SMART: Automated Support for Ontology Merging and Alignment

Natalya Fridman Noy and Mark A. Musen

Stanford Medical Informatics
Stanford University
Stanford, CA 94305-5479
{noy, musen}@smi.stanford.edu

Abstract

As researchers in the ontology-design field develop the content of a growing number of ontologies, the need for sharing and reusing this body of knowledge becomes increasingly critical. Aligning and merging existing ontologies, which is usually handled manually, often constitutes a large and tedious portion of the sharing process. We have developed SMART, an algorithm that provides a semi-automatic approach to ontology merging and alignment. SMART assists the ontology developer by performing certain tasks automatically and by guiding the developer to other tasks for which his intervention is required. SMART also determines possible inconsistencies in the state of the ontology that may result from the user's actions, and suggests ways to remedy these inconsistencies. We define the set of basic operations that are performed during merging and alignment of ontologies, and determine the effects that invocation of each of these operations has on the process. SMART is based on an extremely general knowledge model and, therefore, can be applied across various platforms.

1. Merging Versus Alignment

In recent years, researchers have developed many ontologies. These different groups of researchers are now beginning to work with one another, so they must bring together these disparate source ontologies. Two approaches are possible: (1) merging the ontologies to create a single coherent ontology, or (2) aligning the ontologies by establishing links between them and allowing the aligned ontologies to reuse information from one another.

As an illustration of the possible processes that establish correspondence between different ontologies, we consider the ontologies that natural languages embody. A researcher trying to find common ground between two such languages may perform one of several tasks. He may create a mapping between the two languages to be used in, say, a machine-translation system. Differences in the ontologies underlying the two languages often do not allow simple one-to-one correspondence, so a mapping must account for these differences. Alternatively, Esperanto language (an international language that was constructed from words in different European languages) was created through merging: All the languages and their underlying ontologies were combined to create a single language. Aligning languages (ontologies) is a third task. Consider how we learn a new domain language that has an extensive vocabulary, such as the language of medicine. The new ontology (the vocabulary of the medical domain) needs to be linked in our minds to the knowledge that we already have (our existing ontology of the world). The creation of these links is alignment.

We consider merging and alignment in our work that we describe in this paper.

For simplicity, throughout the discussion, we assume that only two ontologies are being merged or aligned at any given time. Figure 1 illustrates the difference between ontology merging and alignment. In **merging**, a single ontology that is a merged version of the original ontologies is created. Often, the original ontologies cover similar or overlapping domains. For example, the Unified Medical Language System (Humphreys and Lindberg 1993; UMLS 1999) is a large merged ontology that reconciles differences in terminology from various machine-readable biomedical information sources. Another example is the project that was merging the top-most levels of two general common-sense-knowledge ontologies—SENSUS (Knight and Luk 1994) and Cyc (Lenat 1995)—to create a single top-level ontology of world knowledge (Hovy 1997).

In **alignment**, the two original ontologies persist, with links established between them.[1] Alignment usually is performed when the ontologies cover domains that are complementary to each other. For example, part of the High Performance Knowledge Base (HPKB) program sponsored by the Defense Advanced Research Projects Agency (DARPA) (Cohen et al. 1999; HPKB 1999) is structured around one central ontology, the Cyc ontology (Lenat 1995). Several teams of researchers develop ontologies in the domain of military tactics to cover the types of military units and weapons, tasks the units can perform, constraints on the units and tasks, and so on. These developers then align these domain-specific ontologies to Cyc by establishing links into Cyc'c upper- and middle-level ontologies. The domain-specific ontologies do not become part of the Cyc knowledge base; rather, they are separate ontologies that reference concepts in Cyc and use its top-level distinctions.

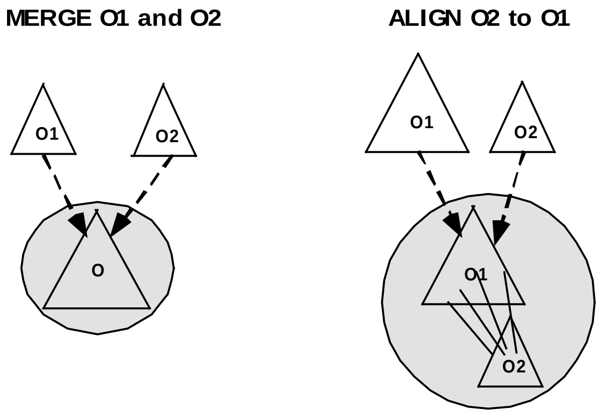


Figure 1. Merging versus alignment. O1 and O2 are the original ontologies, the dotted lines represent the transition that an original ontology undergoes, and the gray area is the result. The result of merging is a single ontology, O; that of alignment is persistence of the two ontologies with links (solid lines in the figure) established between them.

Until now, ontology designers have performed this complicated process of ontology merging and alignment mostly by hand, without any tools to automate the process or to provide a specialized interface. It is unrealistic to hope that merging or alignment at the semantic level could be performed completely automatically. It is, however, possible to use a hybrid approach where a system performs selected operations automatically and suggests other likely points of alignment or merging. The easiest way to automate the process partially is to look for classes that have the same names and to suggest that the user merges these classes. We developed an algorithm for ontology merging and alignment, SMART, that goes beyond simple class-name matches and looks for linguistically similar class names, studies the structure of relations in the vicinity of recently merged concepts, and matches slot names and slot value types.

Our participation in the ontology-alignment effort as part of the HPKB project mentioned earlier was a strong motivation for developing semi-automated specialized tools for ontology merging and alignment. We found the experience of aligning the ontologies without such tools to be an extremely tedious and time-consuming process. At the same time we noticed many steps in the

[1] Most knowledge representation systems would require one ontology to be included in the other for the links to be established.

process that could be automated, many points where a tool could make reasonable suggestions, many conflicts and constraint violations that a tool could check for.

We therefore undertook the following research:

- We defined a set of formalism-independent basic operations that are performed during ontology merging or alignment. Naturally, this set overlaps with the set of all operations performed during ontology development. However, none of these two sets—the set of all operations in a design of single ontology and the set of merging and alignment operations—properly subsumes the other.
- We are currently implementing SMART, an algorithm for semi-automatic ontology merging or alignment. SMART starts by automatically creating a list of initial suggestions based on class-name similarity. Then, for each operation invoked by the user, based on the classes and slots in the vicinity of that operation, SMART performs a set of actions automatically, makes suggestions to guide the user, checks for conflicts, and suggests ways to resolve these conflicts.
- We are currently implementing a specialized tool to perform ontology merging and alignment based on the SMART algorithm. The tool is an extension of the Protégé ontology-development environment (described in Section 2).

The knowledge model underlying SMART (described in Section 2) is compatible with the Open Knowledge Base Connectivity (OKBC) protocol (Chaudhri et al. 1998). We did not make any assumptions about the underlying knowledge model that are more detailed than OKBC-compatibility, so our results are applicable to a wide range of systems.

This paper is organized as follows. We start by describing the Protégé knowledge-modeling environment in Section 2 and presenting an example interaction with SMART and SMART's actions based on the interaction Section 0. We describe our approach to ontology merging and alignment more formally in Section 4. Section 5 gives an overview of the set of basic operations in merging and alignment that we defined, and discusses one such operations. We discuss possible future extensions to our merging and alignment framework in Section 6. In Section 7 we outline special features of SMART, such as types of conflicts SMART identifies, explanations it gives to the user and strategies it uses to prevent the user from loosing focus. Section 8 contains an overview of related work. Section 9 concludes the paper.

2. Protégé Knowledge-Modeling Environment

We designed SMART as part of the Protégé knowledge-modeling environment developed in our laboratory (Musen et al. 1995). In this section, we describe both Protégé as a tool for ontology design and Protégé's knowledge model (which underlies SMART also). We then give an example of representing an ontology in Protégé.

Protégé-2000[2] is a graphical and interactive ontology-design and knowledge-acquisition environment. It is aimed at making it easier for knowledge engineers and domain experts to perform these knowledge-management tasks. Protégé's graphical user interface is based on the premise that it should be as easy to use as possible without any significant loss to expressiveness. An ontology developer can quickly access the relevant information whenever he requires the

[2] Protégé-2000 is the latest implementation of the Protégé approach.

information and, wherever possible, can use direct manipulation for navigating and managing a knowledge base. Tree controls allow quick and simple navigation through a class hierarchy. Protégé uses forms as the interface for filling in slot values. The knowledge model that we use in SMART is an OKBC-compatible knowledge model underlying Protégé (Grosso et al. 1999).

2.1. Protégé Knowledge Model

We will now briefly describe Protégé-2000 knowledge model, concentrating on the knowledge-model elements that are important for understanding our discussion of SMART. Complete specification of OKBC model is available (Chaudhri et al. 1998) as well as a detailed description of OKBC implementation in Protégé-2000 (Grosso et al. 1999).

The knowledge model underlying Protégé-2000 is frame-based (Minsky 1981) and is designed to be compatible with OKBC (Chaudhri et al. 1998). At the top level, there are classes, slots, facets, and instances:

- **Classes** are collections of objects that have similar properties. Classes are arranged into a subclass–superclass hierarchy with multiple inheritance. Each class has own and template slots attached to it (we discuss the distinction later in this section).
- **Slots** are first-class frames. That is, a slot can exist without being attached to a particular class. Slots can be single-valued or multivalued. Each slot has associated with it a name, domain, value type, and, possibly, a set of facets.
- **Facets** are frames that impose additional constraints on an attachment of slot frame to a class frame. Facets can constrain cardinality or value type of a slot, contain documentation for a slot, and so on.
- **Instances** are individual members of classes. Any **concrete** (as opposed to **abstract**) class in Protégé can have instances.

Attaching slots to classes

Even though slots are first-class objects in Protégé-2000, they are used primarily for describing attributes of classes and their instances. **Attaching** a slot to a class establishes a connection between the two. For instance, a knowledge base can have a slot `age` (see Figure 2). The type of the `age` slot is `Integer`; `age` is a single-valued slot and it can be attached to any class that is subclass of `Person` (that is, `Person` is the domain of the `age` slot). The maximum value for the slot is `140`. However, when the `age` slot is attached to specific subclasses of `Person`, it can acquire additional constraints. The constraints on the values a slot can take when attached to a class are determined by **facets.** In the example in the figure, the maximum value of the `age` slot changes for one of the attachments. A `maximum-value` facet specifies that when an `age` slot is attached to a `Teenager` class, the slot's maximum value is limited by `19`. There are no additional restrictions on the attachment to the `Student` class (and therefore, the maximum value for a student's age remains `140`).

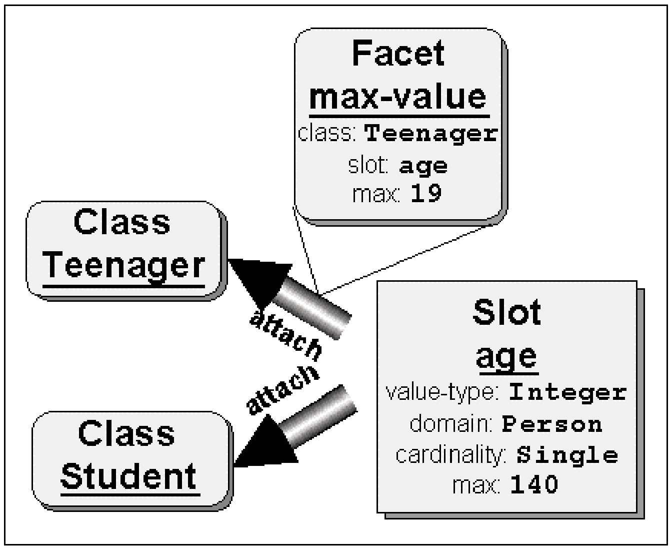


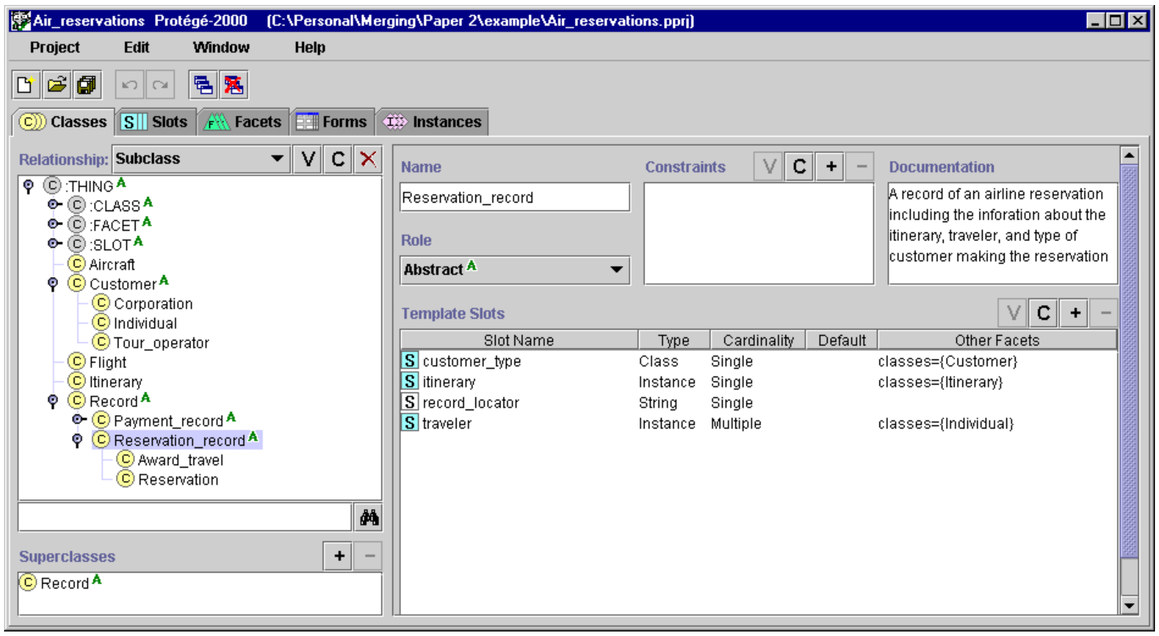
Figure 2. Attaching slots to classes. Facets impose additional restrictions on slot values for the attachment.

The knowledge model that we have just described is an extremely general one, and many existing knowledge-representation systems have knowledge models compatible with it. Therefore, the approach to merging and alignment that we developed can be applied to a variety of knowledge-representation systems.

Adopting an OKBC-compatible knowledge model also makes it possible to implement SMART as an OKBC client, where most of the work in accessing a knowledge base is performed on the server side. We translated the set of basic merging and alignment operations (Section 5) to OKBC using OKBC's scripting language. Most of the operations from this set are simple combinations of one or two operations from the OKBC specification (Chaudhri et al. 1998). Several of the operations, however, require more extensive scripts. If the OKBC front end included these more complex operations, then OKBC would provide better support for merging and alignment.

2.2. Example Ontology in Protégé-2000

We now present an example of a Protégé-2000 ontology: an ontology for an airline-reservation system. In Section 0, we illustrate how the merging side of SMART works. We use this example ontology as one of the sources being merged (at the end of this section we briefly describe the second source ontology for the SMART-merging illustration). The airline-reservation-system ontology (henceforth, "Airline ontology") formalizes concepts for describing airline reservations. We designed it for illustration purposes only and did not attempt to represent fully all the concepts in the domain. You can see how Protégé-2000 presents the Airline ontology in Figure 3. There is a partially expanded class-hierarchy tree on the left-hand side and all the slots for the selected class `Reservation_record` on the right-hand side of the figure.



Air_reservations Protégé-2000 (C:\Personal\Merging\Paper 2\example\Air_reservations.pprj)

Relationship: Subclass

```
:THING
    :CLASS
    :FACET
    :SLOT
    Aircraft
    Customer
        Corporation
        Individual
        Tour_operator
    Flight
    Itinerary
    Record
        Payment_record
        Reservation_record
            Award_travel
            Reservation
```

Superclasses: Record

Name: Reservation_record

Role: Abstract

Constraints:

Documentation: A record of an airline reservation including the inforation about the itinerary, traveler, and type of customer making the reservation

Template Slots

Slot Name	Type	Cardinality	Default	Other Facets
customer_type	Class	Single		classes={Customer}
itinerary	Instance	Single		classes={Itinerary}
record_locator	String	Single		
traveler	Instance	Multiple		classes={Individual}

Figure 3. Airline-reservation-system ontology in Protégé-2000. The left-hand side contains the partial class hierarchy for an ontology of airline reservation system. The right-hand side shows the slots and their values or constraints for a Reservation_record class.

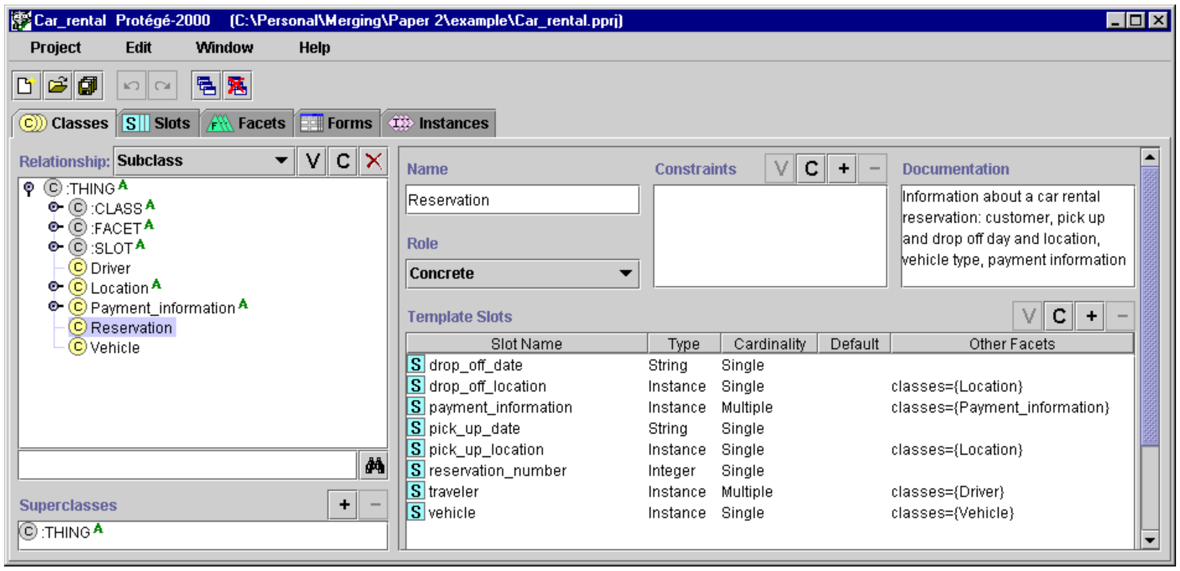
The selected `Reservation_record` class is a subclass of `Record` and has two subclasses of its own: `Award_travel` (for frequent-flyer program members) and `Reservation` (for regular reservations). It has three own slots: `role` (abstract or concrete), `constraints` (not currently filled in) and `documentation`. The list of the template slots attached to the class follows. The `Reservation_record` class inherits the slot `record_locator` from its superclass.[3] The other three slots—`cutomer_type`, `itinerary`, and `traveler`—are attached directly to it.

We now turn to the constraints on the slot values for the `Reservation_record` class. Note that the constraints are constraints on the *attachment* of the corresponding slots to the `Reservation_record` class and not necessarily constraints on the slot frames themselves.

Slot `traveler` has a value-type `Instance`: the value of a slot is an instance of a class in the ontology rather than a simple value such as number, string, or symbol. The values for the `traveler` slot must be instances of the `Individual` class—allowed class for the template slot. The `Itinerary` class is an allowed class for the template slot `itinerary`. Similarly, a slot can have a collection—class—as its value (value-type `Class`). In the latter case, there is a facet specifying allowed parents for the values of the slot. Slot `customer_type` in Figure 3 is of type `Class`: the value of the slot is any *subclass* of class `Customer` (and not an *instance* of `Customer`). We refer to allowed classes for `Instance` slots and allowed parents for `Class` slots as classes **referenced by** a slot.

[3] Protégé-2000 uses color to distinguish inherited slots from the slots that are attached directly to the current class or from the slots that had their values overriden at the current class.

The Airline ontology is one of the two source ontologies that we use in the next section (and in the examples throughout the rest of the paper) to illustrate the ideas behind SMART. We merge the Airline ontology with an ontology for car-rental reservation shown in Figure 4 (henceforth, “Car_rental ontology”). The `Car_rental` ontology describes car-rental reservations in a way similar to the airline reservations in the first ontology. To create a single ontology that knows both about air travel and car rental we merge the two ontologies. The next section demonstrates how SMART goes about implementing the merging.



Name: Reservation

Role: Concrete

Documentation: Information about a car rental reservation: customer, pick up and drop off day and location, vehicle type, payment information

Slot Name	Type	Cardinality	Default	Other Facets
drop_off_date	String	Single		
drop_off_location	Instance	Single		classes={Location}
payment_information	Instance	Multiple		classes={Payment_information}
pick_up_date	String	Single		
pick_up_location	Instance	Single		classes={Location}
reservation_number	Integer	Single		
traveler	Instance	Multiple		classes={Driver}
vehicle	Instance	Single		classes={Vehicle}

Figure 4. Car-rental-reservation ontology in Protégé-2000.

3. SMART at Work

In this section, we give an example of the *merging* (as opposed to the alignment) side of SMART at work. For simplicity, we assume that there are only two source ontologies that need to be merged into one. We show how SMART merges the Airline and the Car_rental ontologies presented in Section 2.2. Figure 3 and Figure 4 respectively show the ontologies. SMART creates the target merged ontology from scratch and the target ontology is different from either of the source ontologies.

SMART starts by comparing class names in the two ontologies and identifying classes with identical names. It then creates its initial list of suggestions to the user by proposing that the user merges the pairs of identically named classes. Operating with the assumption that none of the information should be lost, SMART also proposes that the user simply copies into the merged ontology (using `SHALLOW-COPY` operation) the classes that have unique names. In the future we plan to incorporate more sophisticated comparison of class names which includes comparing parts of complex names, demorphing, finding synonyms (through a connection to WordNet (Miller 1995)) and so on. The original list of suggestions appears in the SMART window presented in Figure 5.

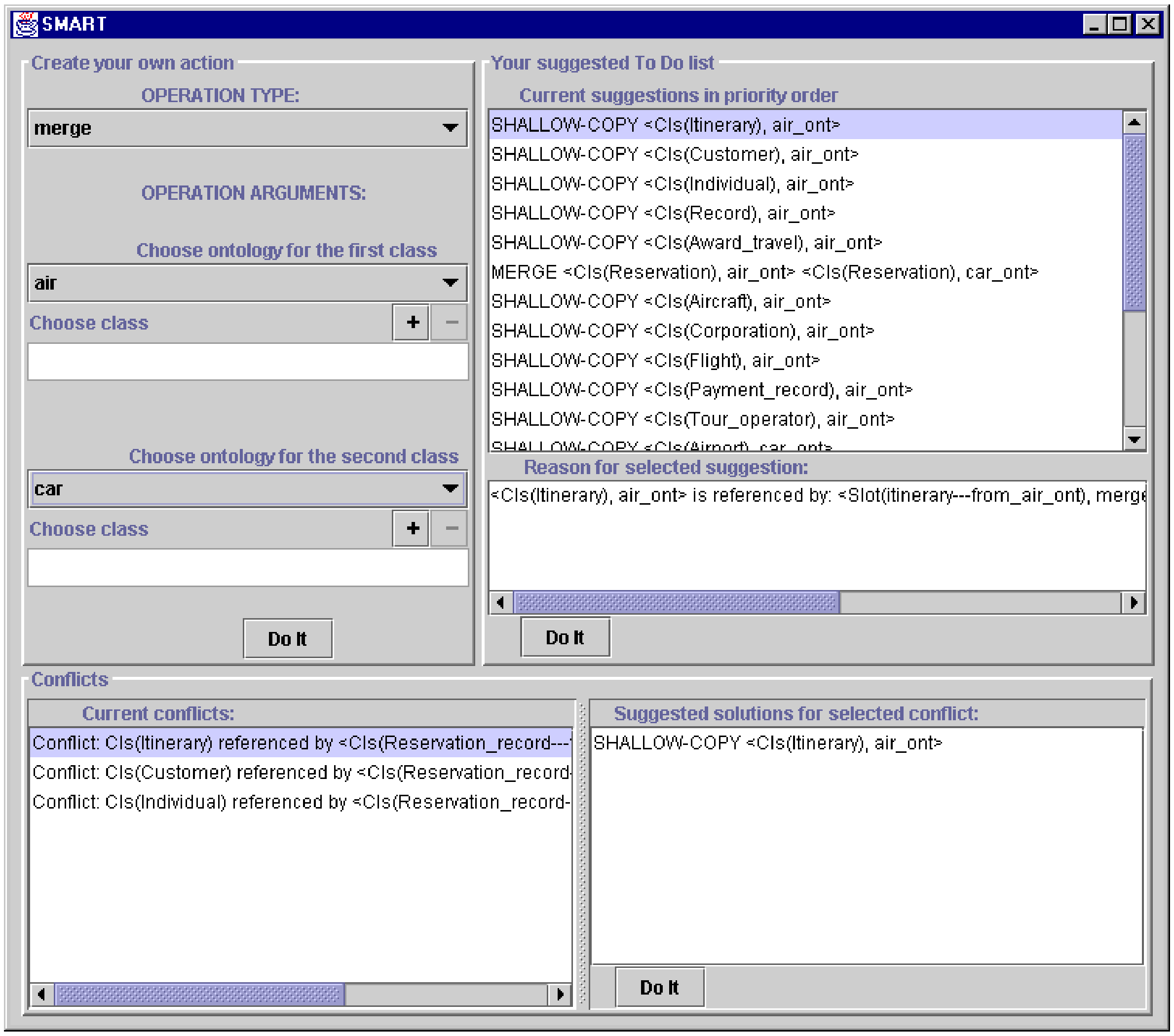
This would be a good time to take a look at the SMART window and discuss various logical elements of it. The top pane has two elements: (1) a panel where the user can create new operations of his own, other than the ones suggested by SMART and (2) a list of current suggestions from SMART (ToDo list). There is also an explanation of why SMART suggested the currently selected item from the ToDo list. The bottom pane of the window contains a list of conflicts that SMART identified and solutions to those conflicts that it suggested.

The state of SMART shown in Figure 5 is the one reached after the first operation has been executed:

    SHALLOW-COPY <Cls(Reservation_record), air_ont>[4]

As the result, the `Reservation_record` class has been copied over from the Airline ontology into the merged ontology.

After the operation, SMART rearranges the ToDo list to move the list items that involve classes referenced by `Reservation_record` to the top. For example, class `Record` is a superclass of `Reservation_record` in the Airline ontology and classes `Award_travel` and `Reservation` are its subclasses. If we go back to Figure 3, we will see that `Reservation_record` class also referred to classes `Itinerary`, `Customer` and `Individual` as allowed classes or allowed


SMART
Create your own action
OPERATION TYPE:
OPERATION ARGUMENTS:
Choose ontology for the first class
Choose class
Choose ontology for the second class
Choose class
Do It
Your suggested To Do list
Current suggestions in priority order
Reason for selected suggestion:
Do It
Conflicts
Current conflicts:
Suggested solutions for selected conflict:
Do It

OPERATION TYPE: merge
Choose ontology for the first class: air
Choose ontology for the second class: car

Current suggestions in priority order:

```
SHALLOW-COPY <Cls(Itinerary), air_ont>
SHALLOW-COPY <Cls(Customer), air_ont>
SHALLOW-COPY <Cls(Individual), air_ont>
SHALLOW-COPY <Cls(Record), air_ont>
SHALLOW-COPY <Cls(Award_travel), air_ont>
MERGE <Cls(Reservation), air_ont> <Cls(Reservation), car_ont>
SHALLOW-COPY <Cls(Aircraft), air_ont>
SHALLOW-COPY <Cls(Corporation), air_ont>
SHALLOW-COPY <Cls(Flight), air_ont>
SHALLOW-COPY <Cls(Payment_record), air_ont>
SHALLOW-COPY <Cls(Tour_operator), air_ont>
SHALLOW-COPY <Cls(Airport), car_ont>
```

Reason for selected suggestion:

```
<Cls(Itinerary), air_ont> is referenced by: <Slot(itinerary---from_air_ont), merge
```

Current conflicts:

```
Conflict: Cls(Itinerary) referenced by <Cls(Reservation_record---
Conflict: Cls(Customer) referenced by <Cls(Reservation_record
Conflict: Cls(Individual) referenced by <Cls(Reservation_record-
```

Suggested solutions for selected conflict:

```
SHALLOW-COPY <Cls(Itinerary), air_ont>
```

Figure 5. SMART at work.

[4] In the SMART window, we refer to the Airline ontology as “air_ont” and to the Car_rental ontology as “car_ont”.

parents for values of `Reservation_record`’s slots. Thus, SMART moved operations involving the referenced classes to the top of its ToDo list as well. Thus, SMART's suggestions that refer to ontology elements closest to the user's latest focus have the highest priority.

The list of current conflicts is not empty either. The conflicts that SMART identified are the dangling references from the `Reservation_record`’s `Instance-` and `Class-`type slots to `Itinerary`, `Customer`, and `Individual`. SMART allowed the creation of these slots and the corresponding references, temporarily leaving the knowledge base in an inconsistent state. It listed the conflicts and suggested possible solutions so that the user can remedy the conflicts or authorize SMART to do it. For example, to resolve the dangling reference to the `Itinerary` class, the user may copy this class from the Airline ontology:

```
SHALLOW-COPY <Cls(Itinerary), air_ont>
```

Other possible solutions (not yet implemented) include changing the value type of the `itinerary` slot or changing the list of allowed classes for the slot.

Now if the user selects the operation

```
SHALLOW-COPY <Cls(Itinerary), air_ont>
```

either from the ToDo list or as a solution to the corresponding conflict, the conflict will be resolved and SMART will remove it from the list (and, naturally, new dangling-reference conflicts may arise).

SMART also keeps track of the original relations between frames in the source ontologies and, if possible, restores these relations when the frames are brought in the resulting merged ontology. For example, if a `Record` class from the Airline ontology is now merged with a `Reservation` class from the Car_rental ontology, the `Reservation_record` class will become a subclass of the merged class—the same relation it had with one of the prototypes for the merged concept in the original ontology.

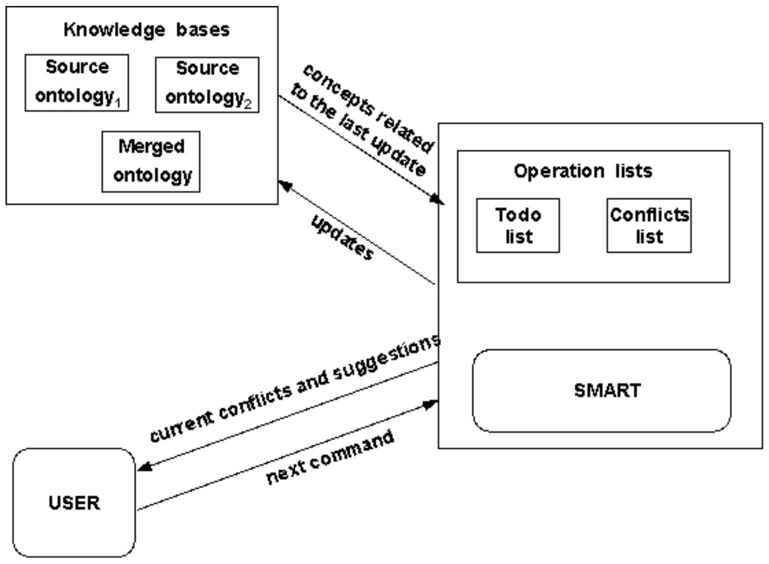


Figure 6. SMART’s architecture.

Currently a user can choose only a single operation to be performed. In the future, it will be possible to select a collection of operations: for example, copy a whole subtree in a class hierarchy from source to target ontology.

SMART's architecture

Figure 6 depicts a general architecture diagram of the merging module in SMART. On the knowledge-base side there are three ontologies: two sources (Airline and Car_rental ontologies in our example) and one resulting merged ontology. The source ontologies remain intact during the process: all the changes are performed in the merged ontology.

SMART itself keeps lists of suggestions (ToDo list), conflicts and their solutions (Conflicts list) as well as all the data structures necessary for keeping track of concepts and their prototypes in the source ontologies. The user selects operations and gets feedback from SMART in the form of updated ToDo and Conflicts list, explanations for elements in these lists and possible solutions for conflicts. SMART in turn implements operations by making changes to the target knowledge base, making new suggestions and determining conflicts based on the information in all three knowledge bases.

4. Formal Description of the SMART Algorithm

Having presented an example of how our SMART-based tool works, we now turn to a more general description of our approach to ontology merging and alignment. We describe an algorithm for merging and alignment that SMART is based on and the data structures that we use to maintain lists of suggestions and conflicts. The algorithm we present here is formalism-independent and can serve as a general framework for implementing ontology merging or alignment in any OKBC-compatible knowledge-modeling environment.

4.1. Description of the SMART Algorithm

In the algorithm description that we present below, the word in parentheses at the beginning of an algorithm step indicates whether a user executes the step (User) or whether SMART can perform the step automatically (SMART).

1. (User) *Setup*: Load two ontologies— O_1 and O_2
2. (SMART) *Generate the initial list of suggestions*: Scan the class names in both ontologies to find linguistically similar names. (Linguistic similarity can be determined in many different ways, such as by synonymy, shared substrings, common prefixes, or common suffixes.) Create the initial list of suggestions to the user.
   - If the process is `merge`, suggest the following:
     - For each pair of classes with *identical* names, either merge the classes or remove one of the classes in the pair.
     - For each pair of classes with *linguistically similar* names, establish a link between them (with a lower degree of confidence than in the first set).[5]

[5] We have not yet implemented the comparison of linguistic similarity of names.

- If the process is `align` and one ontology (say, O_1) has been identified by the user as the less general, then the suggestions are based on the assumption that classes in the less general ontology need to be linked by subclass–superclass relations with the classes in the more general ontology (O_2). For each class `C` at the top level of O_1, the following suggestions are made with high confidence:
  - If there is a class in O_2 with the same name as `C`, merge the two classes.
  - Otherwise, find a parent for class `C.`

3. (User) *Select and perform an operation*: The operation the user selects can be, but does not have to be, from the current list of suggestions. The set of possible basic operations is defined in Section 5.1.
4. (SMART) Perform automatic updates and create new suggestions: For the concepts involved in the operation just performed in step 3, consider linguistic similarity of their class and slot names and the structure of the ontology in their vicinity. Based on the similarity and structural patterns, trigger actions in three categories:
   - Immediately execute additional changes determined automatically by the system.
   - Add to the list of conflicts (described in Section 1) the set of new conflicts in the state of the ontology resulting from the user action and suggestions on how to remedy each conflict.
   - Add to the list of suggestions (described in Section 1) the set of suggestions to the user on what other actions can be performed next.
5. Repeat steps 3 and 3 until the ontologies are fully merged or aligned.

Creation of the initial list of suggestions in step 1 is mostly syntactic: SMART makes its decisions based on *names* of frames and not on *positions* of frames in the ontology or frames' participation in specific *relations*. Step 4 includes semantic analysis as well: SMART considers the *types* of relations involving current concepts and arguments of these relations. In section 1, we give examples of the specific semantic relations that SMART considers in step 4.

We now describe structure and content of the lists that contain the current sets of suggestions. One of SMART's central abilities is to determine new actions and suggestions based on the operation just performed. Thus, for each basic merging or alignment operation, we must define the set of automatic actions that the operation may trigger, the conflicts that it may produce, and the new points of merging that it may create.

4.2. Data Structures

We have referred several times to the two basic data structures that SMART maintains throughout the merging and alignment process and that it uses to guide the user in the process: the list of conflicts that resulted from the user's actions and the list of suggestions for the next action. We call these structures the `Conflicts` list and the `ToDo` list, respectively. The items on the `Conflicts` list represent inconsistencies in the current state of the knowledge base that must be resolved before the knowledge base can be in a logically consistent state. For example, if we merge two slots with conflicting facet values, SMART puts the operation of removing one of the facets in the conflict on the `Conflicts` list. Before closing the knowledge base, the user must perform this operation or ask SMART to decide which facet to remove. Alternatively, if two slots of a merged concept have similar (but not identical) names and the same value type, then merging these slots is an operation that probably should be performed. However, execution of this operation is not required for internal consistency of the knowledge base. SMART places such an operation on the `ToDo` list.

ToDo list

An item on the `ToDo` list is either a single operation or a disjunction of operations. In the latter case, the underlying semantics is that only one of the operations in the disjunction needs to be performed. SMART chooses the operations for the `ToDo` list from the set of basic merging and alignment operations (Section 5.1) and specifies their arguments (and, possibly, suggested results).

In the example in Figure 5, we copied the `Reservation_record` class from the Airline ontology into the merged ontology. The `Reservation_record` class has two subclasses: `Award_travel` and `Reservation`. SMART now suggest that the user copies the subclasses as well:

```
SHALLOW-COPY <Cls(Award_travel), air_ont>

SHALLOW-COPY <Cls(Reservation), air_ont>
```

SMART gives the following explanation for the first of the two suggestions:

```
<Cls(Award_travel), air_ont> is referenced by:
<Cls(Reservation_record---from_air_ont), merged_ont>
```

Conflicts list

An item on the `Conflicts` list consists of a description of a conflict and a suggested action that would remedy the problem. It also contains a default solution that will be invoked if the user asks SMART to resolve the conflict.

For instance, in our previous example, having a reference to a class that is not yet in the ontology is considered a conflict. In the case of the `Reservation_record` class, having a slot `traveler` referencing class `Individual` that has not yet been copied is a conflict:

```
Cls(Individual) referenced by <Cls(Reservation_record---
from_air_ont), merged_ont> is not defined
```

SMART suggests the following action to resolve the conflict:

```
SHALLOW-COPY <Cls(Individual), air_ont>
```

If there is more than one solution, SMART designates one of the solutions as default. If the user asks SMART to resolve all conflicts automatically, SMART executes the default solution.

Other attributes of items in the `Conflicts` and `ToDo` lists

Priority. SMART assigns priority to the items in the `Conflicts` and `ToDo` lists and orders the items in the lists with the higher-priority items more easily accessible. The priority value can be based on one or more of the following:

- SMART’s certainty about the item: The higher the certainty, the higher the priority

- The item's age: Items triggered by the more recent operations have higher priority because they refer to the concepts that are closer to the user's current focus
- The number of references to the concepts in the operation: The more elements in the ontology would be affected if the operation is invoked, the higher the priority of the operation.

Feedback and explanation. Each item in the `Conflicts` and `ToDo` lists has references back to the triggering operation and to the rule that SMART used to put the item on the list (in the formal or natural-language form). The reference and the rule explain to the user why a particular item appears on the list. The user can also disable the rules that he considers not useful or impediments.

5. Basic Operations for Merging and Alignment

5.1. Types of Operations

In this section, we describe the set of basic operations for ontology merging or alignment. We presented a detailed list of the operations in an earlier paper (Fridman Noy and Musen 1999). We will give an overview of these operations here. Then, we present one of these operations `(shallow-copy)` in detail. We discuss the automatic operations and additions to the `ToDo` and `Conflicts` lists that these operations trigger.

OKBC specifies a superset of all operations that can be performed during an ontology-design process (Chaudhri et al. 1998). This superset of operations is sufficient for the processes of ontology merging and alignment, because the latter process can also be treated as the ontology design processes. Therefore, we can argue that the only task that a merging and alignment tool must do is to support the OKBC operations. However, judging from our own experience in merging and alignment, we believe that not all the operations required for general ontology design are used in the more specialized processes of ontology merging and alignment. For instance, during merging or alignment, we rarely create new classes completely from scratch (the new classes are usually modifications of the existing ones); creation of new slots or facets is even less common. Likewise, several operations are performed only during ontology merging and alignment and are usually not present at the ontology-design stage. For example, merging two concepts into one is common in ontology merging. Moving a concept from one source ontology to another source ontology is often performed during alignment. Although these operations can be expressed as combinations of primitive operations from the OKBC set, the user needs to perform these sequences of basic operations in one step.

The following list contains a subset of the operations for ontology merging and alignment that we identified. Several of these operations are normally performed only during ontology merging, several operations are used only in ontology alignment, and most operations apply to both situations. The list that we present here is not exhaustive; for example, we have not represented operations dealing with updates to facets and instances.

- `Merge` (F_1, F_2)—merge frames F_1 and F_2 to create a new frame, F_3. The operation is only applicable if F_1 and F_2 are frames of the same type (e.g., both are class frames).

- `Shallow-copy (F,` O_1, O_2`)`—copy frame `F` from ontology O_1 to ontology O_2. SMART copies only the frame itself and creates "dummy" frames if necessary for any frames `F` refers to (e.g., any frames that slots of `F` reference and that do not already exist in O_2).
- `Deep-copy (F,` O_1, O_2`)`—copy frame `F` from ontology O_1 to ontology O_2; also copy recursively all superclasses of F, superclasses of those classes and so on, until the root of the hierarchy is reached.
- `Remove-frame (F)`—remove frame `F` from the ontology that it belongs to.
- `Remove-parent (`C_1, C_2`)`—assuming that C_2 is a superclass of C_1, remove C_2 from the list of C_1's superclasses.
- `Add-parent (`C_1, C_2`)`—add C_2 from the list of C_1's superclasses.
- `Rename (F, N)`—change the name of frame `F` to `N`.
- `Remove-attachment (`S_1, C_1`)`— assuming that slot S_1 is attached to class C_1, remove the attachment and facets specifying constraints on the attachment.
- `Attach-slot (`S_1, C_1`)`— attach slot S_1 to class C_1.

We translated the operations from the preceding list to OKBC using the OKBC's scripting language. Most of the operations from this set are simple combinations of one or two operations from the OKBC specification. Several of them, however, require more extensive scripts. These operations can extend the OKBC front end to provide support for merging and alignment in OKBC.

Identification of these basic operations and of the consequences of invoking them is the central task in our approach to ontology merging and alignment. For each operation, we can specify the rules that determine the specific members of the three sets updated by the operation: (1) changes that should be performed automatically, (2) conflicts that will need to be resolved by the user, and (3) new suggestions to the user. Having specified these rules, we can have the system automatically create members of these three sets based on the arguments to a specific invocation of an operation.

In the next section, we discuss one of the operations from the preceding list—`shallow-copy`– in detail.

5.2. Example: shallow-copy

`Shallow-copy` operation takes a frame from one of the source ontologies as an argument and copies the frame into the target merged ontology. We describe the `shallow-copy` operation and the design decisions that we made and our rationale for the decisions. Similar issues arise with the rest of the operations in the list in the previous section.

When SMART must copy a frame from one ontology to another, a number of questions arise:

- Should SMART copy all of the frames attached or should it include only the slots that were defined directly at the frame or overridden at the frame? That is, should it copy the inherited slots as well?
- Should it copy the superclasses of the frame?

- If the slots attached to the frame refer to other concepts as their allowed classes or allowed parents, should it copy these referenced frames as well? Or should it remove the reference in the facet if the referenced frame is not yet in the target ontology? Or should it create a temporary frame for the referenced frame and remove the temporary frame later when it copies the actual referenced frame?
- What if the copied frame or one of its slots has the same name as another frame in the target ontology? Should SMART continue the copying operation? Or should it ask the user for a new name for the frame? Or should it create temporary names and flag the conflict?
- What if the slot with the same name as one of the slots attached to the copied concept already exists in the target ontology (and it was *not* brought in from the same source ontology that the concept we are trying to copy)? Should SMART create a new slot with the same name and declare a conflict? Should it attach the existing slot to the frame it is copying? What if facet values describing the attachment contradict restrictions on the existing slot?

And so on.

We realize that there is no one correct answer to any of these questions (and many other similar questions that arose during SMART's development). We based our approach to answering these questions on the following premises:

- We observed what behavior was most common in our own practical experience of manually merging ontologies and designated that as default behavior.
- Interference with the user's current train of thought should be as minimal as possible (see Section 7.3 for the relevant discussion).
- SMART must flag results of any actions that were not explicitly authorized by the user (in a minimally intrusive way possible). The user should be able to change the results easily.
- As much of the SMART's behavior as possible must be customizable.

Therefore we decided to proceed with the shallow copy operation as follows.

Step 1: create frame

Suppose a frame being copied is the `Reservation_record` frame from the Airline ontology. We start by checking that the `Reservation_record` frame has not been already mapped into any other frame in the merged ontology. If it has been mapped, SMART checks with the user if it shall proceed anyway.

If there is a frame with the same name and of the same type (in our example, `Class`) already in the merged ontology, SMART suggests that the user merges `Reservation_record` from the Airline ontology with the existing `Reservation_record` in the merged ontology instead of copying it. If the user insists on copying, SMART flags a duplicate-name conflict.

SMART creates a new frame with the name `Reservation_record` in the target ontology.

Step 2: link the new frame to already existing frames

If any of `Reservation_record`'s parents from the Airline ontology have been already mapped into some concepts in the merged ontology, the concepts that they were mapped into become parents of the newly created `Reservation_record` frame. The same is true for the subclasses of `Reservation_record`. SMART also establishes relations between any other reference to `Reservation_record` that earlier had to refer to a temporary frame with the newly created frame.

Step 3: copy slots and slot attachments

Then SMART copies the slots attached to `Reservation_record`. If the frames for the slots do not yet exist in the merged ontology, SMART creates slot frames and then attaches them to `Reservation_record`. If the frames have been created earlier, SMART just creates the attachment. For each attachment SMART copies the corresponding facets. If facets refer to classes that are not in the merged ontology yet, SMART creates “dummy” temporary frames for them that it will remove later when the user creates the actual frames.

6. Future Directions

The next three sections constitute three threads in our discussion of ontology merging and alignment. The current section presents possible extensions to the general framework we described here and approaches to formal evaluation of ontology merging and alignment tools. In the next section we turn back to our SMART-based tool that we described in Section 0 and overview some of its special features. The third of the discussion sections (Section 8) presents the related work in the ontology-design field and object-oriented programming.

6.1. Extensions to the Model

We describe the following possible extensions to the framework for ontology merging and alignment we presented in the paper: extending the algorithm to handle facets and instances and generalizing our approach for non-OKBC frameworks.

Facets and instances

Facets impose additional constraints on slots in a frame. For example, a facet can define cardinality constraints on a slot. Suppose that we merged two classes and that both classes have a slot with the same name and the same value type. In this case, SMART automatically merges the slots and adds a slot with that name and value type to the newly created concept. However, in one of the source ontologies, the slot that was merged may have had a facet that defined its maximum cardinality as 2. In the second source ontology, the otherwise-identical slot may have had a minimum cardinality of 3. These cardinality restrictions are incompatible, so the algorithm must reconcile them.

The OKBC specification (Chaudhri et al. 1998) lists standard facets for slots, such as maximum and minimum cardinality, inverse, numeric minimum and maximum, documentation, and so on. In the merging and alignment process, it is desirable to consider each of these standard facets, correlation among possible facet values, and ways of reconciling conflicting values. We can also use values of the standard facets to create additional suggestions: For example, if we merge two slots, we probably need to merge the inverses of these slots as well.

Many ontologies include definitions of not only classes (collections of similar objects), but also instances (individual objects). It would be useful to include automation and tool support for instance alignment or merging.

Performing merging and alignment in non-OKBC frameworks

We currently assume that the source ontologies conform to the OKBC knowledge model. More specifically, we rely on having a frame-based model with classes, slots, and facets as first-class objects (see Section 2). In the future, we would like to extend SMART to other knowledge models.

We also pay little attention to axioms. Instead, we rely on the knowledge-representation system's built-in ability to enforce axioms. The next important step is to consider how to apply to heavily axiomatized frameworks semi-automated strategies and generation of suggestions similar to the ones described here.

6.2. Evaluation of the Merging and Alignment Process

For any algorithm or tool, it is important to evaluate performance and to validate empirically achievement of goals. Empirical evaluation generally has not been a strong part of artificial intelligence, although some researchers have undertaken it (Cohen 1995).

We can use the standard information-retrieval metrics of precision and recall to evaluate the performance of SMART. **Recall** can be measured as the ratio of the suggestions on the `ToDo` list that the user has followed to the total number of merging and alignment operations that he performed. **Precision** can be measured as the fraction of the suggestions from the `ToDo` list that the user followed. This metric could be augmented by a measure of how many of the conflict resolution strategies in the `Conflicts` list were satisfactory to the user (and how many he had to alter).

The number of operations from Section 5.1 that were invoked during the ontology-merging or alignment process also could serve as a measure of how closely related or far apart the original ontologies were before the process began. Note that the number of performed operations does not measure the *quality* of the process itself. Collecting information about operations performed during merging or alignment could also contribute to a study of the value of prior knowledge: The number of concepts that were *removed* could be used as a measure of the extra knowledge that a knowledge engineer had to create because he did not use the base ontology from the beginning. This metric would not provide a *direct* indication of the value of using the base ontology from the start, because it does not account for the extra body of knowledge could have slowed the development.

7. Special Features of SMART

We referred to many of SMART's special features in our description of how it works in Section 0. We present the features more formally now and discuss our motivations for choosing particular approaches. We first discuss the feedback that SMART provides for the user to explain its actions (Section 7.1). We then give examples of classes of conflicts that SMART identifies (Section 7.2). We describe the strategies that SMART employs to maintain user's focus in Section 7.3. It also keeps track of relations among concepts in the original ontologies in order to restore them in the resulting ontology (Section 7.4).

7.1. Feedback to the User

Every suggestion and every conflict in SMART comes with an explanation (see Figure 5). Explanations include the original reason SMART came up with the suggestion as well as the latest reference that forced the suggestion to change its priority on the list. Consider a MERGE suggestion in Figure 5:

```
MERGE <Cls(Reservation), air_ont> <Cls(Reservation), car_ont>
```

SMART provides the following explanation:

```
Frames have identical names and <Cls(Reservation), air_ont> is
referenced by: <Cls(Reservation_record), merged_ont>
```

The first part of the explanation is the reason why SMART suggested the MERGE operation in the first place and the second part is the reason the operation buoyed to the top of the list (assuming `Reservation_record` was an argument to a recent operation).

7.2. Conflict Types

We identified several types of conflicts that SMART can recognize and create possible solutions for. The list of conflicts SMART can work with is growing constantly as we continue to develop the algorithm. We discuss two of such conflicts here.

Dangling-reference conflict

We have already mentioned a **dangling-reference conflict** that arises when a class is referred to by a slot of type `Class` or `Instance` as allowed class. Since Protégé (and most other knowledge-management systems) does not allow inconsistent states of a knowledge base at any point, SMART creates a "dummy" frame that is referenced from a slot temporarily. When the user resolves the dangling-reference conflict, SMART replaces the "dummy" frame with a real one (or removes the reference). SMART automatically suggests the following approaches to resolving a dangling-reference conflict:

- Copy the referenced class from the source ontology. If there is already a suggestion to merge that class with another class (such as `Reservation` in Figure 5), then perform the MERGE instead of copy.
- Remove the dangling reference from the collection of allowed classes[6].

Duplicate frame names

Another common conflict is **duplicate frame names**. This conflict often arises when two concepts from different source ontologies had the same name (or when a concept in the merged ontology was renamed and the duplication created). SMART proposes the following ways of resolving the conflict:

- When the frames with duplicate names are of the same type (both are classes or both are slots), SMART suggests that the user merges them.
- If the frames are of different types (for example, a slot and a class), SMART suggests renaming one of them.

[6] Not yet implemented.

It is possible that more sophisticated solutions exist in the latter case that we currently do not deal with. For instance, two identically named frames of different types could manifest different approaches to classification in the two ontologies illustrated in Figure 7. In Figure 7(a) class `Reservation_record` has a Boolean slot named `Award_travel` that is true if the reservation is in fact a frequent-flyer-award reservation. In Figure 7(b), class `Award_travel` is a subclass of `Reservation_record` along with `Regular_reservation`. Therefore, there is a slot frame named `Award_travel` in one ontology and a class frame with the same name in another. And it is likely that the two frames do, in fact, refer to the same concept but the two ontologies adopt different approaches in representing the concept.

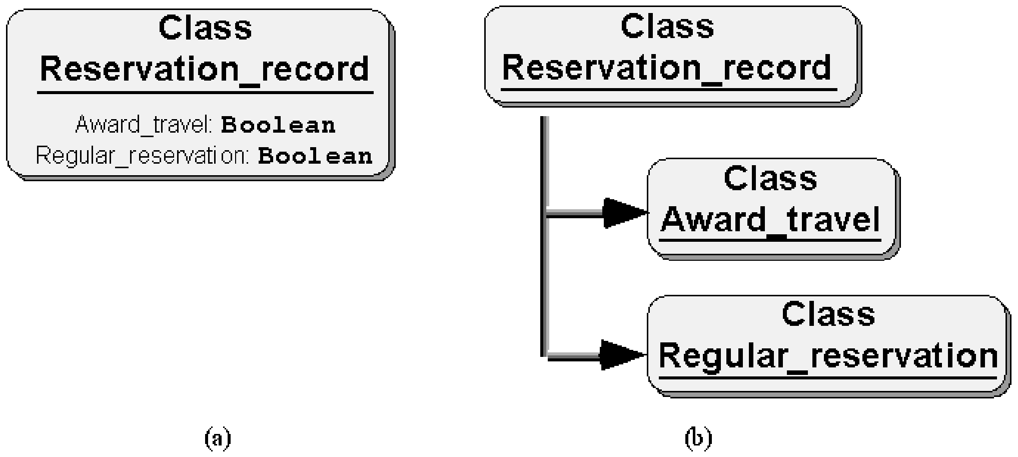


Figure 7.Different modeling approaches. In figure (a) `Award_travel` *is a Boolean slot attached to the* `Reservation_record` *class. In figure (b)* `Award_travel` *is a class that is a subclass of* `Reservation_record` *(solid arrows indicate a superclass–subclass relation*

Even though SMART does not suggest solution based on such modeling difference, it indicates that such a conflict between different frame types exist, enables the user to notice the problem and then solve it.

7.3. Maintaining User's Focus

One of the major reasons that ontology designers need tools that would help them in merging and aligning ontologies is that the source ontologies are usually large and cannot be viewed in one glance. There are just too many concepts and relations to keep track of, think about, and concentrate on. These concepts and relations could be in entirely different parts of the ontology. When we manually aligned ontologies as part of the HPKB project (Fridman Noy and Musen 1999) we observed that the activity is usually happening in one "corner" of the ontologies without switching back and forth to other corners. The same is true for the process of designing a single ontology. With this observation in mind, we designed SMART to maintain user's focus in the current place of interest as much as possible. The **focus** is defined as the location in the target ontology where the latest operation occurred. We use several approaches to keep the focus in place.

The first approach is rearranging ToDo and Conflicts lists after each action to put the operations involving the concepts in the last operation at the top of the lists. For instance, if SMART has just executed an operation that had two arguments C_1 and C_2, it will move operations and conflicts that have subclasses, superclasses, and classes referenced by C_1 and C_2 as their arguments on top of the lists.

The second approach is allowing the knowledge base to be in an inconsistent state temporarily. For instance, if a copied slot refers to a class not yet in the target knowledge base, SMART flags a dangling-reference conflict, which it places at the top of the Conflicts list. SMART does not request that the user solves this conflict immediately, however.

An alternative to leaving a knowledge base in a temporarily inconsistent state would be to interrupt the current operation and ask the user how to resolve the conflict. However, the solution that the user chooses may in turn require more questions and solutions and so on. Not only the process could be cyclical and involve forcing the user to traverse a large part of the ontology without a single operation being completed, but also it forces the user to loose focus and wonder into the parts of the ontology that he was not concentrating on at the moment. Thus, we try to interrupt the user with questions as little as possible and, even if we do, the concepts in the questions are the concepts that are directly referenced by the classes in the current operation. For example, SMART does confirm with the user if it needs to create a frame in the target knowledge base with the name that already exists. The user still has the option of asking SMART to create the frame and resolve the conflict by, for example, renaming the frame later.

7.4. Tracking Relations in the Source Ontologies

We already have referred earlier (Section 0) to the SMART's policy of keeping track of the relationships between frames in the original ontologies and restoring these relationships when it copies the frames into the merged ontology. Even though sometimes this behavior may not be desirable, we believe that most of the times restoring the relations is what the user would have done anyway. For example, suppose we merge the `Driver` class from the Car_rental ontology with the `Individual` class from the Airline ontology (see Figure 3 and Figure 4) to create a class `Individual_customer` in the merged ontology. Suppose that the user then copies the `Customer` class from the Airline ontology into the merged ontology. SMART will automatically make `Individual_customer` a subclass of `Customer` because the prototypes of the two classes had a subclass–superclass relationship before.

Realizing that in some rare cases this may not be the behavior that the user wants and that the user may not notice the newly established relation immediately, we flag all the subclass–superclass relations inferred by SMART. The user can then change several (or all) of the relations and commit to all the other relations SMART established.

We implement the tracking of what the source frames are mapped to by maintaining links from the concepts in the original ontologies (prototypes) to the concepts in the merged ontology into which they were transformed. Because links are between frame objects and are not related to specific frame names or frame locations in the ontology, even if the frame in the merged ontology is renamed or moved to another subtree in the ontology, the link persists and SMART can use it later.

In the future we plan to add to SMART the ability to perform operations on entire trees or collections of selected frames. Working with chunks of ontology rather than individual frames will make some of the tracking not necessary because SMART will keep the relations within the chunks as they were in the original. Moreover, the ability to apply an operation to a collection of frames will allow the user, for example, to rename all classes in a specified subtree of the ontology to conform to the naming conventions of the ontology that the source is being aligned to, for example.

8. Related Work

Researchers in computer science have discussed automatic or tool-supported merging of ontologies (or class hierarchies, or object-oriented schemas—the specific terminology varies depending on the field). However, both automatic merging of ontologies and creation of tools that would guide the user through the process and focus his attention on the likely points for actions are in early stages. In this section, we discuss the existing merging and alignment tools in ontology design and object-oriented programming.

8.1. Ontology Design

One of the few working prototypes of an ontology-merging tool is a system developed by the Knowledge Systems Laboratory (KSL) at Stanford University (Fikes et al. 1999). The system is based on Ontolingua ontology editor developed at KSL (Farquhar, Fikes, and Rice 1996). This system is early prototype that solves initial problems in ontology merging. It can bring together ontologies developed in different formalisms and by different authors.

The Ontolingua approach to merging has similarities to the one described in this paper: The Ontolingua ontology-merging algorithm generates a list of suggestions based on the operations performed by the user. The process starts by running a matching algorithm on class names in both ontologies to suggest the merging points. The matching algorithm looks for the exact match in class names, or for a match on prefixes, suffixes, and word roots of class names. A user can then choose from these matching points, or proceed on his own initiative. After each step performed by the user, the system generates a new set of suggestions.

The Ontolingua's approach to merging is also different from SMART's in many respects. Ontolingua's merging algorithm does not consider slots at all. The process for name matches appears to be depth first and may cause the user to lose focus by finding matches much deeper in the class hierarchy than the current point of concentration. On the other hand, SMART does not consider disjoint classification of categories, whereas the Ontolingua's algorithm does.

The Ontolingua's tool employs user-interface metaphors from the Ontolingua ontology editor—an advantage for the many ontology developers familiar with the ontology editor. The Ontolingua interface is not always easy to use, however, and this drawback seems to carry over to the merging tool.

Another ontology-merging and alignment project at Information Science Institute (ISI) at the University of Southern California (Chapulsky, Hovy, and Russ 1997) attempted to merge extremely large top-level ontologies: Cyc (Lenat 1995) and SENSUS (Knight and Luk 1994). The SENSUS ontology itself resulted from manual merging of the PENMAN Upper Model (Penman 1989), WordNet (Miller 1995), and several other ontologies. In the ISI approach, the creation of an initial list of alignment suggestions relies on more than just class names. The concept features that ISI scores and combines to produce the suggestion list include concepts whose names have long substrings in common; concepts whose definitions (that is, documentation) share many uncommon words; and concepts that have sufficiently high combined name-similarity scores for nearby siblings, children, and ancestors. Experiments have shown that the initial list of suggestions filters out many uninteresting concepts. However, the algorithm stops there: After this initial list is created, it was up to the ontology developer to continue the ontology alignment.

As mentioned in Section 1, medical vocabularies provide a rich field for testing of various ontology-merging paradigms. Not only is there a wide variety of large-scale sources, but also medicine is a field where standard vocabularies change constantly, updates are issued, different vocabularies need to be reconciled, and so on. Oliver and colleagues (Oliver et al. 1999) explored representation of change in medical terminologies using a frame-based knowledge-representation system. The authors compiled a list of change operations that are relevant for the domain of medical terminologies, and developed a tool to support these operations. Many of the operations are similar to the ones described in Section 5. However, the user has to do all the operations manually; there is no automated help or guidance.

8.2. Object-Oriented Programming

One of the better-developed works in the area of ontology merging comes not from the researchers in artificial intelligence, but rather from the researchers in object-oriented programming, who face problems similar to those of ontology researchers. They too may need to bring together several object-oriented class hierarchies (with the associated methods) to create a new product. Subject-oriented programming (SOP) (Harrison and Ossher 1993) supports building of object-oriented systems through composition of subjects. **Subjects** are collections of classes that represent subjective views of, possibly, the same universe. For instance, a class `Shoe` in a shoemaker universe would probably have attributes different from those of the same class in a shoe-seller universe. The class hierarchies that define shoes (and their types) from these different points of view would be the two subjects. If these subjects are to work together (for instance, if the shoemaker supplies shoes to the shoe seller), their corresponding class hierarchies must be merged.

The formal theory of subject-oriented composition (Ossher et al. 1996) defines a set of possible composition rules, these rules' semantics, and the ways that the rules work with one another. Interactive tools for subject-oriented composition are currently under development. There are important differences and similarities in the SOP approach and requirements for class-hierarchy merging and the ontology merging described here.

We begin with the description of the similarities. Merging of class hierarchies covering similar domains is the central task of both ontology merging and subject-oriented composition. The slots of the merged classes (instance variables in object-oriented terminology) must be merged, too. Simply creating union of slots leads to conflicts that must be resolved as well. In fact, our idea of preferred strategy (merge or override) was inspired by the merge and override composition rules used in specifying subject-oriented composition. In both cases, completely automatic matching (or creation of composition rules) is not possible, and interaction with the user (preferably through easy-to-use graphical tools) is needed. In Section 5.3, we introduced the notion of extensions in subject-oriented programming and described the relation of extensions to periodic ontology merging: An extension is a set of rules that defines updates to a class hierarchy. Reapplying extensions after a class hierarchy changes is similar to reapplying the set of logged merging and alignment operations after an ontology changes.

One of the major differences between SOP and our work stems from the fact that subject-oriented composition also needs to compose methods associated with classes. In an ontology, however, classes do not have methods associated with them. On the other hand, ontologies may have axioms that relate different classes or slot values. Object-oriented class hierarchies do not include axioms. Alignment (as opposed to merging) is extremely uncommon in composition of object-oriented hierarchies, whereas it is common in ontology design.

9. Conclusions

We described a general approach to ontology merging and alignment. We presented SMART—an algorithm for semi-automatic merging and alignment. We discussed strategies that SMART uses to guide a user automatically to the next possible point of merging or alignment, to suggest what operations should be performed there, and to perform certain actions automatically. We described the set of basic operations invoked during a merging or alignment process. The more complex operations from this set could be added as front-end operations in OKBC. The strategies and algorithms described in this paper are based on a general OKBC-compliant knowledge model. Therefore, these results are applicable to a wide range of knowledge-representation and ontology-development systems.

Acknowledgements

Parts of this paper appeared in an earlier publication (Fridman Noy and Musen 1999). We are extremely grateful to Ray Fergerson for help with the implementation. Monica Crubézy and William Grosso provided valuable comments on earlier drafts of this paper. This work was funded in part by the High-Performance Knowledge-Base Project of the Defense Advanced Research Projects Agency.

References

Chapulsky, H., Hovy, E. and Russ, T. 1997. Progress on an Automatic Ontology Alignment Methodology. ANSI Ad Hoc Group on Ontology Standards; *available at* http://ksl-web.stanford.edu/onto-std/hovy/index.htm.

Chaudhri, V. K., Farquhar, A., Fikes, R., Karp, P. D. and Rice, J. P. 1998. OKBC: A Programmatic Foundation for Knowledge Base Interoperability. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*: 600-607. Madison, Wisconsin: AAAI Press/The MIT Press.

Chaudhri, V. K., Farquhar, A., Fikes, R., Karp, P. D. and Rice, J. P. 1998. Open Knowledge Base Connectivity 2.0.3, Specification document.

Cohen, P., Schrag, R., Jones, E., Pease, A., Lin, A., Starr, B., Gunning, D. and Burke, M. 1999. The DARPA High-Performance Knowledge Bases Project. *AI Magazine* 19(4): 25-49.

Cohen, P. R. 1995. *Empirical Methods for Artificial Intelligence*. Cambridge, MA: MIT Press.

Farquhar, A., Fikes, R. and Rice, J. 1996. The Ontolingua Server: a Tool for Collaborative Ontology Construction. In *Proceedings of the Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop*. Banff, Canada.

Fikes, R., McGuinness, D., Rice, J., Frank, G., Sun, Y. and Qing, Z. 1999. Distributed Repositories of Highly Expressive Reusable Knowledge. Presentation at HPKB meeting, Austin, TX; *available at* http://www.teknowledge.com/HPKB/meetings/Year1.5meeting/.

Fridman Noy, N. and Musen, M. A. 1999. An Algorithm for Merging and Aligning Ontologies: Automation and Tool Support. In *Proceedings of the Workshop on Ontology Management at the Sixteenth National Conference on Artificial Intelligence (AAAI-99).* Orlando, FL: AAAI Press.

Grosso, W., Eriksson, H., Fergerson, R., Gennari, J., Tu, S., Musen, M. 1999. Knowledge Modeling at the Millenium (The Design and Evolution of Protégé-2000). Submitted to the *Twelfth Banff Knowledge Acquisition for Knowledge-Bases Systems Workshop*. Banff, Canada.

Harrison, W. and Ossher, H. 1993. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'93)*: 411-428. Washington, DC: ACM Press.
High Performance Knowledge Bases 1999: http://www.teknowledge.com:80/HPKB/.

Knight, K. and Luk, S. K. 1994. Building a Large-Scale Knowledge Base for Machine Translation. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*. Seattle, Washington: AAAI Press.

Lenat, D. B. 1995. CYC: A Large-Scale Investment in Knowledge Infrastructure. *Communications of ACM* 38(11): 33-38.

Miller, G. A. 1995. WordNet: A Lexical Database for English. *Communications of ACM* 38(11): 39-41.

Musen, M. A., Gennari, J. H., Eriksson, H., Tu, S. W. and Puerta, A. R. 1995. Protégé-II: Computer support for development of intelligent systems from libraries of components. In *Proceedings of the Medinfo'95*: 766-770. Vancouver, BC.

Oliver, D. E., Shahar, Y., Shorliffe, E. H. and Musen, M. A. 1999. Representation of Change in controlled medical terminologies. *Artificial Intelligence in Medicine* 15: 53-76.

Ossher, H., Kaplan, M., Katz, A., Harrison, W. and Kruskal, V. 1996. Specifying Subject-Oriented Composition. *Theory and Practice of Object Systems* 2(3): 179-202.

Penman 1989. The Penman documentation, Technical report, USC/Information Sciences Institute, Marina del Rey, CA.