Reuse with PROTÉGÉ-II: Adapting Problem-Solving Methods with Mapping Relations

John H. Gennari, Russ B. Altman, and Mark A. Musen

Section on Medical Informatics
Knowledge Systems Laboratory
Stanford University School of Medicine
Stanford, CA 94305-5479, U.S.A

<gennari, altman, musen@camis.stanford.edu>
URL: <http://camis.stanford.edu/protege/>

Abstract[1]

This paper describes the PROTÉGÉ-II architecture for the construction of knowledge-based systems from reusable components: problem-solving methods and knowledge bases. We argue that these components are easier to reuse when their terminologies are described as formal ontologies. We define declarative *mapping relations* that we use to connect pre-existing methods to new domains and knowledge bases. With PROTÉGÉ-II and a set of mapping relations, we are able to reuse the same problem-solving method with two disparate tasks: (1) configuring the parts of an elevator system and (2) identifying plausible configurations of helices in a ribosome molecular strand.

1. Reuse for Knowledge-Based Systems

Software reuse is an appealing solution to the high cost of software construction and maintenance: If a library of reusable software components were available, then developers could use this library to greatly reduce software development time and effort. Since the goal of software reuse is to reduce development *cost*, it is valuable to view reuse from an economic perspective. Thus, the effort needed to build a software component library is the reuse *investment cost*, and the return on that investment is measured by the savings in effort achieved by exploiting reuse over the lifetime of each component. The *benefit* from a single instance of reuse is the difference between development costs with reuse and estimated development costs without reuse. Reuse is successful only when these benefits outweigh the investment costs.

Barnes and Bollinger (1991) outline three ways to make reuse more cost-effective: (1) reduce the initial investment cost of constructing the component; (2) increase the number of times a component is reused; and (3) reduce the cost of selecting, adapting and reusing a component. In this paper, we focus on the third approach, and especially on the cost of adapting a pre-existing component.

We present PROTÉGÉ-II, a development environment and methodology for the construction of knowledge-based systems with reusable components. This environment has been developed within the knowledge-acquisition research community. Thus, it is designed to help developers build systems that include both a *knowledge base* of domain information, and a *problem-solving method* that operates on that knowledge base. For our purposes, these two types of components are the objects for reuse. In particular, we demonstrate the reuse of a problem-solving method across two domains: configuring the parts of an elevator system and finding plausible models for the positions of helices within a ribosome strand.

The elevator-configuration task is a well-studied problem in the knowledge-acquisition research community, originally described and solved by Marcus, Stout, and McDermott (1988). The task is a constraint-satisfaction problem: given a set of building specifications and requirements such as elevator speed and capacity, and given a large body of knowledge about available elevator components and safety constraints, find a configuration of elevator components so that no constraints are violated. This task was chosen for the Sisyphus-2 project: a benchmark for comparing knowledge modeling efforts in the knowledge-acquisition research community. The PROTÉGÉ-II solution to this problem has been described in detail by Rothenfluh, Gennari, Eriksson, and Musen (1994).

The ribosome topology task is another type of constraint-satisfaction problem, but in a very different domain. Given information about the secondary structure of components of the ribosome structure, and distance-constraint information among those components, the task is to locate the position and orientation of those components, relative to a set of known proteins, such that no distance constraints are violated. This problem has been described by Altman, Weiser, and Noller (1994).

These two constraint-satisfaction problems are clearly very different in terminology, and notably different in the size of their search space. Thus, this pair of problems should be a good testbed for software reuse: if a solution can be constructed to solve one problem, it should be adaptable to solve the other. As we will show, PROTÉGÉ-II allows developers to minimize adaptation costs when reusing a problem-solving method.

2. PROTÉGÉ-II: An Environment for Reuse

PROTÉGÉ-II (Puerta, Egar, Tu, & Musen, 1992) can be viewed as a *software architecture* (as in Krueger, 1992) for reuse: an architecture and methodology that make it easier for developers to access and use a library of pre-existing components. For PROTÉGÉ-II, there are two types of components: (1) declarative domain knowledge that may be reused

1. This paper is an edited version of Gennari, Altman and Musen, 1995.

across different application tasks and (2) procedural problem-solving knowledge, such as a problem-solving method, that may be reused across different domains. For example, the domain knowledge in the ribosome task might include facts about the size of helices, a particular distance constraint, or the list of candidate locations of some helix. The problem-solving method is knowledge about what to do with this data: how to manipulate it to solve some task, such as determining helix positions.

Both types of knowledge may be expressed as declarative ontologies. An *ontology* is simply a model of some domain of knowledge; more formally, it is a partial specification of the universe of discourse (Gruber, 1993; Guarino & Giaretta, 1995). Thus, a domain ontology is an explicit list and organization of all the terms, relations and objects that constitute the representational scheme for that domain. Likewise, a method ontology specifies the terms and the inputs and outputs of the problem-solving method. In either case, building an ontology is part of building a model or constructing an abstraction for some knowledge or process. These ontologies are not static, easily defined objects, and are sometimes necessarily incomplete. Nonetheless, a formal description of the representational vocabulary of a reuse component is essential for knowledge reuse. As Krueger (1992) states, "abstraction is *the* essential feature in any reuse technique;" only by understanding the meaning of the terms in the ontology can a developer hope to reuse the corresponding component.

If we wish to support the reuse of both problem-solving methods and domain knowledge bases, then we must provide a way for developers to connect these two components of an application. If methods and domain knowledge bases are both built to be reusable, then we can connect these components into a working application only by adapting and custom-tailoring one of the components or by introducing new elements that developers can use to connect method and domain ontologies. As shown in Figure 1, we explore the latter option, and we call the bridging elements *mapping relations* (Gennari, Tu, Rothenfluh, & Musen, 1994). Mapping relations are a critical PROTÉGÉ-II feature for enabling reuse. Unlike other software reuse repositories that allow and expect developers to make arbitrary adaptations to a selected component, we demand that all modifications be made explicit via mapping relations.

Thus, PROTÉGÉ-II helps developers build knowledge-based systems that include: (1) method-independent domain knowledge, (2) domain-independent methods, and (3) a set of mapping relations that encode any adaptations of these components necessary to create a running system. Figure 2 provides an overview of the processes, objects, and tools included in PROTÉGÉ-II. Our environment includes four tools arranged in the inner ring of the control panel: MAÎTRE, the ontology editor, DASH, the layout meta-tool, MEDITOR, the knowledge-acquisition tool interpreter, and MARBLE, the mapping relation builder tool. Proceeding clockwise around the ring, developers using this environment build and iteratively modify a set of declarative objects that specify the knowledge-acquisition tool, the knowledge base, the mapping relations, and ultimately, the knowledge-based system.

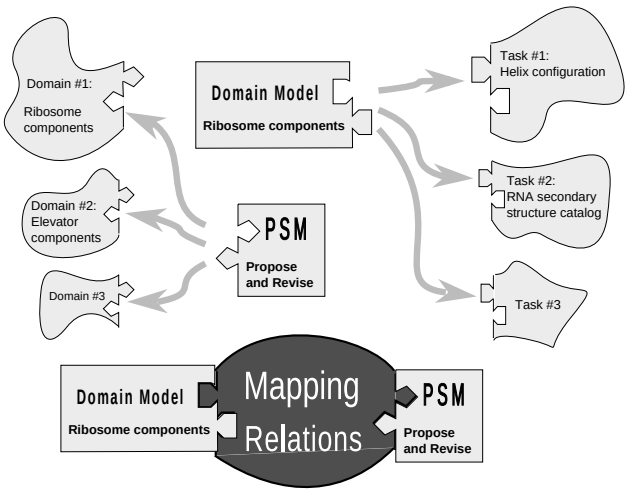

Figure 1. Mapping relations facilitate reuse; they connect problem-solving methods (PSMs) to domain knowledge. If the domain knowledge is method-independent and the PSM is domain-independent, then their inputs and outputs cannot be connected without some bridging relations.

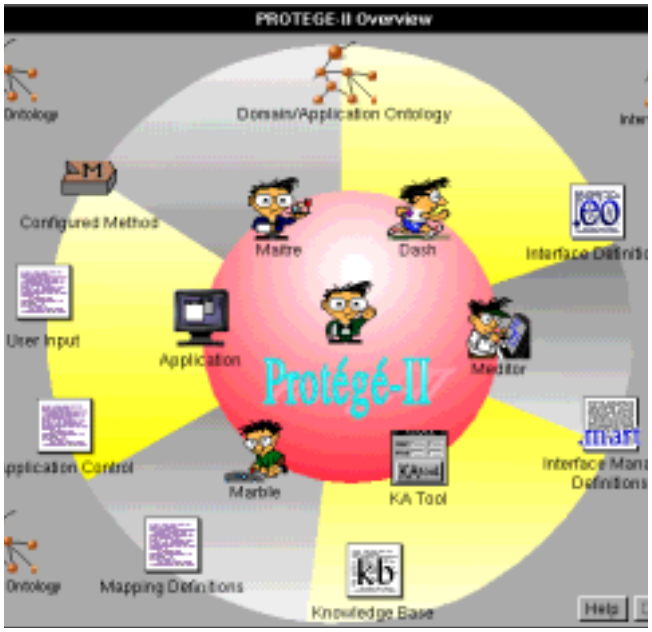

Figure 2. The PROTÉGÉ-II overview and control panel. Tools and products are arranged in the inner ring; the inputs and outputs of those tools are on the outer ring. Development of knowledge-based systems is iterative, and proceeds clockwise around the ring.

We have designed PROTÉGÉ-II for two types of users: domain experts with little or no programming-level knowledge, and developers. Beginning with MAÎTRE, the domain expert provides an initial domain ontology, while the developer selects an appropriate problem-solving method. Next, both work together to create an *application ontology*; this ontology includes knowledge required by the problem-solving method as well as terminology from the domain ontology. This ontology is input to DASH, a meta-tool for

building knowledge-acquisition tools (Eriksson & Musen, 1993). Developer use this system to construct and modify iteratively a domain-specific knowledge-acquisition tool. The resulting tool elicits knowledge in domain-specific terms, allowing a domain expert to create the knowledge base for the given task. Finally, the developer and the domain expert connect the terms in the domain ontology with the requirements specified by the problem-solving method ontology. This is accomplished via mapping relations constructed with the MARBLE tool, which we describe in Section 5.

The ability of developers to represent formally the ontology of either a domain or a problem-solving method is critical to the success of PROTÉGÉ-II as an architecture for reuse. Thus, ontologies should be easy to build and maintain, easy to retrieve and reuse, and also have sufficient expressive power so that a variety of knowledge can be easily incorporated. As might be expected, developing a knowledge representation language that fills these requirements is an ongoing research effort. The Ontolingua language (Gruber, 1993) is one effort to develop a good knowledge representation language for ontology development and sharing.

Rather than attempting to describe the PROTÉGÉ-II methodology in the abstract, we next show exactly how it has been used in the two tasks mentioned earlier: elevator configuration and ribosome topology, applying the problem-solving method known as "propose-and-revise" (Marcus et al., 1988). After describing the method, we discuss the process of building domain and application ontologies, in both the elevator configuration application task, and the ribosome topology application task. Finally, Section 5 describes the mapping relations that connect the application ontologies to the method ontology for propose-and-revise.

3. The Propose-and-Revise Problem-Solving Method

Problem-solving methods are domain-independent and reusable to varying degrees. At a minimum, a developer should identify a domain-independent vocabulary that specifies the requirements of the problem-solving method; in PROTÉGÉ-II, this is the method ontology. The propose-and-revise problem-solving method can be viewed as a simple type of state-space search: (1) an initial solution is proposed, (2) if no constraints are violated, post the solution and quit; otherwise, (3) choose the best fix for a violated constraint, and (4) revise the solution by applying the fix, and return to step (2). The method ontology for our implementation of this algorithm specifies its inputs requirements; see Figure 3. The inputs needed for propose-and-revise are (1) a set of constraints that must be satisfied, (2) a set of fixes to correct violated constraints, and (3) a set of state variables that specify the parameters of the solution and run-time inputs.

The construction of this sort of domain-independent ontology for a given method is part of the investment cost required to make the method reusable. Development of these ontologies may be non-trivial, but this work allows other developers to reuse a problem-solving method. For example, the method ontology in Figure 3 is exactly the target of the mapping relations. As we will see in Section 5, we will map both elevator configuration terminology and ribosome topology terminology to this method ontology. Before defining these mapping relations, we must establish the domain vocabularies of the respective application tasks.

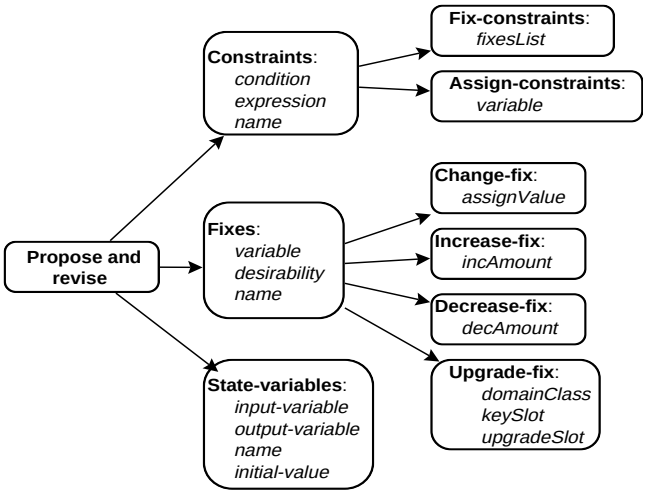

Figure 3. The propose-and-revise method ontology.

4. Building Domain and Application Ontologies

In PROTÉGÉ-II, we distinguish a domain ontology—a task-independent, declarative description of a domain—from an application ontology that may include some method-dependent knowledge. Thus, an application ontology is an augmentation of the domain ontology with information needed by the problem-solving method. This ontology is not the same as the method ontology: The application ontology should use domain-specific terminology, while the method ontology is domain-independent. Recall that the application ontology is the input to the DASH metatool, which produces a knowledge-acquisition tool that uses terminology familiar to domain experts (see Section 2).

4.1 The Elevator Configuration Domain

The task of elevator configuration was a real-world problem, originally defined by interviewing a group of engineers at Westinghouse Elevator Company assigned to produce a specification of elevator parts and dimensions that satisfied both customer specifications and mandated safety constraints. Their task description included an enumeration of available elevator components, a large list of constraints (both safety and simple physical constraints), and information about what action to take when particular constraints were violated. The task description became further codified for the Sisyphus-2 project (see Yost, 1992). Unfortunately, domain experts for this problem are no longer available, and the problem description as it was characterized by Yost is definitely a mix of method and domain knowledge. Thus, it is hard to know what a "method-independent" elevator ontology might look like.

Figure 4 shows our application ontology for this domain; for simplicity, only class names are shown. In this ontology, the `ELVIS-Components` and `ELVIS-Models` subtrees might correspond to a domain model, while classes such as `AssignConstraints` and `StepFixes` are application ontology augmentations needed for the propose-and-revise

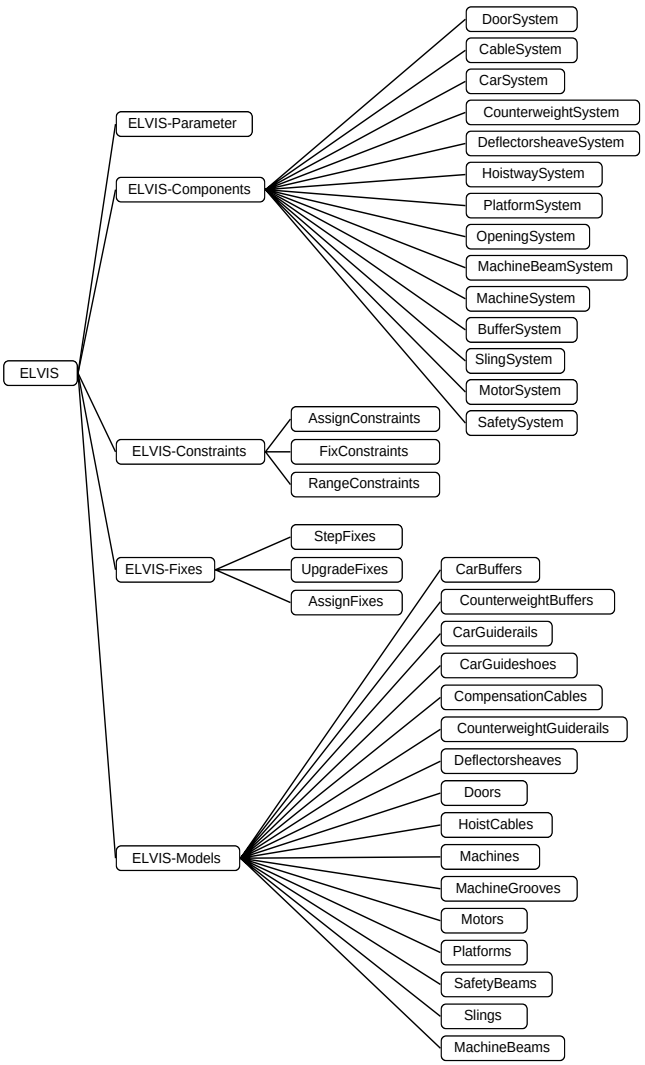

Figure 4. Our application ontology for the elevator configuratio domain and the propose-and-revise problem-solving method.

method. Each of the 14 "systems" underneath `ELVIS-Components` includes a slot for lists of system-specific constraints, parameters, and model information. This implies that each instance of a constraint is not only classified as one of three subclasses of `ELVIS-Constraints`, but is also associated with one of the 14 systems defined by the application ontology. This organization is completely unnecessary for the problem-solving method; this is an example of domain-level information that makes the knowledge-acquisition tool easier to use.

Much of the effort in modeling the elevator domain has to do with the complexity of the domain: with 50 fix-constraints and over 150 assign-constraints, it is hard to know what will happen when a single parameter value is changed. However, because fix-constraints include an ordered list of fixes, the system need not search very long before finding a solution. In other words, although the transition function between states is very complex and difficult to compute, the search space of possible states is not large. As we will see,

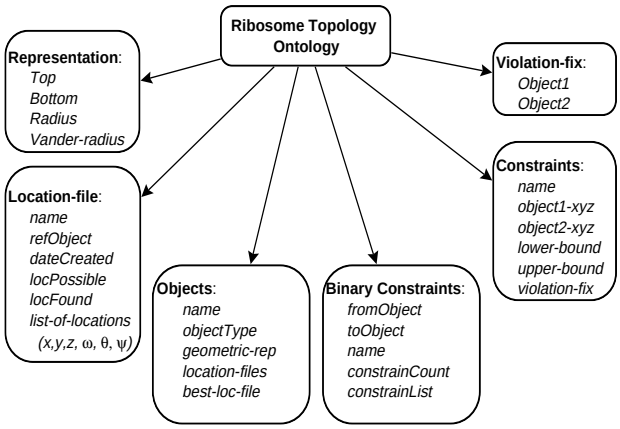

Figure 5. Our application ontology for the ribosome topology domain and the propose-and-revise problem-solving method.

this characterization is significantly different from the task of ribosome topology.

4.2 The Ribosome Topology Domain

In molecular biology, understanding the three-dimensional shape and structure of a ribosome may be critical to understanding its function. A ribosomal subunit of interest (the 30s subunit in prokaryotes) is made of a single chain of RNA bases and a set of 21 unconnected proteins. Current experimental techniques provide four types of information. First, they provide the location in three dimensions of the 21 proteins; this information is used to define a global anchor coordinate system. Second, they provide the primary sequence of RNA bases that form the long chain of connected subunits. Third, they provide the location in the primary sequence of geometric components (secondary structures, such as double helices and coils) that are made of subsets of the RNA bases, and have a known, regular structure. Finally, they provide distance constraints between the components and the fixed proteins, as well as among the components themselves. Thus, given (1) the location of anchoring proteins, (2) information about a set of secondary structures in the ribosome (in this case, 10 helices), and (3) distance constraints between the components and proteins, the task is to find sets of locations and orientations for each component such that no distance constraint is violated.

This domain is notable in the size of its search space: In the original specification, each of the 10 helices had on the order of 10^4 possible locations, giving a total search space of more than 10^{38} possibilities. However, through preprocessing that applies some of the constraint information, and through sampling of location lists, this search space can be reduced to a manageable size of roughly 10^{10} possibilities. Finally, this space is fairly densely populated with solutions—about 10^6 consistent locations (see Altman et al., 1994, for details).

Figure 5 shows our application ontology for this domain; there were only a few augmentations necessary for the propose-and-revise method. The most significant of these was required because the ribosome task does not include any explicit representation of constraint-fixes. Therefore, we augmented the domain ontology with the class `Violation-`

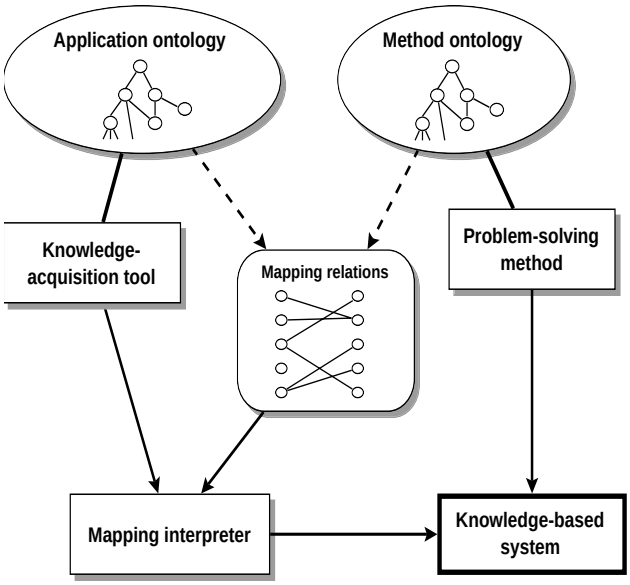

Figure 6. Resolution of differences between method and application ontologies via mapping relations that connect these ontologies. Solid arrows indicate automatic data-flow, while the dashed arrows indicate that the developer must hand-craft the mapping relations.

`fix`, which makes explicit that when a constraint is violated, the "fix" is to try a new location for one or the other helices involved in the binary constraint. Obviously, this type of fix is simpler than fixes in the elevator configuration task. This is one example of the challenge for mapping relations—ribosome fixes and elevator fixes must both be mapped onto the method ontology, where there are four different types of fixes defined (see Figure 3).

5. Specifying Mapping Relations

When information is used by more than one application, it often needs to be modified, transformed, or massaged to fit the different requirements of these applications. For example, by comparing Figures 3, 4 and 5, it is clear that the method and application ontologies have somewhat different definitions of a constraint. Our problem-solving method expects two classes of constraints—assign-constraints that make an assignment to some state variable, and fix-constraints that includes fixes, or actions to take when the constraint is violated. In the elevator configuration task, engineers typically use range-constraints: constraints that specify lower and upper bounds on some variable. In PROTÉGÉ-II, we use mapping relations to view each elevator range constraint as two distinct fix-constraints in the language of the problem-solving method (one fix-constraint each for the lower and upper bounds). This same sort of transformation must occur in the ribosome topology task, since a constraint as defined in the application ontology (Figure 5) includes upper and lower bounds.

Figure 6 is a more detailed picture of the idea of mapping relations; it shows our approach to resolving differences between application and method ontologies. By formally specifying the application and method ontologies, the role of mapping relations is more concrete: Although the application ontology has been augmented to cover all knowledge requirements of the method, it still necessarily includes domain-specific terminology. To allow the problem-solving method to access this domain-specific knowledge base, the information must be renamed or rearranged to match the concepts specified in the method ontology exactly. We resolve this mismatch with declarative mapping relations and a mapping interpreter, an engine that processes the mappings to provide a new view of the knowledge base. Assuming an appropriate set of mappings, this new view can be used by the problem-solving method. Thus, any and all effort needed to adapt the method to a new problem is isolated to the mapping relations, and the reusable method code is not modified.

5.1 An Ontology of Mappings and MARBLE

We claim that the layer of mapping relations is a critical part of an architecture for reuse; this layer isolates application-specific customizations from the components. Nonetheless, constructing a set of mapping relations is part of the overhead cost associated with reuse. To keep this cost at a minimum, we must keep mapping relations simple, and we must provide an editing tool to make their construction easy. For both these reasons, we need a formal, abstract definition for all mapping relations; i.e., we need a *mappings ontology*.

Currently our mappings ontology includes three basic types of mapping relations: (1) renaming mappings, where the semantics between method and application classes match, but the slot names need to be translated; (2) filtering mappings, where the method slots are filled by filtering information from application instances; and (3) class mappings, where method slots are filled from application class definitions rather than from instances. We can use our mappings ontology as input to DASH, and thereby generate a knowledge-acquisition tool for the mapping relations themselves. Currently, this PROTÉGÉ-II–generated tool is our working prototype for MARBLE, the tool for assisting developers in the construction of mapping relations. Although this tool is not ideal, it has proved useful in several domains, including both the elevator configuration and ribosome topology problems. In the next two subsections, we will show examples of mappings and how they are used to transform knowledge from our two application tasks to the requirements of the propose-and-revise problem-solving method.

5.2 Mappings for the Elevator Configuration Task

Mapping the elevator application ontology to the propose-and-revise ontology is relatively straightforward; in this case, it is fairly certain that the problem-solving method is appropriate for the task since, to some degree, the method was originally designed for this domain. For example, the application ontology includes "parameters," which clearly serve the role of "state-variables" in the method ontology. This mismatch can be resolved with a simple renaming mapping.

The mapping between `RangeConstraints` in the elevator ontology and `Fix-constraints` in the method ontology is more complex. Rather than a one-to-one correspondence between slots, filtering mappings can combine

∀ *x*, where *x* is a member of the `RangeConstraints` class,
∃ *y*, a member of the `Fix-constraints` class, such that:

(name *y*)	equals	(constraint-name *x*)
(condition *y*)	equals	(constraint.condition *x*)
(fixesList *y*)	equals	(constraint.lower.fixes *x*)
(expression *y*)	equals	(catenate ">" (constraint.lower.value *x*) (constraint.variable *x*))

Figure 7. A filtering mapping relation from `RangeConstraints` in the elevator application ontology to `Fix-constraints` in the method ontology. In our notation, (<identifier> *x*) is an accessor function to retrieve the slot value of the slot named <identifier> in instance *x*.

multiple source slot values onto a single target slot. Figure 7 presents a formal description of the filtering mapping for the lower bound of the `RangeConstraints` class; there is another, symmetric mapping to deal with the upper bound. We should emphasize that developers do not work with mapping relations at this formal level—instead, they manipulate and edit mappings with MARBLE, and the mappings are stored as instances of the classes defined in the mappings ontology. The mapping interpreter (see Figure 6) then applies these declarative mappings to transform instances from domain to method knowledge bases.

5.3 Mappings for the Ribosome Topology Task

To apply propose-and-revise to the task of determining ribosome topology, and to build the appropriate mapping relations, is less straightforward than adapting the method to the elevator task. For example, "trying a new location for a helix" must be mapped into a type of fix applied in response to a constraint violation. Unfortunately, the list of possible locations for each helix does not include any desirability information; without additional knowledge, the best that this problem-solving method can do is to try the next location in the list. This view of the task also requires filtering mappings to create `State-variables` from each helix, indicating (a) the current location of that helix, and (b) a location-pointer that shows the position of the current location in the list of possible locations associated with that helix. As with the elevator task, the distance constraints are mapped into two fix-constraints, but here, the fix is always to increment the location-pointer for some helix.

As we built up the set of mapping relations, we discovered trade-offs between the complexity of the mapping relations and the simplicity of the design of the knowledge-acquisition tool. For example, we needed to build a relatively complex mapping to guarantee that the location-pointer for a helix never exceeded the number of possible locations for that helix. This filtering mapping would have been simpler if the `Objects` class had included the number of locations as a slot, rather than that information being in the `Location-file` class (see Figure 5). However, this modification would have resulted in a less convenient knowledge-acquisition tool, requiring users to duplicate information in two classes. A similar design decision involved the addition of the class `Violation-fix` into the application ontology. This class is not needed in the domain ontology, but in this case, we decided that by requiring a small amount of inconvenience at knowledge-acquisition time, we would save a great deal of effort at mapping construction time.

6. Conclusions

Our approach to software reuse has been driven by real-world examples—by working with test cases such as described here, we believe we can articulate a theory of mapping relations in support of reuse. With the mapping relations described in Section 5, we were able to apply the same problem-solving method (the same software module) to find solutions to both tasks: elevator configuration and ribosome topology. As should be expected, the propose-and-revise method was most successful with the elevator configuration problem, where the search space is relatively small and where the algorithm can terminate after finding a single solution. With the ribosome problem, the number of valid solutions and the size of the search space meant that, for practical reasons, we could not use this problem-solving method to find all possible solutions. This is appropriate—the method is designed to find a single solution, and modifying it to find all possible solutions is a significant change, suggesting that perhaps developers should use an alternative problem-solving method.

In general, there is a basic tradeoff between reusability and efficiency; if developers wish to use pre-existing software modules to reduce development and maintenance effort, they may pay some price in the efficiency of the final system. Although efficiency is important in the ribosome topology problem, there are many applications where development time and maintenance effort are of greater concern. For tasks where fast prototyping is valuable, the use of an architecture such as PROTÉGÉ-II and the construction of mapping relations to connect components should lead to successful reuse, where the benefit gained is greater than the investment costs.

Without an architecture in support of reuse, and without mapping relations, "reusable" problem-solving methods such as propose-and-revise are not really very reusable. The propose-and-revise method as used by the SALT system for the elevator configuration task was originally described by Marcus and McDermott in 1986, yet the only effort to reuse the system for a different application task was described in 1987 (Stout, Caplain, Marcus, & McDermott, 1990). This effort applied the method to the task of scheduling workshop tasks, and the authors reported that there was "a significant amount of effort" required to adopt the method to the new domain. Our results here suggest that this overhead cost was too high because of the lack of formal ontologies, mapping relations, and an environment for reuse.

We expect that further PROTÉGÉ-II tool development could further decrease adaptation costs for reuse. In particular, the task of adapting components via mapping relations could become less expensive with an improved, more intelligent mapping builder tool. A version of MARBLE that included techniques of apprentice or case-based learning should be able to suggest plausible mappings to developers as they adapt domains to problem-solving methods. A better version of MARBLE also depends on the development of a

better mappings ontology. As we gain experience with different reuse scenarios, we should be able more precisely to define and constrain the set of legal mapping relations.

The benefit of software reuse is that maintenance and development costs are lower because of the ability to adapt pre-existing software modules to new applications. The cost of that adaptation process is critical to the success of reuse. If the adaptation is relatively simple, well-defined, and isolated from the software module, then reuse is more likely to lead to a real savings in effort. A mappings interpreter and a mappings builder tool (both of which require a well-defined mappings ontology) provide developers with the tools needed to achieve exactly those goals: simple, isolated and well-defined adaptations of software for a new purpose.

Acknowledgments

This work has been supported in part by grants LM05157, LM05652, and LM05208 from the National Library of Medicine, by support from the Advanced Research Projects Agency (NRAD contract #N66001-94-D-605), and by gifts from Digital Equipment Corporation. Dr. Musen is the recipient of National Science Foundation Young Investigator Award IRI-9257578. Dr. Altman is a Culpeper Medical Scholar.

This work is a collaborative project by the PROTÉGÉ-II research group. We would like thank Henrik Eriksson, Thomas Rothenfluh, Samson Tu, and Angel Puerta for their comments and discussion.

Appendix: Describing the contribution

A. Reasoning framework

1. What is the reasoning framework?

We consider the reasoning framework to be part of the problem-solving method. Our architecture and approach for reuse is independent of choices made for the problem-solving method. Thus, we are independent of task and domain, and our aim is to characterize and model tasks and domains in such a way as to facilitate their reuse in the future.

5. What are the roles of adaptation, knowledge, and reuse in your approach?

In our approach, the developer is responsible for modeling the knowledge needed in both a declarative description of the domain, and a procedural description of the problem-solving method. Currently, PROTÉGÉ-II includes tools for building models that use a frame-based knowledge-representation language. These tools include the generation of domain-specific knowledge-acquisition tools (see Eriksson & Musen, 1993).

The mapping relations we use can be viewed as a way of adapting either the problem-solving method or the domain knowledge. More accurately, neither the method nor domain knowledge is modified. Instead, mapping relations bridge the differences between the two, allowing the problem-solving method to view domain knowledge in its own terms (and vice versa). Because the original problem-solving code is not modified, it can then more easily be reused in different domains.

C. Evaluation

1. What hypotheses were explored?

Our hypothesis is that the use of mapping relations lowers the cost of adapting an existing component (a method or domain theory) to a new situation. As a corollary, we expect that a theory and model of mapping relations (a mappings ontology) will enable us to build tools to more efficiently allow developers to construct mappings.

2. What type of evaluation?

At present, we are testing our ideas in a variety of domains and settings for reuse and this can be considered a very initial form of evaluation. To date, our work is not yet mature enough for quantitative, empirical evaluation. However, we envision experiments that measure development time using PROTÉGÉ-II and mapping relations versus development time building the product "from scratch", without any reuse. Note that this measures only part of the benefit of reuse—investment costs are recovered over the entire lifetime of a component, and over the maintenance cost as well as the development costs of the product.

References

Altman, R. B., Weiser, B., and Noller, H. F. (1994). Constraint satisfaction techniques for modeling large complexes: Application to central domain of the 16s ribosomal subunit. *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, (pp. 10–18). Stanford, CA.

Barnes, B. H., and Bollinger, T. B. (1991). Making reuse cost-effective. *IEEE Software*, **8**(1), 13–24.

Eriksson, H., and Musen, M. (1993). Metatools for knowledge acquisition. *IEEE Software*, **10**(3), 23–29.

Gennari, J. H., Altman, R. B., and Musen, M. A. (1995). Reuse with PROTÉGÉ-II: From elevators to ribosomes. *Proceedings of the Symposium on Software Reuse*, (pp. 72–80). Seattle, WA.

Gennari, J. H., Tu, S. W., Rothenfluh, T. E., and Musen, M. A. (1994). Mapping domains to methods in support of reuse. *International Journal of Human-Computer Studies*, **41**, 399–424.

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

Guarino, N., and Giaretta, P. (1995). Ontologies and knowledge bases: Toward a terminological clarification. In N.J.I. Mars (ed.), *Towards Very Large Knowledge Bases*, IOS Press, pp. 25–32.

Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys*, **24**(2), 131–183.

Marcus, S., Stout, J., and McDermott, J. (1988). VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine*, **9**(1), 95–112.

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

Rothenfluh, T. E., Gennari, J. H., Eriksson, H., Puerta, A. R., Tu, S. W., and Musen, M. A. (1994). Reusable ontologies, knowledge-acquisition tools, and performance systems: PROTÉGÉ-II solutions to Sisyphus-2. *Proceedings of the Eighth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop* (pp. 43.1–43.30) , Banff, Canada.

Stout, J., Caplain, G., Marcus, S. and McDermott, J. (1990). Toward automating recognition of differing problem-solving demands. In *The Foundations of Knowledge Acquisition*, J. Boose & B. Gaines, Eds., pp. 325–337. Based on a paper presented at *The Second Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1987.

Yost, G. R. (1992). *Configuring Elevator Systems* (Tech. report). Marlboro, MA: Digital Equipment Corporation. See also URL<http://camis.stanford.edu/protege/sisyphus-2/>.