From Wrapping to Knowledge

José L. Arjona, Rafael Corchuelo, David Ruiz, and Miguel Toro

Abstract—One the most challenging problems for Enterprise Information Integration is to deal with heterogeneous information sources on the Web. The reason is that they usually provide information that is in human-readable form only, which makes it difficult for a software agent to understand it. Current solutions build on the idea of annotating the information with semantics. If the information is unstructured, proposals such as S-CREAM, MnM, or Armadillo may be effective enough since they rely on using natural language processing techniques; furthermore, their accuracy can be improved by using redundant information on the Web, as C-PANKOW has proved recently. If the information is structured and closely related to a back-end database, Deep Annotation ranges among the most effective proposals, but it requires the information providers to modify their applications; if Deep Annotation is not applicable, the easiest solution consists of using a wrapper and transforming its output into annotations. In this paper, we prove that this transformation can be automated by means of an efficient, domain-independent algorithm. To the best of our knowledge, this is the first attempt to devise and formalize such a systematic, general solution.

Index Terms—Enterprise information integration, wrappers, semiautomatic annotation.

1 Introduction

Enterprise application integration (EAI) amounts to designing a piece of software that allows several interfaces to work together. Our research focuses on Web applications that need to integrate several sources of information to provide value-added knowledge. In this context, ubiquitous protocols and languages such as HTML, SOAP, or XML provide the foundations since they allow us to connect heterogeneous systems so that they can exchange data. Notice that for these data to be useful, it is commonly necessary to transform them into a common representation with agreed semantics [1], [2], and this is the goal of Enterprise Information Integration (EII).

Many researchers agree that the Semantic Web might soon become a cornerstone for EII since languages such as RDF or OWL and their accompanying technologies provide the foundations for working with knowledge. The idea is to enhance current Web pages with annotations that describe their semantics, both from an intensional and an extensional point of view. As a result, the Web remains human-readable, and the annotations make it easier for a machine to interpret and process its contents [3], [4]; some researchers are also working on making these annotations easier to understand for humans [5]. Common annotations range from simple flat templates to relational metadata that make it explicit the relationships among several knowledge entities; however, some authors have recently pointed out that for the Semantic Web to achieve its full potential other kinds of annotations based on implicit or soft semantics should also be taken into account [6]. Whatever the kind of annotation is, it is clear that the more interrelated, the most valuable the knowledge [7].

However, no technology is effective unless it is integrated into an appropriate methodological framework. CREAM is such a framework, and it provides both guidelines and tools to help software engineers benefit from the Semantic Web [7]. Recently, the authors extended it with a method called Deep Annotation that targets EII projects in which the applications to integrate provide a Web interface to a back-end database [8], [9]. The method can be applied as long as the information providers are cooperative and agree on changing their applications so that the Web pages they produce are annotated with information about the databases and the queries from which they originate. If the information provider is not so cooperative, be it because the costs or the benefits are not worth, or the method is not applicable, be it because the information comes from different sources or the data needs to be processed by a program before it is transformed into a Web page, several researchers have suggested that the solution should rely on using a class of information extractors known as wrappers [10], [11], [12], [13], [14], [15]. A wrapper is an algorithm that extracts a predefined collection of attributes from a Web page, i.e., literals that carry the information of interest, and organizes them into slots and records.

Our research focuses on EII problems in which the applications to integrate provide information that is both structured and local, e.g., guest portfolios, NASDAQ quotes, or medical records. By structured, we mean that the information is presented to the user in a form whose underlying hierarchical structure may be induced by analyzing several related pages; by local, we mean that the information is not likely to be broadly found on the Web, although it may be useful in many different contexts [16]. Fig. 1 sketches part of the architecture of our EII agents. A typical workflow is as follows: A Web application needs a piece of knowledge and issues a query to a retrieval agent,

- *J.L. Arjona is with the Departamento de Electrónica, Sistemas Informáticos y Automática, Escuela Politécnica Superior, Campus de la Rábida, Pabellón Torreumbría, Carretera Huelva-La Rábida, 21071 Palos de la Frontera Huelva, Spain. E-mail: jose.arjona@diesia.uhu.es.*

- *R. Corchuelo, D. Ruiz, and M. Toro are with the Departamento de Lenguajes y Sistemas Informáticos, ETSI Informática, Avda. de la Reina Mercedes, s/n, 41012 Sevilla, Spain. E-mail: {corchu, druiz, mtoro}@us.es.*

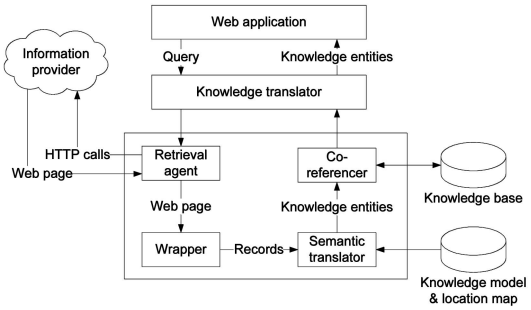


Fig. 1. Architecture of an EII agent.

which transforms it into the appropriate HTTP calls and waits for a Web page to be returned. (Notice that this agent may need to navigate through a series of authentication or cataloging pages before reaching its target.) Once the server returns the Web page, the retrieval agent passes it on to a wrapper that is responsible for extracting the attributes of interest and organizing them into slots and records. The records are then passed on to the semantic translator, which transforms them into interrelated knowledge entities by using a minimal ontology to which we refer to as the knowledge model, and a location map that helps relate a record to a knowledge model. Once a knowledge entity is built, it needs to be coreferenced in an attempt not to produce several versions of the same real-life entity [17]. Notice that a single EII agent may serve several Web applications that must be intimately related but need not share the same ontology. This is the reason why the EII agent works with a minimal ontology that includes information about the structure of the knowledge entities only [18]. Additional constraints are application-dependent and must be dealt with by an appropriate knowledge translator [1], [19], if necessary. Regarding the language used to represent knowledge models and entities, there are a plethora of choices, but we do not commit on any of them [20]. Instead, we commit to a small subset of Description Logics (DL), which seems to be the common root for many such languages [21]; this provides plenty of leeway in selecting the most appropriate tools for each EII problem.

The details on CREAM [7], the retrieval agent [22], the wrappers [11], [15], [23], and the coreferencer [24] are described elsewhere. In this paper, we focus on the semantic translators, and prove that they can be implemented in $O(n \cdot m \cdot \log m)$ time by means of a domain-independent algorithm, where n denotes the number of records to be translated and m the average number of slots per record. As a conclusion, the proposal in this paper complements the CREAM framework with an effective tool that helps solve EII problems in cases in which Deep Annotation is not applicable and the information of interest is structured and local. The rest of the paper is organized as follows: Section 2 provides an insight into the most closely-related proposals; Section 3 provides the foundations of our algorithm, which is presented in Section 4 and analyzed in Section 5; Section 6 concludes the paper, and the Appendix provides a short introduction to the mathematical notation we use.

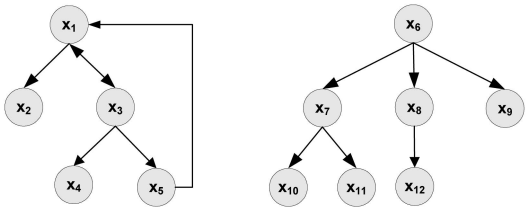
2 Related Work

S-CREAM is a closely related proposal [25] that relies on using taggers, a class of information extractors that recognize literals that denote dates, prices, or named entities such as actors, hotels, or countries. They usually build on techniques that originated in the natural language processing community, which makes them appropriate for unstructured pages in which the information of interest is contained in natural language passages. S-CREAM attempts to transform a Web page into a set of knowledge entities by applying several heuristics to the tags produced by the Amilcare tagger [26]. The method was enhanced recently with PANKOW [27] and C-PANKOW [28] in an attempt to improve its precision to recognize named entities by using redundant information in the Web. Proposals such as KIM [29], Armadillo [30], or MnM [31] are similar in spirit to S-CREAM.

From the above discussion, it is clear that the idea of using information extractors has attracted many researchers, and this has led to quite a broad range of solutions in domains such as news servers, departmental sites, scientific databases, or community portals; the common underlying theme is that the data of interest is unstructured and contains redundant information that can be used to validate the results, to disambiguate conflicting terms, or to infer additional information [30], [32]. The kind of EII problems with which we deal differs in that the information is structured and local, which makes the above systems of little interest since they can neither benefit from natural language processing techniques, nor use other information sources to improve their precision.

Regarding wrapper generators, the most classical proposals include inductive methods such as Wien [12], which can deal with multirecord pages, Stalker [14] and SoftMealy [33], which improve in that they deal with missing attributes and attribute permutations, or NoDoSe [34], which improves in that it deals with records whose slots are organized hierarchically. W4F or RoadRunner range amongt the most recent proposals. W4F is a wrapper generator toolkit that uses a declarative rule-based language called HEL [15]. The rules are XPath-like expressions that describe navigation paths to the attributes of interest. The toolkit also provides a simple template language to map records onto Java objects and XML documents. The mapping onto Java is declarative if simple types such as strings or arrays of strings are used; otherwise, it must be defined programmatically. The mapping onto XML documents is specified by using a simple template language that requires the XML to have the same hierarchical structure as the source records. Although it is very effective, W4F differentiates from the above-mentioned systems in that it does not attempt to infer the extraction rules or the templates, which must be devised by an engineer with the help of a wizard. RoadRunner improves on the rest in that it fully automates the generation of wrappers [11]. It provides a domain-independent algorithm that analyzes the similarities among several Web pages and infers their common grammar in most common cases.

Summing up, there are a variety of wrappers available. Unfortunately, very little has been said on how to transform



(a) (b)

Fig. 2. Graphs and trees lie at the heart of our data structures and algorithms. The circles denote the vertices, and the arrows the edges between them. The vertices are usually referred to by means of a name that is written inside them. (a) Sample graph. (b) Sample tree.

their outputs into knowledge entities. This is not trivial at all since the number of records of information per page may vary, some attributes may be optional or missing, and the hierarchy into which the wrapper organizes them has nothing to do with the structure of the corresponding knowledge entities in most common cases. The reason is that the structure of a record is heavily influenced by the structure of the Web page to which it corresponds, whereas the structure of a knowledge entity must match the structure of its underlying knowledge model [19].

3 Preliminaries

In this section, we define the abstract data types (ADTs) on which our algorithm relies. The definitions are rather operational since our goal was to produce an implementation. This is why we represent records, knowledge models, or knowledge entities using trees instead of a more abstract representation based on logical formulae, for instance. The definitions are, however, abstract enough to be independent from specific programming languages.

3.1 Vertices, Edges, and Paths

Vertices are the simplest data structure we use in our algorithm; they allow us to build graphs and trees and may carry data that range from literals in a slot to the name of a concept in a knowledge model. Building on them, we may form edges or paths by combining pairs or sequences of vertices, respectively.

Their formal definition is as follows:

$$
\begin{aligned}
&[Vertex]\\
&Edge == Vertex \times Vertex\\
&Path == \{p : \text{ seq } Vertex \mid \#p \geq 2\}.
\end{aligned}
$$

3.2 Graphs and Trees

A graph is composed of a nonempty set of vertices and a possibly empty set of edges between them. For instance, in Fig. 2a, we depict a graph that is composed of vertices $\{x_1, x_2, x_3, x_4, x_5\}$ and edges

$$\{(x_1, x_2), (x_1, x_3), (x_3, x_1), (x_3, x_4), (x_3, x_5), (x_5, x_1)\}.$$

In theory, a graph may have an infinite number of vertices and edges, but we restrict our attention to finite graphs since we use them to represent the information extracted from a Web page, which is finite by definition. Notice, too, that the edges are directional and, thus, the existence of an edge between any two vertices does not imply that there must exist an inverse edge between them.

Graph

[Structure]
$vertices : \mathbb{F}\, Vertex$
$edges : \mathbb{F}\, Edge$
[Ancillary definitions]
$paths : \mathbb{P}\, Path$
$children : Vertex \nrightarrow \mathbb{F}\, Vertex$
$leaves : \mathbb{F}\, Vertex$

[Invariants]
[1] $vertices \neq \varnothing$
[2] $\forall v_1, v_2 : Vertex \mid (v_1, v_2) \in edges \bullet \{v_1, v_2\} \subseteq vertices$
[Ancillary definitions]
[1] $paths = \{p : Path \mid \forall i : 1 \,..\, \#p - 1 \bullet (p(i), p(i+1)) \in edges\}$
[2] $\forall v : Vertex \mid v \in vertices \bullet children(v) = \{u : Vertex \mid (v, u) \in edges\}$
[3] $leaves = \{v : Vertex \mid v \in vertices \bullet children(v) = \varnothing\}$

Fig. 3. A scheme to model graphs.

The scheme in Fig. 3 defines a graph ADT formally and introduces several ancillary functions that facilitate the definition of the forthcoming schemes. The first invariant requires the set of vertices to be nonempty, and the second one requires the vertices connected by means of an edge to belong to the same graph. $paths$ is an ancillary set that holds the paths in which every vertex is connected to the next one by means of an edge; $children$ is an ancillary function that returns the set of vertices to which a given vertex is connected by means of an edge; $leaves$ is an ancillary set that holds the vertices that do not have any children. For instance, in the graph depicted in Fig. 2a, $paths$ includes the sequences of edges $\langle x_1, x_2\rangle$, $\langle x_1, x_3, x_4\rangle$, or $\langle x_1, x_3, x_1, x_2\rangle$ amongst others, and $leaves = \{x_2, x_4\}$; similarly, $children(x_1) = \{x_2, x_3\}$ and $children(x_3) = \{x_1, x_4, x_5\}$.

Notice that the set of paths in an arbitrary graph may be infinite if there are cycles, but this is not a problem since we only work with graphs that are trees, i.e., graphs in which there is a distinguished vertex called root that is connected to every other vertex by means of a unique path, which guarantees that cycles may not exist. For instance, Fig. 2b shows a sample tree in which the root is vertex x_6.

Building on the graph ADT, we define the tree ADT in Fig. 4. The first invariant states that the root vertex belongs to the set of vertices, and the second one that no other vertex may be disconnected from the root or connected to it by means of more than one path.

Tree

[Structure]
Graph
$root : Vertex$

[Invariants]
[1] $root \in vertices$
[2] $\forall v : Vertex \mid v \in vertices \setminus \{root\} \bullet \exists_1 p : Path \mid p \in paths \bullet front\, p = root \wedge last\, p = v$

Fig. 4. A scheme to model trees.

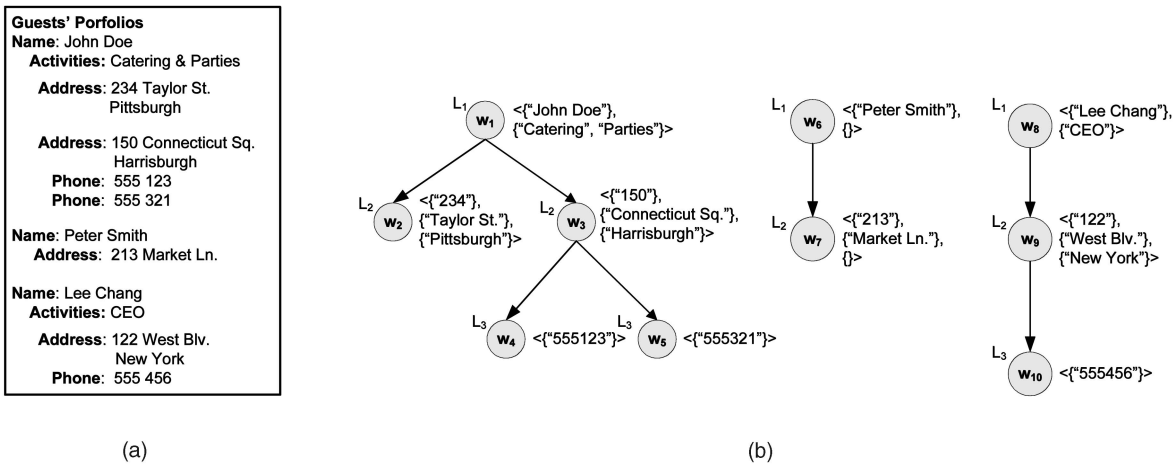


Fig. 5. A sample Web page and the records a wrapper has extracted from it. The circles represent slots, and the edges denote their hierarchical relationships within a record; the labels are shown on the left, whereas the attributes are shown on the right. (a) Sample Web page. (b) Sample wrapper output.

Record

[Structure]

Tree

$attributes : Vertex \pfun \operatorname{seq} Attribute$

$label : Vertex \pfun Label$

[Invariants]

[1] $\operatorname{dom} label = vertices$

[2] $\operatorname{dom} attributes = vertices$

[3] $\forall w : Vertex \mid w \in vertices \bullet \exists a : Attribute \mid \langle a \rangle \operatorname{in} attributes(w) \bullet a \neq \varnothing$

[4] $\forall w_1, w_2 : Vertex \mid \{w_1, w_2\} \subseteq vertices \land label(w_1) = label(w_2) \bullet \#attributes(w_1) = \#attributes(w_2)$

Fig. 6. A scheme to model records.

3.3 Records

A wrapper is an algorithm to extract attributes from a Web page, i.e., sets of literals that carry the information in which we are interested. For instance, in Fig. 5a, we show a page with information about three guest portfolios; the attributes in which we are interested are the guests' names, their professional activities, and their contact addresses, which are composed of a house number, a street, a city, and a list of phone numbers. In practice, many attributes are optional or multivalued, and this is the reason why we represent them as finite sets of literals:

$$[Literal]$$
$$Attribute == \mathbb{F}\ Literal.$$

Most recent wrappers organize the attributes into records, which are trees that resemble the structure of the Web page from which they were extracted. The vertices of such trees are referred to as slots, and they must be associated with a sequence of attributes and a label. Intuitively, each slot must provide information about one or more knowledge entities, and all of the slots with the same label must provide information about the same kind of entity; notice, however, that slots with different labels may still provide information about the same kind of entity. For instance, Fig. 5b shows three records that a wrapper has extracted from our sample page; a slot with label L_1 provides information about a guest's name and his or her professional activities; a slot with label L_2, provides information about a house number, a street, and a city; similarly, a slot with label L_3 provides information about a phone number.

The scheme in Fig. 6 defines the record ADT we use, which builds on the tree ADT and adds two functions called $attributes$ and $label$ that return the sequence of attributes and the label associated with each slot in a record, respectively. The first invariant is trivial since it just requires every vertex in a record, i.e., every slot, to have a label. The second and the third invariant ensure that every slot has at least an attribute, since it does not make sense to create a slot that does not hold any attributes. The fourth invariant attempts to capture the intuitive idea that the slots with the same label must have information about the same kind of entity; since we cannot model this requirement formally, we at least require them to have the same number of attributes. Regarding the example in Fig. 5b, it is easy to realize that $label(w_1) = L_1$, $label(w_3) = L_2$, and $label(w_7) = L_2$, just to mention a few cases; similarly,

$$attributes(w_1) = \langle \{''\text{John Doe}''\}, \{''\text{Catering}'', ''\text{ Parties}''\} \rangle,$$
$$attributes(w_3) = \langle \{''150''\}, \{''\text{Connecticut Sq.}''\}, \{''\text{Harrisbugh}''\} \rangle,$$

i.e., both slots w_1 and w_3 are plenty of information, but $attributes(w_7) = \langle \{''213''\}, \{''\text{Market Ln.}''\}, \emptyset \rangle$ since the Web page does not have any information about the city in which the address "213 Market Ln." is located.

The record ADT is general enough to accommodate both older wrappers that are not able to analyze the hierarchical

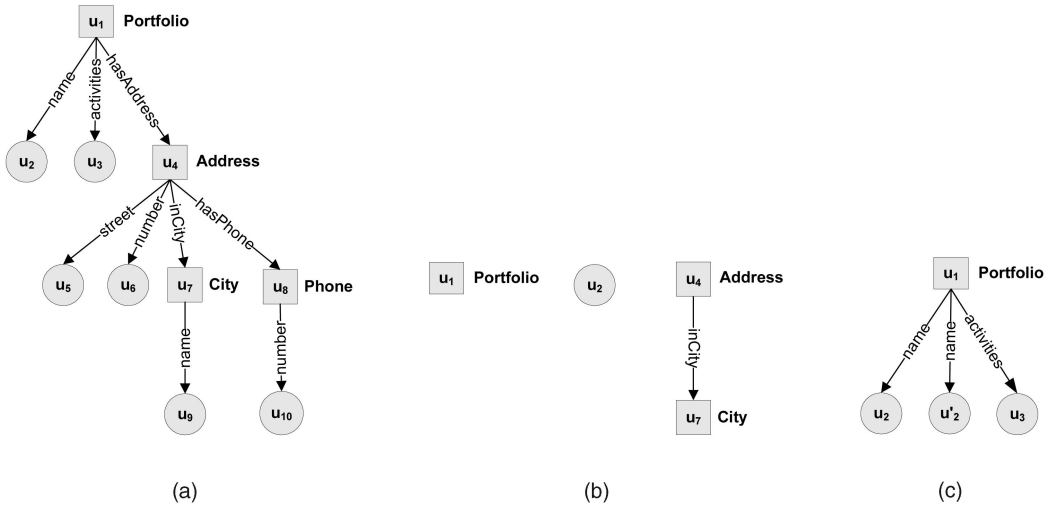


Fig. 7. A valid knowledge model for our portfolios and several invalid ones. The vertices and the edges are labelled with the concepts and properties they represent, respectively; squared vertices represent object vertices, whereas rounded vertices represent data vertices, which makes it unnecessary to label them with concept $DATUM$. (a) A valid knowledge model. (b) Some invalid knowledge models. (c) Another invalid knowledge model.

structure of a Web page and more recent wrappers that benefit from it. The former case is dealt with by means of trees that consist of just one slot, but this is not a special case in our proposal since it is completely independent from whether the record under consideration has just one or several slots.

3.4 Knowledge Models

DL provides a rich set of constructs to represent knowledge models by means of TBoxes, but we just need a structural description of the concepts and properties involved in a given problem. Thus, we first introduce two sets to denote concept names, also known as classes, and property names, also known as roles. The latter can relate either two concepts, also known as object properties, or a concept and a literal, also known as data properties. Furthermore, we assume that there is a predefined concept name called $DATUM$ that denotes literal data exclusively.

$$[ConceptName, PropertyName]$$
$$DATUM : ConceptName.$$

Fig. 7a depicts a knowledge model for our running example; it defines concepts $Portfolio$, $Address$, $City$, and $Phone$ together with several data vertices and properties to hold information about the guests' names, their activities, and so on. Notice that some property names are overloaded, which is very common in practice; for instance, property $name$ is ambiguous since it may refer to either the property that assigns a guest's name to a portfolio or the property that assigns a name to a city. Whenever necessary, we provide enough information to disambiguate these properties.

Our formal definition builds on an ancillary scheme that is presented in Fig. 8. It helps us define a common representation for knowledge that is used both to define knowledge models and knowledge entities. This ancillary scheme builds on the tree ADT and complements it with a function called $concept$, which assigns a concept name to every vertex, and a function called $property$, which assigns a property name to each edge; furthermore, we define two ancillary sets that allow us to refer to the data and object vertices more easily. The first and the second invariants are trivial since they just require every vertex in a knowledge representation to be associated with a concept, and every edge to be associated with a property. The third invariant requires the set of edges not to be empty, which implies that there must be at least a property and, consequently, an object vertex and a data vertex; this is not a shortcoming, but simply a requirement to avoid empty knowledge models. The fourth invariant requires the leaves of the tree to be the only ones that may be associated with concept $DATUM$, i.e., the only ones that may carry the attributes extracted from a Web page; this makes sense since intermediate vertices correspond to object vertices that may have a name, but not an attribute unless it is assigned by means of a data property. Fig. 7b illustrates three invalid knowledge representations; the former is invalid because it represents a knowledge entity without any data properties, which does not seem to be reasonable in practice since this would imply that we would be interested in pages that do

KnowledgeRepresentation

[Structure]

$Tree$

$concept : Vertex \nrightarrow ConceptName$

$property : Edge \nrightarrow PropertyName$

[Ancillary definitions]

$data : \mathbb{F}\ Vertex$

$objects : \mathbb{F}\ Vertex$

[Invariants]

[1] $\mathrm{dom}\ concept = vertices$

[2] $\mathrm{dom}\ property = edges$

[3] $\mathrm{dom}\ property \neq \varnothing$

[4] $\forall v : Vertex \mid v \in vertices \bullet (concept(v) = DATUM \Leftrightarrow v \in leaves)$

[Ancillary definitions]

[1] $data = leaves$

[2] $objects = vertices \setminus data$

Fig. 8. A scheme to model knowledge representations.

KnowledgeModel

[Structure]

KnowledgeRepresentation

[Invariants]

[1] $\nexists s : \mathbb{F}\, Vertex \mid s \subseteq vertices \bullet \exists v : Vertex \mid v \in vertices \setminus s \bullet \exists_1 cn : ConceptName, pn : PropertyName \bullet$
$\#s \geq 2 \land (\forall u : Vertex \mid u \in s \bullet u \in children(v) \land concept(u) = cn \land property(v, u) = pn)$

Fig. 9. A scheme to model knowledge models.

not provide any data; the second is invalid because it consists of just a data vertex, which does not seem reasonable since this data would be isolated; the third case is similar to the first one. The ancillary definitions introduce the set of data vertices as the set of leaves, and the set of object vertices as the set of intermediate vertices, which simplifies the definition of the forthcoming schemes.

Building on this simple knowledge representation, the scheme to define knowledge models is presented in Fig. 9. The invariant avoids knowledge models in which there is a subset of vertices with the same concept that have a common parent with which they all are related by means of the same property. This avoids a knowledge representation from having several definitions of the same property, be them redundant or erroneous, see Fig. 7c.

Our algorithm translates each slot independently and recombines the results in a bottom-up manner; this leads to many partial knowledge entities that need to conform to their own partial knowledge models. For instance, a slot with label L_1 provides information about a guest's name and his or her professional activities, a slot with label L_2 provides information about an address, and a slot with label L_3 provides information about a phone number; it is thus reasonable to have the partial knowledge models in Fig. 10. Next, we introduce a function that formalizes the set of partial knowledge models that can be obtained from a given knowledge model:

$$
\begin{array}{l}
partialModels : KnowledgeModel \nrightarrow \mathbb{F}\, KnowledgeModel \\
\hline
\forall km : KnowledgeModel \bullet \\
\quad partialModels(km) = \{pm : KnowledgeModel \mid \\
\qquad pm.concept \subseteq concept \land \\
\qquad pm.property \subseteq property\}
\end{array}
$$

Notice that we have introduced this function as an independent axiomatic definition because every knowledge model is included in its set of partial models. If we had included the definition of this function in scheme

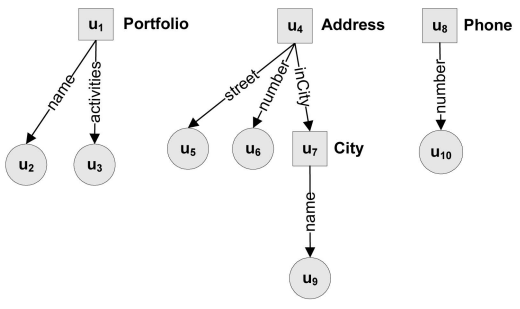


Fig. 10. Some partial knowledge models.

$KnowledgeModel$, this would have led to infinite recursion and the definition of the scheme would have been wrong.

3.5 Knowledge Entities

A knowledge entity is an instantiation of a knowledge model in which every object vertex is identified by means of a universal name, and every data vertex has an associated attribute. For instance, Fig. 11 shows a knowledge entity that has been obtained from the first record in Fig. 5b and conforms to the knowledge model in Fig. 7a.

We begin our formal definition by introducing the set of entity names. We do not commit to a particular representation, which may range from standard URIs to ad hoc UUIDs depending on the context, but require every possible vertex to have a different name. This is accomplished by means of a name generator that must conform to the following specification:

$$
\begin{array}{l}
[EntityName] \\
createName : Vertex \rightarrow EntityName \\
\hline
\forall v_1, v_2 : Vertex \mid v_1 \neq v_2 \bullet \\
\quad createName(v_1) \neq createName(v_2)
\end{array}
$$

The scheme we use to model knowledge entities is presented in Fig. 12. It builds on the

$$KnowledgeRepresentation$$

scheme, too, and adds two functions called $name$ and $attribute$ that map each object vertex onto a name and each data vertex onto an attribute, respectively. For instance, regarding the example in Fig. 11, it is easy to realize that $name(v_1) = ID_1$ or $name(v_4) = ID_2$, for instance, whereas

$$
\begin{aligned}
&attribute(v_2) = \{''\text{John Doe}''\} \text{ and} \\
&attribute(v_6) = \{''\text{Taylor St.}''\}.
\end{aligned}
$$

The invariants are trivial since they just require every object vertex to have a name and every data vertex to have an attribute, but we have added several ancillary predicates and functions that are very useful to determine what the partial knowledge model of a given knowledge entity is. The unary predicate $conformsTo$ checks if a knowledge entity conforms to a given knowledge model; this holds iff there is a total surjective mapping from the vertices of the knowledge entity onto the vertices of the knowledge model so that corresponding vertices have the same concept, and corresponding edges have the same property. For instance, Fig. 13a shows a knowledge entity and its corresponding partial knowledge model. Notice that this knowledge entity corresponds to an address with several phone numbers, which implies that data vertices v_{12} and v_{13} are both mapped onto object vertex u_8 in the knowledge model, for

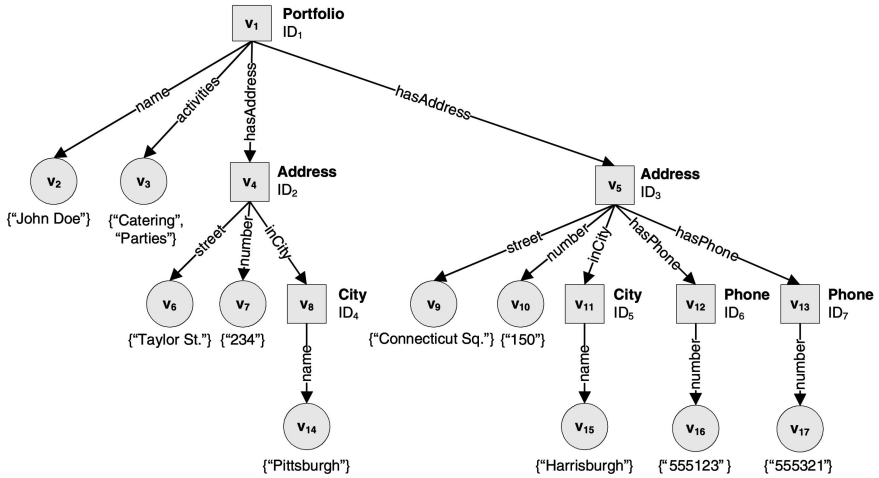


Fig. 11. A knowledge entity for portfolio "John Doe." For the sake of simplicity, we use $ID_1, ID_2, \ldots$ to denote the names of the object vertices, but an actual implementation would rather use URIs, for instance. The attributes that correspond to each data vertex are written below them.

instance. Figs. 13b and 13c, however, illustrate a couple of knowledge entities that do not conform to their corresponding knowledge models due to lack of surjectivity, i.e., the knowledge model provides more information than it is required, or due to lack of totality, i.e., the knowledge model does not provide enough information.

It is easy to realize that for every knowledge entity and model, there may exist one such mapping at most; thus, function $mapping$ returns either a singleton or an empty set, which eases the definition of predicate $conformsTo$. The scheme also introduces a function called $correspondsTo$ that, given a data vertex in a knowledge entity, locates its corresponding vertex in a given knowledge model; obviously, this requires the knowledge entity to conform to the knowledge model and uses the unique mapping between them both to find the result. For instance, v_5 corresponds to u_4 in Fig. 13a, which means that the semantics of the object vertex identified by means of ID_3 in the knowledge entity is defined by vertex u_4 in the knowledge model.

KnowledgeEntity

[Structure]

$KnowledgeRepresentation$

$name : Vertex \pfun EntityName$

$attribute : Vertex \pfun Attribute$

[Ancillary definitions]

$conformsTo\_ : KnowledgeModel$

$correspondsTo : Vertex \times KnowledgeModel \pfun Vertex$

$mapping : KnowledgeModel \pfun \mathbb{F}(Vertex \pfun Vertex)$

[Invariants]

[1] $\mathrm{dom}\, name = objects$

[2] $\mathrm{dom}\, attribute = data$

[Ancillary definitions]

[1] $\forall km : KnowledgeModel \bullet conformsTo(km) \Leftrightarrow mapping(km) \neq \varnothing$

[2] $\forall v : Vertex;\ km : KnowledgeModel \mid v \in data \land conformsTo(km) \bullet$
$correspondsTo(v, km) = (\mu f : Vertex \pfun Vertex \mid f \in mapping(km))(v)$

[3] $\forall km : KnowledgeModel \bullet mapping(km) =$
$\{f : Vertex \pfun Vertex \mid \mathrm{dom} f = vertices \land \mathrm{ran} f = km.vertices \land$
$(\forall v : Vertex \mid v \in vertices \bullet concept(v) = km.concept(f(v))) \land$
$(\forall v_1, v_2 : Vertex \mid (v_1, v_2) \in edges \bullet$
$(f(v_1), f(v_2)) \in km.edges \land property(v_1, v_2) = km.property(f(v_1), f(v_2)))\}$

Fig. 12. A scheme to model knowledge entities.

3.6 Location Maps

A location map is a simple structure that maps a subset of vertices onto a subset of labels and indices. For instance, Fig. 15 illustrates the location map for our running example; it states that vertex u_2 in our knowledge model must be instantiated from the first attribute of a slot with label L_1; similarly, vertex u_5 must be instantiated from the second attribute of a slot with label L_2. Alternatively, the map may also be interpreted by stating that a slot with label L_1, for instance, instantiates data vertices u_2 and u_3 in the knowledge model.

The scheme in Fig. 14 defines this simple ADT formally. Notice that there are no invariants regarding $label$ or $index$. The reason is that the constraints depend completely on the record analyzed and the knowledge model under consideration. Such constraints are thus part of our algorithm, and they are explained in the following section.

4 Semantic Translation

Intuitively, our algorithm consists of two steps and a driver that combines them both: the former analyzes each slot independently and uses the information in a location map to determine which part of the knowledge model under consideration must be instantiated; the second step attaches the results of the first step by means of the appropriate properties; the driver cares of synchronizing both steps so that every slot in the input record is translated appropriately. The following sections report on the details behind this process.

4.1 Configurations

A configuration is a triple composed of a record, a knowledge model, and a location map. The record has the attributes a wrapper has extracted from a Web page, and the knowledge model provides the semantics behind the knowledge entity into which our algorithm transforms the record; the location map provides part of the information required to guide the process.

Building on the previous structures, the configuration ADT is presented in Fig. 16. In the following paragraph,

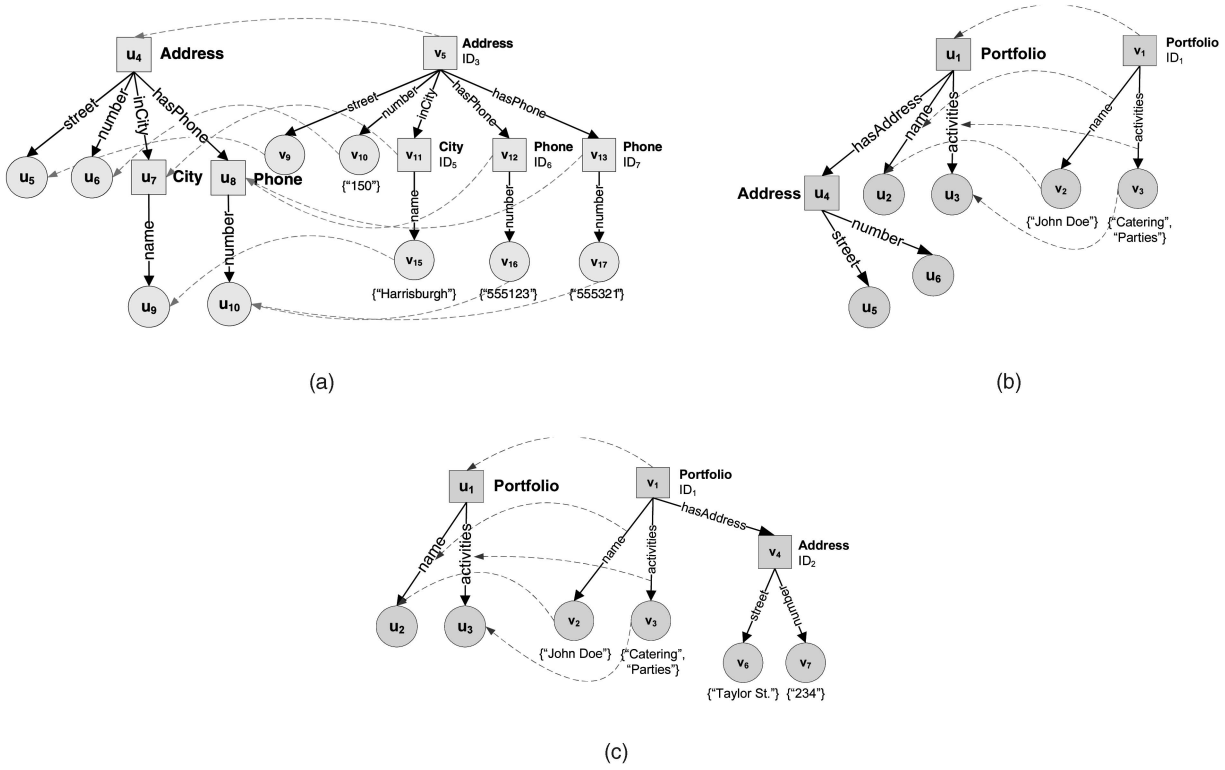


Fig. 13. Valid knowledge entities must conform to a knowledge model. The dotted lines represent the (partial) mappings that associate the vertices and edges in the knowledge entities with their corresponding vertices in the knowledge models. (a) Conformance to a knowledge model. The mapping among the properties is not shown since it would be unreadable. (b) Nonconformance due to lack of surjectivity. (c) Nonconformance due to lack of totality.

we provide an explanation of the invariants, which rely on three ancillary functions, namely, $attribute$, which given a data vertex in a knowledge model and a slot, returns the attribute that the slot holds for that data vertex; $correspondingData$, which is applied to a slot and returns the maximum set of data vertices in the knowledge model that a slot with the same label may instantiate; and $influenceArea$, which is applied to a slot and returns the minimum partial knowledge model that includes the data vertices returned by the previous function. Intuitively, the influence area of a slot denotes the minimum partial knowledge model that describes the semantics of the partial knowledge entity that may be instantiated by using the attributes in that slot. Regarding slot w_1 in Fig. 5b and the location map in Fig. 15, for instance, it is easy to realize that $correspondingData(w_1) = \{u_2, u_3\}$; thus, its influence area is the partial knowledge model shown in Fig. 17a. Regarding slot w_7, it is easy to realize that

$$correspondingData(w_7) = \{u_5, u_6, u_9\};$$

thus, its influence area is the partial knowledge model shown in Fig. 17b. Notice that the influence area for a slot is complete even if the slot does not have enough information to instantiate all of its data vertices, which is the case of slot w_7.

The first and the second invariant in the configuration ADT state that the location map must provide an index and a label for every data vertex in the knowledge model, i.e., none of the data vertices is useless and the location map is complete. The third invariant states that the record must have a subset of the labels used by the location map, i.e., the location map takes every possible label into account, but a record is not required to have information about every possible label since missing attributes are common in practice and it does not make sense to create a slot that does not hold any attributes. The fourth invariant states that the indices assigned by the location map to each data vertex must be different and consecutive, i.e., no attribute in a slot is useless. The fifth invariant states that the number of attributes assigned to each slot must coincide with the number of data vertices the location map assigns to its label, i.e., the slots provide enough information even if some of

Location
[Structure]
label : *Vertex* ⇸ *Label*
index : *Vertex* ⇸ ℕ

Fig. 14. A scheme to model location maps.

Data vertex	Label	Position
u_2	L_1	1
u_3	L_1	2
u_5	L_2	2
u_6	L_2	1
u_9	L_2	3
u_{10}	L_3	1

Fig. 15. A location map.

Configuration

[Structure]

$rec : Record$

$km : KnowledgeModel$

$loc : Location$

[Ancillary definitions]

$attribute : Vertex \times Vertex \nrightarrow Attribute$

$correspondingData : Vertex \nrightarrow \mathbb{F}\, Vertex$

$influenceArea : Vertex \nrightarrow KnowledgeModel$

[Invariants]

[1] $\mathrm{dom}\, loc.index = km.data$

[2] $\mathrm{dom}\, loc.label = km.data$

[3] $\mathrm{ran}\, rec.label \subseteq \mathrm{ran}\, loc.label$

[4] $\forall\, lb : Label \mid lb \in \mathrm{ran}\, loc.label \bullet loc.index(\!|\, loc.label^{\sim}(lb) \,|\!) = 1 \,..\, \#loc.label^{\sim}(lb)$

[5] $\forall\, w : Vertex \mid w \in rec.vertices \bullet \#rec.attributes(w) = \#correspondingData(w)$

[6] $\forall\, w_1, w_2 : Vertex \mid (w_1, w_2) \in rec.edges \bullet (influenceArea(w_1).root, influenceArea(w_2).root) \in km.edges$

[Ancillary definitions]

[1] $\forall\, u, w : Vertex \mid u \in km.data \land w \in rec.vertices \bullet attribute(u, w) = rec.attributes(w)(loc.index(u))$

[2] $\forall\, w : Vertex \mid w \in rec.vertices \bullet correspondingData(w) = loc.label^{\sim}(rec.label(w))$

[3] $\forall\, w : Vertex; \mid w \in rec.vertices \bullet influenceArea(w) =$
$(\mu\, pm_1 : KnowledgeModel \mid pm_1 \in partialModels(km) \bullet pm_1.data = correspondingData(w) \land$
$(\nexists pm_2 : KnowledgeModel \mid pm_2 \in partialModels(km) \bullet pm_2.data = pm_1.data \land pm_2.objects \subsetneq pm_1.objects))$

Fig. 16. A scheme to model configurations.

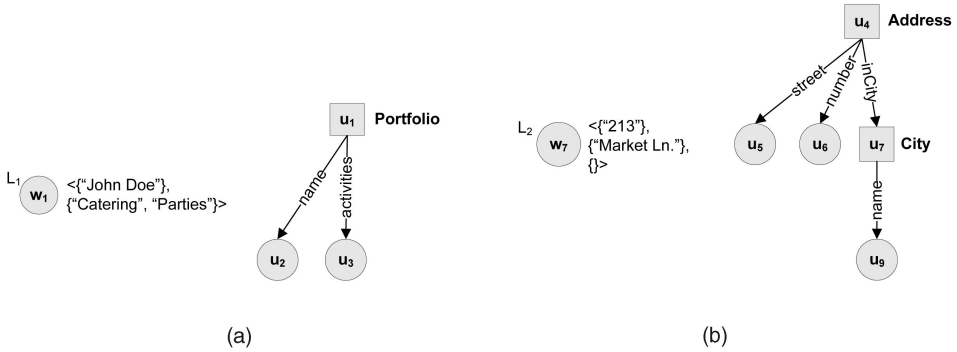


Fig. 17. Influence areas of several slots. (a) Influence area of w_1. (b) Influence area of w_7.

the attributes are missing in the original Web page. Finally, the sixth invariant states that for every two slots that are connected by means of an edge, their corresponding influence areas must be connected in the knowledge model; otherwise, we would not be able to connect the partial knowledge entities obtained from them.

The fourth invariant requires a little explanation: $loc.label^{\sim}$ denotes the inverse of $loc.label$, i.e., when $loc.label^{\sim}$ is applied to a label, it returns the set of data vertices that are mapped onto that label; for instance, if we assume that loc denotes the location map in Fig. 15, then $loc.label^{\sim}(L_1) = \{u_2, u_3\}$, $loc.label^{\sim}(L_2) = \{u_5, u_6, u_9\}$, and $loc.label^{\sim}(L_3) = \{u_{10}\}$. Therefore, when the index function is applied to these results, it must return a sequence of consecutive indices, for instance,

$$loc.index(|\{u_2, u_3\}|) = \{1, 2\},$$
$$loc.index(|\{u_5, u_6, u_9\}|) = \{1, 2, 3\}, \text{ and}$$
$$loc.index(|\{u_{10}\}|) = \{1\}.$$

If the index returned something different, then there would be useless attributes in the slot.

4.2 Instantiating an Influence Area (Step 1)

The first step of the algorithm consists of creating a knowledge entity for each slot in a record, which amounts to instantiating their corresponding influence areas so that each object vertex has a unique name and each data vertex has the right attribute. This step is defined in Fig. 18 as a new scheme that is composed of just a configuration and two ancillary functions.

Function $subareaToInstantiate$ determines what the exact subarea of influence of a given slot is; it returns the unique partial knowledge model in the influence area associated with the slot under consideration whose data vertices correspond to the data vertices for which it has an attribute. Recall that every knowledge model is included in the set of partial knowledge models obtained from it, which makes it easier to formalize this function since the case when there are not any missing attributes is not an exception. For instance, the subarea to instantiate that corresponds to slot w_1 coincides with its influence area since it provides information about every possible attribute for a slot with label L_1. In the case of w_7, however, it does not provide any information about the city in which "213 Market Ln." is located, which implies that the subarea to instantiate does not coincide with its influence area, see Fig. 19a.

$Step_1$

[Structure]

$cfg : Configuration$

[Ancillary definitions]

$subareaToInstantiate : Vertex \nrightarrow KnowledgeModel$

$instantiateInfluenceArea : Vertex \nrightarrow KnowledgeEntity$

---

[1] $\forall w : Vertex \mid w \in cfg.rec.vertices \bullet subareaToInstantiate(w) =$
$(\mu pm : KnowledgeModel \mid pm \in partialModels(influenceArea(w)) \wedge$
$pm.data = \{u : Vertex \mid u \in cfg.correspondingData(w) \wedge cfg.attribute(u, w) \neq \varnothing\})$

[2] $\forall w : Vertex;\ ke : KnowledgeEntity \mid w \in cfg.rec.vertices \bullet$
$instantiateInfluenceArea(w) = ke \Leftrightarrow ke.conformsTo(subareaToInstantiate(w)) \wedge$
$\forall v : Vertex \mid v \in ke.vertices \bullet$
$(v \in ke.objects \Rightarrow ke.name(v) = createName(v) \wedge$
$(v \in ke.data \Rightarrow ke.attribute(v) = cfg.attribute(ke.correspondsTo(v, subareaToInstantiate(w)), w)))$

Fig. 18. A scheme to model the first step of our algorithm.

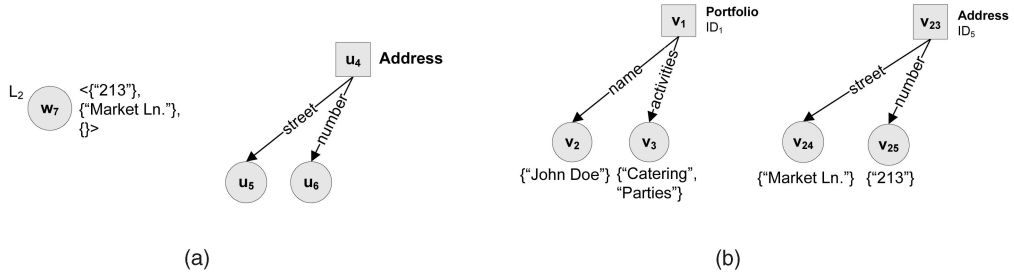


Fig. 19. Sample influence subareas and instantiation. (a) Subarea to instantiate with w_7. (b) Instantiation of w_1 and w_7.

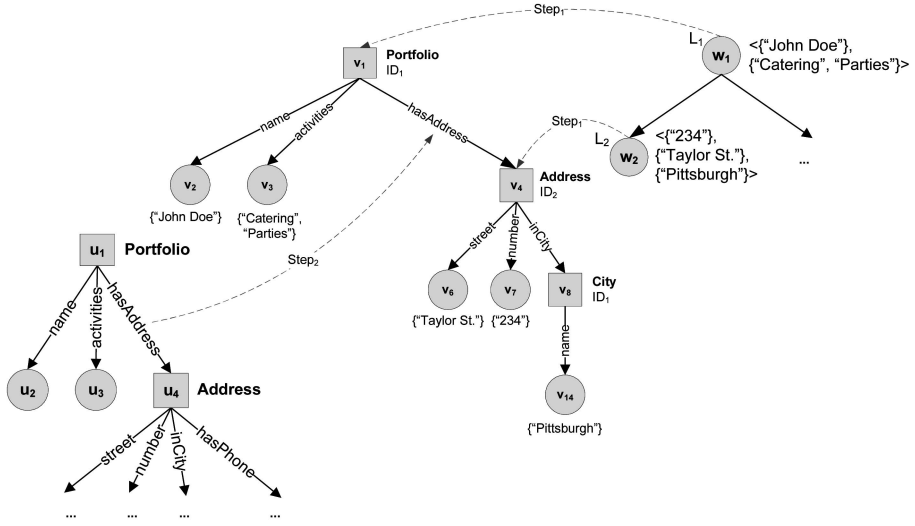


Fig. 20. $Step_1$ and $Step_2$ working together.

Function $instantiateInfluenceArea$ works on a slot and returns a partial knowledge entity that instantiates its corresponding influence (sub)area. The definition is simple since it just requires the resulting knowledge entity to conform to the influence (sub)area of the slot and to provide a name for each object vertex and an appropriate attribute for each data vertex, see Fig. 19b.

4.3 Attaching Knowledge Entities (Step 2)

Each slot produces a parent knowledge entity that correspond to the slot itself, and a set of child knowledge entities that correspond to its children, if any. They must be combined into a new knowledge entity with the same data vertices, edges, concepts, properties, names, and attributes as the knowledge entities to be attached, but additional edges to connect the root vertex of the parent knowledge entity with the root vertex of each child knowledge entity. The property that corresponds to these additional edges must be the same property that connects the roots of the corresponding influence areas. For instance, Fig. 20 shows part of the first record in Fig. 5b and the results of instantiating their corresponding influence areas and attaching them.

This step is defined as a new scheme in Fig. 21. It is composed of just a configuration and three ancillary functions. The unary predicate $attachable$ works on two knowledge entities and determines if they may be attached; this is possible as long as they both conform to two partial knowledge models obtained from the knowledge model

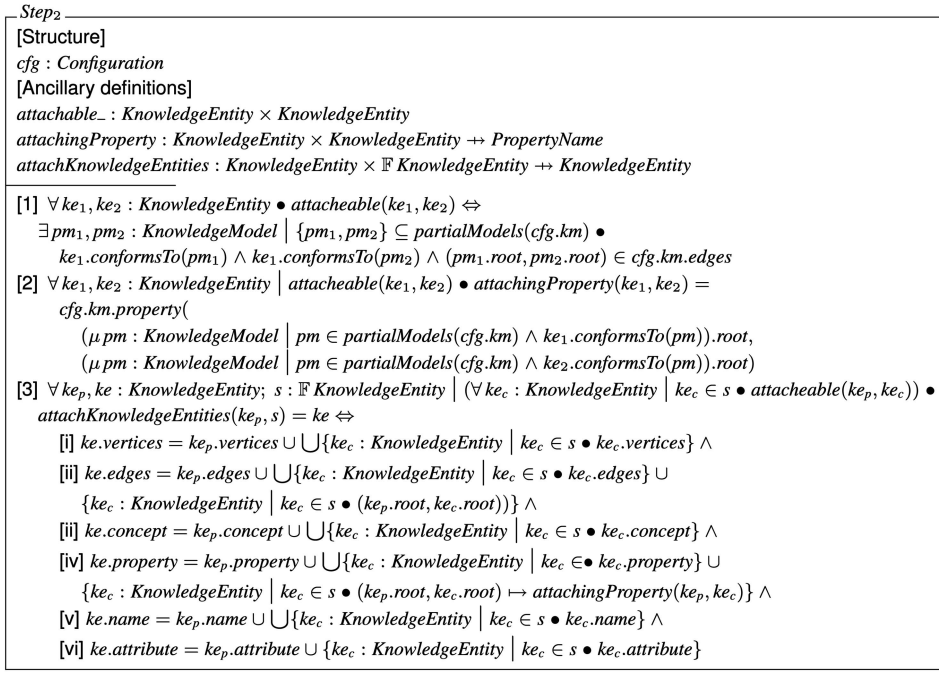


Fig. 21. A scheme to model the second step of our algorithm.

provided by the configuration on which the step is working, and their root vertices are related by means of a property. Building on this predicate, function $attachingProperty$ returns the property that is necessary to attach two given knowledge entities, which is the property that connects the roots of the unique partial knowledge models to which they conform.

$attachKnowledgeEntities$ works on a parent knowledge entity and a set of child knowledge entities and attaches them all. The result is built as follows:

i. its vertices are the vertices of the parent knowledge entity plus the vertices of the child knowledge entities,
ii. its edges are the edges of the parent knowledge entity, plus the edges of the child knowledge entities, plus additional edges to connect them all,
iii. its concepts are the concepts of the parent knowledge entity plus the concepts of the child knowledge entities,
iv. its properties are the properties of the parent knowledge entity plus the properties of the child knowledge entities and additional properties to connect them all,
v. its names are the names used in the parent knowledge entity plus the names used in the child knowledge entities, and
vi. the attributes are the attributes used in the parent knowledge entity plus the attributes used in the child knowledge entities.

4.4 The Driver

The driver is responsible for coordinating the previous steps. From a structural point of view, it is composed of two instances of $Step_1$ and $Step_2$, and a function called $translateSlot$ that transforms a slot into its corresponding knowledge entity. Its formal definition is presented in Fig. 22.

The first invariant requires the two steps the driver combines to share the same configuration. This may seem a little odd at first, but the reason is that we assume that the same EII agent may be working on several pages at the same time, so there might be several instances of $Step_1$ and $Step_2$ working simultaneously on different configurations; obviously, the driver requires the steps to combine to be working on the same configuration. The second invariant defines function $translateSlot$, which is applied to a slot in the record to be translated and traverses it in a deep-first manner: When the recurrence reaches a slot without any children, the function instantiates it by using the first step of the algorithm; when it reaches an intermediate slot, $translateSlot$ is first applied recursively to the children slots, and the results are then combined by using the second step. The ancillary function $translateChildren$ is responsible for translating the children of a given slot.

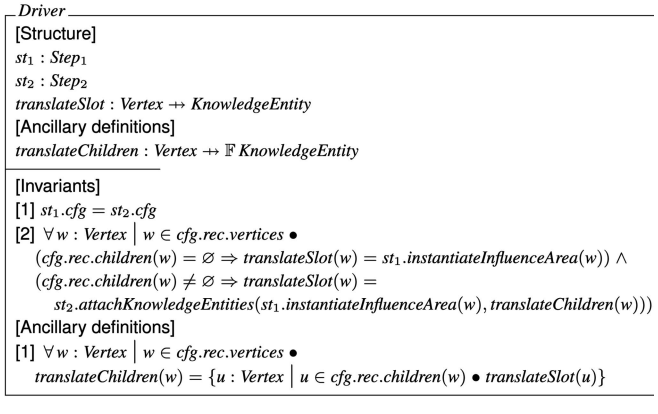


Fig. 22. A scheme to model the driver of our algorithm.

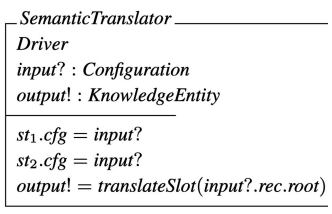


Fig. 23. A scheme to model our semantic translators.

4.5 An Entry Point

The scheme that defines the semantic translator module in Fig. 1 is presented in Fig. 23: It includes the driver, requires an input configuration, and produces an output knowledge entity by applying the $translateRecord$ function to the root slot of the record in the input configuration. Notice that this is just an entry point to our system, but others might be defined similarly, e.g., an entry point that translates a collection of records, but the details are irrelevant.

5 Analysis

Theorem 1. *Our algorithm terminates.*

Proof. The algorithm relies on function $translateSlot$, which is applied to the root slot of the record to be translated. $translateSlot$ instantiates a knowledge entity for the slot being studied, and calls itself recursively by means of a call to $translateChildren$. Since the number of slots in a record is finite, and the recursion ends every time $translateSlot$ is applied to a leaf slot, the algorithm terminates. □

Theorem 2. *The worst-case complexity of our algorithm is $O(m \cdot \log m)$, where m denotes the number of slots in the record to which it is applied.*

Proof. Notice that each label is associated with a unique influence area, and this mapping depends solely on the location information, which implies that this information may be precalculated. The property used to attach the knowledge entities obtained from any two slots can be precalculated, too. In the worst case, a record is a balanced k-ary tree in which every slot has the same number of attributes. Thus, there must exist constant upper bounds a and b to the time required to instantiate an influence area, and the time to attach a parent knowledge entity to its child knowledge entities, respectively.

As a conclusion, the worst-case temporal complexity for a k-ary record that consists of m slots is defined as follows:

$$T(m) = \begin{cases} k \cdot T(\lfloor m/k \rfloor) + a + b & \text{if} \quad m > 1 \\ a & \text{if} \quad m = 1. \end{cases}$$

To solve this equation, we calculate the time contribution of the slots at the same depth. In a k-ary record, there are k^i slots at depth i; thus, each slot contributes $\lfloor m/k^i \rfloor + a + b$ to the total time. This implies that the set of slots at the same level contribute $k^i \cdot (\lfloor m/k^i \rfloor + a + b)$ to the total time. Since a balanced k-ary tree with m vertices has depth $\log_k m$, there is an upper bound to $T(m)$, namely,

$$T(m) \leq \sum_{i=0}^{\log_k m} (m + (a+b) \cdot k^i) =$$

$$\sum_{i=0}^{\log_k m} m + (a+b) \cdot \sum_{i=0}^{\log_k m} k^i =$$

$$m \cdot \log_k m + (a+b) \cdot \left(\frac{k^{\log_k m + 1} - 1}{k - 1} \right).$$

Therefore, the worst-case complexity is $O(m \cdot \log_k m + k^{\log_k m}) = O(m \cdot \log_k m + m) = O(m \cdot \log m)$. □

Corollary 1. *In the worst case, our algorithm takes $O(n \cdot m \cdot \log m)$ time to translate a collection of n records with average number of slots m.*

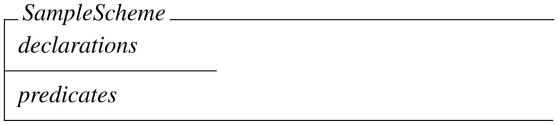
6 Conclusions

The proposal we describe in this paper integrates seamlessly into the CREAM framework, where previous enhancements to target structured information sources required the information provider to be very cooperative and the information to come from a back-end database directly; other enhancements were devised to work with unstructured, redundant information sources. The analysis of the related work proves that our proposal to transform a wrapper's output into knowledge entities is original in the context of structured, local information sources. We also prove that it is quite efficient since its time complexity is $O(n \cdot m \cdot \log m)$; this means that if the number of records to translate is significatively larger than the average number of slots per record, the algorithm then behaves linearly; otherwise, it behaves log-linearly regarding the size of the record to be translated.

Appendix

The Z Notation

Throughout the paper, we use the Z mathematical notation [35]. It extends the use of set theory and first-order predicate calculus languages by allowing a mathematical type known as the scheme type. Z schemes are composed of a declarative part, which declares variables and their types, and a predicate part, which relates and constrains those variables. The type of any scheme can be considered as the Cartesian product of its variables constrained by the predicates. They are typeset using the following graphic notation:



Modularity is facilitated by allowing a scheme to be included within other schemes or by using schemes as data types. For example, the scheme $IncludingScheme$ below includes all the declarations and predicates of scheme $SampleScheme$.

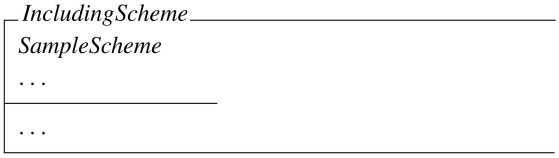


TABLE 1
Summary of the Z Notation Used in This Paper

Declarations	
$x : A$	Asserts that $x \in A$
$x? : A$	Asserts that $x \in A$. By convention, $x?$ is an input variable
$x! : A$	Asserts that $x \in A$. By convention, $x!$ is an output variable
$p\_ : A$	Declares p as a unary predicate over A
$A == B$	A is an alias for B
Quantification	
$\forall x : A \mid q(x) \bullet p(x)$	Equivalent to $\forall x \bullet x \in A \land q(x) \Rightarrow p(x)$
$\exists x : A \mid q(x) \bullet p(x)$	Equivalent to $\exists x \bullet x \in A \land q(x) \land p(x)$
$\exists_1 x : A \mid q(x) \bullet p(x)$	Unique existential quantification
$\mu x : A \mid p(x)$	Denotes the only $x \in A$ such that $p(x)$ holds
Sets	
$\mathbb{P} A$	Powerset of A
$\mathbb{F} A$	Finite powerset of A
$i \,..\, j$	The set $\{x \mid x \geq i \land x \leq j\}$
$\{x : A \mid q(x) \bullet e(x)\}$	The set $\{e(x) \mid x \in A \land q(x)\}$
$\bigcup A$	Generalised or distributive union
$\# A$	Cardinal of set A
Sequences	
$\mathrm{seq}\, A$	The set of sequences whose elements belong to A
$\langle e_1, e_2, \ldots, e_n \rangle$	A sequence composed of $e_1, e_2, \ldots, e_n$
$\mathit{front}\, s$, $\mathit{last}\, s$	First and last element of sequence s
$s(i)$	The i^{th} element of sequence s
$\# s$	Length of sequence s
$\langle e \rangle$ in s	Membership
Functions	
$A \rightarrow B$	Total function
$A \nrightarrow B$	Partial function
$\mathrm{dom} f$, $\mathrm{ran} f$	Domain and range of function f
$f^{\sim}$	The inverse of function f
$f(\!\mid A \mid\!)$	The set $\{f(x) \mid x \in A\}$

When using schemes as data types, we employ a selection operator "." in order to refer to a particular variable. For instance, in the following scheme, $schemeInstance$ is an object of scheme type $SampleScheme$; thus, variable var declared in scheme $SampleScheme$ is denoted by writing $schemeInstance.var$.

SchemeAsDataType

schemeInstance : *SampleScheme*

...

... *schemeInstance.var* ...

To introduce a type in which we wish to abstract away from its actual elements, we use the notion of a given set. For instance, we write $[Identifier]$ to represent the set of all identifiers, without delving into their structure or properties.

A summary of the mathematical notation used in this paper is given in Table 1, where p or q are used to denote predicates, and e to denote expressions; sometimes "$\mid q(x)$" is omitted and it is equivalent to "$\mid true$;" sometimes "$\bullet\, e(x)$" is omitted, and it amounts to "$\bullet\, x$."

ACKNOWLEDGMENTS

The work reported in this paper was supported by the Spanish Interministerial Commission on Science and Technology under grant TIC2003-02737-C02-01 (Project WebMade).

REFERENCES

[1] M. Hepp, "Products and Services Ontologies: A Methodology for Deriving OWL Ontologies from Industrial Categorization Standards," *Int'l J. Semantic Web Information Systems,* vol. 2, no. 1, pp. 2-99, 2006.

[2] O. Kaykova, O. Khriyenko, D. Kovtun, A. Naumenko, V. Terziyan, and A. Zharko, "General Adaption Framework: Enabling Interoperability for Industrial Web Resources," *Int'l J. Semantic Web Information Systems,* vol. 1, no. 3, pp. 31-63, 2005.

[3] N. Bassiliades, G. Antoniou, and I. Vlahavas, "A Defeasible Logic Reasoner for the Semantic Web," *Int'l J. Semantic Web Information Systems,* vol. 2, no. 1, pp. 1-41, 2006.

[4] F. Bry, C. Koch, T. Furche, S. Schaffert, L. Badea, and S. Berger, "Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages," *Int'l J. Semantic Web Information Systems,* vol. 1, no. 2, pp. 1-21, 2005.

[5] A. Naeve, "The Human Semantic Web: Shifting from Knowledge Push to Knowledge Pull," *Int'l J. Semantic Web Information Systems,* vol. 1, no. 3, pp. 1-30, 2005.

[6] A. Sheth, C. Ramakrishnan, and C. Thomas, "Semantics for the Semantic Web: The Implicit, the Formal and the Powerful," *Int'l J. Semantic Web Information Systems,* vol. 1, no. 1, pp. 1-18, 2005.

[7] S. Handschuh, R. Volz, and S. Staab, "Annotation for the Deep Web," *IEEE Intelligent Systems,* vol. 18, no. 5, pp. 42-48, Sept./Oct. 2003.

[8] S. Handschuh and S. Staab, "CREAM: Creating Metadata for the Semantic Web," *Computer Networks,* vol. 42, pp. 579-598, 2003.

[9] R. Volz, S. Handschuh, S. Staab, L. Stojanovic, and N. Stojanovic, "Unveiling the Hidden Bride: Deep Annotation for Mapping and Migrating Legacy Data to the Semantic Web," *J. Web Semantics,* vol. 1, no. 2, pp. 187-206, 2004.

[10] W. Cohen and L. Jensen, "A Structured Wrapper Induction System for Extracting Information from Semi-Structured Documents," *Proc. 17th Int'l Joint Conf. Artificial Intelligence (IJCAI '01),* 2001.

[11] V. Crescenzi and G. Mecca, "Automatic Information Extraction from Large Websites," *J. ACM,* vol. 51, no. 5, pp. 731-779, 2004.

[12] N. Kushmerick, "Wrapper Verification," *World Wide Web J.,* vol. 3, no. 2, pp. 79-94, 2000.

[13] M. Michalowski, J. Ambite, C. Knoblock, S. Minton, S. Thakkar, and R. Tuchinda, "Retrieving and Semantically Integrating Heterogeneous Data from the Web," *IEEE Intelligent Systems,* vol. 19, no. 3, pp. 72-79, May/June 2004.

[14] I. Muslea, S. Minton, and C. Knoblock, "Hierarchical Wrapper Induction for Semistructured Information Sources," *Autonomous Agents and Multi-Agent Systems,* vol. 4, nos. 1-2, pp. 93-114, 2001.

[15] A. Sahuguet and F. Azavant, "Building Intelligent Web Applications Using Lightweight Wrappers," *Data Knowledge Eng.,* vol. 36, no. 3, pp. 283-316, 2001.

[16] A. Halevy, N. Ashish, D. Bitton, M. Carey, D. Draper, J. Pollock, A. Rosenthal, and V. Sikka, "Enterprise Information Integration: Successes, Challenges and Controversies," *Proc. 2005 ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '05),* pp. 778-787, 2005.

[17] A. Doan, Y. Lu, Y. Lee, and J. Han, "Profile-Based Object Matching for Information Integration," *IEEE Intelligent Systems,* vol. 18, no. 5, pp. 54-59, Sept./Oct. 2003.

[18] S. Agarwal, S. Handschuh, and S. Staab, "Surfing the Service Web," *Proc. Second Int'l Semantic Web Conf. (ISWC '03),* pp. 211-226, 2003.

[19] O. Corcho and A. Gómez-Pérez, "A Layered Model for Building Ontology Translation Systems," *Int'l J. Semantic Web Information Systems,* vol. 1, no. 2, pp. 22-48, 2005.

[20] A. Gómez-Pérez and O. Corcho, "Ontology Specification Languages for the Semantic Web," *IEEE Intelligent Systems,* vol. 17, no. 1, pp. 54-60, Jan./Feb. 2002.

[21] F. Baader, I. Horrocks, and U. Sattler, "Description Logics as an Ontology Language for the Semantic Web," *Mechanizing Math. Reasoning: Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday,* D. Hutter and W. Stephan, eds., pp. 228-248, Springer-Verlag, 2005.

[22] C.-H. Chang, H. Siek, J.-J. Lu, C.-N. Hsu, and J.-J. Chiou, "Reconfigurable Web Wrapper Agents," *IEEE Intelligent Systems,* vol. 18, no. 5, pp. 34-40, Sept./Oct. 2003.

[23] D. Embley, D. Campbell, Y. Jiang, S. Liddle, Y.-K. Ng, D. Quass, and R. Smith, "Conceptual Model-Based Data Extraction from Multiple-Record Web Pages," *Data Knowledge Eng.,* vol. 31, no. 3, pp. 227-251, 1999.

[24] A. Harth, "SECO: Mediation Services for Semantic Web Data," *IEEE Intelligent Systems,* vol. 19, no. 3, pp. 66-71, May/June 2004.
[25] S. Handschuh, S. Staab, and F. Ciravegna, "S-CREAM: Semi-Automatic CREAtion of Metadata," *Proc. 13th Int'l Conf. Knowledge Eng. and Knowledge Management (EKAW '02),* pp. 358-372, 2002.
[26] F. Ciravegna, A. Dingli, Y. Wilks, and D. Petrelli, "Adaptive Information Extraction for Document Annotation in Amilcare," *Proc. 25th ACM SIGIR Int'l Conf. Research and Development in Information Retrieval (SIGIR '02),* pp. 367-368, 2002.
[27] P. Cimiano, S. Handschuh, and S. Staab, "Towards the Self-Annotating Web," *Proc. 13th Int'l Conf. World Wide Web (WWW '04),* pp. 462-471, 2004.
[28] P. Cimiano, G. Ladwig, and S. Staab, "Gimme the Context: Context-Driven Automatic Semantic Annotation with C-PANKOW," *Proc. 15th Int'l Conf. World Wide Web (WWW '05),* pp. 332-341, 2005.
[29] A. Kiryakov, B. Popov, I. Terziev, D. Manov, and D. Ognyanoff, "Semantic Annotation, Indexing, and Retrieval," *J. Web Semantics,* vol. 2, no. 1, pp. 49-79, 2004.
[30] F. Ciravegna, S. Chapman, A. Dingli, and Y. Wilks, "Learning to Harvest Information for the Semantic Web," *Proc. First European Semantic Web Symp. (ESWS '04),* pp. 312-326, 2004.
[31] M. Vargas-Vera, E. Motta, J. Domingue, M. Lanzoni, A. Stutt, and F. Ciravegna, "MnM: Ontology Driven Semi-Automatic and Automatic Support for Semantic Markup," *Proc. 13th Int'l Conf. Knowledge Eng. and Knowledge Management (EKAW '02),* pp. 379-391, 2002.
[32] S. Dill, N. Eiron, D. Gibson, D. Gruhl, R. Guha, A. Jhingran, T. Kanungo, K. McCurley, S. Rajagopalan, A. Tomkins, J. Tomlin, and J. Zien, "A Case for Automated Large-Scale Semantic Annotation," *J. Web Semantics,* vol. 1, no. 1, pp. 115-132, 2003.
[33] C.-N. Hsu and M.-T. Dung, "Generating Finite-State Transducers for Semi-Structured Data Extraction from the Web," *Information Systems,* vol. 23, no. 9, pp. 521-538, 1998.
[34] B. Adelberg, "NoDoSE: A Tool for Semi-Automatically Extracting Structured and Semistructured Data from Text Documents," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '98),* pp. 283-294, 1998.
[35] D. Lightfoot, *Formal Specification Using Z.* Palgrave Macmillan, 2001.