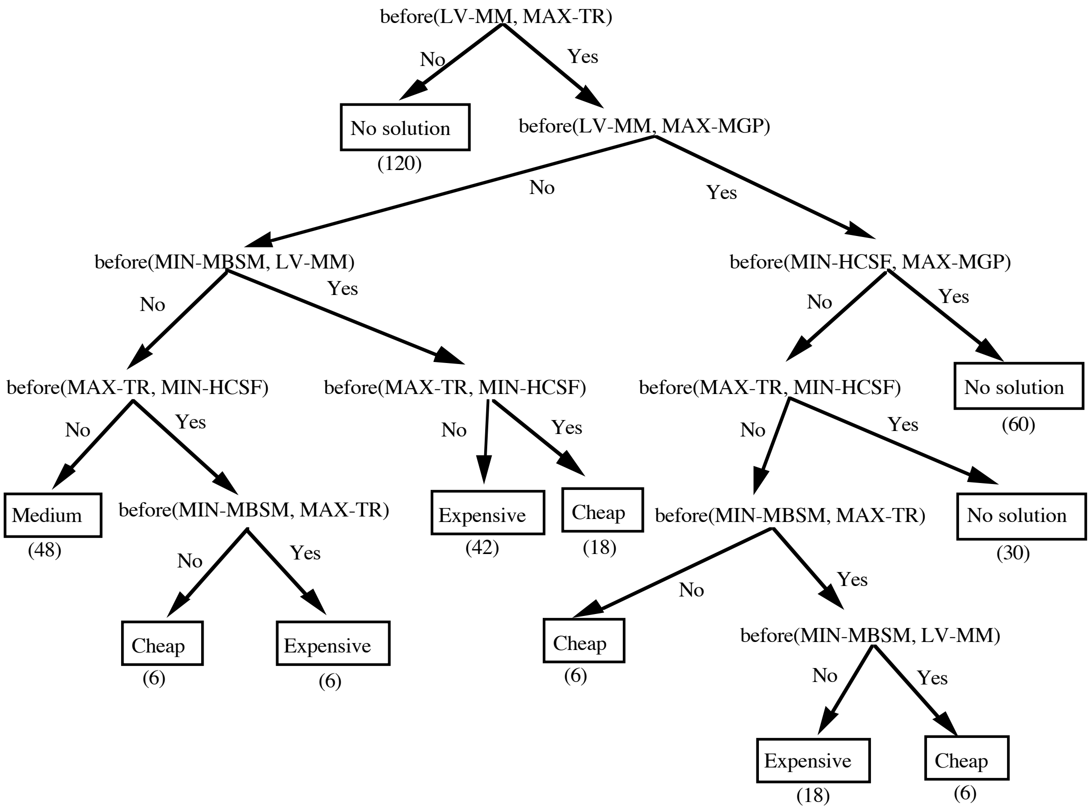

Figure 22. The decision tree for classes: Expensive, Medium, Cheap, No solution

6.3 Sample Trace

The trace below shows the particular sequence of successful fix applications in the 'complete-model-then-revise' implementation which produces the solution given by Yost. The prototype system compiled directly from the design model takes about 45 seconds to generate this solution on a Sun Sparc II workstation. By carrying out some pretty obvious optimizations to the code we have been able to reduce the execution time to less than 10 seconds on the Sparc II, and to 56 seconds on an Apple Powerbook 180.

```
DL 56 > (run-design-generate-complete-model-first)
Initializing design network .....Done
Acquiring Initial Data....Done
Generating values for all parameters....Done
Initial design completed, now checking all constraints...Done
Current CVs:
 MIN-PLATFORM-TO-HOISTWAY-LEFT
```

MAX-MACHINE-GROOVE-PRESSURE
MAX-TRACTION-RATIO
MIN-MACHINE-BEAM-SECTION-MODULUS
LEGAL-VALUE-MOTOR-MODEL
MAX-VERTICAL-RAIL-FORCE
MIN-HOIST-CABLE-SAFETY-FACTOR
All CVs have fixes
Now trying to fix CV: MAX-MACHINE-GROOVE-PRESSURE
Applicable Fix Combinations:
(INC-HOIST-CABLE-QUANTITY-7-16)
Trying Fix Combination: (INC-HOIST-CABLE-QUANTITY-7-16)
Applying Fix Combination...
HOIST-CABLE-QUANTITY set to 4
Propagating fix....Done
CV MAX-MACHINE-GROOVE-PRESSURE not fixed ...Trying (INC-HOIST-CABLE-QUANTITY-7-16) again - it is incremental
Applying Fix Combination...
HOIST-CABLE-QUANTITY set to 5
Propagating fix....Done
CV MAX-MACHINE-GROOVE-PRESSURE now fixed
Updating Design Model...Done
Now trying to fix CV: LEGAL-VALUE-MOTOR-MODEL
Applicable Fix Combinations:
(INC-MACHINE-MODEL-5-14-9)
(INC-MOTOR-MODEL)
(INC-MACHINE-MODEL-5-14-9 INC-MOTOR-MODEL)
Trying Fix Combination: (INC-MACHINE-MODEL-5-14-9)
Applying Fix Combination...
MACHINE-MODEL set to 28
Propagating fix....Done
CV LEGAL-VALUE-MOTOR-MODEL now fixed
Updating Design Model...Done

****
****HERE WE SKIP THE TRACE OF 3 CV FIXING STEPS****
****

Now trying to fix CV: MIN-PLATFORM-TO-HOISTWAY-LEFT
Applicable Fix Combinations:
(INC-OPENING-TO-HOISTWAY-LEFT)
(DEC-CAR-RETURN-LEFT)
(INC-OPENING-TO-HOISTWAY-LEFT DEC-CAR-RETURN-LEFT)
Trying Fix Combination: (INC-OPENING-TO-HOISTWAY-LEFT)
Applying Fix Combination...
OPENING-TO-HOISTWAY-LEFT set to 33
Propagating fix....Done
CV MIN-PLATFORM-TO-HOISTWAY-LEFT now fixed
Updating Design Model...Done
No unsolved CVs left. Design completed.

7. DISCUSSION

The solution to the VT elevator design problem discussed in this paper consists of two distinct components: a detailed conceptual model of the problem and a running system which is able to reproduce the solution given in (Yost, 1994), as well as generating alternative designs. The conceptual model comprises a precise characterization of the VT task, the CMR problem solving architecture, and VT-OCML, our own ontology for the VT domain. It is difficult to quantify the effort gone into our VT enterprise, which now includes i) two domain ontologies, ii) a number of problem solving architectures - see (Zdrahal & Motta, 1995) for details of some other solutions we produced - iii) an in-depth analysis of VT-Onto, and iv) has resulted in a number of publications. A more useful (for the reader) measure of how long it takes to tackle a domain such as VT in the VITAL workbench is probably an estimate of the effort it took us to produce our first solution to the VT problem (Motta et al., 1994); this was between three and four man-months, over a two month period. However, most of this effort went into the domain analysis, because VT-Onto was not (re)used. We estimate that the effort required to reproduce the work of VT-Onto was some 75% of the whole. This figure makes clear the enormous benefits which the KBS community can derive from the work on shareable and reusable knowledge bases, and the importance of facilitating the reuse of ontologies. As discussed in section 3.4, reusability might be better served by developing rich ontologies containing multiple, redundant, possibly inconsistent views of a domain, rather than minimalistic, ‘knowledge-deprived’ ones.

A clear advantage of the VITAL approach is that our solution could be provided in a highly top down manner. Since each model is in some sense an expansion of a previous one, the result is a structure preserving development from an anodyne coarse-grained synthesis model to the specific design model which is the basis of the implementation. Each model is therefore a representation of the problem-solving at a different level of detail, and a fine-grained specification of each fragment of any coarse-grained model can be discovered in the finer-grained models later in the development process. The resulting series of models can therefore be seen as the basis of a highly principled methodology for acquiring knowledge for an application, producing a knowledge-level model of the problem, and designing its code.

An important aim of the Sisyphus II exercise was to test the level of reusability that different methodologies and tools can provide. Given that we have already discussed at length some important aspects concerning the design of reusable models, here we only need to emphasise that the VITAL methodology and the VITAL tools provide a good framework both for developing new, reusable models, and for reusing pre-existing ones. In particular, a comparatively small number of rewrite rules in the GDM library provide an extensive and customizable range of of problem-solving models which can be used for carrying out model-based knowledge acquisition. In addition, the fact that a GDM is developed in tandem with KA (as opposed to being selected beforehand), means that the GDM is more likely to 'fit' the problem-solving in the domain. This entails that reuse of models need not result in inappropriate models being used. In effect, individual inferences, not whole models, are being reused in the GDM approach; this allows the possibility of novel combinations of inferences.

The tripartite organization of VITAL conceptual models is common to other KBS modelling approaches - e.g. Components of Expertise (Steels, 1990) - and facilitates the reuse of domain and method ontologies. Moreover, a number of features of the VITAL KBS modelling language - OCML - facilitate both the reuse of pre-existing ontologies and the construction of new, reusable ones. For instance it is easy to translate from OCML to Ontolingua and vice versa; OCML models can be easily operationalised; and the OCML graphical editor provides an effective environment for developing OCML models. Finally, OCML also includes a design rationale capture facility (Stutt & Motta, 1995), not discussed in this paper, which enables KBS designers to record explicitly the rationale for their design decisions. This approach improves the reusability of KBS models as it makes explicit both how model components have been combined by actual designers (the design rationales) and how they may best be combined (design principles). Finally, software reuse in VITAL is also supported by means of the VITAL-KR, an architecture supporting the development of hybrid, knowledge-based applications as well as the integration of pre-existing software modules into an application.

In conclusion in the course of the VT experiment we have found that the VITAL methodology and workbench provide good support for developing reusable knowledge bases. However, there is still quite a lot to be done. Our current, Unix-based workbench comprises many tools, provides a powerful KBS development environment, but it is not necessarily very easy to use and requires high-specification workstations. We are addressing this problem by developing a scaled-down, more user-friendly PC-based version, which will be more suitable for our students and accessible to a wider audience. In order to make our tools more 'attractive' to KBS developers, we are also planning to develop a customizable translator supporting the mapping of OCML models to CLIPS (Giarratano & Riley, 1993), which is a widely used programming language for expert system development. A library of OCML models is also needed. Rather than developing it from scratch our idea is that it will emerge through application development, exercises such as Sisyphus I and II, and 'cannibalization' (in a friendly sense) of existing ontology development enterprises around the world.

ACKNOWLEDGEMENTS

Many thanks to John Domingue for designing and implementing the tools shown in figure 20 and to James Cupit for his work on the implementation of the GDM Tool and for creating the GDM screen snapshots. This paper would not have been possible without the efforts of all those involved in the Sisyphus I and II initiatives. In particular we would like to acknowledge Gregg Yost and Jay Runkel with whom we had extensive discussions about the VT domain which helped us to clarify our ideas.

REFERENCES

Allemang, D and Van Heijst, G. (1993). Generic Tasks in KEW. In N. Aussenac et al (eds) Knowledge Acquisition for Knowledge-Based Systems: Proceedings of the 1993 European Knowledge Acquisition Workshop. Springer-Verlag.

Anjewierden A., Wielinga B., and Shadbolt N. (1992) Supporting Knowledge Acquisition: The ACKnowledge Project. In L. Steels and B. Lepape (eds.) *ESPRIT-92 Knowledge Engineering*

Breuker, J. A. and Van de Velde, W. (1993). The CommonKADS Library. ESPRIT Project P5248 Deliverable M2, University of Amsterdam and Free University of Bruxelles.

Chandrasekaran, B. (1990) Design Problem Solving: A Task Analysis. *AI Magazine, 11(4)*, pp.59-71.

Domingue, J., Price, B. and Eisenstadt, M. (1992) Viz: a framework for describing and implementing software visualization systems. In D. Gilmore and R. Winder (Eds.) *User-Centred Requirements for Software Engineering Environments*, Springer-Verlag.

Domingue, J.B., Motta, E., and Watt, S. (1993). The emerging VITAL Workbench. In N. Aussenac et al (eds) *Knowledge Acquisition for Knowledge-Based Systems: Proceedings of the 1993 European Knowledge Acquisition Workshop*. Springer-Verlag.

Fassert, C., Lhept, M., Lozinguez, Y., and Rouge, A. (1994). ERUDITE, The VITAL Approach to Expertise Reuse-Driven Requirements Engineering. VITAL Project Report DD251, Syseca, Paris, October 1994.

Gaspari, M., Motta, E, and Stutt, A. (1993). Inferring in Lego-land: an architecture for the integration of heterogeneous inference modules. In Torasso (ed), Advances in Artificial Intelligence, *Lecture Notes in Artificial Intelligence, LNAI 728*, Springer-Verlag. 1993.

Gennari, J. H., Tu, S. W., Rothenfluh, T. E., Musen, M. A. Mapping Domains to Methods in Support of Reuse. *International Journal of Human-Computer Studies*, *41(3)*, September, 1994.

Giarratano, J. and Riley, G. (1993). *Expert Systems Principles and Programming*. PWS, Boston, MA.

Gruber, T. R. (1993a). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, *5(2)*, 1993.

Gruber, T. R. (1993b). Toward Principles for the Design of Ontologies Used for Knowledge Sharing. (1993b). Technical Report KSL 93-04, Stanford: Knowledge Systems Lab., Computer Science Dept. Stanford University, 1993.

Gruber, T. R., Runkel, J. T., and Olsen, G. R. (1994). VT Domain Ontology. Available by FTP from ksl.stanford.edu/pub/knowledge-sharing/ontologies/. Stanford: Knowledge Systems Lab., Computer Science Dept. Stanford University, 1994.

Jonker, W. and Spee, J. W. Yet another formalization of KADS conceptual models. In Wetter et al. (Eds), *Current Developments in Knowledge Acquisition*. Lecture Notes in Artificial Intelligence, 599, Springer-Verlag, Berlin, 1992.

Kontio, J., and Rouge, A. (1991). VITAL Life Cycle Guide. VITAL Project Report DD111.

Leroux, B., O'Hara, K., Shadbolt, N., Outtandy, S., Laublet, P., and Motta, E. (1993). The VITAL Library for Knowledge Modelling. VITAL Project Report DD215, August 1993.

Linster, M. (1994). Problem Statement for Sisyphus: Models of Problem Solving. To appear in the *Knowledge Acquisition Journal, 6(1).* March 1994.

Marcus, S., and McDermott, J. (1989). SALT: A Knowledge Acquisition Language for Propose-and-Revise Systems. *Journal of Artificial Intelligence*, 39(1), 1-37.

Minton S., Johnson M.D., Philips A.B. and Laird P. (1992). Minimising conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence 58*. (1992). pp. 161–205

Motta, E. (1995). KBS Modelling in OCML. Proceedings on the Workshop on Modelling Languages for KBS , Vrije Universiteit Amsterdam, January 30-31, 1995.

Motta E., O'Hara K., Shadbolt N., Stutt A. and Zdrahal Z. (1994). A VITAL Solution to the Sisyphus II Elevator Design Problem. *Proceedings of the 8th KA workshop Vol. 3*. Banff, Alberta, Canada.

Motta, E., and Zdrahal, Z. (1995). The trouble with What: Issues in Method-Independent Task Specification. Proceedings of the 1995 Knowledge Acquisition Workshop, Banff, Canada.

O'Hara, K. (1993) A Representation of KADS-I Interpretation Models Using a Decompositional Approach. In Christiane Löckenhoff, Dieter Fensel and Rudi Studer (eds.) *3rd KADS Meeting* (Siemens AG, Munich, 1993) pp.147-69

O'Hara, K. and Shadbolt, S. (1993a). AI Models as a Variety of Psychological Explanation. In Proceedings of the 13th International Joint Conference on Artificial Intelligence, IJCAI '93, vol.1 pp.188-93. Morgan Kaufmann, San Mateo, California.

O'Hara, K. and Shadbolt, S. (1993b). Locating Generic Tasks. In *Knowledge Acquisition* vol.5 pp.449-81.

Quinlan J., R. (1986). Induction of Decision Trees. *Machine Learning 1*, pp. 81-106

Runkel, J. T. and Birmingham, W. B. (1994). Solving VT by Reuse. Proceedings of the 8th Banff Knowledge Acquisition Workshop, Banff, Canada, 1994.

Schreiber, G. and Birmingham, W. B. (1994). Sisyphus II - VT Elevator design Problem. Volume III of the Proceedings of the 8th Banff Knowledge Acquisition Workshop, Banff, Canada, 1994.

Shadbolt, N., Motta, E., and Rouge, A. (1993). Constructing Knowledge Based Systems. *IEEE Software*, November 1993.

Steels, L. (1990) Components of Expertise. AI Magazine Summer Issue.

Stutt, A. and Motta, E. (1995). Recording the design decisions of a knowledge engineering community to facilitate re-use of design models. Proceedings of the 1995 Knowledge Acquisition Workshop, Banff, Canada, 1995.

Terpstra, P., Van Heijst, G., Shadbolt, N., and Wielinga, B. (1993) Knowledge Acquisition Process Support Through Generalized Directive Models, in J-M. David, J-P. Krevine and R. Simmons (eds.) *Second Generation Expert Systems* (Springer-Verlag) pp.428-454.

Van de Velde, W. Issues in Knowledge Level Modelling. Proceedings of the 8th Banff KA for KBS Workshop, Vol 2, 1994a. 38-1 - 38-11.

Van Heijst G., Terpstra P., Wielinga B. and Shadbolt N. (1992). Using Generalized Directive Models in Knowledge Acquisition. In Th. Wetter, K.-D. Althoff, J. Boose, B.R. Gaines, M. Linster and F. Schmalhofer (eds.) *Current Developments in Knowledge Acquisition - EKAW '92* (Springer-Verlag) pp.112-32

Wielinga B. J., Schreiber A.Th. and Breuker J., (1992) KADS: A Modelling Approach to Knowledge Engineering. In *Knowledge Acquisition* 4(1) pp.5-53

Willoughby, A., Plowman, D., and Motta, E. (1991). Combining Object-Oriented and Rule-based programming by Integrating the Common Lisp Object System with a Rule-based Inference Engine. *IJCAI Workshop on KBS Architectures*. Sydney, Australia, August 1991.

Yost, G. R. (1994). Configuring Elevator Systems . *DEC Project Report*. Available through anonymous ftp from ftp://camis.stanford.edu/pub/protege/s2-word.sea.hqx

Zdrahal, Z and Motta, E. An In-depth Study of Propose & Revise Problem Solving. Proceedings of the 1995 Knowledge Acquisition Workshop, Banff, Canada, 1995.