Reusable Ontologies, Knowledge-Acquisition Tools, and Performance Systems: PROTÉGÉ-II Solutions to Sisyphus-2

Thomas E. Rothenfluh, John H. Gennari, Henrik Eriksson,
Angel R. Puerta, Samson W. Tu, Mark A. Musen

Medical Computer Science Group
Knowledge Systems Laboratory
Stanford University School of Medicine
Stanford, California 94305-5479
U.S.A.

Internet address: protege-staff@camis.stanford.edu

Abstract: This paper describes how we applied the PROTÉGÉ-II architecture to build a knowledge-based system that configures elevators. The elevator-configuration task was solved originally with a system that employed the propose-and-revise problem-solving method (VT; Marcus, Stout & McDermott, 1988). A variant of this task, here named the Sisyphus-2 problem, is used by the knowledge-acquisition community for comparative studies. PROTÉGÉ-II is a knowledge-engineering environment that focuses on the use of reusable ontologies and problem-solving methods to generate task-specific knowledge-acquisition tools and executable problem solvers. The main goal of this paper is to describe in detail how we used PROTÉGÉ-II to model the elevator-configuration task. This description provides a starting point for comparison with other frameworks that use abstract problem-solving methods. Starting from a detailed description of the elevator-configuration knowledge (Yost, 1992), we analyzed the domain knowledge and developed a general, reusable domain ontology. We selected, from PROTÉGÉ-II's library of preexisting methods, a propose-and-revise method based on chronological backtracking. We then configured this method to solve the elevator-configuration task in a knowledge-based system named ELVIS. We entered domain-specific knowledge about elevator configuration into the knowledge base with the help of a task-specific knowledge-acquisition tool that was generated from the ontologies. After we constructed mapping relations to connect the domain and method ontologies, PROTÉGÉ-II generated the executable problem solver. We have found that the development of ELVIS has provided a valuable test case for evaluating PROTÉGÉ-II's suite of system-building tools.

1 PROTÉGÉ-II AND SISYPHUS-2

To evaluate a general architecture for software development, developers must test the architecture with real-world tasks. The best way to validate the strengths and to discover the weaknesses of an architecture is to apply the ideas and tools of the architecture to a task that is similar in size and complexity to tasks found in the real world. The PROTÉGÉ-II architecture is a set of tools and a methodology for developing knowledge-based problem-solving systems. The Sisyphus-2 problem is a large-scale task of configuring elevator systems; it is a variant of the problem solved by the VT system (Marcus, Stout & McDermott, 1988). This paper describes how we used the PROTÉGÉ-II architecture to solve the Sisyphus-2 problem.

The knowledge-acquisition research community selected the Sisyphus-2 task as a benchmark for comparing knowledge-modeling approaches, problem-solving methods, and reusability of knowledge structures. This elevator-configuration task consists of selecting appropriate components and dimensions to configure an elevator system according to user specifications and safety constraints. Our knowledge base for this task is primarily defined by the Sisyphus-2 document (Yost, 1992) and by a proposed ontology for configuration

Figure 1: The table of contents of the Sisyphus-2 document gives an overview of the document that describes the elevator-configuration problem. Sections 1 and 2 provide an introduction to the general concepts of elevator configuration; Sections 3 and 8 define the input and output of the task; Sections 4 through 6 list all the domain components and dimensions, as well as the formulae to calculate them; Section 7 explains which constraints exist and how they can be fixed; and Section 9 gives the input and output values of relevant design parameters for one test case.

design and the Sisyphus-2 task.[1] To provide an idea of the problem description, we list in Figure 1 the section headings of the Sisyphus-2 document (Yost, 1992). This document is pivotal to our work with Sisyphus-2 because it provided all the essential information, concrete data, and well-documented formulae in this complex domain. A secondary source of information about the elevator-configuration task, associated tools, systems, and problems is the literature about SALT, a knowledge-acquisition system developed for the original VT task (Marcus & McDermott, 1989).

A core idea for the success of Sisyphus-2 is the use of *ontologies* that describe terminology and knowledge appropriate for the elevator-configuration task. Such ontologies should facilitate the comparison of problem-solving methods developed in different theoretical frameworks (for related issues about sharing and reuse, see the ARPA Knowledge-Sharing Effort—e.g., Neches, Fikes, Finin, Gruber, Patil, Senator, & Swartout, 1991). In PROTÉGÉ-II, we use the word *ontology* in the same sense as in Ontolingua (Gruber, 1993).

PROTÉGÉ-II is a knowledge-engineering environment that enables developers to define knowledge-acquisition tools and knowledge systems by reusing problem-solving methods and domain ontologies. The PROTÉGÉ-II architecture emphasizes the automatic generation of knowledge-acquisition tools and performance systems from declarative, domain-oriented knowledge structures. The original description of SALT's goals is almost identical to a description of our goals:

> SALT is a program that acquires knowledge from an expert and generates a domain-specific knowledge base compiled into rules. SALT then combines this compiled knowledge base with a problem-solving shell to create an expert system. SALT maintains a permanent, declarative store of the knowledge base which is updated during interviews with the domain expert and which is the input to the compiler/rule generator. It is this intermediate language which represents knowledge by function. (Marcus & McDermott, 1989, p. 3)

[1] The text document as well as the ontologies and the knowledge bases are available as computer files through anonymous ftp from ksl.stanford.edu.

The knowledge-acquisition tools generated by PROTÉGÉ-II are *task-specific*. This characteristic is a critical difference between PROTÉGÉ-II and systems such as SALT; SALT is based on the propose-and-revise method and takes a domain-independent approach to acquiring knowledge. In contrast, PROTÉGÉ-II's architecture is method independent, but builds task-specific tools for knowledge acquisition. We believe that knowledge-acquisition tools will be usable by domain experts only if those tools use the domain-specific terms and concepts that are familiar to the experts.

Sisyphus-2 adds a slightly different focus by stressing the *reusability* of the knowledge used to solve the task. Contributors to Sisyphus-2 were strongly encouraged to use a set of ontologies and knowledge bases distributed with the call for participation. However, much of this knowledge is actually the output of what we would consider to be part of PROTÉGÉ-II's domain-modeling and knowledge-acquisition phases. Thus, to demonstrate our methodology and development tools, we must begin earlier in the task-modeling process, beginning with a domain expert's description of the task—in this case, the Sisyphus-2 document (Yost, 1992).

We used PROTÉGÉ-II to develop a knowledge-based system, ELVIS (ELeVator configuration In Sisyphus), that solves the Sisyphus-2 problem. In addition to a run-time system that solves the task, ELVIS includes various intermediate knowledge structures and editing tools as provided or required by the PROTÉGÉ-II architecture. Although many aspects of PROTÉGÉ-II have been documented elsewhere (for a recent overview, see Puerta, Egar, Tu, & Musen, 1992), we provide, in Section 2, an introduction to PROTÉGÉ-II's main concepts and building blocks—the various ontologies, tools, and development steps—that are employed to produce a knowledge system. In Section 3, we provide a detailed analysis of the knowledge contained in the Sisyphus-2 document. This analysis is the basis for the concrete structure of the domain and method ontologies. Section 4 describes the domain ontology and how PROTÉGÉ-II uses this ontology to generate a knowledge-acquisition tool for the Sisyphus-2 problem. Section 5 presents our configuration of the propose-and-revise problem-solving method, as well as the final run-time solution. Finally, in Section 6, we discuss the strengths and weaknesses of the PROTÉGÉ-II architecture that we discovered while modeling the Sisyphus-2 problem.

2 APPROACH TO KNOWLEDGE MODELING

The PROTÉGÉ-II architecture emphasizes the importance of domain-specific knowledge-acquisition tools, and provides a system-development environment and a development methodology to generate such knowledge-acquisition tools automatically. The knowledge acquired with these tools is then used by the selected and configured problem-solving method to generate a run-time system that solves the given problem.

PROTÉGÉ-II is more than just a methodology; it is an operational system implemented on the NeXT platform. The system consists of a suite of tools that allows developers to define ontologies, to design knowledge-acquisition interfaces, and to generate run-time systems. The main reasoning engine and knowledge-representation format for domain-knowledge instances is provided by the CLIPS production language, a system implemented in C and available on many different computer platforms.[2] The language used to define ontologies in PROTÉGÉ-II, MODEL, is an extension of CLIPS's object-oriented language (for more details, see Walther, Eriksson & Musen, 1992, and Gennari, 1993). The definition of ontologies in the PROTÉGÉ-II architecture is supported by MAÎTRE (see Section 4.3), a special frame-based editor for building and maintaining MODEL structures. These MODEL ontologies are then used by DASH, a subsystem that generates knowledge-acquisition interfaces (see Eriksson & Musen, 1993, and Section 4.4). The development environment and the knowledge-acquisition tools generated with PROTÉGÉ-II depend on the NeXTStep operating system and its user-interface software.

In Figure 2, we summarize the interplay of the different building blocks of PROTÉGÉ-II. The construction of a *knowledge-based system* (sometimes also called a *problem solver*, or a *performance system*) starts from a declarative description of the domain and of the problem-solving method: the domain and method ontologies. The developer merges these ontologies to produce an *application ontology* that is both domain- and method-specific. Next, DASH builds a knowledge-acquisition tool from the application ontology. The domain expert can then enter knowledge via this tool. To generate a run-time system, PROTÉGÉ-II interprets

[2] For details about the CLIPS language and its functionality, see the CLIPS Reference Manual, available from the Software Technology Branch, Lyndon B. Johnson Space Center, NASA, Houston, TX.

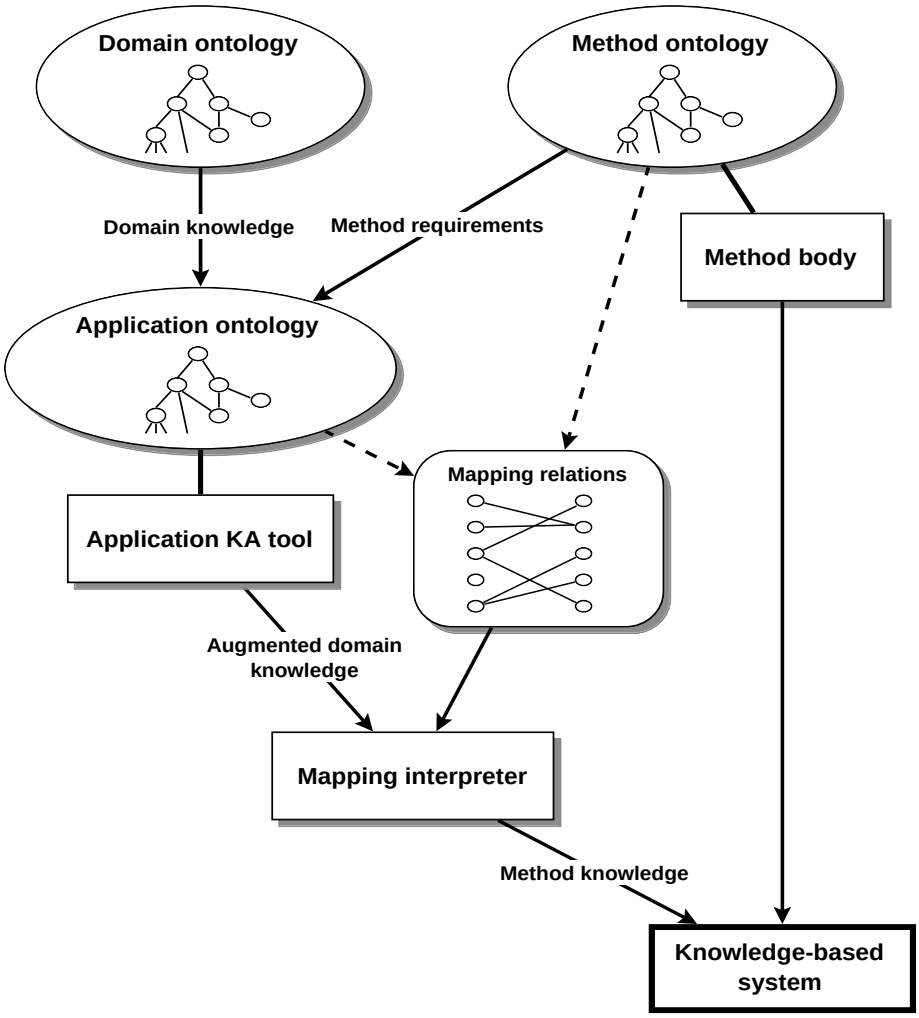

Figure 2: Overview of the building blocks and system-development steps. Reusable domain and method ontologies are combined into a task-dependent application ontology. The knowledge instances are acquired with the generated knowledge-acquisition tool, and then are transformed into an executable knowledge-based system by the mapping interpreter. Finally, a task instance is solved by acquiring runtime input (not shown here) and executing the code supplied by the selected and configured method.

the knowledge base created by the expert as input to the problem-solving method; the system does this by using a set of declarative *mapping relations* and a *mapping interpreter.* In Section 2.1 through Section 2.7, we describe, in more detail, these various building blocks and components of the PROTÉGÉ-II architecture.

2.1 Domain Ontology

Since both the representation of domain knowledge and the adaptation of general problem-solving methods are difficult tasks, an important product of a knowledge analysis is the definition of a *domain ontology.* Such a domain ontology declaratively defines a language that is used to express all kinds of domain knowledge. A domain ontology in PROTÉGÉ-II is a framework that defines the knowledge structures (classes, relations, functions, and object constants) that a domain expert will fill in with domain-knowledge instances. For the Sisyphus-2 problem class, the domain ontology is composed of the definitions of all parts, dimensions, design parameters, and features of an elevator system, as well as of their internal relationships, such as part-of relationships, functional links between components, and constraints.

The basic structure of an ontology modeled in PROTÉGÉ-II is a straightforward IS-A hierarchy. This type of hierarchy allows for different levels of abstraction and representational choices for a given domain. For example, in the Sisyphus-2 domain, we could choose to model just a single level of components, listing each

domain term as a separate class: elevator door, car, car telephone, hoistcable, and so on. Alternatively, we could provide a set of intermediate and high-level abstractions, such as a car assembly, a door system, the cables, or the set of all input parameters. Such abstractions make it easier to work with complex ontologies by taking advantage of inheritance mechanisms, and preserve natural groupings of domain knowledge. In addition, abstract-level classes may be more easily reused across different tasks.

An important feature of a domain ontology is versatility: it should allow for different models and views of the domain. For example, a single domain may be modeled differently depending on the task to be solved. Even with the same task, different domain experts may have different views of the domain. If we can represent these different views of the task, we can build user-specific knowledge-acquisition tools. As well as providing versatility, we must also maintain consistency across different ontologies; thus, all concepts must be defined axiomatically. For example, if a domain ontology includes logical expressions, that ontology must include a declarative definition of legal expressions that lists all the allowable logical terms and defines their relations to other concepts in the ontology.

Despite the relative leeway in defining an ontology, the entities defined in the domain ontology are not to be confused with method-related concepts—on the contrary, it is desirable to keep such procedural knowledge out of the domain ontology. For example, the notion *constraint* is used heavily in the Sisyphus-2 document, presumably originating from general design notions or from the experts' natural-language explanations. However, the use of this *constraint* notion does not necessarily require the use of a constraint-based method to solve the problem. Similarly, the concept *variable* in the description of the elevator domain should not be confused with that of a *state-variable* in a search algorithm. Such correspondences—even if they are simple one-to-one mappings—are established only through explicit *mapping relations* (Gennari, Tu, Rothenfluh & Musen, 1993).

A useful domain ontology should be able to take advantage of specialized knowledge types present in the application domain. For example, a developer may want to specialize the concept of a domain variable to the concept of a design parameter that includes more knowledge about expected values. This knowledge can provide better support for knowledge acquisition and for knowledge-base maintenance or verification. In addition to a thorough knowledge analysis that tries to abstract recurrent patterns from the domain, an important step in the design of a domain ontology is to think about how knowledge will be acquired from domain experts. We illustrate this point in Section 4.4 in more detail.

Finally, the definition of any ontology—which may easily consist of hundreds of objects, classes, and slots—should be supported by a specialized knowledge editor. The MAÎTRE subsystem (Gennari, 1993) allows easy navigation and context-sensitive editing support for ontologies. MAÎTRE produces a MODEL description of the ontology that can be used by other PROTÉGÉ-II modules, such as the DASH module and the run-time system. In Section 4.3, we illustrate MAÎTRE's functionality with our version of the domain ontology.

2.2 Method Ontology

In the PROTÉGÉ-II architecture, all knowledge about method-related concepts is stored in a *method ontology*. This method ontology describes domain-independent method concepts, in contrast to the domain ontology, which describes method-independent domain concepts. Method ontologies are abstract descriptions of the inputs and outputs of the problem-solving method. For example, Figure 3 shows our method ontology for the propose-and-revise problem-solving method. This ontology defines domain-independent concepts such as state-variables, constraints, and fixes. To apply this method to a task, developers must instantiate these concepts with information from the domain. Thus, the method ontology describes the knowledge requirements and the knowledge roles of a given problem-solving method.

We believe that creating a declarative method ontology is an important step toward method reuse. Developers can reuse methods only if they are described in *abstract* terms: as an approach to solving a class of problems, rather than as a specific technique to solve a particular task. Constructing an accompanying method ontology automatically forces the developer to think about the method in more abstract terms. Such ontologies may also clarify differences between similar methods, and may help the developer to recognize when an existing method can be reused with a new domain. For these reasons, the method ontology serves as an important index for method selection.

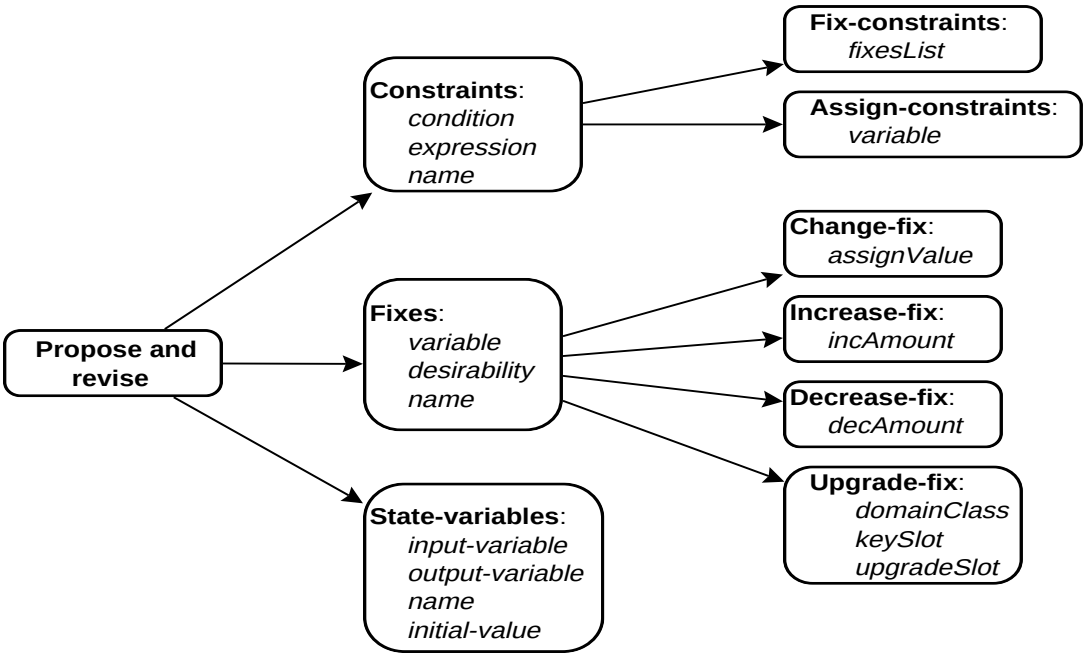

Figure 3: The method ontology for the propose-and-revise method. The method expects particular information that has to be provided by the application ontology.

2.3 Method Selection and Method Configuration

A pre-existing problem-solving method may not be exactly appropriate for a given task in a given domain. Problem-solving methods for specific tasks are defined in a two-step process (Eriksson, Musen, Shahar, Puerta & Tu, 1992):

1. *Method selection*, where the developer selects a generalized problem-solving method from a library of available methods; if many method candidates are available, this step may include *method evaluation* to determine the most appropriate method.
2. *Method configuration*, where the developer adapts the abstract problem-solving method to the task at hand.

The *method-selection* process is constrained by many decisions; usually, there is no simple, direct link between tasks and methods. Often, external decision factors are critical to the choice of method. These factors include the availability of method expertise, access to alternative methods, project resources, and the knowledge engineers' expertise about a given task. In some cases, these external factors may mean that method selection occurs before knowledge analysis, thereby influencing the design of the domain ontology. This scenario can often lead to an inefficient system or to an awkward domain knowledge representation.

Knowledge about the given domain (available in the domain ontology) may lead the developer to make changes in a generic method through *method configuration*. A generic problem-solving method should be decomposable: it should be divisible into some sequence of subtasks, which in turn are solved by other (smaller-grained) methods. We call a method that is not decomposable a *mechanism*; mechanisms can be viewed by the developer as black boxes. Changing or configuring the method to fit a specific task often includes replacing a mechanism with one that is more appropriate or efficient for the given task and domain. (For a thorough discussion of PROTÉGÉ-II's perspective on mechanisms and methods, see Puerta, Tu & Musen, 1993.)

These changes, however, should be made in a principled way and should not include loopholes that allow unconstrained ad hoc programming. For example, domain-knowledge analysis may reveal that there is a static dependency structure among variables. The system developer may then create a method specialization, replacing the mechanism that recalculates the value of all parameters with a mechanism that recalculates only those parameters that depend on changed values. This specialization process is different from ad

hoc programming, because the method ontology and the decomposition of the method into mechanisms explicitly defines and constrains where such changes can be made and what knowledge is involved in a given alteration.

2.4 Application Ontology

In many cases, the knowledge requirements in the selected method ontology will suggest an adaptation of the domain ontology. For example, developers may modify the domain ontology by acquiring and adding method-dependent knowledge, or by transforming some of the knowledge into a representation that is better suited for a particular method. By adapting the domain ontology, the developer builds an *application ontology*: a middle ground between the method and the domain ontologies. This ontology is distinct from the domain ontology because of reuse considerations: Although domain ontologies usually originate from reusable, static sources—such as technical handbooks, databases, and terminology knowledge bases—the intended use of the application ontology is specific to a particular task, and may not be as reusable as other types of ontologies.

For example, an inventory program, a cost-estimation program, a computer-aided drawing program, and an integrated manufacturing system may access the same global domain ontology. However, knowledge about how to fix violations of constraints in a configuration process is not important to any of these programs. Thus, the notion of *violations* and *constraints* are defined in our application ontology, but not necessarily in the general-purpose domain ontology for elevators. Construction of the application ontology is a non-automated task: The developer must compare the domain and method ontologies and construct the application ontology by adding domain-specific distinctions required by the method and removing unneccessary domain concepts. This complex modification process is one reason that it is important to have a single editing tool, MAÎTRE, for working with any ontology. Although we generally view application ontologies as less reusable, over time, an abstraction process from several different tasks may lead to the migration of reusable classes and terms from an application ontology to a reusable domain ontology.

2.5 Mapping Relations

The final connection between the problem-solving method's concepts and the basic terms defined in the application ontology is made by a set of *mapping relations*. For each knowledge role defined in the method ontology, corresponding role fillers have to be established in the application ontology. For example, the functional role of constraint knowledge in a constraint-based method could be embodied in the application ontology as the specification of *ranges* for the values of a design parameter. These ranges would need to be transformed into constraint expressions by mapping relations. Furthermore, such a constraint might be represented as a *rule* in a knowledge base, thus also requiring the identification of the single (constrained) variable for the construction of a dependency network of parameters. Other examples are the transformation of knowledge that is provided in the application ontology in *tabular* form, to a set of rules (one for each table cell) that can be used by the selected method.

Instead of having explicit mapping relations, we could modify the problem-solving method directly to reflect the structure of the domain ontology. However, although this approach avoids the overhead of an additional task-specific ontology (the application ontology), it also makes the problem-solving method and its ontology less reusable. In general, this problem raises questions such as: Should the domain ontology or the problem-solving method be the primary target for reuse? Will the new application ontology be easy to maintain? Is the adaptation of the knowledge base computationally tractable? We advocate the use of explicit mapping relations because they lower reuse cost and keep the various adaptations transparent (Gennari et al., 1993).

2.6 Application-Domain Instances

If the application ontology is in place and the method is configured appropriately, *application-domain instances* provide the actual content of the application problem knowledge. These instances are acquired from domain experts through the use of specialized, domain-specific knowledge-acquisition tools, and define all the concrete knowledge that does not change across specific problems. In PROTÉGÉ-II, knowledge-acquisition tools are generated from the application ontology by the DASH subsystem (see Eriksson & Musen, 1993, and Section 4.4). The design and layout of the knowledge-acquisition tool is domain specific and should enable the domain experts to enter their knowledge and data in a natural way.[3]

In the Sisyphus-2 problem, the configuration knowledge is separated into input specifications (Section 3 of Yost, 1992) and into static domain knowledge (Sections 4 through 6 of Yost, 1992). Whereas the definition of the *domain ontology* depends heavily on knowledge analysis, the acquisition of domain-knowledge instances for the Sisyphus-2 problem is mostly a matter of translating and entering the information provided in the written document. The decisions about what knowledge should be included in the ontologies and what should be supplied either with the knowledge-acquisition tool or as run-time input are dependent on the system's design and application scenarios.

Unfortunately, the structure of the domain knowledge base provided as part of the Sisyphus-2 distribution is almost useless, because it is fully entrenched in a constraint-satisfaction point of view.[4] For example, only a flat list of state variables with cryptic names is provided, and there is almost no structure in the domain knowledge (except for components). Although such a representation might be efficient for certain implementations, it is certainly not a good example of sharable and reusable knowledge. For example, we do not know how and when to acquire all the definitions and values for the state variables, we do not know why they are in the knowledge base and what role they played in the modeling process, we do not know how a domain expert may interpret or use these variables, and we certainly do not know how we should construct a meaningful knowledge-acquisition tool, let alone a task-specific tool to maintain the knowledge base. However, the computer files were useful as a starting point for our work in Sisyphus-2. We were able to translate the knowledge base into a format suitable for PROTÉGÉ-II, and were able to test parts of our method without needing to type in all the domain knowledge. We also used the knowledge base to check for inconsistencies with the textual Sisyphus-2 problem description and to resolve ambiguities in our interpretation.

2.7 Run-Time System

The input data that define different problem instances are supplied to the system as *run-time knowledge*. In addition to input data for a particular problem, this type of dynamically changing knowledge also includes answers to any questions that the knowledge system might ask during its search for a feasible solution. Default values or initial assumptions may be supplied as run-time knowledge or as part of the domain ontology, depending on the anticipated reuse. For example, it makes sense to add special knowledge to the ontology if certain defaults are used across many problem instances. This knowledge may be a general rule, such as to always start with the cheapest elevator component, or it may be a set of default choices that was elicited from the domain expert by the knowledge-acquisition tool.

Regardless of how this knowledge is provided, the ontology should mark this type of information explicitly as input or default knowledge. This explicit representation will provide easy access to this kind of knowledge and will allow changes according to special problem needs. For example, the selected method might automatically check that all input values are present and that all default values are used as a first approximation. Due to the scenario choices made for Sisyphus-2, the current implementation of the run-time system receives all its input parameter values and some default assumptions from a simple text file.[5]

3 INITIAL PROBLEM-SOLVING APPROACH

In this section, we give an overview of PROTÉGÉ-II's approach to structuring a domain problem by discussing the phases encountered during the development of a solution and by outlining the initial task-analysis phase. The details of developing, using, and applying the domain and method ontologies for the Sisyphus-2 problem will be described in Sections 4 and 5.

---

[3] Because of the use of domain-specific terms and of static data entry, this approach may make it difficult for the domain expert to perceive the intended procedural use of the knowledge that has to be supplied.

[4] The knowledge base, as well as the Ontolingua ontologies were provided by Gruber and Runkel (1993). The knowledge base itself is a large computer file (192KB text) that includes the definition of components and constraints for the Sisyphus-2 problem.

[5] This simple file input is based on the idea that the input to the knowledge system can originate from interactive questioning with special run-time tools, from a database, or from other preprocessing modules. Since the Sisyphus-2 scenario calls for a batch type solution with no run-time interaction, there is no need for an elaborate end-user interface other than a routine that acquires all input information and stores the data appropriately. The choice to use a simple text file is also consistent with the Ontolingua representation of the Sisyphus-2 task proposed by Gruber and Runkel (1993). (A simple translation of this file can be used as input to our system.)

3.1 Development Phases

In tackling the Sisyphus-2 project, we were confronted with several conceptual problems and system-development issues. First, many applications of PROTÉGÉ-II and its predecessors are rooted in the medical domain; for example, one of the best explored problem-solving methods originates from the management task for clinical-trial protocols (Tu, Shahar, Dawes, Winkles, Puerta & Musen, 1992). Working with configuration tasks in an engineering domain requires flexibility both from our architecture and from its users. Since PROTÉGÉ-II is developed as a domain-independent and method-neutral architecture, the use of a new domain and a new method is a good test for our approach. Other approaches, such as generic tasks (Chandrasekaran, 1987) or role-limiting methods (McDermott, 1988), advocate the development of task- or method-specific knowledge-acquisition tools and problem-solving methods. The feasibility and success of these frameworks depends on the availability of appropriate methods. If a task (or the appropriate method) is not defined in the respective framework, a new task (or method) has to be added. Since our effort included method development and configuration, it should be compared to the effort needed in other frameworks to add such a method or task.

Furthermore, the development of a system according to the requirements specified in the *Sisyphus-2 Call for Contributions* had to be intertwined with our priorities and work on the general PROTÉGÉ-II architecture. Therefore, our approach to the Sisyphus-2 problem was somewhat different than what it might have been had we undertaken the project in isolation from our research efforts. The following are the major development phases that we used in constructing the current solution for the Sisyphus-2 problem. These steps are described in full detail in the remainder of this paper.

1. We achieved *familiarization with the domain* through individual study of the Sisyphus-2 document (Yost, 1992) and group discussions.
2. We performed a thorough *analysis of domain knowledge* on the distributed document. This step included a major *rewriting* of the document and the construction of an executable calculation model (see Section 4.1).
3. We constructed a *prototype domain ontology* as well as a *prototype run-time system*, without going through the full PROTÉGÉ-II development cycle (i.e., no knowledge-acquisition tool was generated to acquire and enter all the domain knowledge). Sections 4 and 5 describe our work on the domain and method ontologies.
4. We made *ontology revisions* several times. Most revisions were either to accommodate new features of PROTÉGÉ-II (such as changes in the knowledge-acquisition tool generation process) or to reflect conceptual changes that occurred with respect to our work on mapping relations (Gennari et al., 1993).
5. We developed a *production version* of a solution, ELVIS, with the current implementation of the PROTÉGÉ-II system. This production version includes all the architecture's building blocks as outlined in Section 2, and finds a solution for the test case provided in Section 9 of Yost (1992).

In Sections 3.2 and 3.3, we shall present some general issues and problems that we encountered when applying the PROTÉGÉ-II framework to the Sisyphus-2 problem. The development of the actual ELVIS system is discussed and illustrated in Sections 4 and 5.

3.2 Initial Task Analysis

PROTÉGÉ-II views a task as an activity, or an abstraction of an activity, that is performed in the real world. A *task instance* is a particular instantiation of such a task that can be characterized by the types of its input and output. Further decompositions of the top-level task become apparent only after a particular *method* is chosen, because only methods specify the knowledge required to get from the input to the desired output.

The domain problem, as presented in the Sisyphus-2 project documentation (Yost, 1992) and in the *Sisyphus-2 Call for Contributions*, can thus be analyzed at the top-level as a task—namely, elevator configuration. The input to this task consists of a list of configuration parameters, some of which have default values. The output is another, partially overlapping list of parameters (Sections 3 and 8 of Yost, 1992) that need to have defined values after the task is completed. The set of input parameters can be subdivided further into an *assumption set* and a *user-input set*, where only the latter is supposed to vary from task instance to task instance.

Additional task information is provided as a collection of "formulae," "rules," and "constraints" (Sections 4 through 7 of Yost, 1992) that defines a precise set of mathematical relations among the values of the configuration parameters. A subset of these rules defines a *calculation model* whose execution exactly defines the

values of all output parameters for a given set of input variables. Furthermore, another set of rules, called *constraints* (Section 7 of Yost, 1992), includes a list of *constraint fixes*. This list is partially ordered along an ordinal scale of *desirability*, and is designed to correct the problem identified by the associated rule; in other words, a constraint fix should alleviate a constraint violation.

The full Sisyphus-2 task, then, consists of executing the calculation model with a given set of input values, and adjusting the parameters until there are no constraint violations. For most input-value combinations (also called customer specifications), some constraints will be violated after the calculation of the values of all output parameters. To correct this situation, the system must take appropriate actions, which may require the adjustment of "free" parameters in the model.[6]

As far as PROTÉGÉ-II is concerned, this analysis concludes the characterization of the *top-level* Sisyphus-2 task. Only after the selection of a *method* will we be able to specify further subtasks and their knowledge requirements. The *method-selection process* is in turn constrained by (1) the availability of methods, and (2) the availability of required domain knowledge. It usually consists of trading off computational advantages (e.g., efficiency, ease of representation) against the time and effort required to elicit domain knowledge (e.g., costs of knowledge acquisition, availability of expertise). Once an appropriate method is selected, the overall task can be decomposed into smaller subtasks, which in turn need to be analyzed and solved.

3.3 Reusability Considerations

Reusability considerations play an important role in major system-design decisions. Although we might be tempted to ask for reusability of all components all the time, several trade-offs have to be made that enhance or hinder future reusability. Furthermore, reusability for the Sisyphus-2 problem is a somewhat artificial idea, because there is no actual scenario or a real-world demand for elevator-configuration reuse. Only with such real-world scenarios can reuse be evaluated against its overhead costs. These costs include the extra amount of work in development of generic ontologies, as well as that of maintenance and version control to keep ontologies up to date. In addition, the effort required to adapt generic ontologies to a specific problem and the time to learn and use "foreign" ontologies have to be balanced against a development from scratch—that is, against the effort to build a custom-tailored solution with familiar, conventional techniques.

Further system-design questions arise when we take a closer look at the different modalities of reuse: It is important to ask not only *what* gets reused (targets), but also *when* and *by whom* such a reuse is feasible (see also Musen, 1992). Three building blocks of PROTÉGÉ-II are main targets for reuse: the abstract *domain ontologies*, the domain-independent *methods*, and the generic *mapping relations* that link together an application ontology and a configured method. Since both the application ontology and the configured method include highly task-specific knowledge, they are less likely candidates for reuse. However, even within these categories, there are many different levels of abstraction and generality that influence reusability. Different types of ontologies can be reused in different scenarios: *general-purpose domain ontologies* could be used for many different tasks. For example, general physical, mathematical, logical knowledge types or standardized measurement unit ontologies could be employed as background knowledge for the calculation of formulae or to develop domain ontologies that can be used either with SI-units or with the U.S. unit system. In contrast to these general-purpose ontologies, *Domain-specific ontologies* could be reused for different tasks in the same domain. For example, an ontology for elevators could be reused for a stockkeeping, for an accounting, for an ordering, or for a configuration task. Only *application ontologies* are both task- and method-specific, making them too custom-tailored for reuse.

Reuse is also possible at different *phases* of system development. For PROTÉGÉ-II, we envision the following development phases as targets for potential reuse. During *system development*, the reuse of general domain ontologies, shared ontologies, method ontologies, and application ontologies may speed up the construction of new systems. The global ontologies provided as part of the Sisyphus-2 distribution were reused in ELVIS after initial modifications. These modifications consisted of syntactical changes to accommodate differences between Ontolingua and MODEL and to account for our own variable namings. During *knowledge acquisition*, the reuse of previous domain instances and of pre-existing application ontologies may enable a faster start-up. For example, we reused instance information from the Sisyphus-2 distribution

[6] Note that not all non-input parameters are "free" model parameters (i.e., can be modified without restrictions). Furthermore, not all input parameters are fixed—that is, some may be changed during problem solving. We shall return to this point in Section 4.

such as model information and constraint expressions. Finally, *run-time* systems can benefit greatly from data sets that are available in a standardized, method-independent representation. We reused the test-case values from Section 9.1 of Yost (1992) to validate ELVIS's solution.

Tightly coupled to the issue of the time of reuse is the issue of who may benefit the most from different kinds of reuse. *Knowledge engineers* are likely to gain the most from abstract domain and method ontologies for the initial structuring of a new problem. Configured methods and reusable instances would speed up the construction of executable prototypes that can be used to do further knowledge acquisition or to compare different approaches. *Domain experts* could benefit from the structure of reusable domain ontologies that would organize their knowledge in a consistent way. They would also value the availability of instantiated domain knowledge bases in a standardized format. Special knowledge or classified information could then be added to complete the acquisition of domain-knowledge instances.

In conclusion, we can specify the precise benefits that are expected from reuse only after knowing more about the context of such reuse and after evaluating the costs of different options. Crucial to any kind of reuse is, of course, familiarity with the problem to be solved and experience with the formal framework in which reusable knowledge is presented. Otherwise, we would be unable to recognize possibilities of reuse in the first place, and the learning effort to master complex ontologies might not be justified by the effort saved by reuse. In our approach to the Sisyphus-2 problem, we have tried to reuse as much as possible from the supplied material—that is, the written problem description from Yost (1992) as well as the ontologies and knowledge bases that were made available by Gruber and Runkel (1993) as computer files.

4 DOMAIN ONTOLOGIES

When we started to look at the domain knowledge in detail,[7] it became obvious that the complexity of the domain, the entangled network of dependencies, and the vagueness of certain statements created a difficult starting point. We thus started to augment the plain text of the Sisyphus-2 document (Yost, 1992) to create a more useful domain dictionary. We made the following major changes:

- The document was reformatted with a text-processing program. This reformatting allowed us not only to take advantage of text-access and search features, but also to add the necessary code to generate a detailed table of contents, various indexes, lists of tables and figures, and to enumerate automatically the constraints in Section 7 of Yost (1992).
- All the parameters listed in Section 9 of Yost (1992) were changed to a consistent naming scheme. In the main text, they were typographically set apart from the rest, and indexing code was added. The index thus generated allowed easy cross-referencing of parameter usage throughout the whole text. The parameter-naming scheme underwent two major revisions as our ontology evolved, because we wanted the document to stay consistent with the names used in the PROTÉGÉ-II system. We envision, for a future version of ELVIS, that the document could be used in a hypertext-like way to provide help for variable names via links to the Sisyphus-2 document. The revision of names in the application ontology was performed mainly to enable automatic preprocessing of the supplied ontology files.
- The test-case (Section 9.1 of Yost, 1992) was converted into a table, in which all the parameters are listed with their values before and after a solution was found. The value changes were also computed in that table. This restructuring helped us to uncover an incorrect initial assumption about input parameters, which we had thought to be unchangeable, as they appeared to be according to Section 3 of Yost (1992). A quick look at the "change" column in the table revealed that an input parameter's value had changed in the test case. Such a discovery, of course, influences the design of the ontology: in this case, it meant that a parameter may be an input parameter as well as have computations that change its value.

This work helped us to cover the domain knowledge systematically and to structure and manage our analyses of knowledge types (see Section 4.2.2 through Section 4.2.3). However, these analyses were not sufficient to make the domain knowledge transparent. The biggest problem with the description of the Sisyphus-2 problem was the sheer number and the apparent complexity of calculations and formulae and

[7] We should emphasize that we started to look at the domain in February 1993, when no ontologies or knowledge bases were available electronically. We postulate that our approach would have changed the way in which people gain access to Sisyphus-2's domain knowledge. Only a comparison with other approaches at the Banff workshop will allow us to test this hypothesis.

the interdependencies of the parameters. This complexity made it almost impossible to judge appropriately the effects of constraint violations and fixes. We solved these problems by applying a simple calculation model to the domain knowledge, as provided by standard spreadsheet technology.

4.1 A Spreadsheet Solution for Sisyphus-2

To decrease the apparent complexity of the Sisyphus-2 problem, we constructed a spreadsheet system that provided us with a functional model of the domain knowledge in the Sisyphus-2 document. This system provided us with an *executable calculation model* that computed all important design parameters for the Sisyphus-2 domain. This model allowed us to analyze the different types of knowledge provided in natural-language descriptions, formulae, and tables in the Sisyphus-2 document. It also clarified some types of implicit knowledge and demonstrated how intermediate parameters are used to help compute the output parameter values. As we shall illustrate in Section 4.2, such an analysis not only is required to make the system actually turn over, but also heavily influences the design of application ontologies and of knowledge-acquisition tools. We chose a commonly available spreadsheet system (Microsoft Excel) to develop this executable calculation model. We found the following features useful:

- *Representation of mathematical and logical expressions:* After the document editing described in Section 4 and the standardization of variable names, we could define formulae with a simple translation from the text.
- *User interface:* We used tables to build an acceptable user interface for entering component information (see Figure 4a). This interface is almost identical to the tables used in the Sisyphus-2 document (Yost, 1992), with the added benefit that the spreadsheet can also display simple derived parameters and current values immediately. We used built-in functions to design a quick prototype that could acquire all input values with prompting windows. The standard reporting features of spreadsheets were used to display and print the required set of output parameter values.
- *Developer interface:* We defined a database of parameter values and associated knowledge (e.g., formulae, ranges, parameter type; see Figure 4b). This interface allowed us to access, inspect, and access parameters in their functional context. Information could be arranged and displayed in almost any format. Dependencies among variables could be detected and reported with the help of simple macros.
- *Computation of new parameter values:* The spreadsheet updates all parameter values within seconds after the user enters input values or makes changes to parameter values. Constraint violations are also signaled immediately, and we used a small macro program to collect all such violations and to report them in a sorted list (see Figure 4c).

Essentially, we have recreated the functionality of the system mentioned in Section 1.1 of Yost (1992): a simple calculation model used by domain experts in lieu of a Sisyphus-2 solution. Domain experts could use our spreadsheet solution to calculate all parameter values for a given set of input data. Then, the domain expert could manually update parameter values to solve constraint violations, and could use the same system to compute the new output values. In addition to this executable calculation model, we developed a user-input and output interface, and a constraint-checking mechanism. The spreadsheet system signals violations of constraints, and provides a list of possible fixes. However, the automatic application of fixes and a principled exploration of the possible design space were not easily implemented within the spreadsheet framework. Thus, this model does not include an algorithm that automatically finds a solution. The only guidance for modifying parameters is provided by the list of violations and possible fixes, ordered by desirability (Figure 4c). The user can interactively change the values in the way the fixes prescribe, and can recompute a new parameter state and check again for violations, but no help is provided to backtrack to previous states or to trace changes.

Although this “solution” has limitations, in some situations, it may actually be more practical than a highly sophisticated knowledge-based system, such as might be built by PROTÉGÉ-II. For example, although there is no automatic search for a solution, the user has the chance to notice fix sequences with certain characteristics, or to apply independent fixes in parallel. In the real world, decisions about how much of the task should be automated will depend on workplace analyses, usability studies, cost and safety issues, and, in general, a study of the domain expert’s capabilities with respect to real tasks.

Another reason for developing this executable calculation model with a conventional spreadsheet program was to give us a yardstick for measuring and comparing development efforts. Spreadsheets penetrate engineering domains more easily than do knowledge systems, and we believed that the main obstacle for domain experts to develop solutions for the Sisyphus-2 problem was not at the conceptual level, but rather was at

(a)

sling.table	Sling Model	A	B	C	Sling Weight
Table 1	2.5B-18	1.5	1.002	56	*292.20188*
	2.5B-21	1.75	1.002	94	*347.70188*
	4B-GP	2.5	1.6	223	*607.504*
	4B-HOSP	1.8	1.2	223	*506.128*
	6C	3.1	2.2	317	*822.068*
current values	*2.5B-18*	*1.5*	*1.002*	*56*	*292.20188*

crosshead.table	Sling Model	Crosshead Model	Crosshead Height (in)	Rated Crosshead Bending Moment	Rated Crosshead Deflection Index
Table 2 & 12	2.5B-18	W8x18	8.125	705000	84600000
	2.5B-21	W8x21	8.250	850000	103800000
	4B-GP	C10x15.3	10.000	1340000	200700000
	4B-HOSP	C8x11.5	8.000	810000	96900000
	6C	C13x16.55	13.500	1790000	200700000
current values	*2.5B-18*	*W8x18*	*8.125*	*705000*	*84600000*

(b)

S	*Parameter Name*	*Value*	*Entry*	*Fix*	*Lower*	*Upper*
4.3	counterweight.frame.height	138			90	174
4.3	counterweight.frame.thickness	31				
4.3	counterweight.plate.depth	7			7	12
4.3	counterweight.plate.thickness	1				
4.3	counterweight.stack.height	80				107
4.4	car.return.left	25			1	
4.4	car.return.right	3			1	
4.4	counterweight.space	18.25				
4.4	counterweight.to.hoistway.rear	6			*1.5*	
4.4	counterweight.to.platform.rear	5.25			*2.25*	
4.4	counterweight.ubracket.protrusion	0.75				
4.4	door.space	6.5				
4.4	opening.to.hoistway.right	16				
4.4	platform.running.clearance	1.25				
4.4	platform.to.hoistway.front	7.75				
4.4	platform.to.hoistway.left	7		06@2@06=change(!opening.to.hoistway.left,+1)@08=change(!car.return.left,-1)	8	
4.4	platform.to.hoistway.right	13			8	

(c)

03=step(!counterweight.to.platform.rear,-0.5)
04=step(!car.supplement.weight,+100)
04=step(!hoist.cable.quantity,+1)
04=step(!hoist.cable.quantity,+1)
04=upgrade(!car.guiderail.unit.weight)
04=upgrade(!hoist.cable.diameter)
04=upgrade(!machine.beam.model)
06=increase(!opening.to.hoistway.left,+1)
06=upgrade(!compensation.cable.model)
08=decrease(!car.return.left,-1)
08=upgrade(!machine.groove.model)
09=change(!machine.model,"28")

Figure 4: The interface of the spreadsheet solution: (a) Table interface for components*;* the gray areas are designed for user input. Computed values—here the sling weights—are displayed instantly. (b) Representation of the parameters; Column 1 references the place where the parameter is defined in the Sisyphus-2 document. Column 2 contains the standardized parameter name. The third column contains the formulae to compute the values, but only the result is displayed. The formulae are defined such that all references to a parameter name will use the value displayed here in any computation. The fourth column is used to directly enter new values. The fifth column is used to signal any constraint violations. Again, the formula to compute the value shown in the fields is actually accessible in that field. Gray fields indicate that there is no constraint directly associated with this parameter. Columns 6 and 7 contain range values. Italicized values are computed from other parameters; Roman values are entered at knowledge acquisition time. (c) Report of violations; an internal representation of possible fixes for current violations.

the computational level. We developed the spreadsheet model in about 30 person-hours, including an 8 person-hours redesign cycle to accommodate name changes and a better spreadsheet organization with no added functionality. Of course, the initial work invested in rewriting the original Sisyphus-2 document shortened the development time considerably, because we were already familiar with the text and terminology, and had some ideas about problematic issues. Only further experiments will show whether it is feasible to develop systems for similar problems with conventional technologies such as spreadsheets. The effort required to learn, use, and adapt knowledge-system technology has to be balanced against that needed for custom-tailoring new systems with traditional technologies. A major problem for Sisyphus-2 is to produce a system that is accessible and easy to use for the domain expert. Spreadsheets on personal computers may solve this problem more easily than can knowledge-based systems. A major advantage of this technology is the almost instantaneous feedback for many different parameter-value configurations. The ability to recalculate the complex model quickly and the availability of immediate feedback on any actions taken would allow an expert to deal with constraint violations in novel and exploratory ways.

The main disadvantage of the spreadsheet solution, of course, is its ad hoc nature. Although we are able to abstract certain knowledge types (such as table calculations, range checking, and formula decomposition) that are certainly reusable for other problems, there is no automatic way to translate or reuse the domain knowledge encoded in the spreadsheet's cells. A main side effect of this executable calculation model was that we could check the Sisyphus-2 problem description (Yost, 1992, and our modified version) more thoroughly and were able to discover minor inconsistencies and several errors. These discoveries helped us to construct and validate the current ELVIS system. However, this validation was limited, since we could test our model with only one data set. The spreadsheet model can validate the final values as defined in Section 9.1 of Yost (1992)—without certain errors we suspect to be present in the document. We do not know whether the spreadsheet model is correct for other value combinations.

4.2 Knowledge Analysis and Interface Design

Knowledge analysis, the identification of different types of knowledge in a task, influences decisions on how to represent and use domain knowledge in a performance system. One or more forms of knowledge representations have to be chosen from a variety of knowledge-representation schemes, such as frames, object–attribute–value triples, record structures, tables, objects, or rules. Typically, different forms of representations support the expression of different types of knowledge by providing special features that ease the task of translating real-world knowledge into formal structures.

Domain knowledge usually comes in many different apparent structures. Lists of components and part-of hierarchies may seem natural to describe physical objects, and rules allow domain experts to express actions that depend on different conditions. The task of the knowledge engineer is to find appropriate knowledge-representation formalisms that not only support the *encoding* of the domain knowledge, but also help with the *acquisition* of that knowledge from the domain experts. If the surface structure of the representation is too complex or requires programming experience, the domain experts may refuse to provide the relevant knowledge, or at the least, they may find it difficult to provide, update or edit the relevant information. Thus, domain-knowledge analysis is an important task that must find a representation that is both efficient for a computer system and suitable for knowledge acquisition.

PROTÉGÉ-II, like its predecessors OPAL (Musen, Fagan, Combs & Shortliffe, 1987) and PROTÉGÉ (Musen, 1989a), emphasizes the importance of representations that are modeled closely after the domain. The form-based approach of PROTÉGÉ-II evolved from modeling the domain of clinical-trial protocols (Musen, 1989b) and represents knowledge in a way that is familiar to domain experts. These forms can be modeled on existing knowledge representations, such as patient records, laboratory data sheets, or any format with which the domain experts are already familiar. The study of knowledge representations and artifacts employed by problem solvers in their natural environment (e.g. Suchman, 1987, or Hutchins, 1991) takes seriously the problem solver's way of structuring a task, and tries to avoid imposing artificial knowledge representations on real-world tasks. To enable the actual use of knowledge systems and to improve acceptability, we follow as closely as possible the way in which people in the domain deal with the knowledge. Of course, for the purpose of developing a computer system, many compromises or adaptations may be made to accommodate complex knowledge structures. However, it remains the duty of the knowledge engineer to make explicit such changes and assumptions by declaring them in a formal domain ontology. We go further in the use of ontologies than do other proponents who see their primary virtue as providing a rigid, formal

structure. We emphasize the *communication aspect* of the ontology building process. Only when the domain experts can understand and agree with the knowledge engineer about the utility and meaning of an ontology will the effort put into this formal adventure pay off.

Since we had no interaction with human experts, we were severely limited in our attempt to develop an appropriate domain ontology for elevator experts. Our strategy in dealing with this artificial situation was to take the document as seriously as we could with respect to the way the domain knowledge was presented.[8] In the remainder of Section 4.2, we present our knowledge analysis for the Sisyphus-2 problem. We begin with an analysis of different knowledge sources (Section 4.2.1), followed by discussions of parameter types (Section 4.2.2), of constraint knowledge (Section 4.2.2), and of fix knowledge (Section 4.2.3). Finally, we conclude this section by presenting our application ontology for Sisyphus-2.

4.2.1 Domain-Knowledge Sources

For the Sisyphus-2 task, domain knowledge for the elevator-configuration task originates from different *sources*. The origin of information may influence the characteristics of knowledge representation, as well as of knowledge acquisition. For Sisyphus-2, we analyzed knowledge as originating from four different sources:

- *Building specifications:* dimensional information about the layout of the *location* where the elevator will be installed. This knowledge includes facts such as widths, lengths, and locations of architecturally given elements. This information originates with the building designer, and the problem solver should not modify these values.
- *Elevator specifications:* information about the material and components supplied by the *manufacturers* of elevators from which the configuration process can choose. These specifications include all the available parts, as well as parts' pertinent characteristics (such as component dimensions, weights, and allowable forces).
- *External specifications:* knowledge imposed on the configuration explicitly or implicitly as externally supplied constraints for legal or safety reasons. For example, legally prescribed minima and maxima should never be changed by the problem solver.
- *Customer specifications:* certain components and dimensions specified by the customer (e.g., the need for a telephone in the car cab). It was unclear to what degree such specifications could be modified by the problem solver.

Although there is no commitment yet to a specific representation or problem-solving method, certain choices that influence those decisions are already made at this preconceptual stage. The "first impression," the attempt to "just understand what the problem is about," or a "quick summary" reveal certain biases on how to view the problem. For instance, the document was clearly written with a constraint-based solution in mind, and leaves little choice on how else to think about the problem. We could try to ignore this biased information, but unfortunately there is no other domain knowledge available that we could use to develop alternative approaches. Even if such a constraint-satisfaction view is adopted, the material does not allow for much freedom in the selection of a problem-solving method. For example, the knowledge provided seems already to contain specific heuristics concerning the optimization of other aspects of elevator configuration—namely, cost minimization. An example of this hidden parameter is that almost all proposed fixes do upgrades from smaller, presumably cheaper, components to bigger ones. Without an explicit goal of minimizing costs and without detailed knowledge about cost structures, it is impossible to experiment with other problem-solving methods.

Another important attribute of domain knowledge is the time at which the knowledge is used in the overall *control flow* of the problem-solving process:

- *Input specifications*: Information to be supplied externally by the customer is represented by 26 different parameter values that are assessed at run time.
- *Default values*: Many design parameters have default values for an initial configuration or for situations when no values are supplied from other sources. Most of these values are likely to change during run time. The epistemological status of this knowledge should be assessed carefully, be-

[8] However, we did *not* follow the Sisyphus-2 document in certain procedural suggestions, as we explain in Section 5.2.

cause, in many instances, there is hidden, implicit knowledge available that guides the selection of these default values (e.g., knowledge about the price of components). It may be desirable to make explicit the justifications for default values in a refined ontology.

- *Assumptions:* For a variety of similar task instances, it may be advisable to make constraining assumptions about certain design parameters. Assumptions will not change as long as the system's scope remains the same. To enable future extensions and to delineate the competence boundaries of the system, developers should identify and label all assumptions.
- *Calculations:* Many design parameters are calculated from given domain formulae that refer to other design parameters. These parameters include values defined by physical laws, as well as engineering formulae and approximations.
- *Output parameters:* A specified set of parameters is reported when all values of the model are determined and a solution is reached.

Some of the input specifications are *mandatory input values* that cannot be changed throughout a problem-solving episode, but that may be different for different problem instances (e.g., building specifications, user requirements, legal requirements, or safety factors). Other inputs are *desired input values*, with which the problem solver should start, but which may need to be changed to satisfy more important constraints (e.g., optional features, or dimensions specified for noncritical reasons, such as aesthetic factors). A third type of input knowledge is *default input values*, which provide commonly made assumptions about certain components or parameters (e.g., available models, or default sizes of components). These different input types may require different modes of acquisition at run time and should be treated differently by the configured problem-solving method.

4.2.2 Constraint Knowledge

In Section 7 of the Yost (1992) document, 50 explicitly labeled constraints are listed. Similar statements that constrain possible values or calculations for parameters are scattered throughout the document. In addition, knowledge about default or initial values could be represented as constraint knowledge. In fact, we originally included such assumptions as constraints, but later decided that this knowledge was sufficiently different to warrant its own classification. Our classification of constraints now includes three different types: assign constraints, range constraints, and general fix constraints.

Assign constraints: "*The distance* `top.landing.to.underside.machine.beam` *is the* `overhead` *minus the distance from the* `machine.room.floor.to.underside.machine. beam` …" A parameter value is constrained by a formula that may include the values of other parameter values. In many cases, the applicability of a general formula (e.g., computing the weight for all sling models) is further parameterized by model-dependent conditions: "*When the* `machine.beam.support.type` *is of the pocket type, the* … `machine.beam.bearing. plate.thickness` *is one inch.*"

Range constraints: "*The* `car.cab.height` *must be between 84 and 240 inches, inclusive.*" The (interval) parameter's value is constrained by acceptable boundaries. Some of these boundaries may be dependent on preconditions. Such constraints may include a list of fixes that may be applied if the parameter falls outside of the boundaries.

General fix constraints: "*The* `motor.model` *must be compatible with the* `machine.model`." Here, a parameter value (a choice of motor) is constrained by being compatible with the value of some other parameter (the machine model). This category of constraints includes a list of fixes to be applied when the constraint is violated.

These three constraint types seem to cover all the constraints of the Sisyphus-2 document. Assign constraints provide a definition for all basic and intermediate parameters whose values can be computed by calculation. Range constraints are introduced to account for different knowledge-acquisition needs—that is, to provide a defined place to specify simple value boundaries. The last constraint knowledge type, general fix constraints, includes a list of fixes, and covers knowledge about safety, legal configurations and compatibility.

4.2.3 Fix Knowledge

Along with constraint knowledge, Section 7 of Yost (1992) provides information on how to fix constraint violations. This knowledge specifies which parameter values to change and how to change them. We have identified three types of fixes: assign fixes, step fixes, and upgrade fixes.

Assign fixes: *"Increase the* `counterweight.space` *by the amount by which the* `counterweight.to.hoistway.rear` *distance falls short of its minimum."* Parameter values are changed according to some formula. Such fixes may also depend on preconditions that check a related parameter. The system may modify values of constrained parameters by (1) adding or subtracting the newly calculated value to the current value or by (2) assigning a new value. In many cases, the "formula" to compute the difference between the new and the old parameter value is just a fixed increment or decrement—a piece of knowledge that makes the recomputation of new values easier.

Step fixes: *"… increase the* `counterweight.buffer.quantity` *by steps of one."* This fix knowledge assumes that several actions to repair a constraint violation can be taken in a pre-specified sequence. The Sisyphus-2 document proposes a stepwise application of such a fix until the signaled constraint violation is resolved. It remains unclear, however, what should happen if other constraint violations are flagged during such a stepwise value change. Such questions about how to apply a fix depend on the problem-solving method. Regardless of the method used, the knowledge about these step fixes should be included in the domain ontology. Thus, a special fix type provides knowledge about the direction, amount and fashion in which a value can be modified with steps.

Upgrade fixes: *"Upgrade the* `sling.model`*."* Upgrade fixes can be applied only to ordinal parameters, such as component models, where an ordered list of values is given. We could choose to provide the knowledge to upgrade a component in many different places of the domain ontology. An obvious candidate would be the fix knowledge itself, because that is where the information is actually used. For example, for a fix that requires an upgrade of the sling model, we could choose to embed the new, upgraded sling model in the fix rule itself. Such a representation requires a different fix rule for each sling model and explicitly provides the upgraded model in its action part.

Although this representational choice may produce efficient rules in a compiled knowledge system—given the usually powerful matching behavior of most rule-based architectures—the scattering of knowledge may not be a wise choice with respect to elicitation and maintenance of the knowledge. The natural representation of upgrade relations in an ordered table—used many times throughout the Sisyphus-2 document—serves this purpose in a more elegant way. Most of the 14 explicit tables (some table-like information is given in plain text) are database-lookup tables that show the characteristics of different models. At least one column is usually used to supply comparison information that can be used for upgrading. Typically, these are parameters that can take only one of a fixed list of discrete, ordered values (e.g., 10, 15, 20, 25, 30, or 40 horsepower for a motor model). Some tables represent relations between two different parameters (or models). One problem with these tables is that their order relation is given only implicitly, usually via a left-to-right reading order. Only by making inferences from the model's names (e.g., "DS-20" is a deflector sheave model that has a diameter of 20 inches) or by applying other domain knowledge can we identify the correct component. Many of the original VT and SALT publications expressly use this tabular knowledge type and provide specific prompting schemes for table access For example, the database-lookup procedure asks for a "table name", a "column with needed value," and an "ordering column."

Due to the lack of a table representation in MODEL and DASH, we have chosen to model upgrade fixes differently. The upgrade knowledge for every component is specified explicitly in the domain ontology: All components include a separate upgrade slot. The value entered into that slot through the knowledge-acquisition tool is then used by mapping relations (see Section 5.3) to generate individual upgrade rules for every component. Although such a representation may not be the best way to acquire and model this knowledge, we can avoid maintenance and consistency problems with the automatic generation of the respective upgrade rules.

These three fix knowledge types (assign, step, and upgrade fixes) cover all the possible fix applications in the Sisyphus-2 problem. Assign fixes allow the definition of rules that override existing values. Step fixes are special cases of assign fixes that include some control flow information (i.e., they specify what to do after the fix has been applied once). The third category, upgrade fixes, takes advantage of special domain knowledge that allows for changing the selection of a component in a principled way.

The main problem with fix knowledge, as provided in Sisyphus-2, is the fact that fixes represent heuristic knowledge that lacks sufficient justification and thus does not allow additional "depth" in reasoning and fix-application procedures. For example, fixed increments (e.g., "increment by one inch") are suggested in many cases to overcome a constraint violation, but no explicit reason for the amount of change is given. Further knowledge analysis might reveal that these fixed increments depend on external constraints that are unchangeable (e.g., parts supplied by a subcontractor), are used for practical reasons (e.g., precision of cutting

tools), or it may turn out that such increments have only been used to facilitate the computational aspects of recalculating new parameter values (which is not a problem if done with a computer). This kind of knowledge would not only influence the representational format but also enhance the reusability of our abstract knowledge types because it defines applicability conditions and additional knowledge needs in a more explicit way.

4.2.4 Domain Ontology

The most crucial activity in the construction of a domain ontology is to determine a useful backbone: an organizing structure on which developers can place different types and classes of domain knowledge. In many domains, it is necessary to use some natural *grouping of parameters* into sections or logical components. This need to group information is especially true for Sisyphus-2, where there are so many domain relations and parameters to be managed. (The test case in Section 9 of Yost (1992) lists 161 such parameters, and there can be many more intermediate parameters, depending on the complexity of the representation.) Some variable groupings seem to emerge naturally from the experts' domain models. However, there are many different reasonable groupings possible. For example, at first sight, it seemed obvious to group all the information related to a crosshead model as a logical component at the top level of the ontology. A closer look, however, revealed that crossheads matter only in the context of selected sling models and are not relevant in the overall configuration process. Thus, crosshead parameters could be grouped under the sling component.

Such ontology design issues can cause many revisions of the domain ontology's main structure. Some of these revisions may be dependent on the developers' individual style and preferences. For example, whereas one domain expert might prefer to reproduce the part-of hierarchy of an elevator system, another developer might think it more natural to have a simple flat list of state variables. These decisions may not influence the performance of a run-time system, but they certainly will affect the knowledge-acquisition and knowledge-maintenance phases. Thus, representation decisions have to balance several factors: the expected reuse of an ontology and its instances, the efficiency of the representation for a given method, and the adequacy of the ontology's structure for knowledge acquisition. Whereas a part-of hierarchy appears to be useful for reuse (in contrast to a long list of cryptic state variables), a backtracking problem-solving method might not make any use of such "representational sugar" at run time.

In Figure 5, we show our application ontology for the Sisyphus-2 problem. Since this is an application ontology, it includes concepts that could be reused from a domain ontology, such as the classes `ELVIS-Components` and `ELVIS-Models`, as well as concepts that are inherited from information in the method ontology (see Figure 3), such as the classes `ELVIS-Constraints` and `ELVIS-Fixes`. As we show in Section 4.4, this application ontology is also the basis of the knowledge-acquisition tool. Therefore, to construct this ontology, we tried to follow, as closely as possible, the organization of the domain knowledge presented in the Sisyphus-2 document. This strategy should lead to a knowledge-acquisition tool that is natural and easy to use for domain experts.

Every subclass of `ELVIS-Components` is a logical system that groups and provides access to the parameters that are relevant in that context. For example, building dimensions are grouped under the hoistway system if they appeared in the hoistway drawing in the Sisyphus-2 document. These logical systems may also include a list of relevant physical components. For example, the set of available car-guiderail models is accessed under the `CarSystem` class. Information about each instance of a car-guiderail model is stored under the class `CarGuiderails`; however, for knowledge acquisition, this information is accessible only via the `CarSystem` class.

4.3 Definition of Ontologies: MAÎTRE at Work

The MAÎTRE tool supports the definition and maintenance of ontologies and helps the developer to enter and edit ontology objects consistently. Features such as context-sensitive pop-up menus allow the user to define complex information with a minimum of typing, and without overwhelming the user with the syntax details of the MODEL representation language. Figure 6 illustrates the MAÎTRE editor with the ELVIS application ontology. Although building this type of graphical editor is not novel research by itself, MAÎTRE is an essential part of the PROTÉGÉ-II tool set. Because our methodology includes a number of different types of ontologies, and expects developers to revise these ontologies iteratively, we must have a tool that provides a consistent and simple interface for inspecting and editing ontologies.

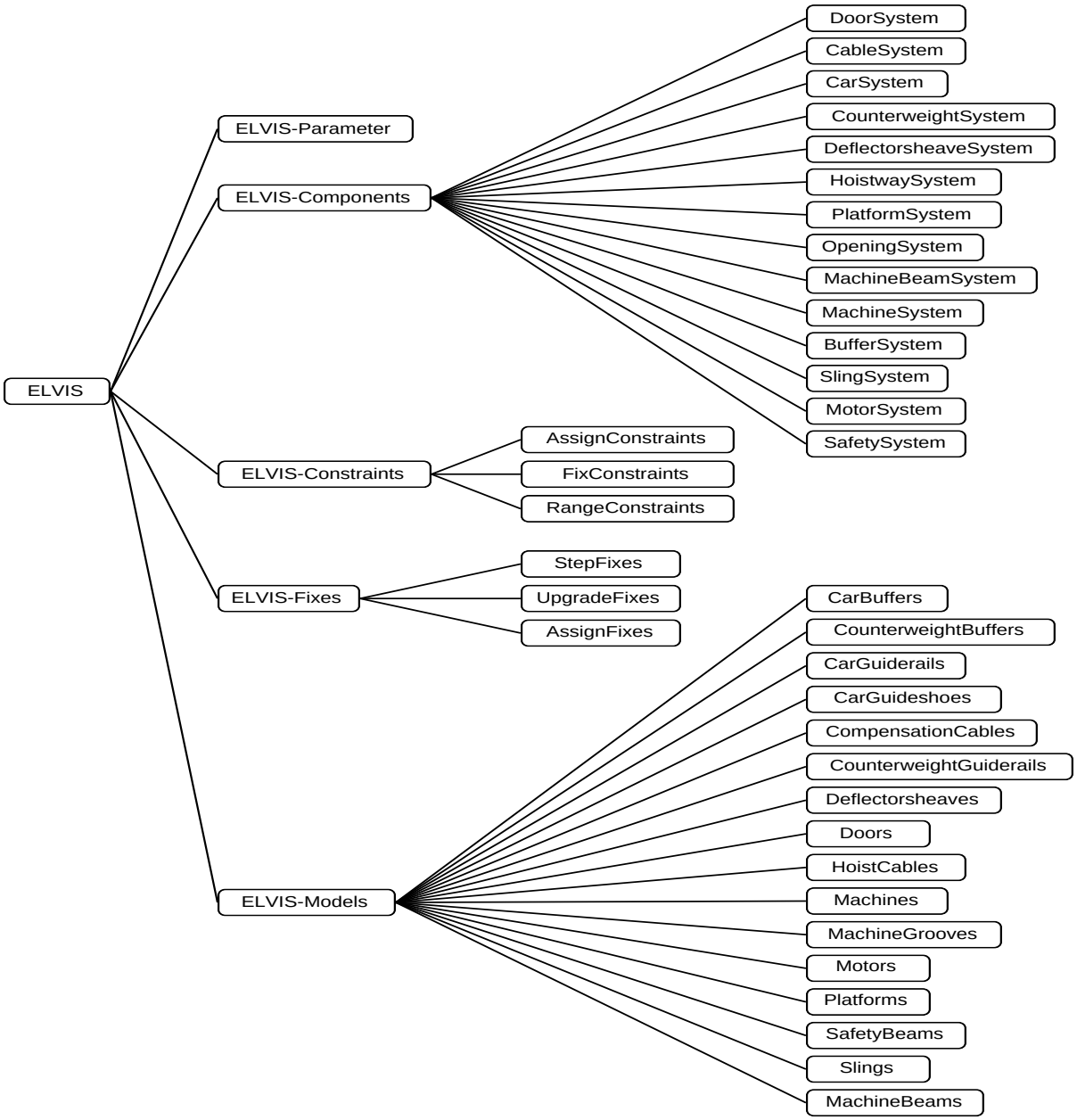

Figure 5: The structure of the application ontology (for simplicity, no slots are shown). Every member of ELVIS-Components includes a list of ELVIS-Constraints and a list of ELVIS-Parameters, as well as information about the appropriate ELVIS-Models.

Additional MAÎTRE features, such as incremental loading of partial ontologies, allow experimentation with different versions of a single ontology. For example, we are currently experimenting with integrating parts of the Ontolingua design ontology into our ELVIS representation. The design ontology defines a terminology suitable for any configuration design; for example, we have designed the classes `ELVIS-Constraints` and `ELVIS-Parameter` to be subclasses of the more general terms “Constraint” and “Parameter,” as defined in the Ontolingua design ontology. This integration is accomplished with the help of layered ontologies, where different abstractions levels are loaded incrementally to produce a composite ontology. In the future, we hope to integrate more of the general design ontology into the ELVIS application ontology.

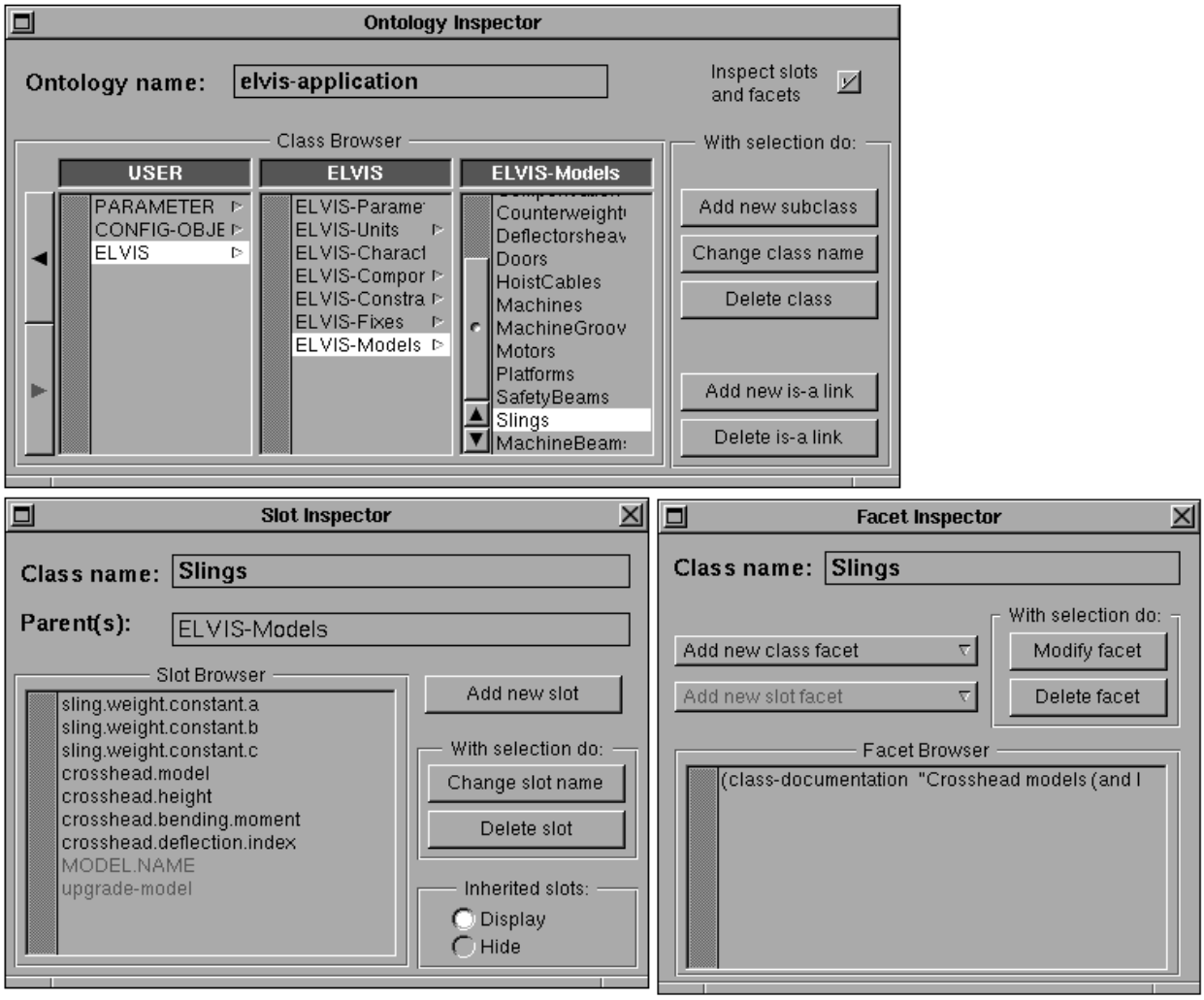

Figure 6: Editing the domain ontology of ELVIS with MAÎTRE. The top panel is used to define class objects and their relations, the lower-left panel slots, and the lower-right panel facets. Note that the ontology does not allow the entry of values; they will be entered with the help of the knowledge-acquisition tool generated from this ontology.

4.4 Design of the Knowledge-Acquisition Interface: DASH at Work

A crucial part of PROTÉGÉ-II's development cycle is the design of a suitable *knowledge-acquisition tool* (also called a *domain-specific editor*) that allows the acquisition of all the required *domain knowledge instances*. As outlined in previous sections, the *structure* of the domain knowledge is already given by the *domain ontology*, but the construction of a useful tool to enter the specific *domain knowledge instances* depends on information about the *form* of individual knowledge pieces. This *interface design knowledge* contains some (potentially) domain- and application-independent knowledge that may be reused across different domains and applications. However, many design choices have to be made based on the specific domain knowledge to be acquired and will depend on layouting and interfacing options and demands of the particular application and even of the specific implementation. For example, the usable size of the screen will affect design choices: Information that might be grouped together may need to be split up for use on a smaller screen.

The design of a knowledge-acquisition tool is in itself a complex design process, most often constrained by available tools and hardware options. To achieve reusability across different domains and portability across different computing environments, PROTÉGÉ-II uses a *generative approach*, where the knowledge-acquisition tool may always be regenerated from a given application ontology. The DASH subsystem (Eriksson, & Musen, 1993) takes the *application ontology* as a starting point for design and constructs a suitable knowl-

edge-acquisition tool. The class hierarchy of the ontology along with user-defined slots provides the initial grouping and layout information for the design of the interface. However, to provide more sophisticated user-interface support for the acquisition process, we should provide DASH with additional information beyond the pure ontology: (1) *data type* information and (2) the intended *dialog structure*. The former information translates application ontology objects into appropriate graphical interaction objects such as text fields, number fields, browsers, push-buttons, or pop-up menus. The dialog structure defines interaction patterns: which objects are accessible directly at the top-level of the interface, how objects should be grouped into windows, and what the sequencing of windows in the knowledge-acquisition tool should be. In the current implementation of PROTÉGÉ-II, this type of information is supplied via a set of special-purpose *slot facets* that is managed by the MAÎTRE editing environment.[9]

DASH suggests an initial layout according to its design rules, but the user can the freely modify this initial design to suit his particular needs and taste.[10] The user of a knowledge-acquisition tool is most often the domain expert. Depending on the complexity of the domain, on the tediousness of knowledge entry, and on the computer literacy of such a user, domain experts may be supported by other knowledge providers and by the knowledge engineers. DASH then stores the current design and layout information, along with a database of the relevant changes (e.g, coordinates of display widgets, changed labels of buttons), and produces as its main output a declarative description of the interface.

4.5 Generation of a Knowledge-Acquisition Tool: MART at Work

Finally, the MART subsystem of PROTÉGÉ-II transforms the declarative interface description into a standalone *knowledge-acquisition tool* that consists of all the windows and forms as defined by DASH. The complete knowledge-acquisition tool generated by DASH and MART for the Sisyphus-2 problem includes 37 different types of knowledge-entry forms. Figure 7 shows the design of screen forms for three objects in the application ontology. In the figure, these windows are filled with *domain-knowledge instances*. The domain expert should then use this tool to enter all the domain-knowledge instances required for solving all the *application problems*. MART is a run-time system based on the MECANO approach (Puerta, Eriksson, Gennari & Musen, 1993) that generates knowledge editors on the NeXT platform.

Although the division into small, separate steps of the process of generating knowledge-acquisition tools may seem unnecessarily complex, it provides clear boundaries among the different types of knowledge required to generate a run-time system. For example, the MECANO approach not only includes the domain ontology, but also allows requires the explicit formulation of an interface model that includes all additional interface-design knowledge. Furthermore, the transparent separation of these different knowledge types allows PROTÉGÉ-II to achieve platform independence that will be required in real-world knowledge-acquisition situations. For example, because DASH produces an implementation-independent description of the knowledge-acquisition interface, we were able to develop another knowledge-acquisition tool on the Macintosh. The tool produced for the Macintosh has the same functionality and the same look and feel as the tool produced on the NeXTstation, because we used the same generative process beginning with the same application ontology.

4.6 Summary of Ontology Design for ELVIS

The final design of the application ontology includes all the subsystems (assemblies) of an elevator system. For all physical components, the ontology contains all the parameters that need to be acquired from the domain expert. For example, all sling models have three specific constants that depend only on the selected model (see Figure 7b). All other relevant information about slings can be provided by the domain expert through the definition of parameters and constraints. For example, the formula to calculate a sling model's weight would be added as an assign constraint. Similarly, range constraints can be added, along with their potential fixes.

Note that developers have to make choices about what knowledge to acquire with the knowledge-acquisition tool and what knowledge to store in the application ontology. For example, in Figure 7(a), the domain expert must provide with the knowledge-acquisition tool the list of all parameters associated with slings. Instead,

---

[9] A better solution would be a separate knowledge-acquisition design-editing environment, because this information does usually not belong into a reusable domain ontology.

[10] On the NeXT platform, the user custom-tailors the layout with Interface Builder, a tool for window and display layout and editing.

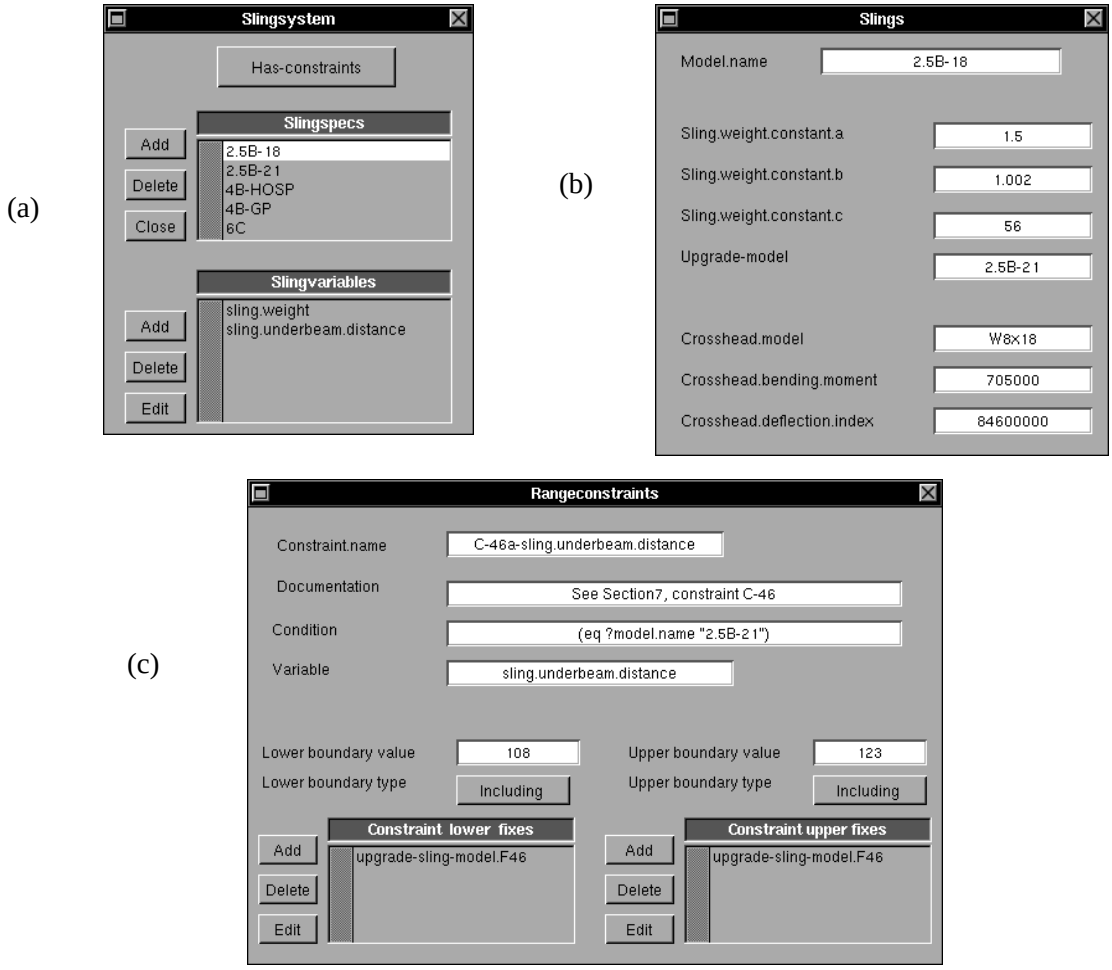

Figure 7: Three forms of the knowledge-acquisition tool generated by DASH. (a) An editor to define multiple instances of sling models and associated variables. (b) A form to enter data specific to a particular sling model. (c) An interface to define a range constraint.

we could have listed these parameters as slots in the application ontology, and have left only the task of adding components and their specifications to the domain expert. At the other extreme, we could have left even the definition of the static component parameters (such as "Sling.weight.constant.a" in Figure 7b) to the domain expert.

The choice of storing knowledge in the ontology versus eliciting that knowledge with a knowledge-acquisition tool should be resolved by the envisioned reuse. Our decisions were based on the assumption that our ontology would be reused by different companies or in different countries, where different guidelines and safety and legal constraints might exist, but where the basic physical structure of elevator components is unchanged. In another scenario, such as internal reuse within the same company, all the parameters, constraints, and fixes might be reused; thus, their inclusion in the ontology would be preferred.

We realize that quantitative information about knowledge types—be it rules, constraints, or knowledge-base entries—is not always meaningful because representational changes may influence these figures. Nevertheless, we would like to give some feeling for the size and complexity of the final application ontology. The following list provides an overview of some important characteristics:

- 14 physical and logical systems, with 16 models and component types, are defined.
- 20 input parameters are provided by the user at run time and are never changed during the system's execution (fixed run-time input parameters).

- 6 input parameters are also provided at the beginning of a problem-solving episode, but are modifiable through fixes (modifiable run-time input parameters).
- 7 default parameters have specific values at the beginning of a configuration problem.
- 50 constraints specify conditions to be satisfied for major parameters. If upper and lower boundary constraints are counted separately, there are 64 constraints.
- 58 fixes are provided. 41 are unique; some fixes can be applied to several violations. 10 fixes are unique upgrade fixes, 14 are unique assign fixes, and 17 are step fixes. The fixes directly affect 31 major parameters (which in turn affect many other parameters). There are at most five different fixes for a single constraint violation; more typically there are only one or two fixes per violation.
- Over 150 formulae specify how to compute values for major parameters.

Our attempt to organize this complex web of knowledge provided for solving the elevator-configuration task is based on the construction of a reusable domain ontology. The domain ontology developed for ELVIS is based on the information provided in the Sisyphus-2 document (Yost, 1992). Although the Ontolingua ontology proposed for Sisyphus-2 did not meet all the requirements for ontology construction in PROTÉGÉ-II, we tried to stay as compatible as possible to the ideas in the ontolingua ontology. Thus, the domain ontology was based on the physical and logical components of an elevator system. This domain ontology includes the knowledge that describes the more static parts of the domain knowledge (discussed in Section 4.2.1 and Section 4.2.2). Procedural domain knowledge is added to the domain ontology through the definition of constraint knowledge (Section 4.2.2). These constraints can define additional internal parameters and can be used to derive values for parameters. The knowledge included so far constitutes the domain ontology. Such a domain ontology is a good candidate for later reuse in other tasks that use different problem-solving methods. The augmentation of the domain ontology with the abstract definitions for signaling violations and for applying fixes (Section 4.2.3) produces the application ontology, since this knowledge is added to solve the configuration task with the propose-and-revise problem-solving method.

5 PROBLEM-SOLVING METHODS

We have modeled the Sisyphus-2 task with the propose-and-revise problem-solving method defined in PROTÉGÉ-II. This method behaves like the propose-and-revise method of VT (Marcus et al., 1988), and only a comparison of the respective method ontologies would tell us whether they are identical; unfortunately, VT did not include an explicit method ontology. Figure 8 shows the main control structure of the propose-and-revise method: propose an initial design, critique the design, revise the design such that it satisfies the design constraints, and repeat this process until there are no constraint violations in the design. This method description is close to the generic task description for routine design (Chandrasekaran, 1990). Although the method is described at a similar level of abstraction, the different terminologies used in generic tasks, in VT, and in PROTÉGÉ-II make it difficult to assess commonalities and differences of the respective methods. We have already described our method ontology for the propose-and-revise method in Section 2.2, Figure 3. In Sections 5.1 through 5.3, we shall discuss the version of the propose-and-revise method that we use in ELVIS, provide a detailed description of the method's behavior, and show how we map the propose-and-revise method to the Sisyphus-2 domain knowledge.

1. Propose an initial solution.
2. Check for any constraint violations;
   if there are none, succeed.
3. Choose the best fix associated with the violated constraint;
   if no fixes are available, backtrack to previous choice point.
4. Revise the solution by applying the selected fix.
5. Go to step 2.

Figure 8: The general structure and control flow of the generic propose-and-revise method. See Figure 8 for a more detailed account of the method.

5.1 Basic Problem-Solving Method

Our implementation of the propose-and-revise method is based on the *chronological-backtracking method* (for a detailed history of these methods in PROTÉGÉ-II, see Eriksson et al., 1992). Chronological backtracking itself can be viewed as state-space search with additional control structure for the generation and traversal of states. Although the authors of VT (Marcus et al., 1988) have chosen to call their method *knowledge-based backtracking*, the core method remains the same.

The main ingredients of a backtracking algorithm are the ability to generate new states and, when the algorithm reaches a dead end, the ability to reverse or *backtrack* to a previous state. The exact nature of the generation step, as well as of the method for detection of dead ends, is dependent on domain knowledge. To improve the algorithm from a blind state traversal, we employed heuristic evaluation functions to select the next state to explore. These functions also are usually domain dependent. For example, we have used a scoring algorithm that provides an evaluation score for each state; the next state to be explored is the state with the highest score. If that path (the state and its possible successor states from the recursive application of the method) is unsuccessful, the state with the next highest score will be considered. This control regime is best-first search. Of course, we could support alternative search strategies, such as depth-first or breadth-first algorithms. In our implementation, a simple modification of the method-control program can achieve these variations without modifying other pieces of code.

An important consideration for a problem-solving method is the *representational requirements* of that method. Chronological backtracking does not impose a strong model onto the data; thus, it is sufficient to have a way to model *states*. For ELVIS, a state is a data structure that holds all the design parameters and their respective values. All knowledge representation requirements of our propose-and-revise method are specified in the accompanying method ontology (see Figure 3).

5.2 Method Configuration

The generic method can be improved to account for specific circumstances in the domain. For example, the fix-application algorithm specified in Section 7 of the Yost (1992) document proposes a specific sequence in which fixes have to be applied. Although that control scheme may be optimal with respect to the acquired domain knowledge—and especially for the given "desirability values"—we have designed a more general control structure that we outline next. Improvements of the general propose-and-revise method (as described in Figure 8) are a matter of optimizing performance, which is here defined as the *time required* to achieve a feasible solution. Since we had to rely on only one set of test data, our optimization steps are subject to further evaluation.

A first deviation from the proposed fix-application algorithm is that we apply all possible fixes in parallel. In addition, we do not follow the algorithm supplied in Section 7 of Yost (1992) for applying step fixes. Instead of explicitly applying a step fix until the constraint violation is solved, we apply a step fix only once per state. However, the same step fix may reoccur as a candidate fix in the next revision state if the constraint is still violated. As can be seen from the program's fix-application trace in Figure 11 (in Section 5.4), the application of two fixes in the test case do follow the incremental schema that is proposed in the Sisyphus-2 document. That is, both fixes for dealing with the violation of the constraints associated with `counter-weight.to.platform.rear` and `car.supplement.weight`, respectively, are applied in sequence.

Our spreadsheet model and the analysis of our first PROTÉGÉ-II prototype allowed us to construct a dependency network that explicitly lists all parameters and their predecessors and successors. The dependencies can become very complex: for example, the parameter `angle.of.contact` depends on four other parameters which in turn depend on other parameters, such that any recalculation of `angle.of.contact` involves 27 other parameters. This dependency knowledge is used in ELVIS to reduce the amount of recalculation when updating parameters. In the Excel solution, the updating of parameter values was achieved as a built-in feature of the spreadsheet and did not require us to make this dependency knowledge explicit. The dependency knowledge could, however, also be used to maintain the knowledge base or to improve it by finding loops and shortcuts in the calculation model. To speed up recalculation, we derived the dependency network from an analysis of calculation formulae, constraint rules, and fix knowledge. Whenever a parameter is changed, this network is inspected and all dependent parameters are recursively flagged for recalculation.[11] Only these parameters need to be recalculated. Although we chose to use a static dependency network, a dynamic construction of such a network could be added as special method code if the domain ontology were to change frequently.

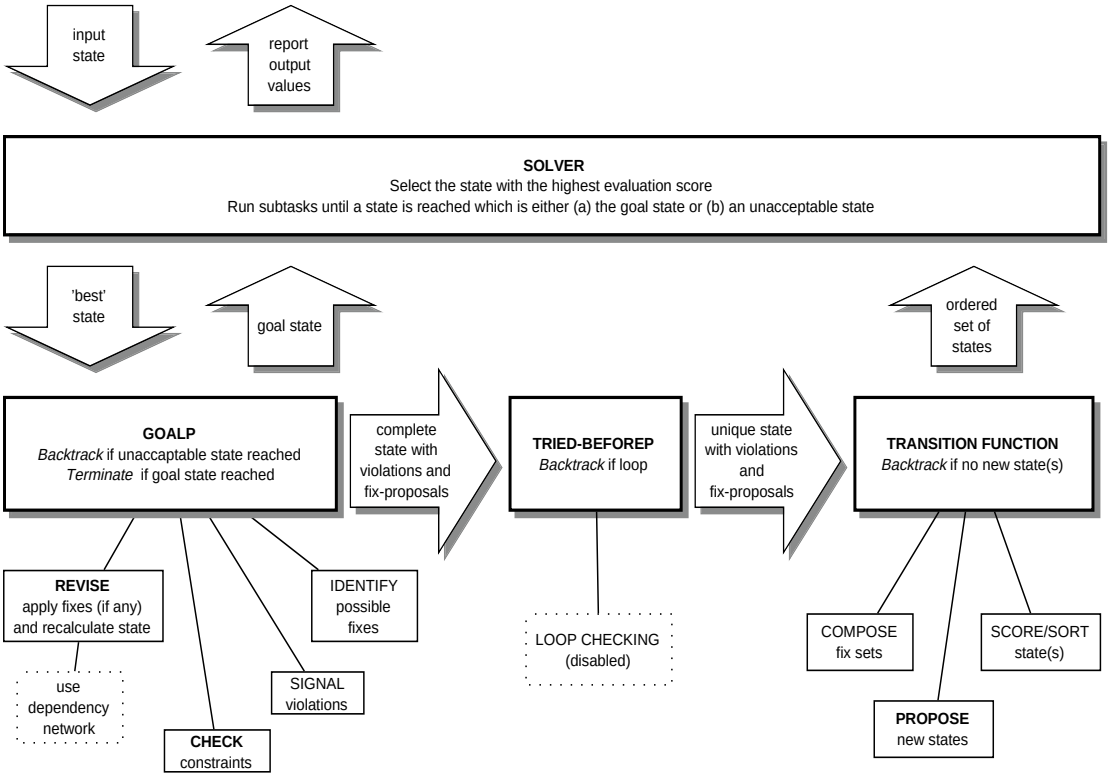

Figure 9: Overview of the configured propose-and-revise method. Arrows denote the flow of states through the recursive application of the main control procedure (SOLVER). The three subtasks (GOALP, TRIED-BEFOREP, and TRANSITION FUNCTION) are further split up in conceptual subparts. Dashed rectangles show modifications to the generic propose-and-revise method (e.g., the use of a dependency network or the skipping of the expensive state-comparison) or the support of with domain-dependent heuristics (e.g., some special procedures to deal with multiple fixes).

A detailed description of the current version of ELVIS' problem-solving method is illustrated in Figure 9. The figure shows the control flow (i.e., the transition of states of parameters and their values) as well as the main processes performed in subtasks. These processes, like *CHECK constraints*, specify the knowledge roles for the domain knowledge. These knowledge roles can be used to guide method selection as well as they can cause knowledge acquisition activities to elicit the required knowledge. The particular order of the subparts is somewhat arbitrary and subject to implementation details because of the recursive nature of the basic algorithm (and because of the way rules are executed in CLIPS). For example, the *REVISE* process (i.e., the application of fixes and the recalculation of parameter values) is actually executed as part of the *GOALP* subtask, but it could as well be located in the *TRANSITION FUNCTION*.

5.3 Mapping Relations

Mapping relations relate the knowledge in the application ontology to the knowledge required by the selected method. These relations are responsible for translating the knowledge instances into a representation that is usable by the configured problem solver. Because this transformation process can vary in complexity, PROTÉGÉ-II supports a number of different types of mapping relations (for more details, see Gennari et al., 1993). In particular, for the elevator-configuration task and the propose-and-revise method, we use the following:

[11] This bookkeeping mechanism, with the help of a dependency network, allowed us to deal with the problem of not updating certain parameters (e.g., the parameters `car.buffer.blocking.height` and `counterweight.bottom.reference` are special cases; see Sections 5.7 and 5.9 of Yost, 1992).

For all instances of the `Slings` class, generate an instance of the `Assign-constraints` class with slot values as follows:

`name:`	`<Model.name>`CrossheadRule
`condition:`	(eq ?Sling.model `<Model.name>`)
`expression:`	`<crosshead-model>`
`variable:`	?Crosshead.model

Figure 10: A filtering mapping that creates assign constraints. These constraints will establish the correct crosshead model for each sling model.

- *Renaming* mappings, where the semantics between method and application classes match, but the slot names need to be translated
- *Filtering* mappings, where the method slots are filled by filtering information from application instances
- *Class* mappings, where method slots are filled from application class definitions, rather than from instances

For example, a simple renaming mapping is used to transform instances of the application ontology class `ELVIS-parameters` to `State-variables` in the method ontology (see Figures 5 and 3). Figure 10 shows an example of a filtering mapping: The assign constraints generated by this relation enforce the rule that each sling model requires a particular crosshead model. The sling and crosshead model names are provided by the domain expert with the knowledge-acquisition tool generated from the application ontology by DASH. Similarly, we used conditional renaming mappings to translate the most types of constraint and fix knowledge as described in Section 4.2 to the knowledge types required by the method ontology. For example, assign fixes (Section 4.2.3) were translated to increase or decrease fixes (see Figure 3) based on the existence and on the sign of the fixed change parameter.

These mapping relations provide instances required by the method ontology. Once the developer has built the mappings, and the domain expert has entered information as application-ontology instances, then the mapping relations are applied by a *mapping interpreter*, and are given as input to the instantiated method (see Figure 2). These mappings allow the developer to adapt an application ontology to arbitrary method ontologies.

The construction of mapping relations can be an expensive process. To make it as easy as possible to build instances of mapping relations, PROTÉGÉ-II includes a tool for guided input and editing of the mapping relations needed to connect application and method ontologies (see Gennari et al., 1993). We expect that the method-configuration tool will allow developers to build a wide variety of mapping relations: mappings from task to submethod, mappings from method to domain (for interpreting the output of the method), and mappings from the application ontology to method ontology. All these mappings must be created by the developers during the method-configuration stage, since the type and number of mapping relations depend on the method ontology, which in turn depends on the knowledge engineer's selection and configuration of methods and mechanisms.

5.4 Run-Time System

The run-time system solves the test case with the data provided in Section 9.1 of Yost (1992). Although meaningful performance measures can not be assessed based on the single test case that is available, we include the following numbers as rough indications. The system takes about 20 seconds to load all the system's components (ontologies, domain knowledge instances, method code, and input values) and about 60 seconds to reach a goal state. The summary of fix applications in Figure 11 shows the principal changes made to the initial values by the propose-and-revise method. As we can see, five fixes were applied in the first state *[gen1]*, and two step fixes changed two parameters in the next 11 states. In the final state *[gen55]*, the resulting values of all parameters were equal to the ones supplied as a solution for the test case.

```
; [gen1] Apply increase fix: opening.to.hoistway.left from 32 to 33
; [gen1] Apply change fix: machine.model := machine28
; [gen1] Apply change fix: hoist.cable.model := hoistcable4_5
; [gen1] Apply change fix: machine.beam.model := s10x35_0
; [gen1] Apply change fix: car.guiderail.model := carguiderail_2
; [gen10] Apply change fix: hoist.cable.model := hoistcable4_625
; [gen16] Apply decrease fix: counterweight.to.platform.rear from 5.25 to 4.75
; [gen20] Apply decrease fix: counterweight.to.platform.rear from 4.75 to 4.25
; [gen24] Apply decrease fix: counterweight.to.platform.rear from 4.25 to 3.75
; [gen28] Apply decrease fix: counterweight.to.platform.rear from 3.75 to 3.25
; [gen32] Apply decrease fix: counterweight.to.platform.rear from 3.25 to 2.75
; [gen36] Apply decrease fix: counterweight.to.platform.rear from 2.75 to 2.25
; [gen40] Apply increase fix: car.supplement.weight from 0 to 100
; [gen43] Apply increase fix: car.supplement.weight from 100 to 200
; [gen46] Apply increase fix: car.supplement.weight from 200 to 300
; [gen49] Apply increase fix: car.supplement.weight from 300 to 400
; [gen52] Apply increase fix: car.supplement.weight from 400 to 500
; [gen55] Goal state reached.
```

Figure 11: Trace extract: application of fixes for the test case.

6 DISCUSSION

We have shown that PROTÉGÉ-II solves the Sisyphus-2 task. Since the PROTÉGÉ-II architecture is designed to facilitate the definition and use of ontologies and to design useful knowledge-acquisition interfaces, we spent more time on performing a thorough knowledge analysis and on configuring an appropriate knowledge-acquisition tool than on refining the details of the problem-solving method. We defined a reusable domain ontology for the elevator-configuration task that provides a basis for building a knowledge-acquisition tool. We modified the domain ontology, creating an application ontology, to accommodate the requirements of the selected problem-solving method, propose-and-revise. PROTÉGÉ-II then generated a domain-specific knowledge-acquisition tool based on the application ontology. We built mapping relations that transformed the domain knowledge instances provided by the domain expert into instances that could be understood by the propose-and-revise method. Finally, using the CLIPS inference engine, we tested the run-time system with the test case provided in Section 9.1 of Yost (1992).

The importance of working with a running system and real-world domain knowledge cannot be understated. At several points during our development of ELVIS, we encountered problems that we could not have predicted had we been working from a theoretical perspective. For example, we could not have anticipated the need for a special upgrade fix and the complexities such a fix introduces in the application ontology, in the method and in the mapping relations. In fact, working with the Sisyphus-2 task forced us to re-evaluate certain aspects of the general-purpose PROTÉGÉ-II architecture. In particular, we are more acutely aware of the problem of *overhead costs* for reuse. That is, before developers can reuse problem-solving methods and domain ontologies, a number of task-specific adaptations may be necessary. In the case of Sisyphus-2, these costs include time spent on method modifications such as a dependency network, and the work needed to adapt the domain ontology to create the application ontology. If these adaptations are particularly time consuming, then there is less benefit from reuse. Our future work with PROTÉGÉ-II will be aimed at reducing these overhead costs.

One drawback of the Sisyphus-2 task is that there was only one source of real-world domain knowledge. Several times during our development efforts, we wondered to what extent the knowledge as presented in the Sisyphus-2 document influenced our knowledge analysis and system building. We profited from the immense amount of knowledge acquisition, knowledge structuring, and verification that must have gone into the production of the Sisyphus-2 document; we used this document not only as the starting point for our knowledge analysis, but also as our reference for domain expertise. However, we are aware of the dangers of this situation. Even if the knowledge described in the Sisyphus-2 document is correct and complete (i.e., if it covers all the possible scenarios and produces feasible solutions), we have no evaluation procedure for the usability of our knowledge-acquisition tools and the usefulness of our performance system. Such an evaluation would require more test cases and a set of evaluation scenarios that include detailed feedback

from domain experts and other system users. Only empirical evidence—collected from the use of ELVIS in everyday, real-world situations—can answer the question of whether this system is of practical use in the given scenarios.

The crucial question for Sisyphus-2 is, of course, to what extent our approach produces reusable and sharable knowledge. Again, only concrete reuse scenarios, such as exchanging knowledge bases and ontologies with other developers, can provide answers to this important question. Even given a real example of reuse, we must have a way of measuring the *effort* required to develop a system that solves the Sisyphus-2 problem. Such a measurement could lead to a quantitative metric of the benefit gained from reuse.

As we have seen in Section 4.1, the definition of what counts as a solution is not straightforward. If we chose a rigid criterion, we would claim that the spreadsheet solution fails to solve the Sisyphus-2 task because it requires user intervention. Likewise, although our production version of ELVIS solves the task correctly, the domain-knowledge base was not constructed exclusively with PROTÉGÉ-II tools (i.e., the instances were translated from the computer files and not by using the knowledge-acquisition tools generated by DASH from the application ontology). However, both solutions provide important improvements over a plain constraint-satisfaction method or over existing, method-based systems. An efficient constraint-satisfaction method may be able to reach a solution state quickly, but may lack an adequate knowledge-acquisition facility to define, maintain, and edit all the domain knowledge. A domain-specific knowledge-acquisition tool is a critical part of any solution: If the system is not easily accepted by the user community of domain experts, then it will not be used.

Our expectation is that the domain-specific tools produced with the PROTÉGÉ-II architecture are suitable and easy to use for domain experts. For example, we believe that our tools are more usable than are those of SALT, the knowledge-acquisition system developed for the original VT task that maps the domain knowledge into an OPS5 production system. To verify this claim, we would need objective measurements of the amount of time spent with different systems. The effort required to develop VT with the SALT knowledge-acquisition tool (Marcus & McDermott, 1989) are estimated to be somewhere near 46 person-hours (although it is unknown how familiar the domain expert was with the SALT system). A similar attempt based on the Soar architecture is reported to have required only 35 hours (Yost, 1993; this Soar-based solution started from knowledge that was almost identical to that of Sisyphus-2). However, this figure is valid only for an expert in the TAQL representation language used by the Soar solution.

Only about 22 hours were spent on an Excel implementation that modeled the configuration system. This figure does not include the amount of time spent rewriting the document, a crucial precondition to the success of the spreadsheet solution. In addition, the Excel solution is incomplete: it includes the calculation of constraint violations and the proposal of fixes, but the automatic selection and subsequent application of fixes seem to be beyond the easily implementable functionalities in a spreadsheet model. This effort seems to be about in the same range as the 19.5 hours spent to complete the part of the TAQL system that is comparable to our Excel solution (Yost, personal communication).

For PROTÉGÉ-II, we cannot evaluate accurately our development effort based on a fully implemented system, since (a) the PROTÉGÉ-II tools are under development and were modified during the construction of the ELVIS system, (b) we did not actually enter all the knowledge into the knowledge-acquisition tool because we lacked a robust instance save and load mechanisms that would have allowed us to modify ontologies and mappings after entering all the domain knowledge, (c) the mapping tool did not provide interactive support to generate all the necessary rules and facts for the knowledge base, and (d) some efforts involved group development and others were performed by individual members. The following figures are thus provided only as rough indications of major development steps—we hope that these figures will encourage other developers to report their efforts and that we can start to define suitable measuring methods. We spent about 20 person-hours for understanding the domain and for drafting and modifying the ELVIS application ontology. The addition of the propose-and-revise method to PROTÉGÉ-II's method library required 19 hours of CLIPS programming (this figure includes the time spent for method configuration, i.e., for optimizing the method's performance to the task at hand). The adaptation of the domain knowledge base (provided as a computer file in Ontolingua format) took 69 hours. This figure includes manual and macro-supported editing of the domain knowledge, transformation into a representation that conforms to the requirements of PROTÉGÉ-II and CLIPS, addition of fix knowledge, as well as debugging, testing and optimizing. These 69 hours represent the effort needed to hand-code the domain knowledge without support from either the PROTÉGÉ-II tools nor from taking full advantage of the reusable Ontolingua representation.

This last step of entering the domain knowledge instances would be shortened considerably with appropriate translators from Ontolingua into MODEL. However, an automated translation would have to rely on the adequacy of the original representation and any addition (e.g., of fix knowledge) would require additional efforts to ensure consistency and correctness. Furthermore, the Ontolingua representation was based on idiosyncratic variable names and did not always follow the written problem specification. Thus, additional effort had to be put into the mapping from Ontolingua back to the plain text and several interpretational ambiguities had to be resolved. However, we believe that—with the full functionality of PROTÉGÉ-II in place—a domain expert *without* experience in programming would spend less time building the application ontology and entering the domain knowledge with the help of the DASH-generated knowledge-acquisition tool than would be required to learn and understand the representational format of either Ontolingua or TAQL. In addition to the time spent by the domain experts, we have also to include the effort required by system developers for proposing a domain ontology, selecting and configuring an appropriate method and constructing the mapping relations. Future research has to show that the use of the PROTÉGÉ-II tools actually shortens the overall system development time compared to the effort needed to build a system from scratch (e.g., an improved version of the spreadsheet solution).

Of course, we need objective, controlled experiments to measure precisely the effort expended on a particular task. In the absence of quantitative measurements, approximate indications of effort are essential; such estimates are currently the only way to assess the usability of the PROTÉGÉ-II architecture. A main objective of future research and tool development is the study of evaluation procedures and the inclusion of measuring tools in the PROTÉGÉ-II architecture. However, only work with real problems and with real domain experts allows us to discover the strengths and weaknesses of PROTÉGÉ-II. Our experience with elevator configuration—a task completely unfamiliar to our developers—has validated the appropriateness of the PROTÉGÉ-II suite of tools. Although the Sisyphus-2 problem has certain shortcomings, we believe that this task has allowed us to evaluate our architecture with a substantial real-world problem.

Support

This work has been supported in part by grant LM05157 from the National Library of Medicine, and by gifts from Digital Equipment Corporation and from the Computer-Based Assessment Project of the American Board of Family Practice. Computing resources were also provided by the Stanford University CAMIS project, which is funded under grant number IP41 LM05305 from the National Library of Medicine of the National Institutes of Health. Preparation of this article was also supported in part by Swiss National Science Foundation grant 8210-028342 to Dr. Rothenfluh. Dr. Musen is recipient of National Science Foundation Young Investigator Award IRI-9257578.

Acknowledgments

The work reported here is part of a collaborative project by the PROTÉGÉ-II research group. We thank the other members of that group for their contributions to the PROTÉGÉ-II architecture. We are indebted to Gregg Yost for producing the comprehensive Sisyphus-2 task description and for providing helpful suggestions on an earlier draft. He also encouraged us to think more about our own attempts to collect and provide comparable quantitative data on development efforts. Discussions with Tom Gruber and Jay Runkel helped us to understand the ontologies for Sisyphus-2; their encoding of the domain knowledge in computer files provided a useful starting point for our prototyping. We also thank Lyn Dupré for editorial assistance.

References

Chandrasekaran, B. (1987). Generic tasks as building blocks for knowledge-based systems: The diagnosis and design examples. *Knowledge Engineering Review*, *4*, 183–219.

Chandrasekaran, B. (1990). Design problem solving: A task analysis. *AI Magazine*, *11*, 59–71.

Eriksson, H., & Musen, M.A. (1993). Metatools for knowledge acquisition. *IEEE Software*, *10*, 23–29.

Eriksson, H., Musen, M.A., Shahar, Y., Puerta, A.R., & Tu, S.W. (1992). *Task modeling with reusable problem-solving methods* (Tech. Rep. No. KSL-92-43). Stanford, CA: Stanford University, Knowledge Systems Laboratory.

Gennari, J.H. (1993). *A brief guide to MAÎTRE and MODEL: An ontology editor and a frame-based knowledge representation language* (Tech. Rep. No. KSL-93-46). Stanford, CA: Stanford University, Knowledge Systems Laboratory.

Gennari, J.H., Tu, S.W., Rothenfluh, T.E., & Musen, M.A. (1993). *Mapping domains to methods in support of reuse.* (Tech. Rep. No. KSL-93-57). Stanford, CA: Stanford University, Knowledge Systems Laboratory.

Gruber, T.R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition, 5*, 199–220.

Gruber, T.R., & Runkel, J. (1993). *Ontolingua files for the elevator-configuration task.* [Machine-readable data files]. Stanford, CA: Stanford University, Knowledge Systems Laboratory.

Hutchins, E. (1991). The social organization of distributed cognition. In L.B. Resnick, J.M. Levine & S.D. Teasley (Eds.), *Perspectives on socially shared cognition* (pp. 283–307). Washington, DC: American Psychological Association.

Marcus, S., & McDermott, J. (1989). SALT: A knowledge acquisition language for propose-and-revise systems. *Artificial Intelligence, 39*, 1–37.

Marcus, S., Stout, J., & McDermott, J. (1988). VT: An expert elevator designer that uses knowledge-directed backtracking. *AI Magazine, 9*, 95–112.

McDermott, J. (1988). Preliminary steps toward a taxonomy of problem-solving methods. In S. Marcus (Ed.), *Automating knowledge acquisition for expert systems* (pp. 225–256). Boston, MA: Kluwer.

Musen, M.A. (1989a). Conceptual models of interactive knowledge-acquisition tools. *Knowledge Acquisition, 1*, 73–88.

Musen, M.A. (1989b). *Automated generation of model-based knowledge-acquisition tools*. San Mateo, CA: Morgan-Kaufmann.

Musen, M.A. (1992). Dimensions of knowledge sharing and reuse. *Computers and Biomedical Research, 25*, 435–467.

Musen, M.A., Fagan, L.M., Combs, D.M., & Shortliffe, E.H. (1987). Use of a domain model to drive an interactive knowledge-acquisition tool. *International Journal of Man–Machine Studies, 26*, 105–121.

Neches, R., Fikes, R., Finin, T., Gruber, T., Senator, T., & Swartout, W. (1991). Enabling technology for knowledge sharing. *AI Magazine, 12*, 36–56.

Puerta, A.R., Egar, J., Tu, S.W., & Musen, M.A. (1992). A multiple-method knowledge acquisition shell for the automatic generation of knowledge-acquisition tools. *Knowledge Acquisition, 4*, 171–196.

Puerta, A.R., Eriksson, H., Gennari, J.A., & Musen, M.A. (1993). *Beyond data models for automated user interface generation* (Tech. Rep. No. KSL-93-57). Stanford, CA: Stanford University, Knowledge Systems Laboratory.

Puerta, A.R., Tu, S.W., & Musen, M.A. (1993). Modeling tasks with mechanisms. *International Journal of Intelligent Systems, 8*, 129–152.

Suchman, L.A. (1987). *Plans and situated actions: The problem of human–machine communication*. Cambridge U.K.: Cambridge University Press.

Tu, S.W., Shahar, Y., Dawes, J., Winkles, J., Puerta, A.R., & Musen, M.A. (1992). A problem-solving model for episodic skeletal-plan refinement. *Knowledge Acquisition, 4*, 197–216.

Walther, E., Eriksson, H., & Musen, M.A. (1992). Plug-and-play: Construction of task-specific expert-system shells using sharable context ontologies. *Proceedings of the AAAI Workshop on Knowledge Representation Aspects of Knowledge Acquisition* (pp. 191–198). San Jose, CA.

Yost, G.R. (1992). *Configuring elevator systems* (Tech. Rep.). Marlboro, MA: Digital Equipment Corporation.

Yost, G. R. (1993). Acquiring knowledge in Soar. *IEEE Expert, 8*, 26–34.

Appendix: Sample Trace

The following text is the output of the production version of ELVIS that we have edited slightly to improve its readability. Repeated text that contains only variations of previously shown information (i.e., new states with different instantiation of constraints and fixes) is shown as points of ellipsis […]. Some trace information and CLIPS-specific loading and display information were deleted to shorten the output. The original sequence of the run-time system is maintained, but is broken up into several sections.

Loading ontologies and code

```
CLIPS> (load clipsfixes.clp) ; CLIPS enhancements
CLIPS> (load method-ont.clp) ; Generic Method
CLIPS> (load method-config.clp) ; Configured Method
CLIPS> (load elvis-system.clp) ; Application domain & instances
CLIPS> (load elvis-dependencies.clp) ; Dependency network
CLIPS> (load elvis-input.clp) ; Runtime input
```

Reading the input values of the test case

```
CLIPS> ; (elvis) ; Start run-time system with [gen0]
; [Reading input values]
; [gen0] Signal violation platform_to_hoistway_left-low
; [gen0] Signal violation hoist_cable_safety_factor-low
; [gen0] Signal violation traction_ratio-high
; [gen0] Signal violation vertical_rail_force-high
; [gen0] Signal violation machine_beam_section_modulus-low
; [gen0] Signal violation machine_groove_pressure-high
; [gen0] Signal violation motor_model-incompatible
;; [gen0] SCORE-FIX-A Best score for viol traction_ratio-high is 3
;; [gen0] SCORE-FIX-A Best score for viol hoist_cable_safety_factor-low is 4
;; [gen0] SCORE-FIX-A Best score for viol platform_to_hoistway_left-low is 6
```

Applying the method

```
> SOLVER ([gen0])
> GOALP [gen0]
> TRIED-BEFOREP [Skipping comparison of states.]
> TRANSITIONFUNC [gen0]
>> DUPLICATE: Generate new state [gen1]
; [gen1] Adding a multi-fix for platform_to_hoistway_left-low
;; 6 increase opening.to.hoistway.left 1
; [gen1] Adding a single-fix for motor_model-incompatible
;; 8 change machine.model machine28
; [gen1] Adding a single-fix for machine_groove_pressure-high
;; 4 change hoist.cable.model hoistcable4_5
; [gen1] Adding a single-fix for machine_beam_section_modulus-low
;; 4 change machine.beam.model s10x35_0
; [gen1] Adding a single-fix for vertical_rail_force-high
;; 4 change car.guiderail.model carguiderail_2
```

Generating new states

```
>> DUPLICATE: Generate new state [gen2]
; [gen2] Adding a multi-fix for platform_to_hoistway_left-low
;; 8 decrease car.return.left 1
; [gen2] Adding a single-fix for motor_model-incompatible
;; 8 change machine.model machine28
; [gen2] Adding a single-fix for machine_groove_pressure-high
;; 4 change hoist.cable.model hoistcable4_5
; [gen2] Adding a single-fix for machine_beam_section_modulus-low
;; 4 change machine.beam.model s10x35_0
; [gen2] Adding a single-fix for vertical_rail_force-high
```

```
;; 4 change car.guiderail.model carguiderail_2

>> DUPLICATE: Generate new state [gen3]
[...]
>> DUPLICATE: Generate new state [gen4]
[...]
>> DUPLICATE: Generate new state [gen5]
[...]
>> DUPLICATE: Generate new state [gen6]
[...]
>> DUPLICATE: Generate new state [gen7]
[...]
>> DUPLICATE: Generate new state [gen8]
; [gen8] Adding a multi-fix for traction_ratio-high
[...]
>> DUPLICATE: Generate new state [gen9]
; [gen9] Adding a single-fix for motor_model-incompatible
;; 8 change machine.model machine28
; [gen9] Adding a single-fix for machine_groove_pressure-high
;; 4 change hoist.cable.model hoistcable4_5
; [gen9] Adding a single-fix for machine_beam_section_modulus-low
;; 4 change machine.beam.model s10x35_0
; [gen9] Adding a single-fix for vertical_rail_force-high
;; 4 change car.guiderail.model carguiderail_2

>> SORT-SCORE of states ([gen1] [gen2] [gen3] [gen4] [gen5] [gen6] [gen7] [gen8]
   [gen9])
<< SORT-SCORE of states ([gen1] [gen2] [gen3] [gen4] [gen5] [gen6] [gen7] [gen8]
   [gen9])
< TRANSITIONFUNC ([gen1] [gen2] [gen3] [gen4] [gen5] [gen6] [gen7] [gen8] [gen9])

> SOLVER ([gen1] [gen2] [gen3] [gen4] [gen5] [gen6] [gen7] [gen8] [gen9])
```

Revising a state

```
> GOALP [gen1]
; [gen1] Enable recomputation of opening.to.hoistway.left and dependents
; [gen1] Apply increase fix: opening.to.hoistway.left from 32 to 33
; [gen1] Enable recomputation of machine.model and dependents
; [gen1] Apply change fix: machine.model := machine28
; [gen1] Enable recomputation of hoist.cable.model and dependents
; [gen1] Apply change fix: hoist.cable.model := hoistcable4_5
; [gen1] Enable recomputation of machine.beam.model and dependents
; [gen1] Apply change fix: machine.beam.model := s10x35_0
; [gen1] Enable recomputation of car.guiderail.model and dependents
; [gen1] Apply change fix: car.guiderail.model := carguiderail_2
; [gen1] Signal violation hoist_cable_safety_factor-low
; [gen1] Signal violation traction_ratio-high
;; [gen1] SCORE-FIX-A Best score for viol traction_ratio-high is 3
;; [gen1] SCORE-FIX-A Best score for viol hoist_cable_safety_factor-low is 4
;; [gen1] SCORE-FIX-A Best score for viol hoist_cable_safety_factor-low is 4
> TRIED-BEFOREP [Skipping comparison of states.]
> TRANSITIONFUNC [gen1]
>> DUPLICATE: Generate new state [gen10]
; [gen10] Adding a multi-fix for hoist_cable_safety_factor-low
;; 4 change hoist.cable.model hoistcable4_625
>> DUPLICATE: Generate new state [gen11]
; [gen11] Adding a multi-fix for hoist_cable_safety_factor-low
[...]
< TRANSITIONFUNC ([gen10] [gen11] [gen12] [gen13] [gen14] [gen15])

> SOLVER ([gen10] [gen11] [gen12] [gen13] [gen14] [gen15])
```

Generating and checking more states

```
> GOALP [gen10]
; [gen10] Enable recomputation of hoist.cable.model and dependents
; [gen10] Apply change fix: hoist.cable.model := hoistcable4_625
; [gen10] Signal violation traction_ratio-high
[...]

> SOLVER ([gen57])
> GOALP [gen57]
; [gen57] Enable recomputation of compensation.cable.model and dependents
; [gen57] Apply change fix: compensation.cable.model := compensationcable1_4chain
; [gen57] Signal violation compensation_cable_unit_weight-high
;; single-fix: 1 change compensation.cable.model compensationcable5_16chain
> TRIED-BEFOREP [Skipping comparison of states.]
> TRANSITIONFUNC [gen57]
>> DUPLICATE: Generate new state [gen58]
; [gen58] Adding a single-fix for compensation_cable_unit_weight-high
;; 1 change compensation.cable.model compensationcable5_16chain
>> SORT-SCORE of states ([gen58])
<< SORT-SCORE of states ([gen58])
< TRANSITIONFUNC ([gen58])

> SOLVER ([gen58])
```

Showing the solution state

```
> GOALP [gen58]
; [gen58] Enable recomputation of compensation.cable.model and dependents
; [gen58] Apply change fix: compensation.cable.model :=
    compensationcable5_16chain
 [gen58] Goal state reached.

[gen58]
```

Reporting the output parameters of the solution state

```
CLIPS> (report-variables t [gen58])

hoistway.bracket.spacing(165)
car.buffer.model(carbuffer_oh1)
car.buffer.blocking.height(18)
car.buffer.footing.channel.height(3.5)
car.guiderail.unit.weight(11)
car.supplement.weight(500)
compensation.cable.model(compensationcable5_16chain)
compensation.cable.quantity(2)
compensation.cable.length(993)
controlcable.controlcablewtperinch(0.167)
crosshead.model(crosshead_w8x18)
counterweight.between.guiderails(28)
counterweight.frame.weight(508.75)
counterweight.frame.thickness(31)
counterweight.plate.depth(7)
counterweight.plate.quantity(91)
counterweight.buffer.model(counterwtbuffer_oh1)
counterweight.buffer.quantity(1)
counterweight.buffer.blocking.height(0)
deflector.sheave.model(ds_25)
door.model(door_2sso_rh)
governor.cable.diameter(0.375)
governor.cable.length(2130)
hoist.cable.model(hoistcable4_625)
```

```
hoist.cable.length(1058.96)
machine.beam.model(s10x35_0)
machine.beam.length(124)
machine.beam.load.front.left(8529.4918670544484)
machine.beam.load.front.right(8529.4918670544484)
machine.beam.load.rear.left(6060.7244190455522)
machine.beam.load.rear.right(6060.7244190455522)
machine.model(machine28)
machine.gear.ratio.symbolic(55:1)
machine.groove.model(msheavegroove_k3269)
motor.model(motor20hp)
motor.generator.model(motgen_46_230v)
platform.model(platform_4b)
safety.model(safety_b1)
safety.beam.between.guiderails(72.25)
sling.model(sling_25b_18)
sling.stile.length(130.94)
sling.underbeam.space(21)
car.cable.hitch.to.counterweight.cable.hitch(51.75)
car.cable.hitch.to.platform.front(38.0)
car.return.left(25)
car.return.right(3)
counterweight.bottom.reference(55.539999999999999)
counterweight.to.hoistway.rear(9.0)
door.space(6.5)
platform.running.clearance(1.25)
platform.to.hoistway.front(7.75)
car.cab.height(96)
car.capacity(3000)
car.intercom(FALSE)
car.lantern(FALSE)
car.phone(TRUE)
car.position.indicator(TRUE)
door.opening.type(side)
door.speed(double)
hoistway.floor.height(165)
hoistway.depth(110)
machine.beam.support.front.to.hoistway(3)
hoistway.width(90)
machine.beam.support.distance(118)
machine.beam.support.type(pocket)
machine.beam.support.bottom.to.machine.room.top(16)
opening.height(84)
door.opening.strike.side(right)
opening.to.hoistway.left(33)
opening.width(42)
opening.count(6)
hoistway.overhead(192)
hoistway.pit.depth(72)
platform.depth(84)
platform.width(70)
car.speed(250)
hoistway.travel(729)
CLIPS> [...]
```

KSL Technical Report

Cover Page Information

Tech Rep. No. KSL-93-65

Revision December 1993

Title Reusable Ontologies, Knowledge-Acquisition Tools, and Performance Systems: PROTÉGÉ-II Solutions to Sisyphus-2

Authors Thomas E. Rothenfluh, John H. Gennari, Henrik Eriksson, Angel R. Puerta, Samson W. Tu, Mark A. Musen

Affiliation Medical Computer Science Group
Knowledge Systems Laboratory
Stanford University School of Medicine
Stanford, California 94305-5479
U.S.A.

Email protege-staff@camis.stanford.edu

Additional info Paper to be presented at the Knowledge Acquisition Workshop 1994 (KAW-94), January 30–February 4, 1994. Banff, Canada.