Mediating Knowledge between Application Components

Monica Crubézy, Zachary Pincus and Mark A. Musen
Stanford University, CA 94305, USA
crubezy@smi.stanford.edu

Abstract

In such contexts as the Semantic Web, the components of an application increasingly rely on ontological models and content knowledge developed and maintained by independent contributors. These components also are designed to be building blocks of various applications. We advocate the use of a mediating component that defines and processes the knowledge transformations required to enable application components to exchange, and inter-operate on, knowledge and data. We present our approach and associated tools to support developers (1) in defining mapping relations between the ontologies involved in their application and (2) in running a mapping interpreter to mediate content knowledge and data among the corresponding ontology-based components.

1 Interoperation of Application Components

As a multi-contributor environment, the World-Wide Web fosters the formation of applications that involve multiple, distributed components. In light of the Semantic-Web approach, ontologies—models that define the concepts, properties and relations of a domain of discourse—are the communication interface (if not the backbone) of these components meant to be assembled in various applications. Increasingly, however, such ontology-based application components are contributed independently and hence cannot be expected to adhere to shared models nor to integrate with one another gracefully. Instead, different components impose different semantic, structural and syntactic views and expectations on knowledge and data, expressed by means of independent ontologies. For example in a travel-planning application, a flight-booking component would conceive travel time as the exact day and time of a flight (e.g., "Outbound on 05-01-2003 at 14h25min"), whereas a car-reservation component might only need the approximate rental period (e.g., "From Monday May 1st 2003 early evening to Sunday May 7th mid-morning plus or minus 1 day"). Such conceptual and representational mismatches need to be resolved at the ontological level in order to enable application components to exchange, and to interoperate on, a common set of data and knowledge elements.

Our solution centers around the design of a *mediating component*—one that isolates and processes the knowledge needed for configuring different knowledge-based components to work together in a particular application. This middle component encodes declarative *mapping relations* that express rules to resolve mismatches between the concepts and properties defined in the ontologies of two application components. The mediating component interprets mapping relations to transform knowledge and data from one component into knowledge and data expected by another component. We have developed associated tools for creating and processing mapping relations between any two ontologies, based on the Protégé[1] knowledge-modeling environment.

Our approach offers the advantages of maintaining the integrity of the original independent application components (hence increasing the reusability of the components in different knowledge systems) while localizing and making explicit the knowledge transformations involved in adapting the components to work together (hence reducing the effort needed to encode and modify the transformation operations). It is important to note that our solution accounts for ontology-level alignment operations as well as for content-level transformation operations. While the former is the focus of much of the current ontology-management research, the latter is more traditionally found in database integration approaches.

2 Ontology-based Mediation of Knowledge

Our approach to mediating knowledge and data between application components centers around the definition of a set of *mapping relations* that both bridge gaps between different components' ontologies and transform instance knowledge and data from one component's ontology to another. First, our solution introduces a generic ontology of the kinds of mapping relations that can be defined between the ontologies of any two application components. Instantiated mapping relations represent the correspondence links

[1] http://protege.stanford.edu

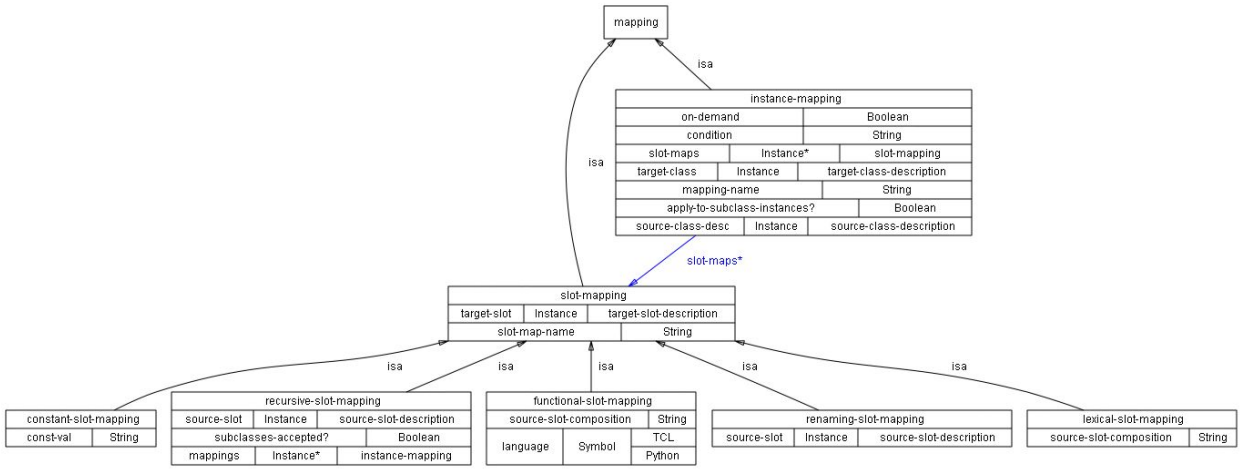

Figure 1: **Our generic ontology of mapping relations.** Each *instance-mapping* relation connects one or more source classes to one target class, and expresses how one instance of the target class is computed from each instance of the source class. Actual transformations of data and knowledge are specified in an associated set of *slot-mapping* relations that each defines the rules for computing the value of one slot of the target instance, possibly from the values of the source instance's slots. Types of slot mappings span the scope of operations that source knowledge can undergo to fit the format and semantics specified by the target ontology: from simple slot-value renaming to lexical expressions, to functional transformations. Recursive slot mappings are used for calculating instance-valued target slots, through a dependent instance mapping only processed in that context (*on-demand* flag). Finally, an instance mapping can be conditional upon properties of instances being mapped (*condition* slot), thus allowing for one-to-many instance-level mapping relations, and can be propagated to instances of subclasses of the source class (*apply-to-subclass-instances?* flag).

and transformation rules between the concepts and property values of the two components' underlying ontologies. Second, our approach includes a mapping interpreter that processes a set of mapping relations defined for two ontologies and migrates instantiated contents from one ontology to the other. Our approach is based on earlier work in our group that was aimed at studying the composition of knowledge systems from reusable domain knowledge bases and problem-solving methods [3, 4, 1].

We adopt a frame-based modeling view of ontologies. Accordingly, a set of *classes* are organized in a subsumption hierarchy to represent concepts in the domain of interest, and have *slots* attached to them to represent their properties. The values that slots can take are restricted by *facets*, such as cardinality, type and range. Classes are templates for individual *instances*, that have particular values for slots. Here, we adopt the notion of a knowledge base as an ontology populated with instances.

An Ontology of Mapping Relations

Mapping relations are defined between the ontologies of two—a *source* and a *target*—application components. Mapping relations hold the transformation operations to be applied on the source component's knowledge so that the target component is given the pieces and aspects of knowledge that it can operate on. According to a set of custom mapping relations, instances of the source component's concepts that are of interest to the target component are transformed (by a *mapping interpreter*, see below) into instances of corresponding target concepts, on which the target component is able to operate directly. Note that source and target roles for application components are dependent on the application's knowledge flow and are easily reversible.

It makes sense to categorize the types of mapping relations that can be expressed in any situation that requires mapping knowledge from one ontology to another. Such categorization allows us to conceptualize mapping relations in a better way and to design appropriate tool support for their definition and interpretation (see Section 3). We hence designed a small, generic *mapping ontology* that provides a structure for defining mapping relations between a source and a target ontology, in terms of conceptual alignment, of instance migration and of slot value computation. Figure 1 details the main aspects of our mapping ontology. To configure two components to work together in a system, a developer instantiates our mapping ontology with the set of mapping relations that link the ontologies of the source and target components. The developer thus creates a *mapping knowledge base* that contains rules to compute the target instances from source instance knowledge.

A mapping relation can be as simple as a one-to-one renaming correspondence between a source class and its slots, and a target class and its slots. In the travel-planning example, the ontologies of the flight-booking and car-reservation components might exactly share the notion and representation format of

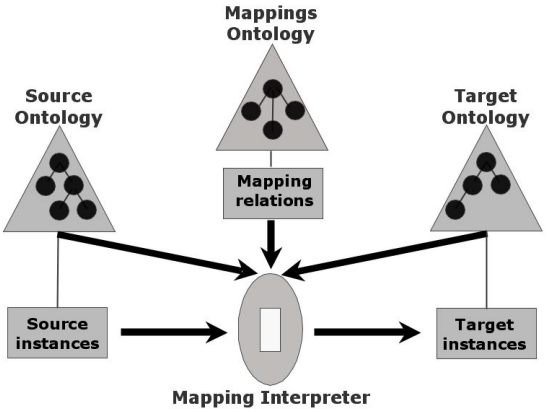

Figure 2: **Knowledge transformation and mediation performed by the mapping interpreter.**

a billing address. More complex instance-level mappings can express many-to-one, or many-to-many, aggregation relations between source and target concepts, as well as one-to-many concept-decomposition relations. Slot-level mappings also can express aggregation and decomposition operations, and include lexical, numerical and functional transformations of slot values. In the same example, the translation of the notion of time from one component to the other would involve several of those more complex operations, such as calculating a date interval from outbound and inbound flight dates, changing the date encoding, deriving approximate moments of the day from more precise flight times, etc.

Our mapping ontology provides the basis for expressing the adaptation knowledge needed to configure two components to work in a certain application. It is important to note that the core knowledge that is needed to create target instances out of source instances resides in the set of slot-level transformation operations attached to an instance-level mapping relation—operations that change the format and resolution of the source slot values to compute the required values of target slots. Eventually, a software component needs to operate on data structures that are derived from the filled-in instances of its ontology.

A Mapping Interpreter

We have developed the mapping interpreter as a piece of software associated with our mapping ontology that performs mediation of knowledge and data inside of a component-based application. As sketched on Figure 2, the mapping interpreter processes a given set of mapping relations between two ontologies—a mapping knowledge base—on a set of instances of the source ontology to produce a corresponding set of instances of the target ontology.

Specifically, in its default mode of operation the mapping interpreter cycles through all instance-mapping relations defined in the mapping knowledge base and creates one instance of the specified target class for each instance of the specified source class in a given instance mapping. The interpreter computes and fills-in the target instance's slot values according to each slot-mapping relation associated with the current instance mapping. A specific syntax that can be used in slot-value mapping expressions and in other mapping code such as conditions enables the interpreter to have local access to the source (sub-)instance's slot values. The mapping interpreter is also able to execute custom scripting and functional procedures (in TCL and Python), that provides additional mapping flexibility.

The mapping interpreter is written in Java and can be included in any component-based application. The mapping interpreter has a complete API for accessing its representation of a knowledge base (i.e., an ontology with instances). The mapping interpreter currently handles knowledge bases in the form of Protégé knowledge bases, Java collections of objects organized as in a frame-based knowledge base, and knowledge bases accessible from an OKBC [2] server. These formats can be extended with new ones.

3 Tool Support for Knowledge Mapping

We have developed an initial tool—the *Knowledge Mapping Tool*—to support an application developer in configuring ontology-based components to work together in a system, or simply to migrate knowledge from one ontology to another. Our tool is based on the Protégé knowledge-modeling environment. Ontologies have been at the heart of the Protégé methodology and tools since very early versions of the system [2]; Protégé hence is suited to provide the basis for the tool support that is necessary for mapping ontologies of application components. Protégé supports domain experts in modeling

[2] `http://www.ai.sri.com/~okbc/`

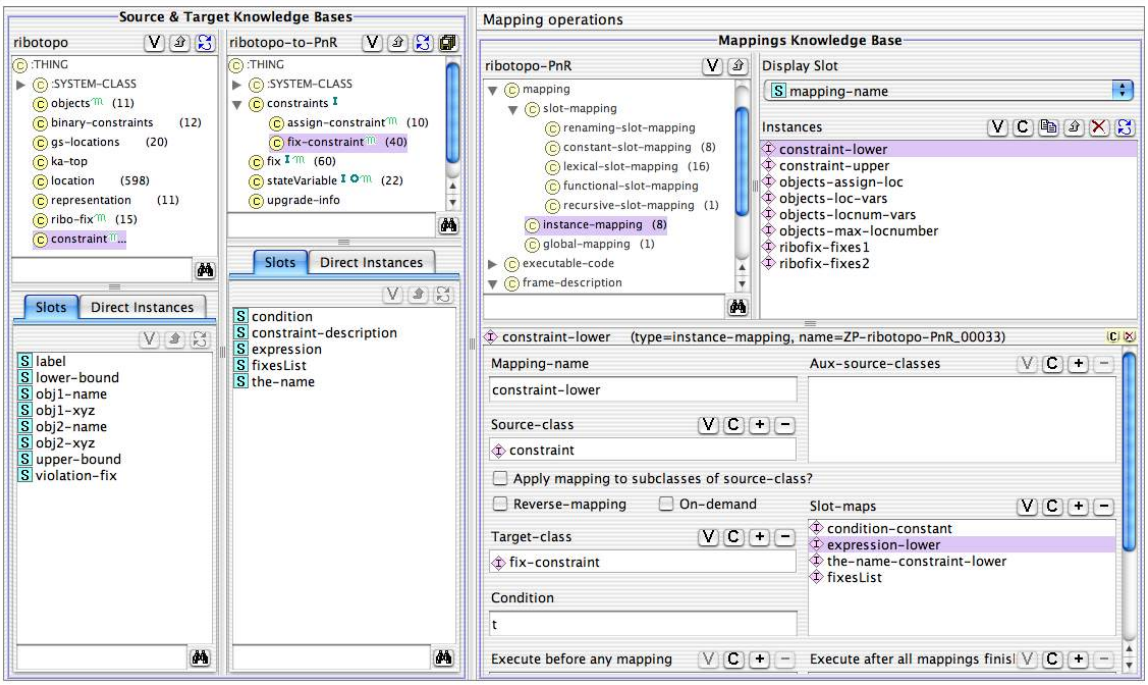

Figure 3: **Main view of the Knowledge Mapping tool.** The two left columns display, side-by-side, the source and target knowledge bases (classes in the upper panels, slots or instances in the lower panels). A small "m" icon next to a class name means that the class is part of a mapping relation with a class in the other ontology. At the right, the mapping panel displays the mapping knowledge base: At the top, the mapping ontology (left) and existing instances of mapping relations (right); below, the contents of the selected mapping relation instance, including its set of slot-level mappings. Double-arrow buttons at the top of each knowledge bases synchronizes all three panels according to the mappings defined for a selected class. For example, this screenshot shows the mapping relation "constraint-lower," from the "constraint" (source) class of the ribosome topology domain ontology and the "fix-constraint" (target) class of the propose-and-revise method ontology, for which four slot-level mapping relations have been defined to compute the values of the "condition," "expression," "the-name" and "fixesList" target slots. This example mapping relation specifies how to transform the lower bound value for the location of a ribosomal object into an actual distance-comparison expression and associated value-modification fixes to use when the expression is violated (see [1]). Note the "Mapping operations" menu at the very top of the mapping panel, that enables developers to create mapping relations from classes or slots selected in the source and target ontologies; to save the mapping knowledge base; and to run the mapping interpreter.

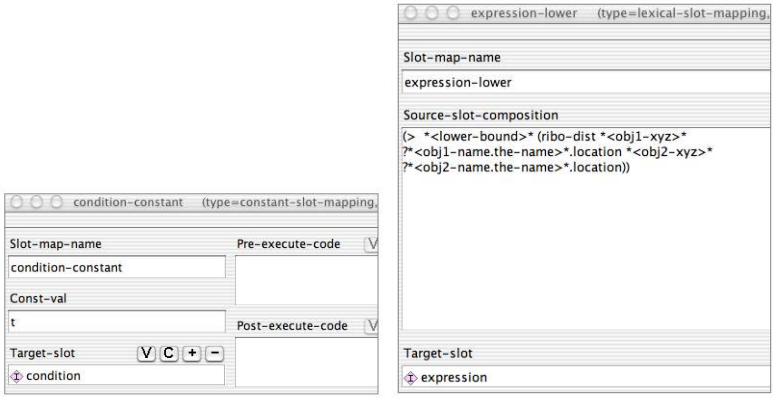

Figure 4: **Two particular slot-mapping relations**, defined in the scope of the instance mapping shown in Figure 3. (1) Left, is a simple constant slot mapping that specifies that for each instance of the target "fix-constraint" class created from an instance of the source "constraint" class, the value of the target slot "condition" should be filled-in with the value "t." (2) Right, is a lexical slot mapping that specifies a comparison predicate as value for the target "expression" slot involving the values of the source slots "lower-bound," "obj1-xyz," "obj2-xyz," "obj1-name," and "obj2-name." The $* < ... > *$ notation is used to access the actual values of the source (sub)instances' slots.

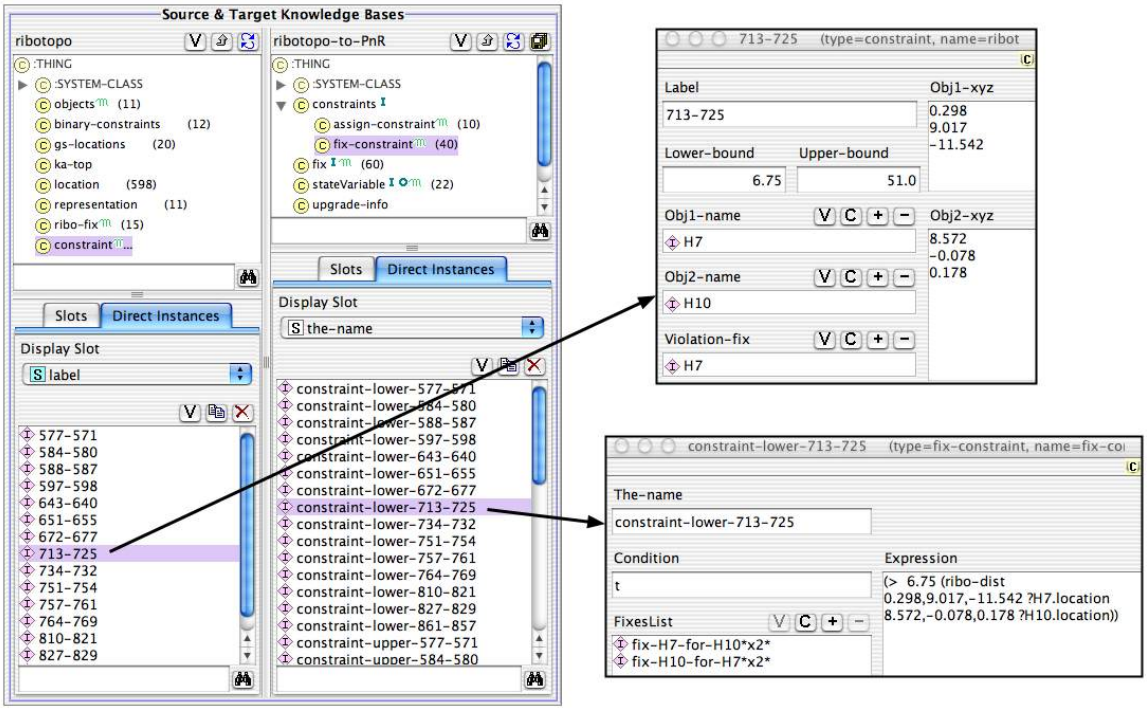

Figure 5: **Side-by-side inspection of the resulting target instances and their corresponding source instances.** As a result of running the mapping interpreter, the target ontology is populated with instances that are computed from the source ontology's instances and filled according to the set of mapping relations defined for these two ontologies. Note that instances of the target class can be the result of either instance mapping that has that class as a target class. Highlighted mapped instances are shown on the right: The contents of the instance of the source class "constraint" have been mapped partially to an instance of the target class "fix-constraint," according to the mapping relation shown in Figure 3. In particular, the value of the target slot "expression" contains the result of the lexical expression involving several source slots, as defined by the slot mapping shown in Figure 4.

relevant knowledge in an ontology and in customizing an associated knowledge-entry tool. We extended this native support of Protégé with a tool to help in creating and processing mapping relations between two ontologies. In particular, our mapping tool accesses knowledge bases from Protégé and reuses user interface elements of the base environment to provide a familiar, yet customized, interaction with system developers, as can be seen from our subsequent screen shots (Figures 3, 4 and 5).

The knowledge mapping tool allows application developers to perform the non-trivial activity of creating mapping relations between the entities of two ontologies. The tool provides a developer with an integrated and synchronized support for browsing and managing all three (source, target and mapping) ontologies involved, instead of switching manually between multiple ontology-editing windows. The mapping tool supports a developer in browsing the source classes and instances and the target classes side-by-side, and in creating or visualizing their mapping relations easily, as shown in Figure 3. A developer can populate, browse and edit the corresponding mapping knowledge base—a custom set of instances of the mapping ontology—that reflects the rules of mediating knowledge between the two ontologies.

The tool automatically creates a new mapping knowledge base for the two ontologies, or loads an existing one if available. Concretely, the developer then first creates a set of instance-level mapping relations between pairs of concepts of the two ontologies—relations that mean that for each instance of a source concept, an instance of the target concept will be created. For each instance-mapping relation between a source class and a target class, the developer also creates a set of slot-mapping relations—relations that express the way to compute the values of each slot of that target class, possibly from values of slots of that source class. The mapping tool helps the developer in making sure that all mappings are specified. The developer then can save the mapping knowledge base.

The knowledge mapping tool finally incorporates support to invoke the mapping interpreter on the three knowledge bases involved. Based on the mapping relations defined for two particular ontologies, the mapping interpreter computes a set of target instances that it fills with knowledge transformed from the source instances. After running the mapping interpreter, the knowledge mapping tool enables developers to inspect the computed instances in the newly populated target knowledge base (see Figure 5)—these instances hold the actual knowledge on which the target component will be able to operate directly.

4 Conclusion

We originally designed our solution for the task of assembling reusable domain knowledge bases with generic problem-solving methods into a working knowledge system, where a method defines a domain-independent ontology for its inputs and outputs, to be mapped to specific domain knowledge [1]. Our approach was key in assessing that PSMs and domain components could be reused in different applications. More generally, our generic mapping approach and tools are now applied to mediating knowledge and data between other kinds of knowledge-based application components in a wide range of situations. Recent applications of our tools include a high-performance architecture in which multiple public-health data sources and multiple analysis programs interact to perform syndrome-outbreak surveillance; a system for query transformation and dispatch to heterogeneous information sources; the migration of protocols from several clinical guideline and biological process formalisms to a generic workflow model. Provided adjustments of our tools to new ontology-modeling formalisms such as OWL [3], we are encouraged to believe that our approach will play a key role in Semantic-Web technology, along with ontology-management and database-integration solutions.

[3] `http://www.w3.org/2001/sw/WebOnt/`

Acknowledgements

This research is based on seminal work by John Gennari and John Park. Recent work was supported by the Defence Advanced Research Projects Agency.

References

[1] M. Crubézy and M. A. Musen. Ontologies in Support of Problem Solving. In Staab, S. and Studer, R., editor, *Handbook on Ontologies in Information Systems*, International Handbooks on Information Systems. Springer, In press. Also available as SMI Report SMI-2003-0957.

[2] J. H. Gennari, M. A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubézy, H. Eriksson, N. F. Noy, and S. W. Tu. The Evolution of Protégé: An Environment for Knowledge-Based Systems Development. *International Journal of Human-Computer Studies*, 58(1):89–123, 2003.

[3] J.H. Gennari, S.W. Tu, T.E. Rothenfluh, and M.A. Musen. Mapping Domains to Methods in Support of Reuse. *International Journal of Human-Computer Studies*, 41:399–424., 1994.

[4] J.Y. Park, J.H. Gennari, and M.A. Musen. Mappings for Reuse in Knowledge-Based Systems. In *Eleventh Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW'98)*, Banff, Alberta, 1998.