Chapter 9
Web System Development Using Polymorphic Widgets and Generic Schemas

Scott Britell, Lois M. L. Delcambre and Paolo Atzeni

Abstract Current tools allow non-technical users to create systems to store, display, and analyze their data on their own using whatever schema they choose. At the same time, developers of these systems can create generic widgets that may work across any number of domains. Unfortunately, to use a generic widget an end-user (the domain expert) must make their data conform to the schema of the widgets, possibly losing meaningful schema names. This paper presents a solution to this problem in the form of generic widget models (*canonical structures*), local schemas for domain experts, and an intermediate model (*domain structures*) that—through the use of mappings between the different models—allows generic functionality while preserving local schema. We present the three user roles in our system: widget developers, domain experts, and domain developers (people who develop and map domain structures). We introduce the concept of canonical structures and show how they are mapped to domain structures. We introduce a new relational query operator for writing queries against canonical structures and show how those queries are rewritten against the domain structures. We also provide an evaluation of the overhead of our system compared to custom code solutions and a modern web development framework.

Scott Britell
Computer Science Department, Portland State University, PO Box 751, Portland, OR 97207 USA
e-mail: `britell@cs.pdx.edu`

Lois M. L. Delcambre
Computer Science Department, Portland State University, PO Box 751, Portland, OR 97207 USA
e-mail: `lmd@pdx.edu`

Paolo Atzeni
Dipartimento di Ingegneria, Università Roma Tre, Via della Vasca Navale 79, 00146 Roma, Italy
e-mail: `atzeni@dia.uniroma3.it`

J. Cabot et al. (eds.), *Conceptual Modeling Perspectives*,
https://doi.org/10.1007/978-3-319-67271-7_9

9.1 Introduction

Early on, if an end-user created a website using a word-processing tool like Microsoft Word® that website would be little more than just documents on the Internet. To go beyond that, they would have needed to work hand-in-hand with a developer who had the expertise to take the client's conceptual model and realize it in an application.

Today, technologies such as web development frameworks have democratized the creation of complex systems by allowing non-technical (non-developer) users to define their own content types and create complex data models (i.e., conceptual models) while abstracting away the complexities of database and application creation. Thus, end-users who are experts in their own data, can choose schema names that are meaningful. We call end-user-created schemas *local schemas*.

Modern web frameworks also allow developers to create widgets that can be plugged into any site built upon that framework. These widgets use a conceptual model of the developer's choosing and are typically related to the functionality of the widget.

Traditionally, in order for a widget to work there are two choices. Developers may rewrite the same widget multiple times for the different conceptual models of the end systems. For example, in the case of a calendar widget the developer could modify the widget to work with each different event type. Or, the end systems must conform to the model of the widget; in the case of the calendar widget, each end-user would have to use the event type defined by the widget. This is the common case in use today by most web development frameworks.

Here, we present a different way to solve this problem. We begin by introducing intermediary conceptual models (that we call *domain structures*) between the end-user models (*local schemas*) and the widget models (*canonical structures*). We then define mappings (such as that used in traditional information integration and schema mapping) between the different levels (local schema$\leftrightarrow$domain structures and domain structures$\leftrightarrow$canonical structures). We allow end-users to create local schemas with meaningful names and allow widget developers to create generic widgets with canonical structures. And, we allow those generic widgets to show the local schema names using what we call *local radiance*.

Our system has three main roles. We call the end-user a *domain expert* since we consider someone creating an application for their data to be an expert in their data. The domain expert is responsible for deciding the local schema and data which will be used in the system. This person will enable instantiated widgets by creating mappings between the local schemas and the domain structures.

We call the developer responsible for creating generic widgets described above the *widget developer*. This person writes widget code that interacts with generic schemas, the canonical structures, that produce information that can be displayed on a webpage or used elsewhere in a web framework.

We add a third role to the two traditional roles: the *domain developer* whose responsibility is to create mappings between the generic schemas of the widgets and the schema of the domain expert. The domain developer usually has some (possibly

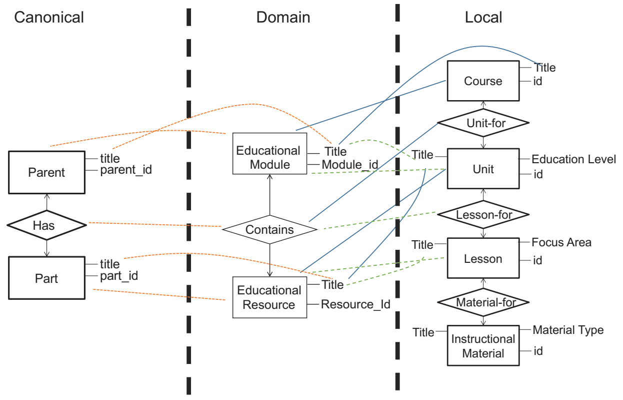


Fig. 9.1 An example use of a generic canonical structure (left), an educational domain structure (middle), a local educational schema (right), and mappings between them.

in-depth) knowledge of the domain but their main responsibility is more likely IT-based (database/web/application development) rather than domain analysis. Domain structures will typically be defined by domain developers. Domain structures are small schemas with names that are understandable to a domain expert. This person may work with the domain expert to create the website or may work with widget developers to allow the generic widgets to be used in specific application areas.

The rest of this paper is structured as follows. Section 9.2 describes the background of our previous work that contributed domain structures, local schemas, the mappings between them, and our query language. In Section 9.3 we explore the widget schemas that we call canonical structures, their mappings to the intermediary model (domain structures), and query rewriting. In Section 9.4 we evaluate the cost of using our system compared to a generic web framework and hard-coded widgets. We present related work in Section 9.5. Section 17.5 concludes the paper.

9.2 Background

In our earlier work we developed a system called information integration with local radiance (IILR) [8] which consists of three main parts: (1) domain structures (schema fragments with domain appropriate names), (2) mappings comprised of simple correspondences from local schemas to domain structures, and (3) a query algebra to allow queries against the domain structures to retrieve data from the local schemas—including the ability to retrieve local schema names. IILR corresponds to the middle and right parts of Figure 9.1.

Figure 9.1 shows the three levels of schema used in our system and mappings between them. In this example we have an hierarchical canonical structure with a domain structure and local schema from an educational domain. These three levels correspond to the three roles described above.

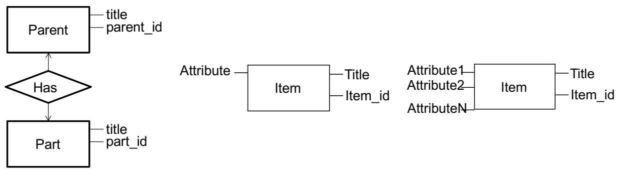


Fig. 9.2 Three examples of canonical structures.

On the left side of the figure, there is a canonical structure for a "Parent" and "Part" related by "Has" that will be used for a hierarchical navigation widget.

A domain developer then may create a domain structure (center of Figure 9.1) to work with the canonical structure of the widget. Note that the domain structure (or a subset of the domain structure) must be isomorphic to the canonical structure. The main difference between the domain structure and the canonical structure is the use of schema names that should be recognizable to a person working in the educational domain, for this example. The domain developer then creates a set of mappings between the canonical structure and the domain structure to instantiate the widget in a domain. In this case, the domain structure represents a hierarchical setting of an "Educational Module" that contains "Educational Resources". This domain structure is identical to the canonical structure albeit for the changing of names.

The domain expert has a local schema (shown on the right of Figure 9.1) and is able to use the instantiated widget in their website by creating mappings from the local schema to the domain structure. Here we see that the local schema is mapped multiple times to the domain structure allowing the widget to show "Units" inside a "Course" (the blue-solid lines between local and domain), and "Lessons" in a "Unit" (the green-dashed lines between local and domain).

A canonical structure is usually rather simple, essentially a "data pattern", on top of which widget code is implemented. A canonical structure often involves a single entity (like those shown in the middle and right in Figure 9.2), to be used by widgets that manage (search, analyze, update, ...) objects of a given data type (phone books, recent messages, calendars, ...).

Figure 9.3 shows a small sample of domain structures across a number of domains. On the left we see two domain structures for an educational domain. We use these structures throughout the rest of this paper. The "Educational Module" structure shown previously is on top and on bottom there is a structure for an educational resource.

In the middle of Figure 9.3 there are two domain structures from a financial domain. The top structure shows "Organizations" and their "Sub-organizations" which may be used for company schemas with departments, divisions, or labs. Below that there is a domain structure for a "Financial Instrument" which can be used for grants, budgets, or other financial entities. Being isomorphic to the educational structures, these structures will work with any widgets that the educational ones do (once mappings are in place between the canonical and domain structure).

On the right of Figure 9.3 there are two domain structures for the sports domain. The top structure represents a "Team" that has people in both coaching and par-

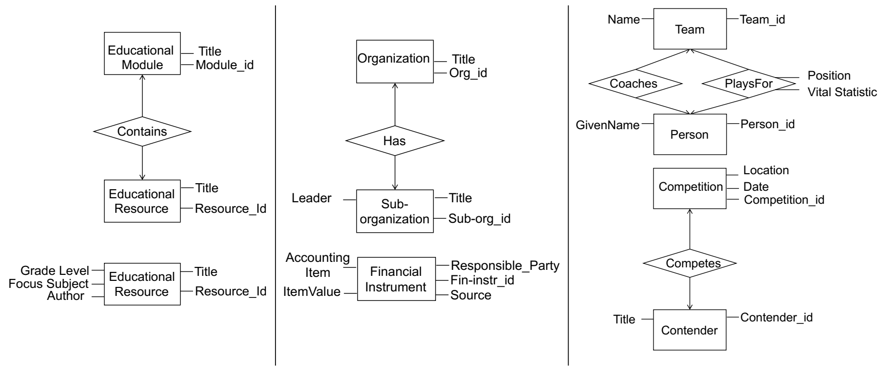


Fig. 9.3 Examples of domain structures from the educational (left), financial (middle), and sports (right) domains.

ticipant roles. The bottom structure represents “Competitions” and “Contenders” which can be used for local schemas ranging from football games to tennis and boxing matches.

Figure 9.4 then shows how the various domain structures from the financial and sports domains can be mapped to the canonical structures. The mappings on the right side of the figure are straightforward. The mapping in the upper left shows how a subset of the domain structure can be isomorphic to a canonical structure. In this case, both the coaches of the teams and the players are mapped to the canonical structure separately so that they can both show up in the hierarchical widget, but only an isomorphic part of the structure is mapped at a single time. The bottom left of the figure shows a more complex mapping where multiple attributes of the domain structure are mapped to a single attribute in the canonical structure. This will perform an operation similar to an unpivot[19] of the local schema (when local types are included in a query result, a feature supported by IILR).

Figure 9.5 shows an example of an educational local schema on the right and a domain structure on the left. There are two mappings between the two schemas. The blue-solid lines show the mapping between the Course/Unit-For/Unit structure in the local schema and the domain structure while the green-dashed lines show the mapping between the Unit/Lesson-For/Lesson structure and the domain structure. In our previous work [9], we performed a user-study that showed that domain experts with and without technical expertise could understand and create these mappings using simple and complex schemas.

We defined a query language at the domain level to enable information integration and querying of multiple local schemas with a single domain query. This enables both integration and data analysis and enables the widgets described later in this paper. Our query language extends the nested relational algebra (σ, π, $\bowtie$, ν, $\ldots$, plus γ for grouping [11]) with two operators: apply (α) and type (τ). Our *apply* operator ($\alpha(DS)$) is the basis of every query in our system. The *apply* operator uses correspondences that comprise the mappings between local schemas and a domain structure to perform information extraction/integration/transformation. The result of the *apply* operator is a set of relational tuples which can be passed to other relational

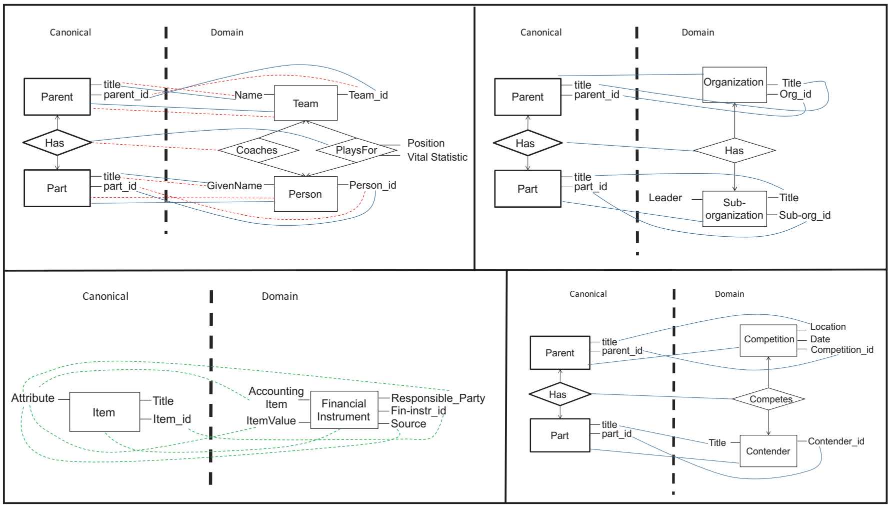


Fig. 9.4 Mappings of all domain and canonical structures in Figures 9.2 and 9.3

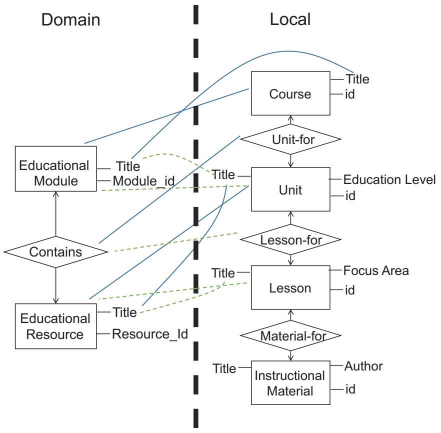


Fig. 9.5 A domain structure (left) and local schema (right) with two mappings between them (the blue-solid mapping and the green-dashed mapping). Each mapping consists of a number of correspondences (single lines).

algebra operators as part of more complex queries. For example, Figure 9.6 shows sample data in the form of the local schema from Figure 9.5. Then the left and middle parts of Figure 9.7 show the use of the apply operator against the "Educational Module" and "Contains" parts of the domain structure from Figure 9.5.

The local *type* operator ($\tau_n(\chi)$) takes a domain structure component (n) and a query (χ) and introduces an attribute into the query result containing the local structure name to which the domain structure component (entity, attribute, or relationship) was mapped. For example, the right part of Figure 9.7 shows the type operator being used after the apply on the "Instructional Resource" part of the domain structure from Figure 9.5. The type operator allows the local names to come to the

Course

Title	Id
Intro to CS	324

Unit-For

Course_id	Unit_id
324	834
324	982

Unit

Title	Id	Education Level
Python	834	12th Grade
Java	982	12th Grade

Lesson-For

Unit_id	Lesson_id
834	835
834	836
982	983

Lesson

Title	Id	Focus Area
Intro to Python	835	CS
Advanced Python	836	CS
Intro to Java	983	CS

Fig. 9.6 Sample local data using the local schema from Figure 9.5.

α(Educational Module)

Title	Module_id
Intro to CS	324
Python	834
Java	982

α(Contains)

Module_id	Resource_id
324	834
324	982
834	835
834	836
982	983

$\tau_{Instructional\_Resource}\,\alpha$(Instructional Resource)

Title	Resource_id	Intructional_ Resource.type
Python	834	Unit
Java	982	Unit
Intro to Python	835	Lesson
Advanced Python	836	Lesson
Intro to Java	983	Lesson

Fig. 9.7 Use of the apply and type operators on the domain structure from Figure 9.5 using the local data from Figure 9.6.

domain level in a generic fashion meaning that generic widgets can display local schema names; in essence the local names radiate to the domain level hence the name information integration with local radiance.

9.3 Canonical Structures

A canonical structure is a generically named schema fragment used by a widget developer. As an example, the left side of Figure 9.1 shows the canonical structure that is used to build the navigation widget described below.

Another basic canonical structure is a single entity with a small set of attributes such as that shown in Figures 9.2 and 9.8. This simple schema allows a variety of different generic widgets to be built. We say that the widget is polymorphic because it can be used with multiple domain structures (and multiple local schemas in turn).

9.3.1 Widgets

We describe three polymorphic widgets that use canonical structures to give the reader some idea of what widgets are and how they can represent different local schemas.

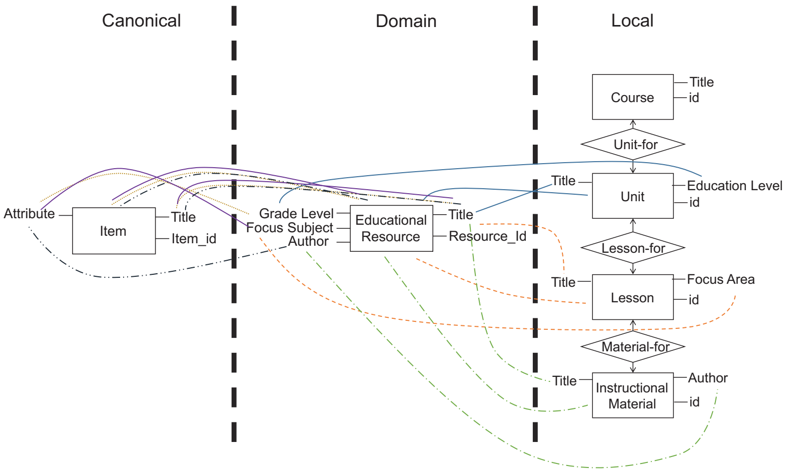
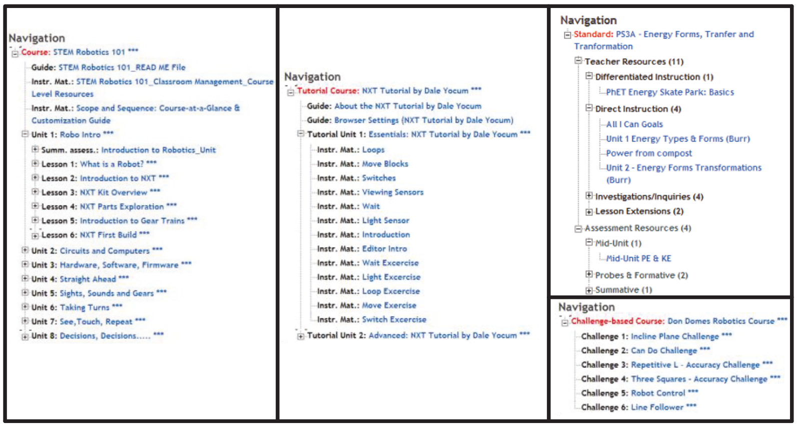


Fig. 9.8 A set of mappings is shown for the data analysis widget. The mappings between local and domain are straightforward. The mappings between the canonical and domain perform an unpivot operation.



Navigation
- Course: STEM Robotics 101 ***
  - Guide: STEM Robotics 101_READ ME File
  - Instr. Mat.: STEM Robotics 101_Classroom Management_Course Level Resources
  - Instr. Mat.: Scope and Sequence: Course-at-a-Glance & Customization Guide
  - Unit 1: Robo Intro ***
    - Summ. assess.: Introduction to Robotics_Unit
    - Lesson 1: What is a Robot? ***
    - Lesson 2: Introduction to NXT ***
    - Lesson 3: NXT Kit Overview ***
    - Lesson 4: NXT Parts Exploration ***
    - Lesson 5: Introduction to Gear Trains ***
    - Lesson 6: NXT First Build ***
  - Unit 2: Circuits and Computers ***
  - Unit 3: Hardware, Software, Firmware ***
  - Unit 4: Straight Ahead ***
  - Unit 5: Sights, Sounds and Gears ***
  - Unit 6: Taking Turns ***
  - Unit 7: See,Touch, Repeat ***
  - Unit 8: Decisions, Decisions..... ***

Navigation
- Tutorial Course: NXT Tutorial by Dale Yocum ***
  - Guide: About the NXT Tutorial by Dale Yocum
  - Guide: Browser Settings (NXT Tutorial by Dale Yocum)
  - Tutorial Unit 1: Essentials: NXT Tutorial by Dale Yocum ***
    - Instr. Mat.: Loops
    - Instr. Mat.: Move Blocks
    - Instr. Mat.: Switches
    - Instr. Mat.: Viewing Sensors
    - Instr. Mat.: Wait
    - Instr. Mat.: Light Sensor
    - Instr. Mat.: Introduction
    - Instr. Mat.: Editor Intro
    - Instr. Mat.: Wait Excercise
    - Instr. Mat.: Light Excercise
    - Instr. Mat.: Loop Excercise
    - Instr. Mat.: Move Exercise
    - Instr. Mat.: Switch Excercise
  - Tutorial Unit 2: Advanced: NXT Tutorial by Dale Yocum ***

Navigation
- Standard: PS3A - Energy Forms, Tranfer and Tranformation
  - Teacher Resources (11)
    - Differentiated Instruction (1)
      - PhET Energy Skate Park: Basics
    - Direct Instruction (4)
      - All I Can Goals
      - Unit 1 Energy Types & Forms (Burr)
      - Power from compost
      - Unit 2 - Energy Forms Transformations (Burr)
    - Investigations/Inquiries (4)
    - Lesson Extensions (2)
  - Assessment Resources (4)
    - Mid-Unit (1)
      - Mid-Unit PE & KE
    - Probes & Formative (2)
    - Summative (1)

Navigation
- Challenge-based Course: Don Domes Robotics Course ***
  - Challenge 1: Incline Plane Challenge ***
  - Challenge 2: Can Do Challenge ***
  - Challenge 3: Repetitive L - Accuracy Challenge ***
  - Challenge 4: Three Squares - Accuracy Challenge ***
  - Challenge 5: Robot Control ***
  - Challenge 6: Line Follower ***

Fig. 9.9 The navigation widget

9.3.1.1 The Navigation Widget

is designed to provided a tree-based navigation browser across a site. As can be seen in Figure 9.9, the widget works across various different schemas like the "Course" on the left and the "Educational Standard" in the upper right. The widget exploits the part-whole relationships in the system.

The widget is written against the "Parent-Part" canonical structure by the widget developer. A domain developer creates a domain structure for "Educational Module-Educational Resource". A domain expert then creates mappings between their local schema and the domain structure.

9.3.1.2 The Data Analysis Widget

shows aggregated information about attributes in a system. For example, the left of Figure 9.10 shows the different focus area of resources within the "Robo Intro" unit while the right side shows aggregated data for the authors of resources within a course.

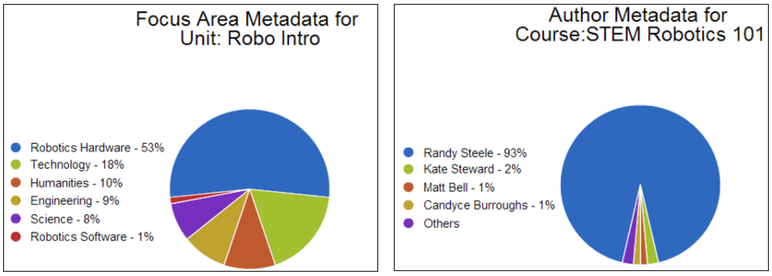


Fig. 9.10 The data analysis widget.

This widget performs an unpivot operation where multiple attributes at the domain level are mapped to a single canonical attribute; this is seen in Figure 9.8 where the different attributes of the domain structure ("Grade Level","Focus Subject", and "Author") all all mapped to the single "Attribute" in the canonical structure. This allows the widget to be written generically for all possible attributes that may appear in the local schemas. The widget builds off the Parent-Part structure used in the navigation widget described above and adds the "Item" canonical structure shown in Figures 9.2 and 9.8 and uses the type operator to bring the local type names into the widget.

9.3.1.3 The Faceted Navigation Widget

uses the canonical structures of the navigation and data analysis widgets together to create an hierarchical tree structure that is able to be restructured by the attribute data used in the analysis widget. Figures 9.11 and 9.12 show the functionality of this widget. The widget starts with a hierarchical view of a collection, in this case, it is for a digital library of computing resources but it could also be a course like those displayed above using the hierarchical widget. The widget allows a user to facet the collection by any of the local schema attributes that have been mapped. Figure 9.12 shows the collection faceted by class week. Each of the subtrees below the values of the class week facet may then be further faceted.

The faceted navigation widget uses the same canonical structures as the data analysis widget but performs a different task. Canonical structures and their mappings to domain structures may be reused multiple times.

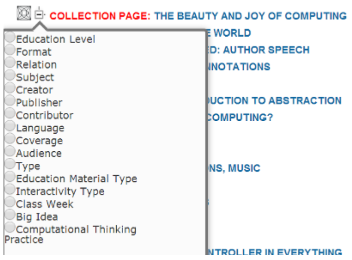


Fig. 9.11 The faceted navigation widget. Attributes that the tree can be faceted by are shown after clicking the diamond symbol next to the tree.

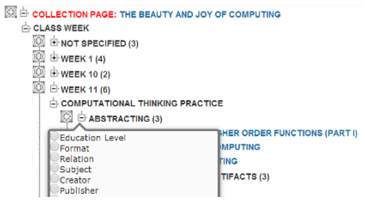


Fig. 9.12 The tree has now been faceted by “Class Week” and “Week 11” has been faceted by “Computational Thinking Practice”.

9.3.2 Mappings

In order to instantiate the navigation widget, the domain developer creates a widget specification which includes the mapping (such as those shown between the left and center parts of Figure 9.1) between the canonical structure with which the widget is associated and the domain structure.

A domain expert can then enable a widget for use in their website by creating mappings between a local schema and a domain structure such as those shown between the center and right of Figure 9.1. Here we see one mapping between the “Course-Unit” part of the local schema to the domain structure and a second mapping between the “Unit-Lesson” part of the local schema. Similar mappings are created for the various different local schemas that may exist in the educational domain. For the sake of brevity we do not show those mappings but intuitively it follows that each relationship and its entities in the local schemas can be mapped to the domain structure to enable the different widgets shown in Figure 9.9. As mentioned above, we impose one constraint on the mappings between canonical and domain structures which is that the mapped portion of the domain structure must be isomorphic to the canonical structure.

Since our implemented systems use relational databases, we have built our information integration with local radiance system on top of that and use the nested relational model and algebra to store our mappings and perform our queries. We use a straightforward translation between the Entity-Relationship model shown in the figures in this paper and relational tables in our implemented system. Mappings between canonical structures and domain structures are stored in the nested relation

$$CSDSmap(ID, CR, DR, CScorr(ID, CA, DA))$$

where each mapping has an id, the canonical relation and domain relation in the mapping, and a nested relation of the correspondences between the canonical attributes and the domain attributes. An example tuple for the mapping between the “Has” canonical relationship and the “Contains” domain relationship shown in

Figure 9.1 would be

$$
\begin{aligned}
(1,'Has','Contains',((&1.1,'Parent\_title','EducationalModule\_Title'),\\
&(1.2,'Parent\_parent\_id','EducationalModule\_Module\_id'),\\
&\ldots))
\end{aligned}
$$

9.3.3 Query Rewriting

In order for our widgets to work we must perform query rewriting from queries addressing the canonical structure to queries addressing the local schemas at the time of execution. As described above, in our previous work we defined the apply operator to translate queries against domain structures into queries against local schemas. We use our mappings and introduce a new operator perform the next step in rewriting a query against a canonical structure into a domain structure-level query. The rewrite operator (θ) is defined as follows, given a canonical relation *cr*,

$$
\theta(cr) = \bigcup_{\substack{\forall id \in \pi_{CSDSMap.ID}(\\ \sigma_{CSDSmap.CR=cr} CSDSmap))}} \rho_{\substack{CSDSmap.CScorr.DA \rightarrow \\ CSDSmap.CScorr.CA, \\ \tau(CSDSmap.CScorr.DA) \rightarrow \\ CSDSmap.CScorr.CA\_type}} \alpha(CSDSmap.DR)
$$

The rewrite operator works by using all the mappings between the given canonical relation and all mapped domain relations. For each mapping it performs the apply operation on the domain relation and then renames the domain attribute names to the canonical attribute names such that they will work in the widget using the canonical names. It also bring the type information from the apply operator so that generic widgets can show local type information as desired.

9.4 Evaluation

In our previous work [9] we have shown that people with and without technical expertise can perform the mappings between domain structures and local schemas required in our system. Here, we evaluate the overhead imposed by our system from our extra layers of modeling and mappings.

We compare our system against a hard-coded custom widget which performs queries directly against its own schema and stores all data in a single table requiring no joins in the resultant query. For the results in Table 9.4, this system is referred to as HC (hard-coded). Since the hard-coded system does not perform any of the overhead associated with our system we consider this to be a good target for fast performance that we would hope to achieve in our best-case. Our best-case scenario (USb) only has simple mappings that require no extra joins to perform.

We also compare ourselves to the default Drupal rendering system (labelled D in Table 9.4). As mentioned above, Drupal stores each attribute of an entity in a separate database table, so in order to render a page it must create a join query joining all the tables of all of the attributes. This is similar to our worst-case (USw) performance because if a user has composed complex mappings that involve the unpivot operation, our system must perform a similar join query. Note also that like Drupal (and most other web systems) these costs are usually one-time costs, since the output of these queries can be cached.

Table 9.4 shows the results of the performance test. Our system is shown in both the best-case (USb) and worst-case (USw) scenarios. All systems were tested with 2, 10, and 20 attributes and on a database with 100, 1000, and 10000 entries. Times are shown in milliseconds and are the average of 10 runs each. All tests were performed on a server with an Intel I7 processor and 8GB of RAM.

Table 9.1 Performance comparison of our system in a best-case scenario (USb) and worst-case scenario (USw) to a hard-coded (HC) single query widget (an optimal but most labor intensive solution) and to the Drupal (D) page rendering system (a generic widget that can render arbitrarily complex types). All three systems tested with 2, 10, and 20 attributes. All times in milliseconds.

Rows	HC2	HC10	HC20	D2	D10	D20	USb2	USb10	USb20	USw2	USw10	USw20
100	6.2	7.2	8	6.6	29.6	47	6.5	9.9	12.6	7.3	33.5	52.6
1000	8.8	16.9	19.9	7.5	40.3	72.9	9.4	27.4	39.5	9.9	53.3	93.7
10000	31.5	79.1	129.6	40	145.7	326.5	46.9	174.5	322.9	67.9	245.3	524.8

From Table 9.4 we see that, in our best-case scenario, we are competitive to a hard-coded solution for a smaller number of rows which is a great result for our naive implementation directly written against the IILR formalism. This naive implementation introduces constants for mapping and type information for every attribute in every row which, unsurprisingly, leads to the slower performance at larger row and attribute sizes. Even with this overhead we are comparable to Drupal in our worst-case scenario and the same or better in our best-case, even at larger row sizes. Our performance can be improved by storing the constant data in the database and optimizing queries using standard relational algebra equivalences. Note that our system is performing local radiance which cannot be done by either the hard-coded or Drupal system.

9.5 Related Work

Generic schemas and functionality have been explored extensively in programming and data management and bring with them many benefits. Generic schemas aid in development by allowing functions, code, and constraints to be defined generically. It also allows reuse and aids in the definition and creation of new (more complex) schemas and systems and allow for a greater reuse of schema [17]. Using generic schemas can provide faster development even with complex models while minimizing development complexity [17]. Generic types in programming language like Java [4] or C# [1] can provide common functionality to many different heterogenous types. We take this approach and add the ease of use of schema mapping systems like CLIO [14] to enable non-technical users to make use of generic functionality.

Web development frameworks [2] also often provide a generic relational mapping to convert complex user defined schemas into generic formats in their database backends. Often an instance of a content type created by a user in the web front-end is stored in the database with a table for each field of the object plus an instantiation of some base class. This is in contrast to Object-Relational Mappers (ORMs) [12] that provide an algorithmic mapping between objects and relational tables that contain attributes for each of the fields in an object. Web development frameworks can provide some basic generic functionality for building pages and websites, but more complex widgets are limited to predefined models.

Work has been done to create reusable semantic web widgets [13, 16]. While these widgets are reusable in a number of sites and can leverage the genericity of self-describing models like big data document stores and triple stores [10] and web models like XML [3] or RDF [6]; they are still limited to predefined models stored in the model or application.

A hybrid approach is often used in electronic medical records (EMR) [15] where there is a predefined schema for many of the entities in the system such as doctors, patients, or vital signs and generic (triple-store-like) tables that allow an EMR to be customized; and, a similar approach in SAP [5] which has transparent, pooled, and clustered tables. While this allows the data storage to be predefined while allowing heterogeneity of end-systems, the conceptual model is usually built into the application logic of the systems.

Our canonical structures are similar to data model patterns [7]. These patterns often are used for common reoccurring schema elements. Our canonical structures are also very similar to generic relationship types in information systems [17, 18]. Generic relationship types like the part-whole relationship or is-a relationship are often instantiated repeatedly in an information system, for example, a book entity has chapters which have sections which have paragraphs. If we know that the relationships between books, chapters, sections, and paragraphs are all instantiations of the part-whole relationship, we can then pre-define constraints and functionality on the part-whole relationship that will apply to all of its instantiations. If IILR was used in a system with such known relationship types we could automatically generate mapping from relevant canonical structures to the local schema.

9.6 Conclusions

We have implemented our system on top of the Drupal framework. As part of our future work, we hope to expand this to other frameworks and potentially create a framework of our own based on these principles.

We have shown how using canonical structures it is possible to write generic widgets that can be used in any number of systems while still maintaining local schema. We believe that the added overhead in terms of runtime costs and personnel is both minimal and justified. Our evaluation shows that in the worst-case scenario we still perform competitively. The notion of having three roles in our system is easily analogous to the different roles in a web framework where there are framework developers (writing completely generic code), community module developers (often writing domain specific widgets), and end-users instantiating frameworks in whatever domain they wish. We believe that this is an important step in allowing end-users to maintain more control over how their data is stored and presented.

We also hope to explore how we could use this paradigm to enable non-technical users to accomplish even more technical tasks, e.g., programming or complex query writing. We believe that by empowering end-users we may encourage them to increase their technical knowledge and possibly help solve the problem of a shortage of developers.

Acknowledgements This work was supported in part by National Science Foundation grants 0840668 and 1250340. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. C# | Microsoft Docs, `https://docs.microsoft.com/en-us/dotnet/csharp/csharp`
2. Drupal, `http://drupal.org`
3. Extensible Markup Language (XML), `https://www.w3.org/XML/`
4. Java 8, `https://java.com/en/download/faq/java8.xml`
5. Pooled and Cluster Tables
6. RDF - Semantic Web Standards, `https://www.w3.org/RDF/`
7. Blaha, M.: Patterns of Data Modeling. CRC Press, Boca Raton,FL (jun 2010)
8. Britell, S., Delcambre, L.M.L., Atzeni, P.: Flexible Information Integration with Local Dominance. Information Modelling and Knowledge Bases XXVI, 21–40 (2014)
9. Britell, S., Delcambre, L.M.L., Atzeni, P.: Facilitating Data-Metadata Transformation by Domain Specialists in a Web-Based Information System Using Simple Correspondences, pp. 445–459. Springer International Publishing (2016)
10. Cattell, R., Rick: Scalable SQL and NoSQL data stores. ACM SIGMOD Record 39(4), 12 (may 2011)
11. Gupta, A., Harinarayan, V., Quass, D.: Generalized projections: A powerful approach to aggregation. In: Proc. 21st VLDB Conf. pp. 11–15 (1995)
12. Keller, A.M., Jensen, R., Agarwal, S.: Persistence software: bridging object-oriented programming and relational databases. ACM SIGMOD Record 22(2), 523–528 (jun 1993)

13. Mäkelä, E., Viljanen, K., Alm, O., Tuominen, J., Valkeapää, O., Kauppinen, T., Kurki, J., Sinkkilä, R., Kansala, T., Lindroos, R., Others: Enabling the Semantic Web with Ready-to-Use Web Widgets. In: FIRST. pp. 56–69 (2007)
14. Miller, R.J., Hernández, M.A., Haas, L.M., Yan, L., Howard Ho, C.T., Fagin, R., Popa, L.: The Clio project. ACM SIGMOD Record 30(1), 78–83 (mar 2001), `http://dl.acm.org/citation.cfm?id=373626.373713`
15. Nadkarni, P.M., Brandt, C., CS, J., A, S., M, D., WE, H.: Data Extraction and Ad Hoc Query of an Entity–Attribute–Value Database. Journal of the American Medical Informatics Association 5(6), 511–527 (nov 1998)
16. Nowack, B.: Paggr: Linked Data widgets and dashboards. Web Semantics: Science, Services and Agents on the World Wide Web 7(4), 272–277 (dec 2009)
17. OliveÌĄ, A.: Conceptual modeling of information systems. Springer (2007)
18. Olivé, A.: Representation of Generic Relationship Types in Conceptual Modeling, pp. 675–691. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
19. Wyss, C.M., Robertson, E.L.: A formal characterization of PIVOT/UNPIVOT. In: Proceedings of the 14th ACM international conference on Information and knowledge management - CIKM ’05. p. 602. ACM Press, New York, New York, USA (oct 2005)