T Veld, L. (1994). Configuring Elevator Specified in $K_{BS}SF$. In Schreiber and Birmingham.

Wielinga, B., Schreiber, G. & Breuker, J., A. (1992). KADS: A Modeling Approach to Knowledge Engineering. In *Knowledge Acquisition*, vol 4, no 1.

Yost, G., R. (1992). *Configuring Elevator Systems.* Technical report, Digital Equipment Co., Marlboro, Massachusetts, 1992.

European Workshop (EKAW´93, Toulouse, France, September 6-10, 1993), Lecture Notes in AI 723, Springer-Verlag, Berlin.

Neubert, S. (1993). *Model Construction in MIKE - Methods and Tools*. Ph.D. Thesis, University of Karlsruhe, 1994 (in German).

Poeck, K. (1991). Solving the YQT-Problem with the Assignment Shell COKE. Linster, M. (ed.): *Sisyphus ´91: Models of Problem Solving*, Arbeitspapiere der GMD 630.

Poeck, K. (1992). Tackling the Sisyphus Problem with COKE. Linster, M. (ed.): *Sisyphus ´92: Models of Problem Solving*, Arbeitspapiere der GMD 663.

Poeck, K. & Gappa, U. (1993). Making Role-Limiting Shells More Flexible. Aussenac, N. et al. (eds.), *Knowledge Acquisition for Knowledge-Based Systems, Proceedings of the 7th European Workshop (EKAW´93,* Toulouse, France, September 6-10, 1993), Lecture Notes in Artificial Intelligence 723, Springer-Verlag, Berlin.

Poeck, K. & Puppe, F. (1992). COKE: Efficient Solving of Complex Assignment Problems with the Propose-And-Exchange Method. In *Proceedings of the 5th International Conference on Tools with Artificial Intelligence*, Arlington, Virginia, November 10-13.

Poeck, K. (1995). *Configurable Problem Solving Methods for Assignment and Classification*., Ph.D. Thesis, University of Würzburg (in German).

Puppe, F. & Gappa, U. (1992). Towards Knowledge Acquisition by Experts. In *Industrial and Engineering Applications of Artifical Intelligence and Expert Systems, Proceedings of the 5th International Conference IEA/AIE-92*, Paderborn, June 9-1.

Puppe, F. (1993). *Systematic Introduction to Expert Systems: Knowledge Representation and Problem-Solving Methods*, Springer-Verlag, Berlin.

Rothenfluh, T., E., Gennari, J., H., Eriksson, H., Puerta, A., R., Tu, S., W. & Musen, M., A. (1994). Reusable Ontologies, Knowledge Acquisition Tools, and Performance Systems: PROTÉGÉ-II Solutions to Sisyphus-II. In Schreiber and Birmingham (1994).

Schreiber, G. (1992). *Pragmatics of the Knowledge Level*, Ph.D. Thesis, University of Amsterdam.

Schreiber, G., Birmingham, B. (Eds, 1994). SISYPHUS-II-VT Elevator design problem, *Proceedings of the 8th Banff Knowledge Acquisition for Knowledge Based Systems Workshop*, Vol. 3, Banff, Canada.

Sprau, R. (1993). *Modellierung eines Aufzug-Konfigurationssystems in KARL (Modelling a Configuration System for Elevators with KARL)*. Master's Thesis, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, (in German).

Schreiber, G., Terpstra, P., Magni, P. & van Velzen, M. (1994) Analysing and Implementing VT Using CommonKADS. In Schreiber and Birmingham (1994).

Schreiber, G., Wielinga, B., Akkermans, H., Van de Velde, W. & de Hoog, R. (1994). CommonKADS. A Comprehensive Methodology for KBS Development, *IEEE Expert*, vol 9, no 6.

A-27, Elsevier, Amsterdam, 139-168.

Angele, J., Fensel, D. & Studer, R. (1990). Applying Software Engineering Methods and Techniques to Knowledge Engineering. D. Ehrenberg et al. (eds.), *Wissensbasierte Systeme in der Betriebswirtschaft, Reihe betriebliche Informations- und Kommunikationssysteme, no 15*, Erich Schmidt Verlag, Berlin, 285-304.

Angele, J., Fensel, D. & Studer, R. (1994). The Model of Expertise in KARL. *Proceedings of the 2nd World Congress on Expert Systems*, Lisbon/Estoril, Portugal, January 10-14.

Brazier, F., M., T., van Langen, P., Philipsen, A., W., Wijngaards, N., J., E. & Willems, M. (1994). DESIRE: Designing an Elevator Configuration. In Schreiber and Birmingham (1994).

Chandrasekaran, B. & Johnson, T., R. (1993). Generic Tasks and Task Structures: History, Critique and new Directions. In: *Second Generation Expert Systems*, David, J.-M. et al. (Eds.), Springer.

Fensel, D., Eriksson, H., Musen, M., A. & Studer, R. (1993). Description and Formalization of Problem-Solving Methods for Reusability: A Case Study. In *Complement Proceedings of the 7th European Knowledge Acquisition Workshop (EKAW´93)*, Toulouse, France, September 6-10.

Fensel, D. & Poeck, K. (1994). A Comparison of Two Approaches to Model-Based Knowledge Acquisition. In Steels, L., Schreiber, G., and Van de Velde, W. (Eds.) *A Future for Knowledge Acquisition, 8th European Knowledge Acquisition Workshop, EKAW-94,* Lecture Notes in Artificial Intelligence 867.

Fensel, D. (1995a): *The Knowledge Acquisition and Representation Language KARL*, Kluwer Academic Publisher, Boston, to appear.

Fensel, D. (1995b). Assumptions and Limitations of a Problem-Solving Method: A Case Study. In *Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´95)*, Banff, Canada, February 26th - February 3th.

Gappa, U. (1995). *Graphical Knowledge Acquisition Systems and their Generation.*, Ph.D. Thesis, University of Karlsruhe, (in German).

Gruber, T. & Runkel, J. (1993). Ontolingua Theories for Sisyphus Elevator-Design, In: *ftp://ksl.stanford.edu//pub/knowledge-sharing/ontologies/ol-ontologies.tar.Z.*

Landes, D. (1994). DesignKARL - A Language for the Design of Knowledge-Based Systems. In *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering SEKE'94*, Jurmala, Latvia, June 20-23.

Landes, D. & Studer, R. (1995). The Treatment of Non-Functional Requirements in MIKE. In *Proceedings of the 5th European Software Engineering Conference ESEC'95*, Barcelona, Spain, September 25-28.

Marcus, S. (1988a). SALT: A Knowledge-Acquisition Tool For Propose-And-Revise Systems. In Marcus (1988b).

Marcus, S. (ed., 1988b). *Automating Knowledge Acquisition for Expert Systems*, Kluwer, Boston.

Neubert, S. (1993). Model Construction in MIKE (Model-Based and Incremental Knowledge Engineering). In Aussenac, N. et al. (eds.), *Knowledge Acquisition for Knowledge-Based Systems, Proceedings of the 7th*

computational framework.

The suite of models that we developed during the VT-experiment especially allowed us to come up with a characterization of the *competence* of the propose-and-revise method and a list of its *assumptions* (some examples are given in section 5.4). These assumptions are further described in Fensel (1995b), but the basic idea is that these assumptions allow for a strong problem-solving method to solve a task more efficiently than the weak generate-and-test paradigm only operating on the problem specification and not using any heuristics for the generation of a solution.

Making these assumptions explicitly allows us on the one hand, to prove properties of problem-solving methods, for example that they correctly solve the problem, when these assumptions hold. Therefore, they can be used during the construction and adaptation process of reusable methods or reusable building blocks of such methods. On the other hand, the assumptions allow us to test, whether a problem-solving method is really applicable for a given domain knowledge base. Therefore, they can be used to support the method selection process, i.e. the actual process of method reuse. Such an analysis is mandatory for building successful application system from reusable components.

The assumptions of a problem-solving method for the required domain knowledge can be defined together with the external part of the method ontology. This combined definition defines all the requirements for the method's domain knowledge. At present, both aspects cannot be represented in KARL. Therefore, KARL is a specification language for models of expertise and for their analysis, but the representation of reusable problem-solving method making their ontological commitments explicit is still not really supported by KARL.

Acknowledgements

We would like to thank the anonymous reviewers for their very helpful and detailed comments and Constanze Reinders for improving the English of this paper.

References

Angele, J., Fensel, D., Landes, D., Neubert, S. & Studer, R. (1993). Model-Based and Incremental Knowledge Engineering: The MIKE Approach. *Knowledge Oriented Software Design*, J. Cuena (ed.), IFIP Transactions

the CRLM approach which supplies powerful shells eliminating or drastically reducing the implementation effort, but provides less support for the early knowledge acquisition phases. By combining both approaches, a description of a system and of the used knowledge at different complementary levels can be achieved: The knowledge is described at the *conceptual level* in a semiformal manner by the different layers and primitives of a model of expertise. It is described at the *formal level* to define a precise and unique meaning. This formal description enables us to exactly define our interpretation of the problem-solving method *propose-and-revise* without referring to implementational aspects. The knowledge is described at the *implementational level* by a running system. The domain knowledge can comfortably be acquired and efficiently be executed by the shell.

One cannot state in general that a specific approach is best suited to solve a knowledge-engineering task like VT. Probably any structured approach will do, given that it provides the means to deal precisely enough with the peculiarities of the domain and the method. Somehow, the *process of reasoning about the system* is as important as the product of that process, independent of whether that product is a formal specification or an implemented system. This reasoning about the competence of a system can also be done during the implementation. Using a formal specification for this process has the following consequences:

- It requires additional effort as a semiformal and formal specification have to be built up. Actually, most of this effort is not really additional effort, as it has to be spent during the implementation otherwise.
- During implementation, the main concern is the product and not the process of understanding the system better. The product of the process is an implementation of the system. During specification, the emphasis lies in understanding the system better. This improved understanding is the important product of the process.
- The formal specification in KARL keeps the conceptual structure of the problem-solving method as the point of reference for discussing and understanding its formal details. This need not hold for the implementation which is concerned with symbol-level efficiency.
- The formal specification abstracts from implementational details, which are not related with the detailed specification of the method but with its realization in a specific

8.2 Experience with the Implementation

The adaptation of the problem solver worked as expected, but the generation of the knowledge editors raised some problems, since the problem solver and the knowledge-acquisition-generation system made significantly different assumptions about the internal representation of the domain knowledge. This is no problem if one assumes two different environments for knowledge acquisition and problem solving and writes a translator for both representations, but we prefer an integrated approach. We will adapt both, problem solver and knowledge-acquisition-generation system, in that respect. Additionally, we have noticed that our approach presently does not allow for the specification of the runtime environment, for example the explanation component, but that we have to code this explicitly.

It was a bit of a surprise that it took way longer to arrive at a sufficiently bug-free domain knowledge base compared to the effort it took to build the interpreter, especially considering that a reverse engineered OPS5 knowledge base in Yost (1992), a formal KARL specification in Sprau (1993), and a formal Ontolingua specification in gruber and Runkel (1993) were given. Some problems can be covered with automatic analyzer tools, but others are introduced by translating natural language descriptions into a formal representation. The correspondence of an informal and a formal specification cannot be proven formally. A good example for this is the compensation cable constraint, which was misrepresented by several modeling groups (including the Ontolingua specification) or the wrong use of radius instead of diameter in the KARL specification of the domain knowledge.

8.3 Experiences concerning the Combination of MIKE with CRLM

To some extent, both approaches are complementary and supplement each other very well. Fensel, Eriksson and Musen (1993) already report on a fruitful combination of a role-limiting method approach and MIKE. In its current stage, MIKE offers significant tool support for the early phases in knowledge engineering. The hyper model can be used to semiformally describe a model of expertise and this description can be further refined and operationalized by the specification language KARL. Language and tool support for the design phase is under way. Currently, there is no support for the implementation of a final system. The contrary holds for

of the domain layer and, thus, are defined twice.

- *Evaluate* knowledge: A constraint is modeled as an object of the class *constraints* and by a rule which defines the conditions for a constraint being violated. A constraint violation is indicated by deriving an object which denotes the respective constraint as an instance of the unary predicate *constraint_violated.* This predicate is used as a switch which turns fix rules on and off. Such usage of flags looks like assembler programming in logic.
- *Revise* knowledge: A fix is modeled as an object of the class *fixes*. The binary predicate *related_fixes* models the relationship between a constraint and its associated fixes and the ternary relationship *fixed_value* relates the fix to the parameter it changes and to the new value which the fix derives for the parameter. A fix is further described by a rule which intensionally defines the relation *fixed_value*. The representation of the "old value" of a parameter (i.e. as a value of the attribute which models the parameter) and the "new value" of a parameter (i.e. as a value of an instance of the predicate *fixed_value*) by different modeling primitives is necessary due to the monotonicity of the domain layer. Value changes cannot be expressed directly unless modal logics such as, for example, temporal logic were used.

It is no surprise that K_{BS}SF Veld (1994) runs into some similar problems. A specification language like DESIRE, which explicitly provides the object/meta distinction, allows a more explicit representation of these types of meta knowledge Brazier, van Langen, Philipsen, Wijngaards and Willems (1994). But even DESIRE has some of these problems, for example in the context of how to derive and represent the knowledge given by the dependencies of the propose rules at the domain layer implicitly without encoding it a second time.

We often claim that modeling with KARL is modeling at the knowledge level because the knowledge can be described declaratively without referring to specific algorithmic solutions for the inference steps, cf. Schreiber (1992). The knowledge is described declaratively without referring to symbol level control defining an algorithmic solution for elementary inference actions. But sometimes modeling with KARL seems to be modeling in the *basement* of the knowledge level.[1]

1. Especially when looking at the restriction on Horn logic which is necessary to operationalize KARL.

effort. This indicates the feasibility of method adaptation instead of simple method selection, even for a family of methods. In the following, we will present lessons which we have learned from using KARL, from using CRLM, and from combing them.

8.1 KARL: The Basement of the Knowledge Level

The domain layer was developed independently from the problem-solving method for the first time. Therefore, it was a crash-test for the mapping via extended Horn clauses to see whether the domain layer could be mapped onto the method ontology. The domain layer uses a hierarchy of several classes, whereas the problem-solving method works on a flat set of parameters. The intelligibility of the domain layer is significantly improved by this hierarchical structure. Related domain knowledge is grouped together and the hierarchy defines different levels of abstractions. Other problem-solving methods like hierarchical skeletal-design methods might require such a hierarchy for their reasoning process. On the one hand therefore the hierarchy improves *usability* and *reusability* of domain knowledge. On the other hand, in our case, this hierarchy of classes with attributes had to be mapped onto a flat set of parameters. Furthermore, the problem-solving method should not make assumptions about the number of parameters. Therefore, every attribute of the domain layer has to be mapped to an object at the inference layer. This is the only way to handle a set of attributes with unknown cardinality. The mapping is accomplished by clauses like the following:

$$par(attribute, domain\_class)[value: V] \in parameter \leftarrow object[attribute : V] \in domain\_class.$$

These clauses already embody *meta-reasoning*, because attribute names (i.e. terminological knowledge) at the domain layer is mapped to objects (i.e., constants) at the inference layer. For every class at the domain layer, such a clause has to be defined.

The domain layer itself contains several types of meta-knowledge which are not represented appropriately in KARL:

- *Propose* knowledge: The computation of values for attributes depends on the values of other attributes. This circumstance is reflected in the binary predicate *dependency,* which ranges over attributes. Such a definition of predicates between attributes was originally not possible in KARL. In addition, the dependencies are given implicitly in the propose rules

The next constraint violation (MINIMUM_PLATFORM_TO_HOISTWAY_LEFT) is caused by the parameter PLATFORM_TO_HOISTWAY_LEFT, since its value should be at least 8. Increasing OPENING_TO_HOISTWAY_LEFT by 1 to a value of 33, i.e. by the missing amount of PLATFORM_TO_HOISTWAY_LEFT, implies recomputation of PLATFORM_TO_HOISTWAY_LEFT to 8 and fixes the violation.Then the HOIST_CABLE_SAFETY_FACTOR is below its minimum (C-29_1). This is simply fixed by increasing the number of hoist cables from 3 to 5. At this stage, the value of vertical force of the car_guiderail is well above its upper bound (C-50). Again, the first fix is successful by upgrading the CARGUIDERAIL to the next model. Now all parameters are computed, but the last value violates the constraint C-48_2 concerning the traction ratio. Fixing this constraint is not trivial and requires a significant amount of search. After trying about 500 combinations, upgrading the MSHEAVEGROOVE.MODEL finally solves the problem. To get the same solution as in Yost (1992), we have to declare the fix upgrading the machine beam model as appropriate for C-48_2. Then, after about 1000 search steps, a combination of four fixes is found for the constraint: The distance from the counterweight to the rear of the platform is decreased to 1.75 inches, the car supplement weight is increased to 500 pound, the "3/16-chain" is selected as compensation cable and "S10x35.0" as machine beam. About 99% of the whole run-time of the configuration are used to fix this last constraint, the other parameters are computed pretty instantaneously. The *entire* runtime of the system is about 10 seconds on a Quadra 700 using Macintosh Common Lisp.

8 Discussion

The integration of KARL and CRLM resulted in a running solution exhibiting fast response times and a formal specification of the propose-and-revise method and the available domain knowledge. The mechanisms employed are generic and reusable in other domains or applications. The specification language KARL can obviously be (and already has been) used to describe other knowledge models. Its application in this experiment indicates that its use is feasible for real life tasks. The implementation resulted in a generic shell for the class of propose-and-revise problems which will be reused for similar problems. The shell was not been constructed from scratch. Instead, a shell for a related problem was adapted with relatively small

as described in Schreiber, Terpstra, Magni and van Velzen (1994).

The implemented system is generic w.r.t. the propose-and-revise method and not specific to the VT domain. The developed environment is suitable for other simple design problems. We validated this by successfully using the system for the U-Haul domain, another parametric design problem developed by John Gennari, c.f. Rothenfluh, Gennari, Eriksson, Puerta, Tu and Musen (1994).

7 Sample Trace

The following table show parts of a sample trace of the system with the VT domain. The parameters are computed in the order in which they appear in the table.

Parameter/Constraint	Value
Propose for COUNTERWEIGHT_BUFFER_BLOCKING_HEIGHT	0
...	...
Propose for ELEVATOR.MOTOR.MODEL-ID	MOTOR_MODEL_M03
Constraint C-34_1	
Propose for MOTOR_PEAK_CURRENT_REQUIRED	207.71633163639518
...	...
Propose for PLATFORM_TO_HOISTWAY_LEFT	7
Constraint MINIMUM_PLATFORM_TO_HOISTWAY_LEFT	
Propose for PLATFORM_TO_HOISTWAY_RIGHT	8
...	..
Propose for HOIST_CABLE_SAFETY_FACTOR	6.865653330712384
Constraint C-29_1	
Propose for MACHINE_GROOVE_PRESSURE	207.84.54248617777776
...	...
Propose for CAR_GUIDERAIL_VERTICAL_FORCE	6328.686463333333
Constraint C-50	
Propose for COUNTERWEIGHT_PLATE_QUANTITY	80
...	...
Propose for HOIST_CABLE_TRACTION_RATIO	1.8534565074966143
Constraint C-48-2	

Figure 24. Trace of the configuration for the test case.

The first constraint violation being detected is C-34_1, since the selected motor.model is not compatible with the actual machine.model. The fix simply upgrades to a compatible machine.

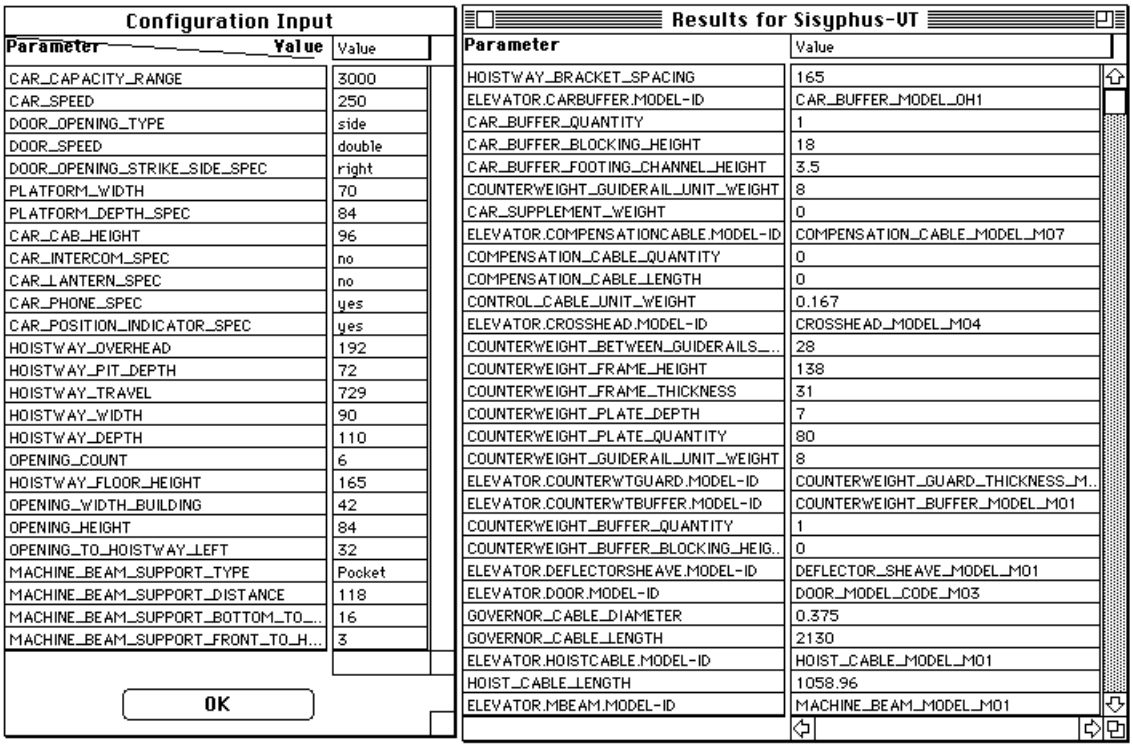


Figure 22. Input and results of the configuration.

Since in the VT-Experiment two formal ontologies, Sprau (1993) and Gruber and Runkel (1993), were given, we defined mappings from both ontologies to the knowledge representation of the finally implemented system, although this is not typical for our approach. The ontology in Sprau (1993) contained sufficient knowledge for our propose-and-revise implementation. Part of the entire Sisyphus project was to evaluate the reuse of domain knowledge as provided by the ontology of Gruber and Runkel (1993). Therefore, we also defined a transformation of this ontology. It contained large parts of the knowledge actually needed for our propose-and-revise system, but we had to add some missing propose rules, add the fixes and correct some minor bugs. The mapping was handled automatically, following suggestions by Guus Schreiber

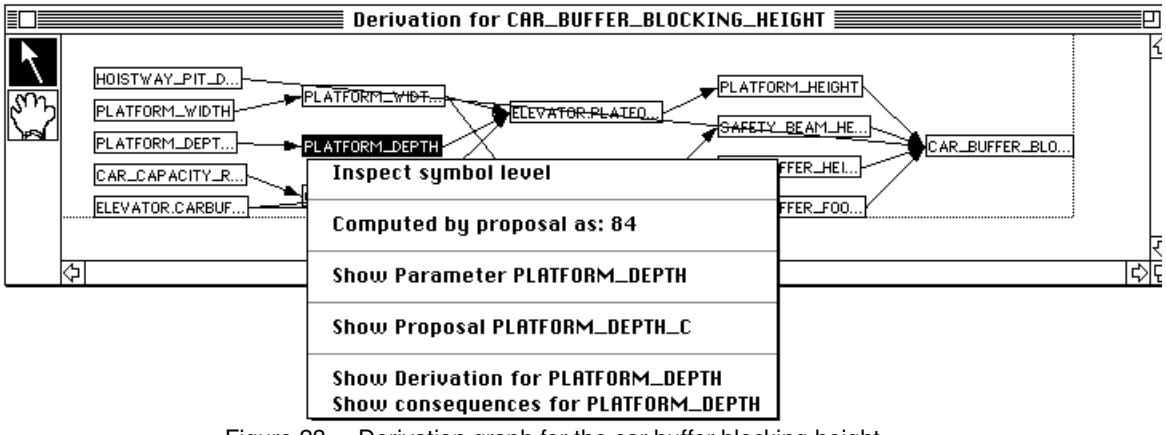


Figure 23. Derivation graph for the car buffer blocking height.

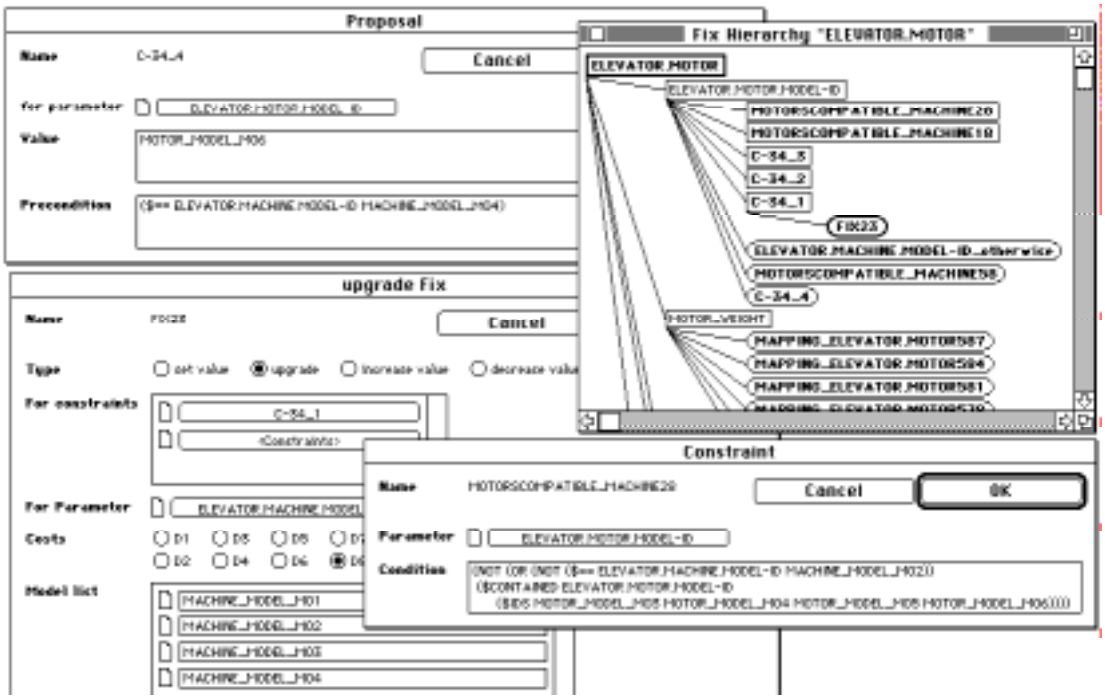


Figure 21. A hierarchy and some corresponding forms generated by the knowledge acquisition environment.

acquired. Again, the specification can be enriched with layout information.

The declaration of the graphical knowledge acquisition environment took about half a day. Some of the generated forms are shown in figure 21. The three forms show the acquisition of a propose rule, a constraint, and an upgrade fix related to the computation of the motor model of the elevator. The hierarchy in the upper right part of the figure shows a more global view of the knowledge base and is used to acquire and present the relations between components and parameters, parameters and propose rules, parameters and constraints, and ,finally, constraints and fixes.

In contrast to the knowledge acquisition environment, the end user environment, for example the table to enter the configuration data, the table for the configuration results, and the explanation component is not generated but hard coded. However, the coding was rather easy given our library of graphical primitives that are also used for knowledge acquisition. In figure 22 the tables containing inputs and outputs are shown. The user may either get an explanation for the computed values or he may change parameters as long as the configuration remains consistent w.r.t the constraints or he accepts the violations. The explanation shows the abstracted derivation graph where only the parameters used to compute the value are shown, but not the actual rules (see figure 23).

interpreter from scratch. Rather, large parts of the propose-and-exchange method used for the previous Sisyphus problems, cf. Poeck (1991, 1992), could be reused. A detailed description of this method and the shellbox COKE in which it is realized can be found in Poeck (1995). Since the propose-and-revise method has already been described in detail in section 5, we will focus in the following on the differences to propose-and-exchange.

In the adapted implementation, propose-and-revise consists of four main steps stemming from propose-and-exchange that are repeated until a complete solution is found or until the configuration task terminates unsuccessfully: Select a parameter (see section 5.2), propose a value for this parameter (see section 5.2), test the constraints for this parameters (see section 5.3), and revise the configuration by applying fixes if constraints do not hold (see section 5.3). The first step is exactly the same as in the propose-and-exchange method. No explicit knowledge for this step is contained in Yost (1992), but at least a partial ordering is defined by a topological sort of the dependencies via the propose- and constraint- and fix rules. The second step is slightly different to propose-and-exchange in that a value is computed rather than selected. While the third step is completely identical again, the correction of constraints in the last step is quite different and had to be implemented entirely anew.

The actual realization of the propose-and-revise-method as an adaptation of propose-and-exchange was rather straightforward and mainly meant implementation of the revise step and adaptation of the control flow. The whole implementation effort for the method took about 2 days.

Although our actual VT-knowledge base was mapped from the ontology in Sprau (1993) and later alternatively from Gruber and Runkel (1993), we also generated a graphical knowledge acquisition environment as it is normally used within our approach. This is done as described in Poeck and Gappa (1993) and Gappa (1995), by precisely declaring the internal knowledge representation, c.f. figure 20, i.e. the object types with attributes and their syntax, relations between the objects and dependencies of attributes. In addition to that, we generated knowledge editors like hierarchies, for example, by specifying relations to be acquired. The specification of the relations can be extended by further layout informations, for example the shape of the boxes. Forms can be generated by simply specifying the object type and the attributes to be

procedure in the shell environment and to a logical rule in KARL. In KARL, rules cannot be the value of an attribute. In the shell environment, the calculation rule for a parameter can used as value for the parameter directly.

6 Implementation Aspects

The implementation of our solution of the configuration task was carried out according to the configurable role-limiting shell approach described in Poeck and Gappa (1993) and Poeck (1995), which extends earlier work on role-limiting methods, c.f. Marcus (1988b) and Puppe (1993). We decided to follow the problem-solving method *propose-and-revise* by Marcus (1988a) as closely as possible. The specification of propose-and-revise in section 5 is a formal reverse engineered version of the implementation and should be used as a reference.

Based on an initial understanding of the problem-solving method, we first designed the already mentioned external method-oriented knowledge representation shown in figure 20. To simplify knowledge-acquisition and user-interaction, we later added some minor details like grouping of parameters in components for better organization in the corresponding hierarchy, etc.

In a second step, after defining a suitable knowledge representation, an interpreter and graphical environment for this knowledge representation was developed. There was no need to build an

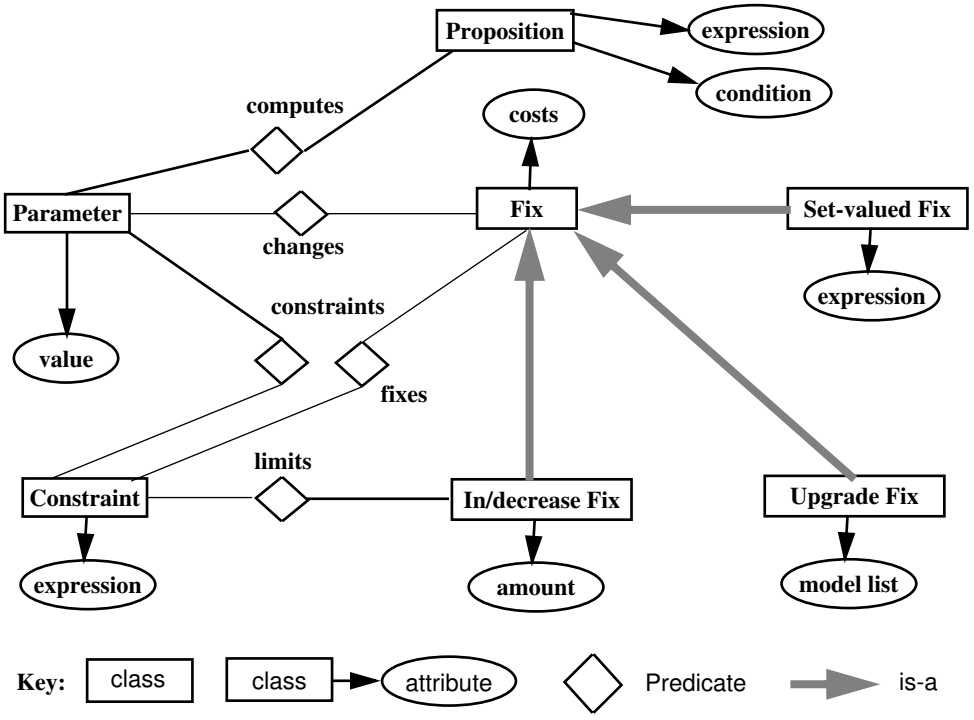


Figure 20. Knowledge representation for propose-and-revise.

applied does not matter). A *critical control decision* of propose-and-revise is that it cannot backtrack beyond a selected constraint violation. If a selected constraint violation cannot be repaired by the fix knowledge, the method stops with a failure instead of selecting another constraint violation and trying to fix this. The propagation of the later repair activity could possibly also repair the former violation. As no heuristic selection knowledge for constraint violations is provided at the domain layer, this is a very significant and critical restriction of the method. For more details see Fensel (1995b). Becoming aware of these features is also possible by implementing rather than formalizing the method. The main advantages of a specification in KARL are:

- The formal specification maintains the conceptual structure of the problem-solving method as point of reference for discussing and understanding formal details of it. This need not hold for the implementation, which is concerned with an efficient realization by a computational agent.
- The formal specification abstracts from implementational details which are not related to the detailed specification of the method, but with its realization in a specific computational framework.

In section four we presented the domain model of the VT-domain. In section five, we defined a method ontology by terminological definitions in inference actions, stores, views, and terminators. The union of all these definitions defines the *method-specific ontology*. When looking only at views and terminators, we have that part of the method ontology which is visible to an external observer of the method. This *external method-specific ontology* defines the knowledge types required by propose-and-revise as input from the domain layer. As KARL does not explicitly provide such a collected view on its method ontology, we will use the representation of the external method ontology of the shell and the knowledge acquisition environment which will be described in the following section. The external part of the method ontology is depicted in figure 20.

A technical difference between the distributed specification of this ontology in the KARL model and the specification in figure 20 is caused by the fact that KARL cannot refer directly to rules as syntactical entities. What is called an *expression* in figure 20 corresponds to a

that were violated before the application of the fixes (these constraints are still accessible in the store *violated constraints*).

The result of the comparison carried out by the inference action *compare constraints* is written into the store *resulting constraints*. In particular, this result indicates which constraint violations have disappeared due to the application of the fixes and, respectively, which other constraints had not been violated before the fix was applied. A combination of fixes is considered successful if it has corrected some constraint violations without causing new constraint violations.

In this case, the set of parameter values in the store *virtual partial design* is copied to the store *partial design* and the contents of *new violated constraints* substitute the previous contents of *violated constraints*, i.e. the effect of the fixes is made permanent. Otherwise, the contents of the stores *partial design* and *violated constraints* remain unchanged, and a new combination of fixes is tried, i.e. the effects of the combination of fixes tried last are discarded.

In either case, the main loop in figure 16 continues trying combinations of fixes until all constraint violations disappear or until the set of applicable combinations of constraints has been exhausted without removing all constraint violations. In the latter case, the problem-solving method stop, since this means that the current problem cannot be solved in this way.

5.4 Concluding Remarks

The *conceptual* and *formal* specifications of the different types of knowledge required for problem-solving provide insight into several important aspects of propose-and-revise. Propose-and-revise makes *strong assumptions* about available domain knowledge. For instance, in our understanding, the method assumes that the graph formed by propose rules is cycle-free and that the fixes do not interact (i.e. the order in which fixes resulting from a fix combination are

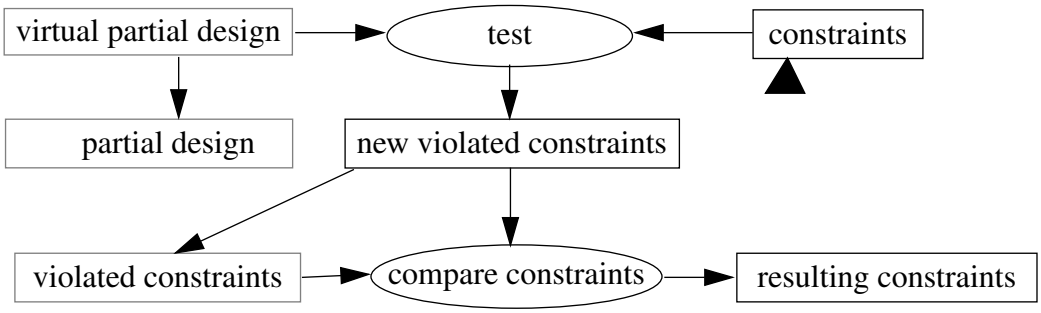


Figure 19. Inference structure of *evaluate fix*.

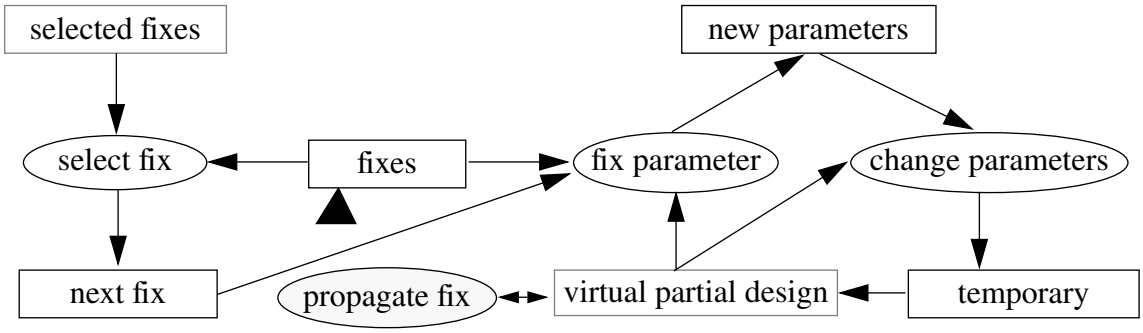


Figure 18. Inference structure of *apply fix*.

layer which is accessible through the view *fixes*. The parameter and its fixed value are stored in *new parameter*.

The inference action *change parameters* substitutes the old value of the corrected parameter in the *virtual partial design* and stores the updated set of parameter values in the store *temporary*. The contents of *temporary* are then copied to the store *virtual partial design*.[1]

Whenever a fix involved in the current combination of fixes has been applied for the specified number of times, the effects of these applications are propagated to other parameters by the inference action *propagate fix*, which again is a composed inference action. The new value has to be propagated through the network of propose rules as defined at the domain layer. For reasons of limited space, we will skip the refinement of this complex inference.

5.3.3 Evaluating Constraints after the Application of a Fix

The inference action *evaluate fix* in figure 15 is also composed. Its refinement is shown in figure 19. When a combination of fixes has been applied and its effects have been propagated to other parameters, the resulting set of parameter values has to be checked again in order to find out if the fix has actually improved the situation with respect to the amount of violated constraints. The constraints' re-evaluation is accomplished by the composed inference action *evaluate fix*.. The set of current parameter values in the store *virtual partial design* is evaluated by the inference action *test*, using the view *constraints* in order to access the constraints formulated in the domain layer. The constraint violations are written into the store *new violated constraints*. In order to find out if the applied combination of fixes results in an improvement, the current constraint violations in the store *new violated constraints* have to be compared to the constraints

1. The respective value cannot be replaced in the store *virtual partial design* directly, since this would run counter to the monotonicity of individual inference actions.

fixes in a combination may be applied more than once (for instance, upgrading a part of the elevator). *Combine fixes* also takes precautions for this. Thus, combinations of fixes may differ in the fixes involved as well as in the number of iterations of individual fixes.

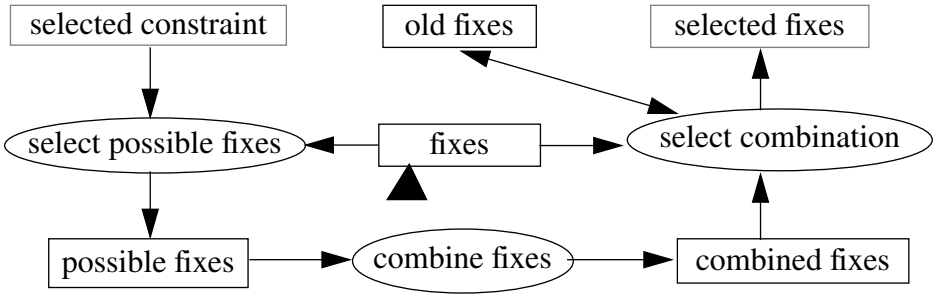


Figure 17. Inference structure of *deliver fix*.

Combinations of fixes causing only minor changes should be tried before any fixes implying severe modifications. Yost (1992) provides a scheme supporting the use of fix combinations. This scheme is described in the view *fixes* by means of several predicates used by the inference action *select combination* for identifying the most promising combination of fixes. *Select combination* also takes into account that only such a combination is selected which has not already been tried on the current set of constraint violations. It does so by keeping track of already tried combinations of fixes in the store *old fixes*. The fixes involved in the chosen combination are then stored in the store *selected fixes*.

5.3.2 Applying Fixes

The inference action *apply fix* in figure 15 is also composed. Its refinement is shown in figure 18. The store *selected fixes* contains a collection of individual fixes, which make up a single composite fix. Each of the individual fixes is tagged with a number indicating how many times the fix should be applied. The inference action *select fix* identifies the fix within the combination which should be applied next. This is again based on the preference relationship available through the view *fixes* which indicates that fixes causing minor side-effects should be applied first. This seems to be fairly reasonable as Yost (1992) does not give an indication in which sequence to apply fixes *within* a combination.

Which fix is to be applied next is recorded in the store *next fix*. This store serves as input for the inference action *fix parameter* which computes a modified value for the parameter affected by the chosen fix. The computation of this value relies on the predicate *fixed_value* at the domain

```
SUBTASK revise
    /* Check for constraint violations. */
        violated constraints := test(partial design);
    /* The logical variable deadlock becomes true if a constraint violation could not be resolved. */
        deadlock := false;
    WHILE ¬∅(violated constraints) ∨ deadlock
    DO
        /* Initialize a copy of partial design. */
            virtual partial design:= partial design;
        /* Select one of the violated constraints. */
            selected constraint := select constraint(violated constraints);
        REPEAT
            /* Generate applicable combinations of fixes and select one of them. */
                selected fixes := deliver fix(selected constraint);
            /* Apply the combination of fixes and propagate its effects. */
                virtual partial design := apply fix(selected fix, virtual partial design);
            /* Check if the applied fixes improve the situation. */
                improvement := false;
                (violated constraints, partial design) :=
                        evaluate fix(violated constraints, virtual partial design)
        UNTIL ∅(selected fixes) ∨ improvement;
        IF ¬improvement THEN deadlock := true ENDIF
    ENDDO
ENDSUBTASK;
```

Figure 16. Controlflow of *revise*.

particular constraint in the store *selected constraint*. The association between constraints and fixes is described at the domain layer by the predicate *related_fixes* which is accessible through the view *fixes*. In some cases, a single fix may not be sufficient for correcting a constraint violation, but combinations of fixes must be considered. To that end, the inference action *combine fixes* generates all possible combinations of the fixes in the store *possible fixes*, which roughly corresponds to computing the power set of applicable fixes. Furthermore, some of the

in the VT-application, selection will be random (again we use the lexicographical ordering of the object-id terms). The selected constraint (which is stored in the store *selected constraint*) is then used by the composed inference action *deliver fix* to find fixes that might remove the particular constraint violation. It then selects the fix or the combination of individual fixes that seem to be most appropriate in the given situation. *Apply fix* then applies the selected (elementary or composite) fix to the *partial design* (actually to *virtual partial design*, which is the internal copy of *partial design* in *revise*). As a result, a new value for one parameter is computed as indicated by the respective fix. Since other parameters may depend on the value which has just been modified, *apply fix* also propagates the effect of that modification onto dependent parameters, thus giving rise to a new set of currently computed parameter values in the store *virtual partial design*

The application of a fix may have side-effects on constraints on other parameters. Therefore, after propagating the effects of the fix, the current collection of parameter values is evaluated again with respect to the given constraints by executing the composed inference action *evaluate fix*. If the situation has improved, the contents of the store *virtual partial design* are copied to the store *partial design*, i.e. the effects of the fix are made permanent, and the set of violated constraints in the store *violated constraints* is updated. If the application of a fix combination has not lead to an improvement, *deliver fix* is used to deliver the next fix combination as long as fix combinations for the selected constraint violation exist. If no further combinations exist, the revise step stops and sets the boolean variable *deadlock* to “true”.

Even in the case of an improvement, violated constraints could still exist. In this case, the selection of one violation, the determination and application of a fix, and the propagation and evaluation of its effect is repeated once again. The internal control of the inference action *revise* is expressed in KARL as shown in figure 16.

5.3.1 Determining Applicable Combinations of Fixes

The inference action *deliver fix* in figure 15 is also composed. Its refinement is shown in figure 17. After a constraint violation has been detected, the first step for its removal is the execution of the composed inference action *deliver fix* in order to identify applicable fixes. To that end, the inference action *select possible fixes* determines all individual fixes associated with the

heuristics with respect to the ordering of parameters which could get a value according to the propose rules and the already derived values of other parameters. Due to this fact, the corresponding mapping expression states that an instance of *parameter* is preferred over another if its object-id term is lexicographically smaller. If there were any domain-specific heuristics, the mapping could be easily adapted to take them into account.

5.3 Revise: Resolving Constraint Violations

After a previously unknown parameter value has been computed by *propose*, the extended set of known parameters is checked with respect to the constraints. If a constraint violation is detected, attempts to remove the problem are made. This is done until no more constraint violations can be found or no more repair activities are available. The inference structure of the composed inference action *revise* is given in figure 15.

The elementary inference action *test* checks, whether the new value together with the already derived partial design lead to constraint violations. The view *constraints* is used to deliver the required domain knowledge. All violations are put in the store *violated constraints*. An attempt to restore consistency proceeds by first choosing *one* of the violated constraints which will be tackled next. This is accomplished by the inference action *select constraint* which selects one of the instances in the store *selected constraint* using a preference relationship defined in the view *constraints*. Since no elaborate heuristics for preferring a constraint over others are given

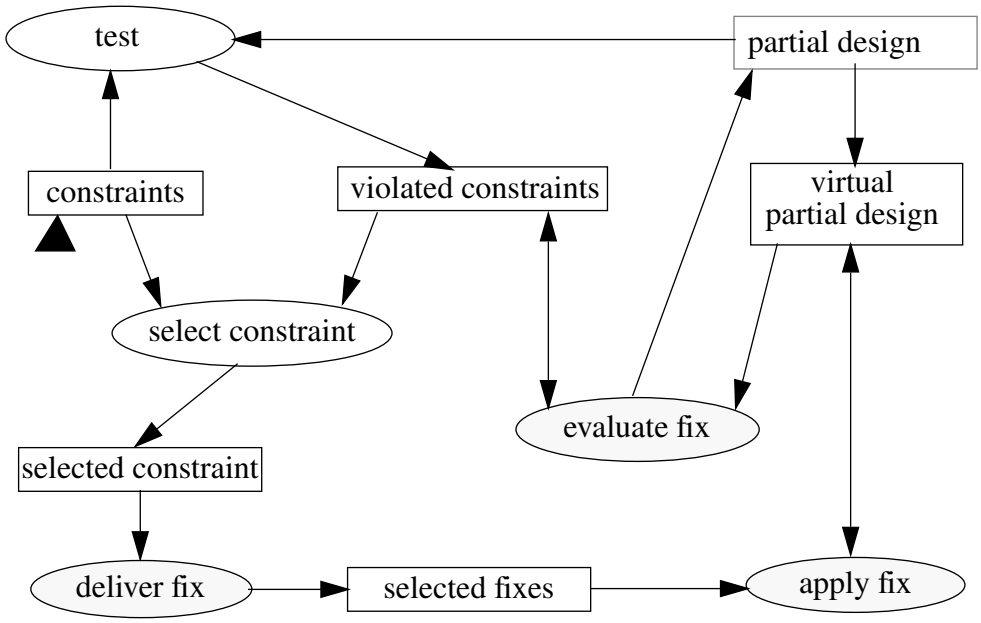


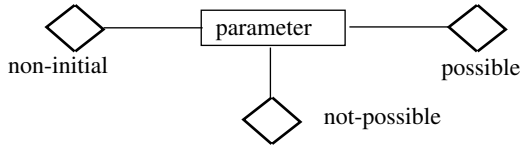
Figure 15. Inference structure of *revise*.

INFERENCE ACTION select possible parameters

PREMISES parameters, partial design;

CONCLUSIONS possible parameters;

DEFINITIONS



RULES

/* Parameters which do not depend on other parameters are possible candidates. */

non-initial(X) ← X[*depends*:: {Y}] ∈ *parameter.*

possible(X) ← X ∈ *parameter* ∧ ¬*non-initial*(X).

/* Parameters which depend only on parameters already having a value are possible candidates. */

not-possible(X) ← X[*depends*:: {Y}] ∈ *parameter* ∧ ¬Y ∈ *partial-design.*

possible(X) ← X ∈ *parameter* ∧ ¬*not-possible*(X).

/* Parameter already having a value are not a candidate. */

X ∈ *possible parameters* ← *possible*(X) ∧ ¬X ∈ *partial design*.

END;

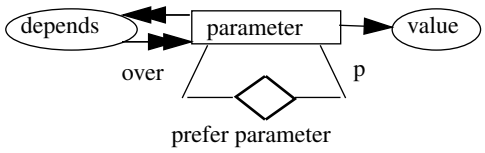
Figure 14. The inference action *select possible parameters*.

In the particular case of a view or a terminator, there is also a specification how this terminology is related to domain specific terms. For instance figure 13 indicates that an instance of the class *parameter* underlying the view *parameters* will be generated for each attribute of the class *elevator* at the domain layer (see clauses 1,2,...,*n*). Also, each attribute of one of the classes describing a particular part of the elevator to be configured corresponds to an instance of the class *parameter* at the inference layer. The set-valued attribute *depends*, which is defined by clause *n*+1, establishes a dependency network of parameters in a similar way as outlined in Marcus (1988a), cf. section 4.3.1. It is used by the inference action *select possible parameters* in order to determine the parameters for which a value can be derived.

Furthermore, the view *parameters* is associated with a predicate, *prefer parameter*, which expresses an ordering of instances of the class *parameter* (see clause *n*+2). This is necessary, as propose proposes only one value for *one* parameter per step and does not select all parameters which could be given a value according to the partial design. There are no domain-specific

VIEW Parameters

DEFINITIONS



UPWARD MAPPING

(1) $par(A,elevator)[value: V] \in parameter \leftarrow$

$O[A: V] \in elevator.$

(2) $par(A,building)[value: V] \in parameter \leftarrow$

$O_1[elevator\_build : O_2[A: V]] \in elevator.$

...

(n+1) $par(X_1, Y_1)[depends:: \{par(X_2, Y_2)\}] \leftarrow dependency(attribute: X_1, on: X_2).$

(n+2) $prefer\ parameter(p: X, over: Y) \leftarrow X \in parameter \wedge Y \in parameter \wedge (X \prec Y).$

END;

Figure 13. Detailed specification of the view *parameters*.

the values of which are already known (i.e. which are in partial design). The store *possible parameters* then contains these parameters, for which values can now be computed. One of these parameters is selected by the inference action *select parameter.* Only one single parameter is selected. The selection is based on the predicate *prefer parameter* associated with the view *parameters* (cf. figure 13). Finally, the value of the selected parameter is computed by the inference action *derive value* according to the respective rules at the domain layer, which are accessible through the view *propose rules*. The computation of the parameter is accomplished on the basis of already known parameter values. The specification of the inference layer of propose includes the formal specification of the elementary inference actions *select possible parameters*, *select parameter*, and *derive value*, the formal specification of the stores *possible parameter* and *new parameter*, and of the views *parameters* and *propose rules*. In the following, we will present the formal definitions for the view *parameters* (see figure 13) and the inference action *select possible parameters* (see figure 14).

The view *parameters* is specified as indicated in figure 13. Each role is associated with class or predicate definitions determining the terminology to be used by the problem-solving method.

```
TASK propose & revise
    REPEAT
        /* Select the next parameter and ompute the value of a previously unknown parameter. */
            partial design := propose(partial design);
        /* Derive a new consistent partial design. */
            partial design := revise(partial design)
        /* The logical variable new value becomes true in propose if a new value could be derived.
            The logical variable deadlock becomes true in revise if a constraint violation could not
            resolved.*/
    UNTIL ¬new value ∨ deadlock
        IF ¬deadlock THEN solution := partial design;
ENDTASK;
```

Figure 11. Controlflow of *propose* & *revise*.

additional parameter values, evaluating constraints, and, if required, fixing constraint violations is repeated until propose does not propose a new parameter value (i.e., the logical variable *new value* becomes false). If this is the case, the partial design is regarded as complete design, i.e. as *solution*.

5.2 Proposing Additional Parameter Values

The composed inference action *propose* of figure 10 is refined to yield the inference structure depicted in figure 12.[1] The inference action *select possible parameters* in the refinement of *propose* identifies parameters with currently unknown values which only depend on parameters

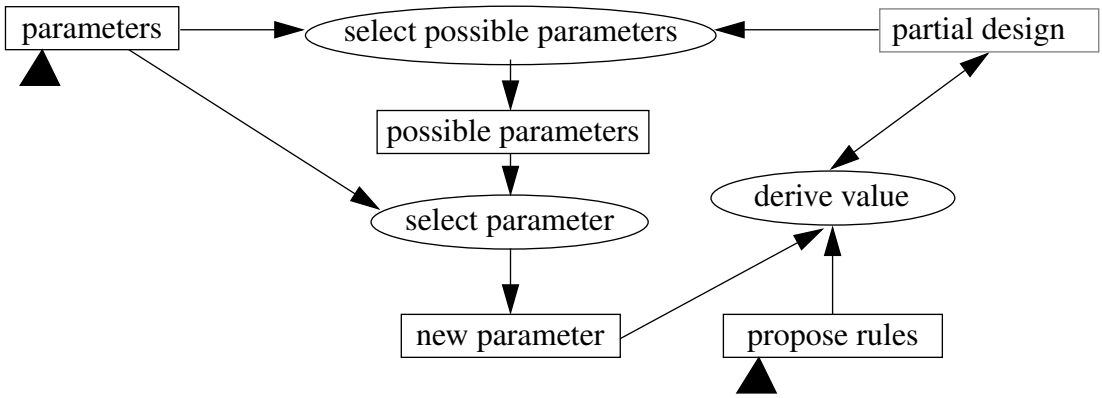


Figure 12. Inference structure of *propose*.

1. Roles are drawn as boxes with dotted borders if they also appear on more abstract levels of refinement.

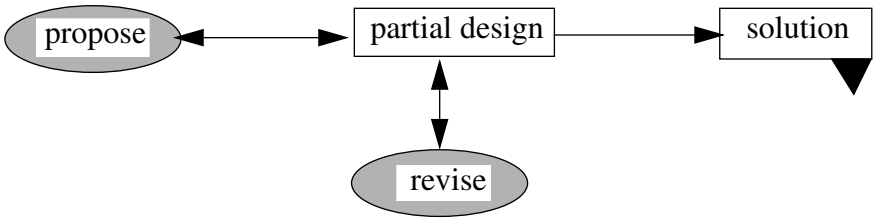


Figure 10. Inference structure of *propose & revise*.

In the following, parts of the specification of the method in KARL will be given. In order to reduce complexity, KARL supports the notion of hierarchical refinement of parts of the problem-solving method. Therefore, the presentation of the configuration task's specification with KARL starts with an abstract view, which is then detailed further on.

5.1 An Abstract View on Propose-and-Revise

At an abstract level, the chosen problem-solving method for the configuration of elevator systems can be depicted in an inference structure as shown in figure 10. The circles in the figure denote inference actions, i.e. problem-solving steps. The inference actions *propose* and *revise* in figure 10 are subject to further decomposition (indicated by shaded bubbles), i.e. they are an abstraction of more detailed levels of inference structures (see the following subsections).

Boxes indicate *roles* which supply input to inference actions or collect their output as indicated by arrows. KARL distinguishes three types of roles: *Views* (a box supplemented with a small triangle pointing upwards) are used to deliver knowledge from the domain layer for the reasoning process. *Terminators* (a box supplemented with a small triangle pointing downwards) are used to write results of the problem-solving process back to the domain layer. They are used to rephrase the generic terms of the problem-solving methods in domain-specific terms. *Stores* (boxes without a triangle) define data stores, which model the dataflow between inferences.

The control flow of the method is defined in figure 11. The two inference actions *propose* and *revise* work on a *partial design*. *Propose* determines the next parameter which should get a value and additionally proposes a value for it. *Revise* checks, whether the union violates constraints. It modifies parameter values as long as there are constraint violations, and no constraint violation is identified that cannot be repaired by any of the available fixes. If a constraint violation cannot be repaired, *revise* stops without a solution (i.e. the logical variable *deadlock* becomes true). If a consistent parameter assignment is found, the process of proposing

48_2 for the *hoist_cable_traction_ratio*. Therefore, it cannot be treated as a constraint.

Nevertheless, neither the *model* of the domain knowledge nor its *representation* by KARL is totally method-specific, since propose-and-revise does not handle components with attributes, but only a flat set of parameters. The domain layer of KARL conserves the conceptual structure of the domain by hierarchically grouping the components and their attributes. Keeping the hierarchical structure of the domain knowledge can be helpful for other types of tasks like (model-based) fault diagnosis or other types of problem-solving methods like hierarchical design methods. Also, it can be used to hierarchically structure the knowledge acquisition tool for domain knowledge (see figure 21). The link between the domain model and the representation implied by the problem-solving method can be specified with flexible mappings in KARL (see section 5). That is, KARL enables the method-independent representation of domain knowledge.

5 The Problem-Solving Method

The problem-solving method employed for configuring elevators closely follows the approach outlined in Marcus (1988a) and Yost (1992). *Propose-and-revise* assumes that a configuration is simply described by a set of parameters. Parameters represent features whose values determine attributes of the elevator to be configured and which may change during the configuration process. For instance, the value of a particular parameter might indicate that the *compensation cable* (the parameter) currently is a *model 5/16-chain* (the value). Configuration then amounts to computing the values of the output parameters on the basis of the actual input parameters. This is done with the help of so-called propose knowledge, which either represents good guesses like *"try to use compensation cable model 3/16"* or facts like *"the total weight of the cable system equals the sum of the weights of the four components"*. The consistency of the configuration is supervised by constraint knowledge. When one or more of the constraints do not hold, no simple backtracking occurs, but specific fix actions are applied until all constraint violations are resolved or until it has been ensured that a constraint violation cannot be resolved by any of the available fix actions (in which case the configuration process aborts with failure). Fix actions modify previously guessed values of some parameters.

must be modeled by an additional attribute *new-value* containing the new value for an attribute.

4.4 Is the Domain Layer Independent of the Problem-Solving Method?

The KARL model of the VT domain was developed independently of the Ontolingua description in Gruber and Runkel (1993), since the latter was not available when we started modeling. The main difference between the KARL model and the model in Ontolingua is that Gruber and Runkel (1993) tries to define a domain ontology that should only contain the domain knowledge necessary to define the problem specification. That is, only the domain knowledge required for the functional specification of the knowledge-based system should be covered by it. The KARL model is more complete and contains the complete domain knowledge required for the problem-solving process as given in Yost (1992). Therefore, the KARL model additionally contains fixes and knowledge for component selection. Furthermore, it clearly separates propose rules and constraints (as they are treated differently by the problem-solving method) both uniformly represented as constraints in Gruber and Runkel (1993). In the case of knowledge-based systems, an important part of the expertise is not only concerned with *what* a solution is, but also with *how* to achieve a solution in an efficient manner. Therefore, this knowledge is an important part of the domain knowledge and is therefore included into our domain model.

The distinction between problem specification and problem-solving knowledge as made by Gruber and Runkel (1993) is not as simple as it might look, as the fixes (which are used by the problem-solving method propose-and-revise) also implicitly represent preferences between solutions. Fixes have costs, and their application leads to a less preferred solution if they have high costs. The preference knowledge on solutions is hardwired in the way to find them by applying fixes. Excluding fixes implies therefore to miss parts of the functional specification.

Additionally, the uniform representation of propose rules and constraints as constraints in Gruber and Runkel (1993) is misleading, since some propose rules are heuristics and only define initial values that may be changed during the configuration. An example for this is the "constraint" *counterweight_to_platform_rear_c* that defines an initial value for the *counterweight_to_platform_rear*. This value may later be decreased in order to fix the constraint *c-*

(the next platform model according to the order *platform_order*) for the platform. The upgrade order has been defined extensionally in the relationship *platform_order* (see section 4.2).

$$fixed\_value(fix: fix_{21}, attribute: selected\_platform, new\_value: N) \leftarrow$$
$$P[selected\_platform : B] \in platform \wedge$$
$$platform\_order\ (current : B, next : N).$$

Figure 9. Fix_{21} and the new derived value.

The domain model of the VT-domain contains 43 fixes.

The relationship between fixes and constraints could also be modeled by a set-valued attribute like $c_{11}[related\text{-}fixes :: \{fix_{21}, ...\}]$ such as it is done in the implementation later on.[1] Still, a significant problem of the representation in KARL remains. In KARL, we had to model each constraint and each fix twice:

- Each constraint and each fix is represented by a rule which expresses when a constraint is violated and what a fix does if it is applied.
- Each constraint and each fix is represented by a constant which defines a name, i.e. a denotation for it.

These denotations are necessary to define the relation between constraints and fixes. It is not possible to directly refer to a rule, as a rule is not an entity of the alphabet (i.e. not a term) of the language. Pointing from a constraint directly to a rule which repairs it would require meta-logic, where formulas from the object logic can be treated as terms of the meta-logic. Even $(ML)^2$, which provides such a relationship between domain and inference layers, has not provided such a powerful mechanism for modeling the domain layer. In KARL, aspects of meta-logic were included in a bottom-up manner[2], but this was the first case where an extended object-meta relationship would have been useful.

A second problem with fixes is that they introduce non-monotonicity by changing a value. As KARL does not provide a variant of modal logic, e.g. temporal logic, for the domain layer, this

1. In the implementation a constraint is linked to the actual fix procedures objects and not just their identifiers

2. Predicates can range over classes and classes can be values of attributes. As classes correspond semantically to predicates, this already includes a partial syntactical extension of first-order logic. See Fensel (1995a) for more details.

some cases, several design modifications are applicable for fixing a constraint violation. In a similar way to constraints, fixes are modeled as elements of the class *fixes*. Some fixes may have severe consequences, such as modifications of building dimensions or of contract specifications. Therefore, fixes are labeled with their *cost*: the higher the costs, the less desirable. Furthermore, some fixes such as, for example, upgrading to another model, may be applied repeatedly.

In order to be able to apply a fix it one has to know which fix might resolve which constraint violation. This relationship between fixes and constraints is modeled by the relationship type *related_fixes*. *Related_fixes* is a m:n relationship, i.e. different fixes may exist for a single constraint and one fix may be related to different constraints. Conditions for the applicability of fixes are defined by KARL rules. Such a rule yields an instance of a relationship type *fixed_value* indicating which fix has been applied, which attribute is affected by the fix, and what the new value for the attribute should be (cf. figure 9).

For instance, the fix fix_{21} may potentially resolve the violation of the above mentioned constraint c_{11} (see figure 8).

fix_{21}[*name*: "*upgrade the platform*", *cost*: 8, *max_iterations*: 2] ∈ *fixes*.

related_fixes(*constraint*: c_{11}, *fix*: fix_{21}).

Figure 8. Fix_{21} and its relation to constraint c_{11}.

The *cost* of 8 for fix_{21} indicates that it has major implications on equipment selection or sizing (*cost* ranges from 1 to 10). As there are three different models of the *platform* (i.e., *2.5B*, *4B*, and *6B*), upgrading can occur at most twice, i.e., the value of the attribute *max_iterations* is 2. Fix_{21} upgrades the *model* of the *platform* as shown in figure 9. This rule yields the new value *N*

constraint_violated (*constraint*: c_{11}) ←

E[*elevator_car* : *C*, *elevator_drive* : *D*] ∈ *elevator* ∧

C[*car_platform* : *P*[*selected_platform_model* : "*2.5B*"]] ∈ *car* ∧

D[*drive_cable_system* : *S*[*compensation_cable* : *X*]] ∈ *drive* ∧

X[*car_top_load* : *L*] ∈ *compensation_cable* ∧ *L* > 600.

Figure 7. Violation rule for *c-11*.

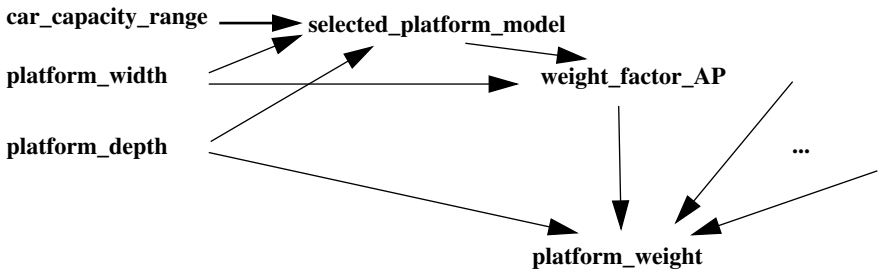


Figure 6. Parameter dependencies.

The dependencies between the parameters mentioned in the propose rules above yield the dependency graph depicted in figure 6. The problem-solving method described in the next section makes some strong assumptions about this graph. It has to be cycle-free and for each parameter not provided by the user, one unique propose rule has to be applicable. The first assumption can be easily checked statically, whereas the second condition depends on the already derived values of the other parameters, see Fensel (1995b) for more details.

4.3.2 Constraints

A valid elevator configuration must meet constraints resulting from security requirements, i.e. the requirements given by the building and the compatibility of different components. During problem-solving, these constraints are checked for violations. In case of constraint violations, the elevator must be reconfigured until the constraint violations disappear. Constraints are modeled as elements of the class *constraints*. Constraint violations are indicated by instances of the unary relationship *constraint_violated*.

For instance, one of the constraints states that the working load for the compensation cable must not exceed 600 pounds if the car is at the top and the selected model of the platform is "*2.5B*". This is expressed by a specific instance of the class *constraints*. A KARL rule as shown in figure 7 then states that c_{11} is violated if the *platform model* of the elevator is "*2.5B*" and the load for the *compensation_cable* is greater than *600* if the car is at the top. For the VT-domain, 67 constraints have been modeled, each of which is described by at least one rule.

4.3.3 Fixes

If the current elevator configuration violates some constraints, some parameter values have to be determined anew in order to restore consistency. Heuristics indicating how to cope with constraint violations are expressed in so-called *fixes*, i.e. rules defining design modifications. In

X[*selected_platform_model* : "*2.5B*"] ←

Y[*elevator_car*: Z, *car_capacity_range*: C] ∈ *elevator* ∧

Z[*car_platform*: X] ∈ *car* ∧

X[*platform_width*: W, *platform_depth*: D] ∈ *platform* ∧

$C \leq 2500 \wedge W \leq 84 \wedge D \leq 60$.[1]

1. The expression $X[a{:}Y,b{::}\{Z_1,...,Z_n\}] \in C$ has the following interpretation: X is an element of class C, C defines the single-valued attribute a and the set-valued attribute b for its elements. X has the value Y for the attribute a and the set of values $Z_1,...,Z_n$ for the attribute b.

Figure 5. Propose rule for the *platform base*.

by four KARL rules, one of which is shown in figure 5. The variable Y addresses the only available elevator as an element of the class *elevator*. The variable C refers to the *car_capacity_range* of this elevator. The variable Z denotes the car of the elevator which is the only instance of the class *car*. The variable X addresses the platform of the car which is modeled as an instance of the class *platform*. The variables W and D indicate the required width and depth of the platform. The model "*2.5B*" of the platform is chosen if the values bound to the variables C, W, and D satisfy the condition $C \leq 2500 \wedge W \leq 84 \wedge D \leq 60$.

The width and depth of the platform and elevator *car_capacity_range* are requirements supplied by the customer. Conversely, other parameters depend on the platform model. For instance, the platform *weight_factor_AP*, which is used to compute the overall weight of the platform, depends on the chosen model of the platform (*selected_platform_model*) and the width of the platform (*platform_width*). In addition, the *selected_platform_model* depends on *car_capacity_range*, *platform_width*, and *platform_depth*. The binary relationship of one parameter directly depending on another induces a graph with input parameters (i.e, requirements) being the sources of the graph. These dependencies can be derived from the propose rules and are used to incrementally compute the different parameters of an elevator configuration. Dependencies between parameters are expressed by means of instances of the relationship *dependency*. For instance, the fact that the *weight_factor_AP* depends on the *platform_width* is expressed as:

dependency(*attribute*: *weight_factor_AP*, *on*: *platform_width*).[1]

1. Predicates can have named arguments in KARL.

class: *platform_model*

Object ID	*platform_height*
"*2.5B*"	6.625
"*4B*"	6.625
"*6B*"	6.6875

predicate: *platform_order*

current	*next*
"*2.5B*"	"*4B*"
"*4B*"	"*6B*"

Figure 4. Platform models and order.

for the *platform_model* and the values for their attribute *platform_height* are described by elements of the class *platform_model*. The order of these three models is defined by instances of the relationship *platform_order* (see figure 4).

4.3 Rules

Three different kinds of rules for configuring elevators, namely propose rules, constraints, and fixes, are modeled. As these rules are domain-specific, they are defined at the domain layer. Examples of each kind of rules in the VT-domain are given below.

4.3.1 Propose Rules

All configuration parameters, i.e. all attributes expressing properties of the different parts of an elevator, are dependent on the input parameters and on some additional assumptions such as, for example, that the only engine available for moving the door weighs 135 pounds. In KARL, the interdependencies of parameters are expressed by Horn rules extended by stratified negation.

The selected platform model, for instance, is determined by the *car_capacity_range* of the elevator and the width and depth of the platform. The dependencies are the following:

selected_platform_model = "*2.5B*"

if *car_capacity_range* ≤ 2500 and *platform_width* ≤ 84 and *platform_depth* ≤ 60

selected_platform_model = "*4B*"

if *car_capacity_range* > 2500 and *platform_width* ≤ 128 and *platform_depth* ≤ 108

selected_platform_model = "*4B*"

if *car_capacity_range* > 2500 and *platform_width* ≤ 115 and *platform_depth* ≤ 126

selected_platform_model = "*6B*" in all other cases

The four different cases of the platform model's dependency on other parameters are expressed

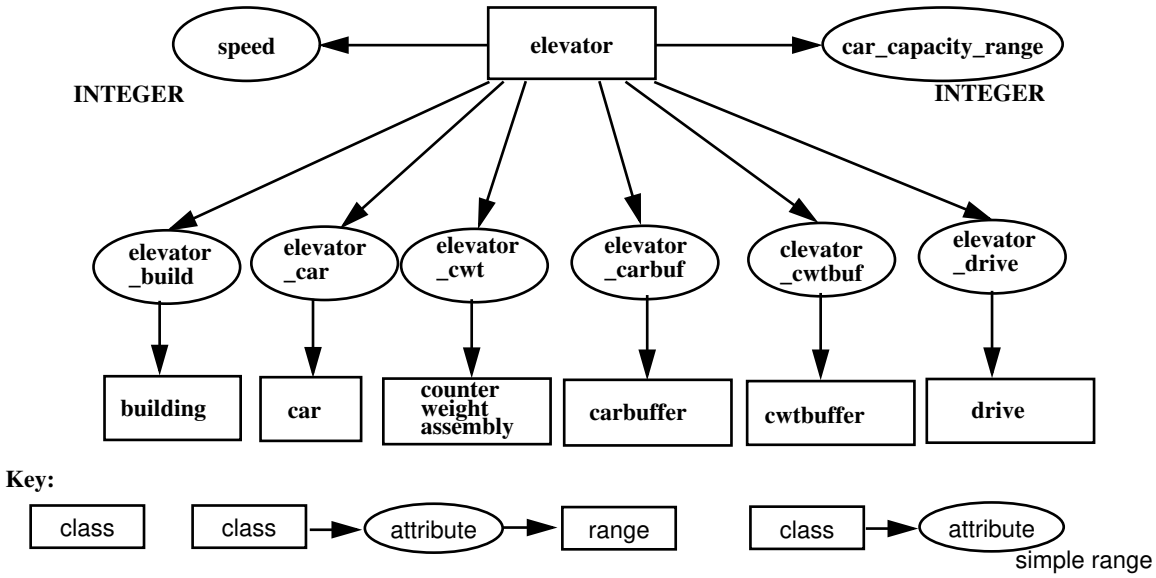


Figure 3. Configuration of an elevator

The top-level object to be configured is an elevator. The particular elevator in question is modeled as an element of the class *elevator*. The parts of an elevator are represented by the attributes *elevator_build, elevator_car, elevator_cwt, elevator_carbuf, elevator_cwtbuf*, and *elevator_drive*. The attributes *car_capacity_range* and *speed* denote additional properties of the elevator, the values of which must be supplied by the customer wanting to configure an elevator. Classes, part-of relationships and attributes of the class *elevator* are described in the graphical notation of KARL as shown in figure 3.

Each class related to the class *elevator* by a part-of relationship can in turn be described by means of subparts and attributes. To that end, we proceed towards elementary classes (elements which are not composed of smaller components). One advantage of such a structured description of a complex system is that it is quite natural to consider a complex system as being made up of components which in turn are made up of more basic components. Furthermore, structuring supports the abstraction principle in a natural way, as it allows for viewing a complex system at different levels of abstraction.

4.2 Factual Knowledge

Knowledge about the facts of a given domain, such as the different available models of the platform, is modeled by facts in KARL. Facts define elements of classes together with values for their attributes or instances of predicates. For instance, the three different models available

Classes are arranged in a specialization/generalization hierarchy via an is-a relationship. The similarity of objects, which are elements of the same class, refers to their structure which is determined by the objects' attributes and parts. These structuring principles are well known from object-oriented data modeling.

KARL provides corresponding language primitives for these two structuring principles: Similar objects are grouped together by means of classes which also describe common attributes and which are arranged in is-a hierarchies. Attributes are inherited according to this relationship. While attributes are shared by all elements of a class, the values of the attributes may differ for each element. Attributes either describe properties of objects such as the *car_capacity_range of an elevator* or express a relationship to other objects. The latter can be used to express part-of relationships between objects. The range of attributes may either be a single value, object or class or a set of values, objects, or classes. In addition, arbitrary relationships between objects may be described by predicates in KARL.

Since an elevator may intuitively be viewed as a hierarchical assembly of components, the relationships between objects expressed by attributes can mostly be interpreted as part-of relationships in the model of the VT domain. In the original problem description , there is no grouping of parameters referring to the same part of the elevator, nor are the parts organized in a part-of hierarchy. The KARL model of the VT-domain is determined by the following modeling decisions:

- Parameters describing properties of the same elevator component are grouped together and constitute the attributes of a class describing this type of object.
- Loads and moments are described as parameters associated to those classes of objects on which they are placed.
- Parameters, which cannot be assigned to a class using the rules above are assigned to classes so that their value can be computed from attributes of only a few other classes. This reduces the complexity of the propose rules.

For reasons of brevity, only parts of the terminological knowledge for the VT-domain will be described in the following.

customer requirements. The *knowledge part* contains the domain-specific knowledge used to solve the problem, i.e. to configure an elevator from the input knowledge. The contents of this part are independent of the particular case to be solved and therefore do not change during problem-solving. In the considered domain, this knowledge consists of the following parts:

- *Terminological knowledge* about elevators and their components, such as buildings, buffers, drives etc. and their attributes and knowledge about their interconnections.
- *Factual knowledge*, for instance the fact that the counterweight always uses a Model 82 frame.
- *Intensional knowledge* includes the rules for proposing new values from the given ones, the constraints that must hold, and the fixes which are used to repair a configuration if some constraints are violated.

The results of the problem-solving process are stored in the *output data part* of the domain layer. This part contains the values for parameters such as *Hoistway_bracket_spacing, Counterweight_guiderail_unit_weight,* etc.

A domain layer must provide *all* knowledge required by the problem-solving method described in section 5. On the one hand it therefore includes method-specific knowledge, such as fixes which are used by the method to repair an intermediate design. On the other hand, the domain knowledge can be represented independently from the ontology of the method, as KARL provides a mapping mechanism which can be used to link the domain terminology to the method-specific ontology.

In the following, some examples of these various knowledge types in the VT domain are presented. A complete description can be found in Sprau (1993). In order to enable comparisons with other ontologies, we will be using the terminology of Gruber and Runkel (1993) whenever possible.

4.1 Terminological Knowledge

For complex systems, different kinds of structuring principles are known. Objects consist of parts and these parts in turn consist of parts again. The part-of hierarchy is broken down to its elementary objects on the bottom level. Furthermore, similar objects are collected in classes.

editors as described in Gappa (1995).

This approach allows for a very flexible process model, c.f. figure 2. The knowledge engineer selects or adapts an appropriate shell for the domain expert who creates and refines the knowledge base. If the expert requires additional features, e.g. because the evaluation of the end-user has revealed some flaws, the shell can be adapted again and so on.

3 Initial Problem Analysis

Although the design of elevator systems is a real life task, the Sisyphus situation is rather unusual. All the early steps of the knowledge engineering cycle had already been performed, the domain knowledge had been described informally quite clearly in Yost (1992) and even formally in Gruber and Runkel (1993), and the problem-solving method already had been presented in Marcus (1988a). Therefore, no real problem analysis activity was necessary.

The following two activities were performed as the initial problem analysis:

- Analysis and formalization of the VT-domain knowledge as described in Yost (1992). The result was a conceptual and formal model of the domain layer in KARL. The domain model is a formalization of Yost (1992) which already contains method-specific knowledge like fixes. A student writing his masters thesis took about six months to achieve this task but the effort included training in KARL and in the corresponding tools.
- Analysis of the propose-and-revise method as described in Marcus (1988a). We selected the propose-and-exchange method, which we had used to solve the previous Sisyphus room-allocation problem, described in Poeck (1991) and Poeck (1992), as a starting point from our library of implemented problem-solving methods. This choice was rather obvious, since we originally developed propose-and-exchange as a special variant of propose-and-revise. The configuration and adaptation of propose-and-revise starting from propose-and-exchange was rather easy and took only a few days.

4 The Domain Layer

The domain layer in KARL consists of three different parts. The *input data part* contains case specific input data for the problem-solving process. In the VT- domain this part contains the

system, etc. Representing such design decisions in the design model narrows the gap between the model of expertise and the implementation of the final system. For instance, the informal and the formal but declarative description of an inference action is supplemented by appropriate data structures and algorithms supporting an efficient computation. The final description is achieved by *implementing the system* in the given hardware and software environment.

CRLM, c.f. Poeck and Gappa (1993) and Poeck (1995), is based on the role-limiting method approach, cf. Marcus (1988b), Poeck and Puppe (1992), Puppe and Gappa (1992) and Puppe (1993). The idea of this approach is to build reusable shells for specific tasks, e.g. classification. Each shell should provide one problem-solving method, a consultation environment and, probably most important, a graphical knowledge acquisition environment allowing domain experts to develop knowledge bases by themselves. These shells obviously have a very limited scope and this disadvantage is to some degree overcome by CRLM.

The additional main idea of CRLM is to represent and implement the problem-solving methods in flexible task structures, c.f. Chandrasekaran and Johnson (1993), which can be reconfigured and customized. These task-structures are built as AND/OR-Trees. For one task, several alternative methods can be specified (OR-links) that either solve the task or decompose it into several subtasks (AND-links). Configuration of a problem-solving method therefore means making decisions for every OR-link.

To be able to provide strong knowledge acquisition support for custom-configured problem-solving methods, the knowledge acquisition environment is generated automatically from a declarative specification of the corresponding knowledge representation and the knowledge

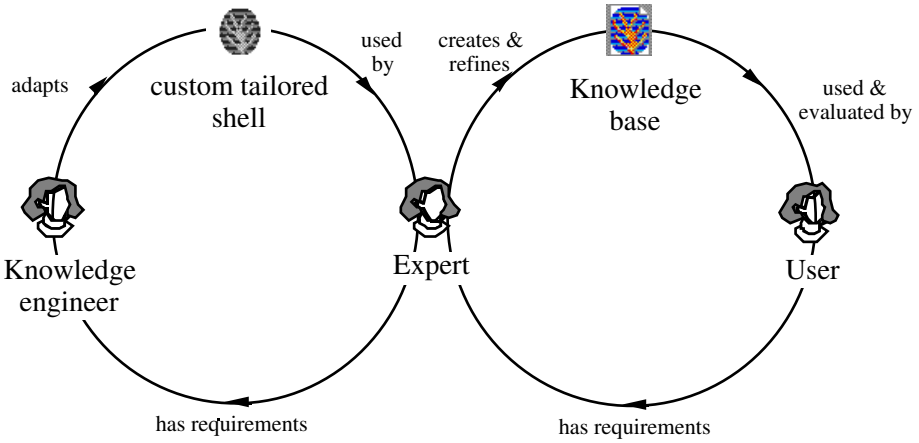


Figure 2. Process model for CRLM

thus may be used for the maintenance of the final system.

The third type of description is accomplished with KARL. Knowledge represented informally or semiformally is formalized during the *knowledge formalization step*. The main benefits of formal descriptions of expertise compared to informal or semiformal representations are the following: The vagueness and ambiguity of natural language descriptions become avoidable, the formalized problem-solving method can be used to guide the further collection of domain knowledge, the formal description may help to get a clearer understanding of single problem-solving steps as well as of complete problem-solving methods and thus supports their reuse; and a formalized specification can be mapped to an operational one, which allows testing to evaluate the knowledge, thus supporting incremental modeling.

Formalization results in a formal and operational description of the model of expertise. Since a KARL specification is based on the structure of the KADS model of expertise, there is a smooth transition from a semiformal to a formal description. The KARL model is constructed by refining the semiformal model of expertise, e.g., by augmenting an informal description of an elementary inference step in the semiformal model by a formal description. Formal descriptions should not replace informal ones, but rather define their meaning precisely and uniquely. On the one hand, natural language is very useful to outline the general idea of an inference since, in a formal description, one often cannot see the wood for the trees. On the other hand, it is very difficult, if not impossible, to define the exact meaning of an inference in a precise and unique manner by natural language only.

The fourth description level is defined by the *design model*, c.f. Landes (1994) and Landes and Studer (1995). The model of expertise finally includes all functional requirements posed on the desired system. For the realization of the final system, additional requirements have to be considered, which are still independent of the system's final implementation. These requirements are non-functional requirements, such as efficiency of the problem-solving method's realization (algorithmic efficiency is independent of the final implementation language), maintainability of the system, persistency of data etc. The design model enriches and refines the model of expertise by taking these issues into account, e.g. by introducing appropriate algorithms and data structures, by taking care of a suitable modularization of the

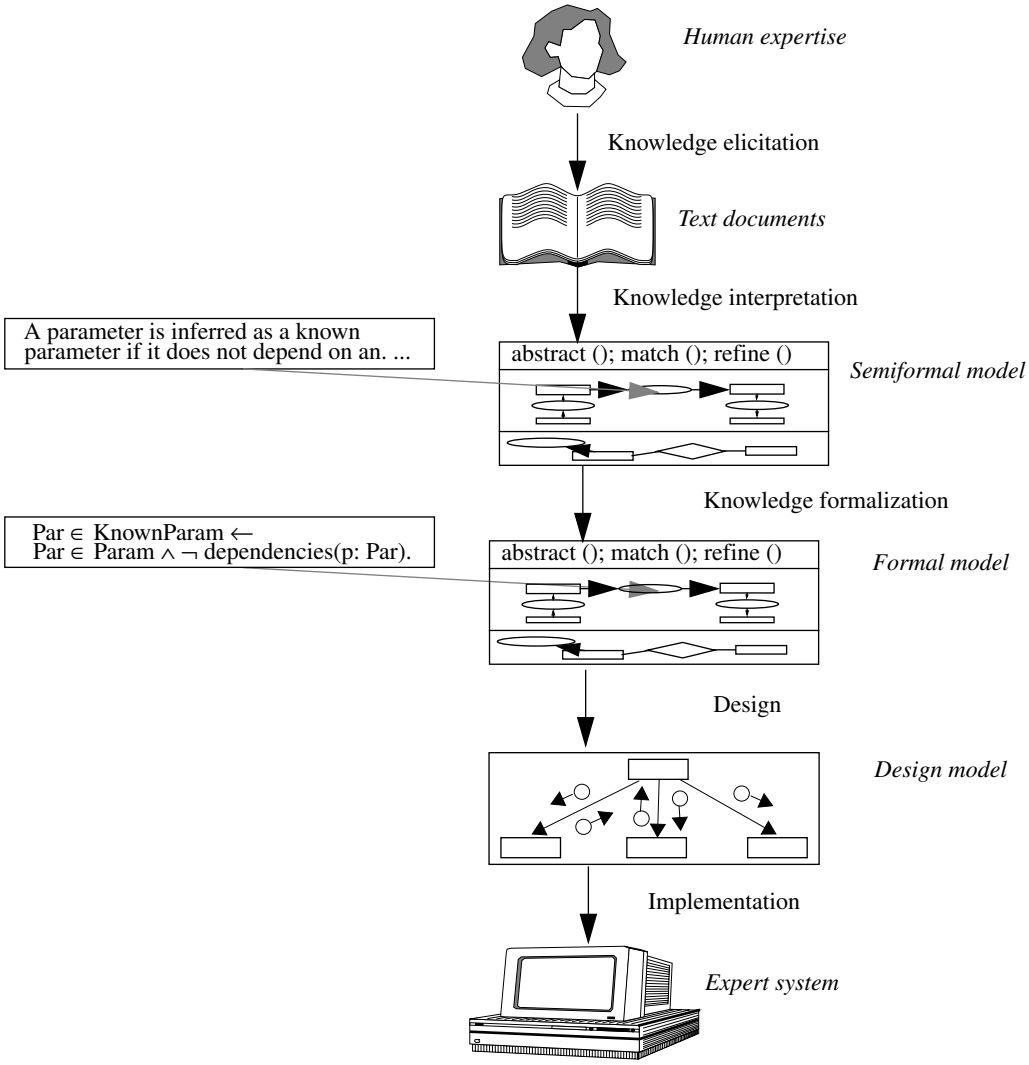


Figure 1. The different description levels of MIKE

informal descriptions are transformed into a *semiformal representation*, the so-called *hyper model*, c.f. Neubert (1994). The hyper model's construction is supported by the tool set MeMoKit, c.f. Neubert (1993). As a result, the knowledge and the task are described along the lines of a model of expertise as it is defined in KADS. The description of knowledge is structured in different layers using appropriate primitives which are also associated with a suitable graphical representation. The semantics of elementary knowledge pieces is still defined in natural language. Such a mediating representation has the following advantages: The structuring process for creating the mediating representation itself provides early feedback for the knowledge engineer and the expert, the semiformal representation of the expertise provides a good basis for communicating with the expert, the contents of the model may be exploited for the explanation facility of the final system, and the model documents modeling decisions and

parts: Section four describes the model of the domain knowledge, while section five focuses on the problem-solving method which is a variant of *propose-and-revise* Marcus (1988a). Section six describes the implemented system and section seven supplies a sample trace. The paper concludes with an evaluation and discussion of the achieved solution and, especially, the advantages of the combination of both approaches.

2 Knowledge Modeling Approaches

In the following, we will shortly sketch the two different approaches we have combined in this experiment. Both approaches are *model-based* in the sense that they *explicitly distinguish different types of knowledge* and use *generic problem-solving methods as the behavior model* of an expert system. In spite of their similarities, the underlying principles and points of interest differ significantly in the two approaches. A detailed comparison of both approaches can be found in Fensel and Poeck (1994).

MIKE is strongly influenced by the results of the KADS-I and CommonKADS projects, c.f. Wielinga, Schreiber and Breuker (1992) and Schreiber, Wielinga, Akkermans, van de Velde and de Hoog (1994), and by work in software engineering and information system design, cf. Angele, Fensel and Studer (1990). It is based on the distinction of different phases in the software development process such as, e.g., analysis, design, and implementation. An important means of MIKE is the *formal and operational knowledge specification language KARL*, cf. Angele at al. (1994) and Fensel (1995a), which allows a precise description of a model of expertise resulting from the analysis phase.

MIKE assumes that, during modeling expertise, *a large gap has to be bridged* between informal descriptions of the expertise gained from the expert using knowledge acquisition methods and the final realization of the expert system. Decomposing this gap into smaller ones reduces the complexity of the whole modeling process, since in every step particular aspects may be considered independently of other aspects. MIKE provides five different description levels of a task and the required knowledge (see figure 1).

First, knowledge and task are described in *natural language documents*. These documents may result from interviews or observations or can already exist as manuals, books, etc. Second, these

1 Introduction

The paper presents a solution for the Sisyphus elevator-design problem based on the combination of two quite distinct approaches to model-based knowledge acquisition. A formal description of the task and the required knowledge using the knowledge specification language KARL, c.f. Angele, Fensel and Studer (1994) and Fensel (1995a), was combined with an implementation by a configurable role-limiting method CRLM, c.f. Poeck and Gappa (1993) and Poeck (1995). KARL was developed in the MIKE project *Model-based and Incremental Knowledge Engineering*, cf. Angele, Fensel, Landes, Neubert and Studer (1993), and allows a formal and operational specification of knowledge-based systems. CRLM *Configurable Role-Limiting Method Approach* emerged from experiences with other role limiting method shells (RLM) like D3/CLASSIKA, c.f. Puppe and Gappa (1992) and COKE, c.f. Poeck and Puppe (1992), over the last years. CRLM tries to preserve the advantages of RLMs such as strong knowledge acquisition support and rapid prototyping while extending their scope by being more adaptable and therefore less brittle.

Although approaches based on specification languages like KARL, on the one hand, and role-limiting methods, on the other hand, are often discussed as contradictory in literature, we experienced that both approaches supplement each other very well. Because both approaches emphasize different aspects in the development process of a knowledge-based system, their combination emhances the power of the achieved results. Fensel, Eriksson, Musen and Studer (1993) already showed how an implementation of the board-game method, i.e. a role-limiting method, can be combined with a semiformal and formal description using KARL.

In this special experiment we started with the formal modeling of the domain knowledge base, configured a corresponding propose-and-revise specific shell, and finally did a formal reverse engineering of the implemented problem solver. Only practical and no methodological issues were the reason for this ordering of tasks.

In the following, we will first briefly sketch the two different philosophies on which the approaches are based. Section three then outlines the activities we performed in the initial problem analysis and how much effort was spent on these activities. Sections four and five describe the developed expert system's model of expertise. The specification is divided into two

Combining KARL and CRLM for Designing Vertical Transportation Systems

In International Journal of Human-Computer Studies (IJHCS), 44(3-4), 1996.

Karsten Poeck (1), Dieter Fensel (2), Dieter Landes (3), and Jürgen Angele (4)

(1) Lehrstuhl für Informatik VI, University of Würzburg, Allesgrundweg 12, 97218 Gerbrunn, Germany

(2) Department SWI, University of Amsterdam, Roeterstraat 15, 1018 WB Amsterdam, The Netherlands

(3) Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, 76128 Karlsruhe, Germany

(4) Fachhochschule Braunschweig-Wolfenbüttel, 38302 Wolfenbüttel, Germany

This paper describes a solution to the Sisyphus-II elevator-design problem by combining the formal specification language KARL and the configurable role-limiting shell approach. A knowledge-based system configuring elevator systems is specified and implemented. First, the knowledge is described in a graphical and *semiformal* manner influenced by the KADS models of expertise. A formal description is then gained by supplementing the semiformal description with *formal specifications* which add a new level of precision and uniqueness. Finally, a generic shell for propose-and-revise systems is designed and *implemented* as the realization of the final system. This shell was derived by adapting the shellbox COKE, also used for the previous Sisyphus office-assignment problem. As a result of this integration, we get a description of the knowledge-based system at different levels corresponding to the different activities of its development process.