Hypertextual Programming for Domain-Specific End-User Development

Sebastian Ortiz-Chamorro[1,4], Gustavo Rossi[1,2], and Daniel Schwabe[3]

[1] LIFIA, Universidad Nacional de La Plata, Argentina
[2] CONICET, Argentina
[3] Departamento de Informática, PUC-Rio, Brazil
[4] Departamento de Electrónica e Informática, Universidad Católica de Asunción, Paraguay
`{sortiz,gustavo}@lifia.info.unlp.edu.ar, dschwabe@inf.puc-rio.br`

Abstract. Domain-specific languages (DSLs) have successfully been used for end-user development. However, dealing with language syntax poses significant learning challenges. In this paper, we introduce hypertextual programming, a technique that represents language syntax as hypertext. With this technique, instead of dealing with textual languages, users can inspect and construct their programs mainly by using navigation. Beyond merely representing the syntax, hypertext can be used to provide various views of a single program code. Nevertheless, to reap the benefits of this technique, adequate hypertextual editors must be built. This paper argues that many of the lessons learned in the web engineering area can be used to deal with this problem. Millions of users navigate the World Wide Web. Hypertextual programming leverages this widely available end-user skill to facilitate the construction of computer programs.

Keywords: hypertextual programming, end-user development, interfaces for end-user development, domain-specific languages, web engineering.

1 Introduction

Domain-specific languages (DSLs, a.k.a. scripting languages) have successfully been used for end-user development [1,2,3]. These languages help domain experts construct, inspect and test computer programs that operate within defined realms. Part of their success may be attributed to the fact that they present a set of familiar concepts to the end-user. However, DSLs force the user to "learn the arcane syntax and vocabulary conventions of the language" [2]. This initial step constitutes a difficult and undesirable challenge for the end-user.

Even in the case of DSLs, language syntax may be very complex. Consider the case of writing business rules in Jess [4], a popular rule-engine. The following is an example of a valid sequence of Jess commands:

```
(defglobal ?*threshold* = 20)
(bind ?age = 15)
(if (> ?age ?*threshold*) then
          (printout t "adult" crlf)
       else
          (printout t "minor" crlf))
```

The syntax of these commands is correct, but one missing or extra parenthesis would render the whole program syntactically invalid. We also have to take into account that dealing with language syntax in order to write a program goes way beyond avoiding syntactic errors. Executing the above set of commands in Jess 7.0 would actually generate a runtime error: *Not a number: "="*

This is because of a subtlety: even though syntactically correct, the second line actually assigns the string "=", not the number 15, to the *age* variable. If the user's intention was to assign the numeric value 15 to this variable, the following would be the correct Jess instruction:

```
(bind ?age 15)
```

Other language conventions may involve constantly memorizing and recalling (or at least searching through) an ever changing set of available elements to use. For example, in Jess, the set of available functions at a given program point contains all the predefined language functions and also any functions that the user has already defined. This is representative of many other languages where users are required to declare, define or import variable declarations, functions and other language elements before using them.

A different dimension of dealing with the language syntax in programming involves understanding the program code. Programming is an iterative process where, typically, the programmer has to read and understand existing code, identify the part or parts of the program that will be modified in a particular iteration and then perform the changes. Going back to the Jess example, a beginner will need significant effort to understand the complex language syntax. This adds a heavy burden to the authoring process.

Visual programming techniques have been developed to mitigate this problem by giving users graphic representations that may be more easily recognizable in some cases. These techniques have been used for end-user development [3]. However, visual programming has problems of its own. Among other things, some authors argue that visual programming may have scalability problems [5].

Graphic or not, the length and complexity of the end-user's programs, together with the limitations stated above and the need to focus on the specific parts that are undergoing modification call for a representation of the program code as a set of manageable pieces that the user can browse for inspection.

If a program is divided into units to be presented to the user, this user will need an intuitive and consistent way to select the specific parts to be viewed and modified.

This paper presents a technique based on the use of hypertext development environments that embody the syntax and conventions of the underlying language and use navigation as the main tool to inspect and modify programs. Hypertext systems [6] provide interactive environments where users can navigate through defined pieces of information (nodes). Beyond code browsing, in this paper we argue that by adequately using widely available tools like the World Wide Web, users can be provided with explicit controls that present them with a carefully chosen set of modification options for each specific node given the underlying language syntax. This has the potential to greatly reduce the programming learning effort.

This technique grew out of more than a decade of experience in the construction of web-applications that had various end-user development features. These ranged from small business process management rule definitions to the complete development environment presented in this paper as a concrete example of hypertextual programming: Benefit Catalog and Benefit Configurator. This dyad of applications constitutes a complete end-user development, versioning, testing, deployment and run-time environment for dynamic health-care insurance policy programming.

Expressing language syntax through navigation poses a significant engineering challenge. Hypertextual programming draws heavily on ideas from web engineering [7]. Several web design methodologies address the problem of expressing an underlying structure (usually a domain model) through a web-application [8,9,10,11,12]. In this case, the underlying structure is the syntax of a domain-specific language. The description of hypertextual programming that we present in this paper is a first attempt towards applying the lessons learned in web engineering to the problem of constructing hypertextual programming environments for a given DSL.

The structuring of this paper loosely follows the chronological development of hypertextual programming. In our experience, it is easier to understand this technique by starting with a concrete example and then exploring the general ideas and definitions behind it. Section 2 contains a description of Benefit Catalog and Benefit Configurator and includes some general requirements, architecture and hypertextual programming characteristics. In section 3, we present a more general description of hypertextual programming and some web engineering ideas that may help in the construction of hypertextual developing environments. Section 4 discusses related work. Finally, the conclusions of this paper and future research are presented in section 5.

2 Benefit Catalog and Benefit Configurator

Health-care insurance is a fertile ground for domain-specific end-user development. The process of administering health care insurance policies involves complex decision-making based on knowledge gained throughout decades of industry experience. Domain experts in this area may take years to learn the intricacies of just the subareas of the business that they work on. Merely the first step of the process from the client's point of view, which is helping to choose, customize and issue a health care policy, involves maintaining a sizable catalog of products that can be tailored to a specific client's needs. The health insurance products rendered by this process must comply with a considerable number of company guidelines and policies, and also any applicable laws.

Benefit Catalog and Benefit Configurator allow domain experts (*benefit engineers*) to: i)collaboratively develop a dynamic catalog of health-care insurance products (each dynamic product definition is called a *product template*); ii)maintain a library of parts to be used by different product templates; iii)test the product templates; iv)promote the approved versions of product templates for use in a production environment; v)run the product templates developed as an interactive sequence of questions to be asked to specialized company sales representatives; vi)based on the answers, generate and store health-care insurance policy specifications (called *answer sets*) as the output of these interactive questionnaires; and vii)provide support for the full product development life cycle, including the management of different versions of product templates and reusing previous answer sets for health care policy renewal.

An important requirement of this application is that the whole process described above has to be done without the intervention of professional programmers or company IT staff. This project required benefit engineers, that is, domain experts that do not have any professional programming background, to develop, test and manage product templates by themselves using a web environment. This requirement clearly prompted us to focus on the construction of tools that facilitate end-user development.

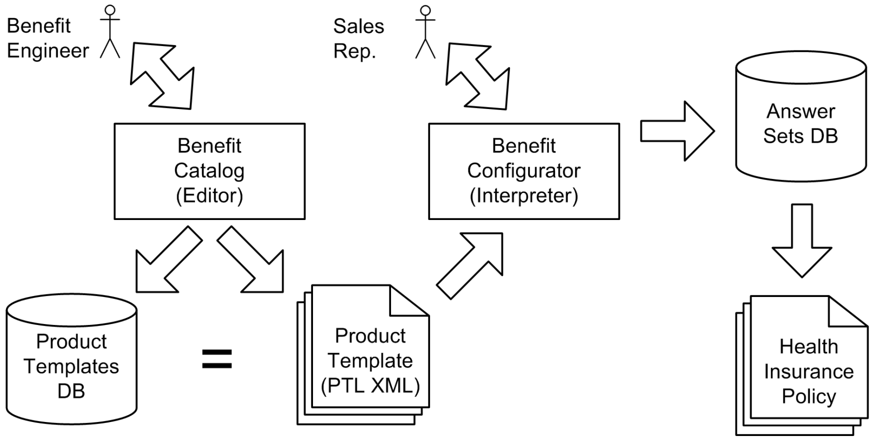


Fig. 1. Benefit Catalog and Benefit Configurator architectural diagram. Product templates are stored in the product template database, then, they are rendered as "executble" PTL XML.

An architectural diagram of the applications is shown in Fig. 1. Benefit Catalog is a fully web-based product template development environment. This tool saves product templates both in the product template database and also as programs written in the PTL[1] XML language. This allows users to query the product template database and obtain information about the various product templates that have been developed.

Additionally, benefit engineers can test and manage product templates' versions and the product part libraries that are used to build them.

Benefit Configurator is a PTL interpreter that runs these programs and generates an interactive series of questions to be answered by specialized sales representatives; then, based on the answers provided, it produces a health care insurance policy for the client (an answer set) and saves it to the answer set database.

The answer sets database is then transformed and imported into various downstream systems, including legal (text) contract generators and various claims systems among others.

[1] PTL stands for Product Template Language.

2.1 Product Template Language

We created an internal domain-specific language for product template specification. The Product Template Language (simply called *PTL*) is an XML [13] language that was built as an extension of the Cytera.Rules language [14]. Cytera.Rules is a Cytera Systems Inc. proprietary XML business rules language that allows the creation and evaluation of basic string, mathematical and boolean-based rules. PTL allows the representation and processing of rules involving objects and operations that are specific to the health-care insurance area.

In order to run PTL, almost all of the Cytera.Rules language interpreter had to be rewritten. To avoid this inconvenience in future projects, we developed a business rules language called AtOOmix with the aim of allowing the creation and implementation of XML domain-specific languages as extensions of existing AtOOmix languages without the need to fully rebuild the original languages and interpreters [15].

2.2 Benefit Catalog Hypertextual Programming Environment

It is important to point out that the benefit engineers never had direct contact with PTL. A fully web-based PTL editor was developed as the core of the Benefit Catalog application. This editor has several features aimed at facilitating the benefit engineers' tasks. Benefit Catalog represents the PTL code of an existing product template as a hierarchical collection of web pages that the user can navigate through.

Benefit Engineers never had to learn PTL, they only had to use the web-application that serves as user-interface. This is similar to the case of users that employ a web-application to populate a database: these users never have to deal directly with the database tables; they merely have to interact with the web-application.

Benefit engineers can also create and modify PTL code with Benefit Catalog. Fig. 2 shows how a new question is created. First, the user activates the *Attach Plan Choice Question* anchor in the *Grouping* node. This takes the user to the *Question* node. In this case, since it is a new question, users must fill-in the appropriate fields and then click on the *Attach* button. A benefit engineer can also create a new question by copying and then modifying an existing one. The *Copy* link is also shown in Fig. 2.

Right after a question has been created, and also throughout subsequent development sessions, benefit engineers can use navigation to go back to the question to inspect or modify it, e.g. change the grouping it belongs to, add a rule to turn-on the question or change the set of possible answers.

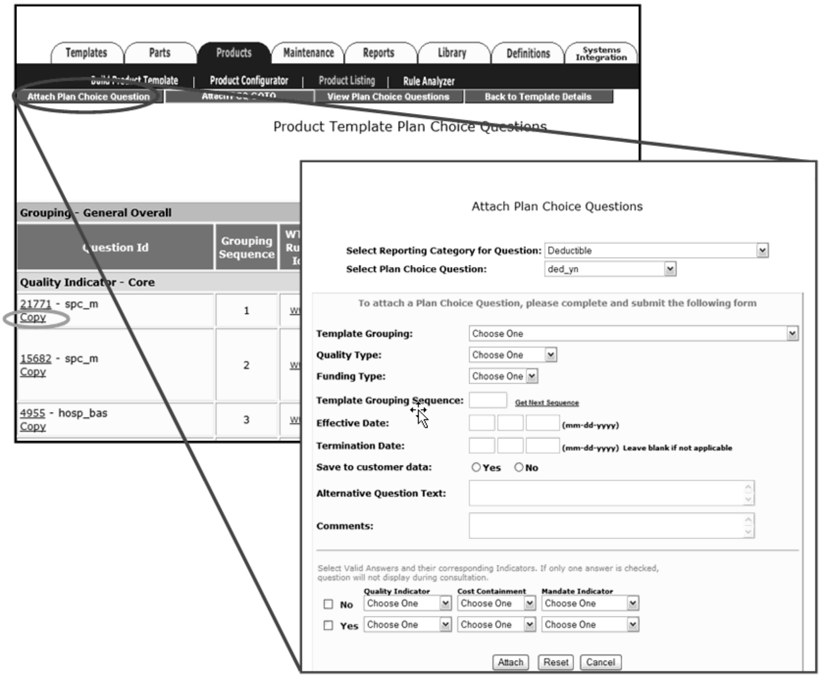
As a comparative example, the following code is a simplified PTL representation of a plan choice question:

```
<grouping name="Deductible">
  <pcq question_part="fam_ded">
    <when_turned_on_rule>
      <operation op="=">
          <var type="String">ded_yn</var>
          <const type="String">Yes</var>
          </operation>
        </when_turned_on_rule>
        <quality_type>core</quality_type>
        <funding_type>SI</funding_type>
        <seq>1</seq>
        <eff_dt>05/11/2005</eff_dt>
        <trm_dt>05/11/2007</trm_dt>
        <save>Y</save>
        <txt>Do you want a family deductible?</txt>
        <answer type="String" qi="Core" cc= "Y"
               mndt="Federal">Yes</answer>
        <answer type="String" qi="Core" cc= "Y"
               mndt="Federal">No</answer>
      </pcq>
   ...
```

Writing these rules manually requires an important effort. The syntax is complex and many language conventions have to be taken into account. For example, all questions have to reference previously defined question parts and answers. This is also true of quality types, funding types, quality indicators, cost containment and mandate indicators. In all these cases, with Benefit Catalog, values are assigned by simply choosing them from select lists. The application interface enforces the language conventions instead of leaving that burden to the user.

Reading questions directly from PTL would also a problem for the end-user, especially as the number of questions becomes large (a template with more than 200 rule-activated questions is not unusual). The background web page shown in Fig. 2 is an example of high-level code visualization. Questions belonging to a specific group are displayed on a single page. At this level, only the most critical question information is displayed to provide the user with a comprehensive view of the set of questions that form the group.

Using this web interface, benefit engineers can add other constructs used in product templates like cost sharing components and define rule-driven properties for them. Users can also define benefit options, benefit service levels for the benefit options and rules to populate them with the dynamic cost sharing components previously defined. For sake of space and simplicity, we do not provide the details of all these programming primitives in this paper. The number of these additional primitives is at least five times higher than the ones related to plan choice questions and involve more health-care insurance-specific concepts that are not as easy to explain as questions and answers. The main features of hypertextual programming on this system are adequately illustrated with plan choice questions.


Templates
Parts
Products
Maintenance
Reports
Library
Definitions
Systems Integration
Build Product Template | Product Configurator | Product Listing | Rule Analyzer
Attach Plan Choice Question
View Plan Choice Questions
Back to Template Details
Attach
Reset
Cancel

Product Template Plan Choice Questions

Grouping - General Overall

Question Id	Grouping Sequence
Quality Indicator - Core	
21771 - spc_m Copy	1
15682 - spc_m Copy	2
4955 - hosp_bas Copy	3

Attach Plan Choice Questions

Select Reporting Category for Question: Deductible
Select Plan Choice Question: ded_yn

To attach a Plan Choice Question, please complete and submit the following form

Template Grouping: Choose One
Quality Type: Choose One
Funding Type: Choose One
Template Grouping Sequence: Get Next Sequence
Effective Date: (mm-dd-yyyy)
Termination Date: (mm-dd-yyyy) Leave blank if not applicable
Save to customer data: Yes No
Alternative Question Text:
Comments:

Select Valid Answers and their corresponding Indicators. If only one answer is checked, question will not display during consultation.

	Quality Indicator	Cost Containment	Mandate Indicator
No	Choose One	Choose One	Choose One
Yes	Choose One	Choose One	Choose One

Fig. 2. Creating a new question in Benefit Catalog

To complete the program lifecycle, benefit engineers can run their product templates in a test environment, manage different versions of the same product template and activate it for it use in a production environment where it is used to interactively configure health-care plans. It is important to point out that all of this process is done by the benefit engineers themselves through Benefit Catalog and without the intervention of IT staff or professional developers.

3 Hypertextual Programming

Benefit Catalog and other applications that provide similar features cannot be adequately characterized neither as visual programming tools nor as text or structure based editors either. Rather, Benefit Catalog can be seen as an example of hypertextual programming.

We define *hypertextual programming* as a form of programming that uses navigation as the primary tool to inspect and edit the application code, and is supported by a computer system that: i) represents the entire program source code as hypertext; and ii) allows all the possible finite language instances to be generated as navigation paths through it.

In contrast to hypertextual programming, visual programming provides the user with a set of mainly graphic (as opposed to purely textual) elements that users can manipulate in order to develop a program. Benefit Catalog does not provide a graphic representation of programs (in this case product templates); it rather provides an interactive system where the users can explore and modify the program code by using navigation.

At the same time, this application is no traditional text editor either. Text editors usually present programs as collections of characters divided in files. Development is achieved mainly by adding and deleting characters in those files. Integrated development environments like Eclipse [16] provide some forms of navigation between different portions of the program code and features like auto-complete; however, we consider that they do not provide all the necessary features for hypertextual programming. First, navigation is not the primary means for source-code browsing and –most importantly– editing. Second, the source code structure at large is not represented as hypertext.

One basic definition of hypertext describes it as text structured in such a way that it has several possible reading paths. An example that satisfies this definition is the famous novel "Rayuela" by Argentinian writer Julio Cortazar. However, several authors insist on having automated navigation support for an artifact to be considered a hypertext system [6]. In the same fashion, we view hypertextual programming as an activity that is inseparable from a computer system that provides automated support for its key aspects. We call this computer system a *hypertextual editor*.

This definition requires program inspection and editing to be done primarily through navigation, but in our experience, the combination of this and other programming and interface construction techniques offer bigger potential. As an example, we found that mathematical and logical formulas may not always be well suited for hypertext representation. Breaking up such formulas in various nodes would lead to unnecessarily long navigation paths that contain very little information in each one of them. Consider the following formula:

```
1 + (2 * (Math.cos(a + b)))
```

If we represent it as eight nodes (1, +, 2, *, Math.cos, a, +, b) and provide the corresponding navigational links between them, very little information would be displayed in each node and the user would have to traverse a long navigation path just to read it.

This example is representative of other cases where better results might be obtained simply by using text to represent sub-parts of a language. In these cases, the text subparts can be used as node components.

In other cases, graphic elements may be more expressive to represent sub-parts of a language. Again, these graphic elements may also be used as node components.

When creating or altering language elements (e.g. adding a question or a group), users are creating or modifying node and link instances; they expect these changes to be reflected in the space that they are navigating (the specific instance of the navigational model at a given time). In other words, with hypertextual programming, the development of a computer program can be viewed as the construction of a navigation space, or more formally, as the iterative instantiation of a given navigational model.

Our definition requires the language syntax to be represented through hypertext. In the next section, we give a more detailed description of how a widely used language syntax definition can be represented in this way.

3.1 Expressing Language Syntax through Hypertext

Several web engineering methodologies separate conceptual design from navigational design in such a way that the nodes and links in navigational models are based on the objects, attributes and relations found in the conceptual model [8,9,10,11,12]. For example, Fig. 3 shows the conceptual and navigational models for part of a health care information website in OOHDM [8].

In OOHDM, navigational objects (nodes and links) are explicitly defined as views on conceptual objects. Nodes are composed of attributes that potentially belong to several classes in the conceptual model. In the conceptual model shown in Fig. 3, a *Medical Condition* class has as attributes the *Name* and *General Information* about it. The symptoms associated with a condition are a related but separate class. Treatment is also on a separate class.

In the navigational model, a node based on the *Medical Condition* conceptual class shows more than merely the *Name* and *General Information*. A list of *Symptoms*, *Tests* and available *Treatments* are also displayed in this node. Here, only *Test* names are displayed (other attributes are hidden at this level), and these names are anchors that trigger navigation to the *Test* node. A similar thing occurs with the Treatment node. However, not necessarily all conceptual classes become nodes. In our example, there is no node corresponding to the *Symptom* class.

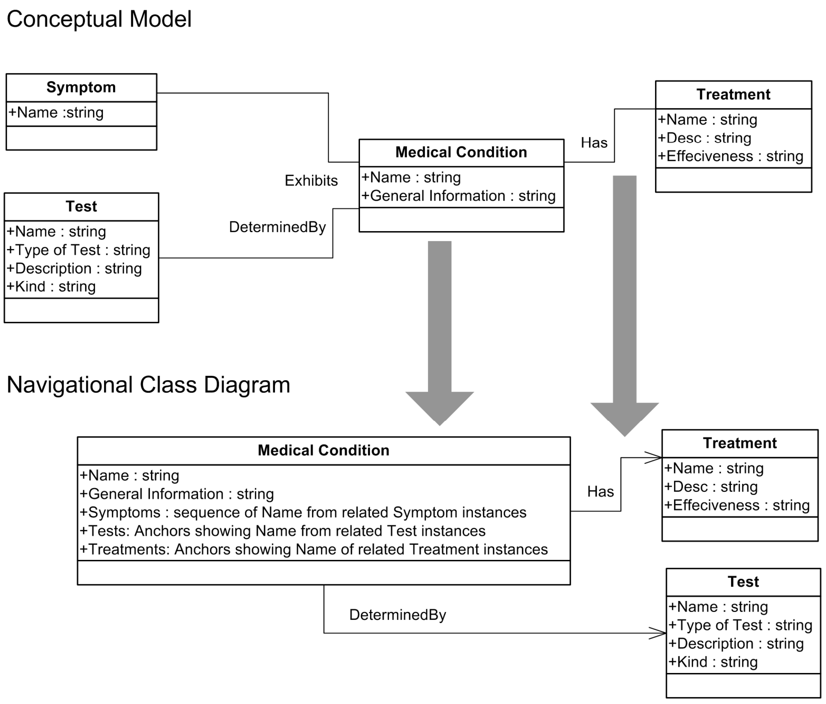


Fig. 3. Conceptual and Navigational models for a health care information website

Links are the navigational realization of relations appearing in the conceptual model. In our example, the *Has* relation between the *Medical Condition* and *Treatment* conceptual classes becomes the *Has* link between the *Medical Condition* and *Treatment* nodes.

This separation of concerns allows web developers to deal with the understanding of complex domains and the creation of a navigational scheme that expresses this underlying domain as separate issues. At the same time, these practices force the developers to elaborate a solid and coherent underlying foundation (the domain model) that will be rendered to the web site user in the form of a concrete navigation structure.

A hypertextual editor's navigational design should also express an underlying structure. The key difference is that the underlying structure being expressed through navigation is not an object model, but rather the syntax of a domain-specific programming language. In order to do this, there must be a correspondence between language syntax and navigational design. Fig. 4 is an example of the correspondence between PTL syntactic elements as defined in XML Schema [17,18] and part of Benefit Configurator's navigational classes.

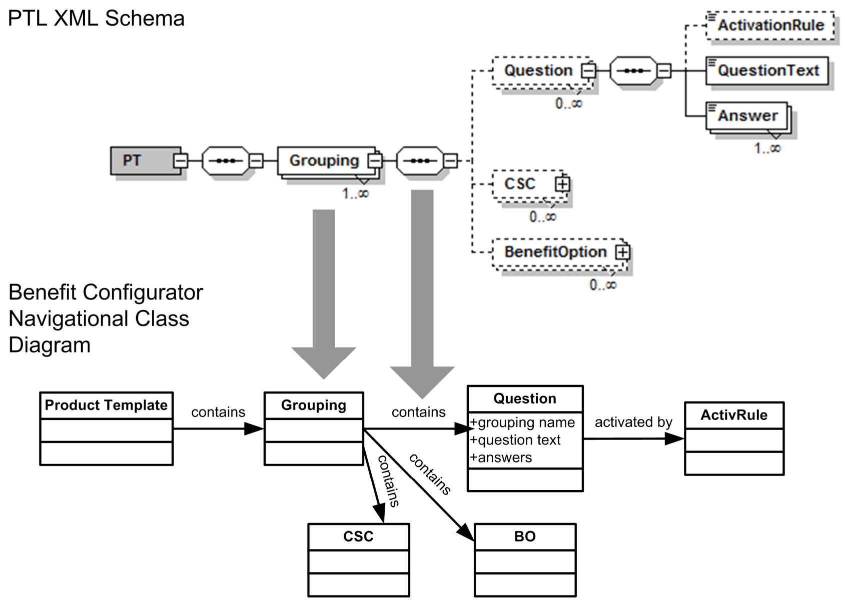


Fig. 4. An example of correspondence between PTL's XML Schema-defined syntax and Benefit Catalog's navigational class diagram. PTL's XML Schema is presented using XML Spy's visual schema notation. Tag attributes are not shown.

Nodes have a correspondence with XML language tags. The node's content may come from the data contained in the tag that it represents and also from related tags. For example, the *Question* node contains, among other things, the *grouping name*, from the parent *Grouping* tag; the *question text* attribute, from the *QuestionText* child tag; and the list of *answers* for the question, from the *Answer* tags below.

Not all tags become nodes. For example, there is no node that corresponds to the *Answer* tag. The contents of these tags are displayed in the *Question* node. Also note that not all the complete contents of a tag are shown in the node that represents it. For example, the *Grouping* node does not show all the details of the *Question* tags that it contains. The question of what tags should constitute nodes and what information to include in them are design choices that have to be decided by the software engineers in charge of the project. General Web design and usability guidelines should be taken into account [19].

Nodes are weaved by the links that connect them in such a way that links correspond to the syntactic rules that define the language structure. The links in Benefit Catalog correspond to the XML Schema definitions that specify the tag structure. For example, the *contains* link from *Grouping* to *Question* corresponds to the *xs:sequence* XML Schema definition that specifies the content of the *Grouping* tags to include a sequence of *Question* tags.

One last element that needs to be defined in order to complete the navigation design is the context diagram. The context diagram groups navigational objects into sets and defines access structures that the user can employ to reach and move through these objects. Fig. 5 shows part of the context diagram for Benefit Catalog.

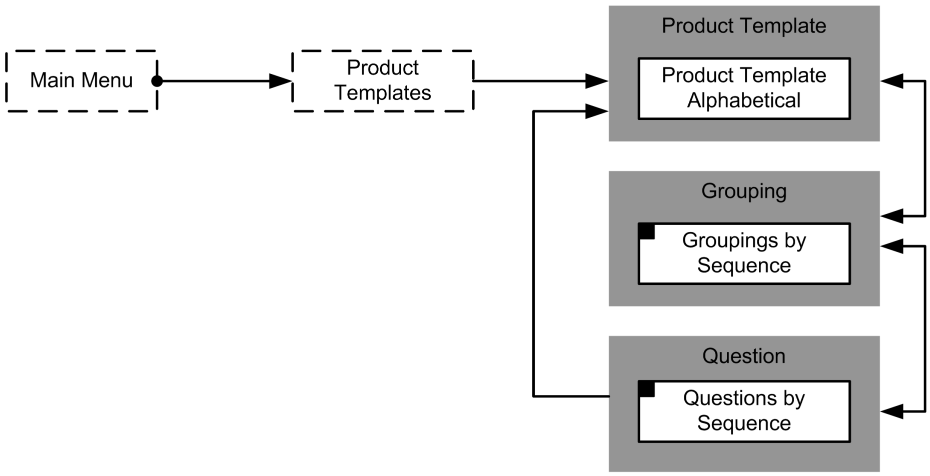


Fig. 5. Part of the Benefit Catalog context diagram

From the main menu, users have direct access to an index of product templates, listed alphabetically. When users access this index, they enter into the *Product Template Alphabetical* context, where product templates are listed by name. From that context, the user can go to the *Groupings* context where groupings are displayed by sequence order.

Although Benefit Catalog does not use this feature, in OOHDM, navigational classes may be decorated when appearing in a particular context. Decorating navigational classes may become important as more potent context diagrams are built.

3.2 Navigating beyond Syntax

Benefit Catalog has a very simple context diagram that stems directly from PTL's hierarchical XML Schema definition, e.g. groupings and questions are only displayed by sequence.

It is important to point out that many other contexts can be built around these navigational elements, providing the user with different views of the program code. For example, a possible improvement for Benefit Catalog could be displaying questions indexed by the variables that are used in its activation rules, or by its answers. This would help visualize what questions a certain variable helps turn on and off, or in what questions a certain answer is used in.

In fact, navigational design in web engineering in general is to a large extent, the definition of the various navigational contexts that the user will be traversing while performing the various tasks the applications purports to support. Therefore, the natural place to look for them is in the task descriptions (for example, in user interaction diagrams).

The potential features of hypertextual editors go beyond merely representing the underlying language syntax. The various tasks performed by end-users should be an important guide for organizing sets of navigational elements.

4 Related Work

4.1 Hypertext CASE Tools

There are different ways in which hypertext can support the software engineering process. Sometimes, these approaches are hard to compare because they may all use hypertext but they use it in completely different ways or to address different software engineering problems.

Østerbye developed a system to explore the idea of using hypertext for literate programming [22]. The goal of this work was to use the linking capabilities of hypertext to help weave together smalltalk code and documentation to facilitate inspection. In classic literate programming spirit, the aim was to construct a human-oriented representation of code and documentation. By using hypertext, the program can go beyond a linear document.

However, the advantages of this technique come at a great cost. The developer has to design all the navigation for the hypertext program representation. Even the authors point out that a drawback of this technique is "the well-known problem of hypertext, that one looses the feeling for where the information presently available at the screen belongs in the overall document".

First, it is important to point out that this system, and literate programming in general, assumes that there is an underlying programming language that will be used in the development process (Smalltalk in this case). Literate programming (with or without hypertext) uses this basic tool –the programming language(s)–, rearranging and combining source code with documentation in order to make them easy to absorb by a human reader; we can say that literate programming is at least one-level above purely textual programming languages. Hypertextual programming is proposed as an alternative to textual programming languages. Moreover, hypertextual programming could be used for literate programming.

There are some similarities between this literate programming system and the hypertextual environments described in this paper. In this literate programming system, some of the Smalltalk constructs are represented as nodes and some of the syntax rules as links. However, this relationship is not strict and the nodes contain important portions of textual code.

This Smalltalk literate programming system does not conform to our definition of hypertextual programming. Although the result of the programming process is hypertext (a program-document that developer can navigate through) and navigation may be used throughout the development process, programming is done primarily by editing text, not by using navigation. The most important criterion or our definition is not met. By using this or a similar system, the end-user would still have to learn a textual programming language. That's precisely what we are trying to avoid.

Using hypertext for end-user development also has to address the user disorientation problem. In order to do this, the web engineering techniques discussed in this paper were developed in part to deal with this problem. However, using these techniques for designing navigation is in turn a costly task usually done by professional web engineers. As opposed to this Smalltalk literate programming system, the present proposal does not leave navigational design to the developer (in our case, an end-user doing development). When designing a hypertextual editor, engineers have the responsibility of transforming language syntax into navigational design and create an application where the user is less likely to get lost.

Then, when end-users add or modify navigational elements, they may create links and nodes, but these actions do not alter the underlying navigational model (they simply instantiate it).

The Chimera open hypermedia system [23] uses hypertext to help manage and combine heterogeneous software engineering tools. Some of these engineering tools are programming language IDEs. Chimera also uses hypertext at a level above programming languages. The same can be said about Ishys [24].

Hypertextual programming editors may be one of the many systems combined by Chimera and other open hypermedia systems.

4.2 Visual and Textual Programming

Visual programming languages provide "some visual representations (in addition to or in place of words and numbers) to accomplish what would otherwise have to be written in a traditional one-dimensional programming languages" [25]. Despite its advantages, visual programming may have scalability problems [5], including scalability from a program-size standpoint and also scalability from a problem-domain standpoint.

From a program-size standpoint, the Benefit Catalog example that we presented was successfully used by end-users to develop programs (dynamic health care products) that are sizable and complex by various measures: i) the programs had several thousand rules; ii) they were collaboratively developed; iii) the development process of these programs typically takes several months; iv) these programs went through several maintenance cycles.

The various levels of abstraction and potentially different views of the source code given by a well-designed hypertextual editor, provide an adequate tool to deal with large programs.

From a problem-domain standpoint, this paper has provided guidelines to build hypertextual editors for any character-based, domain-specific language. Since domain-specific languages have been used in several areas, this suggests that hypertextual programming may also be scalable on this respect. In fact, we have used this technique in health-care insurance, small business rules definitions and programming email alerts for an academic system.

Visual programming techniques may be more appropriate to express some programming concepts. Even in these cases, hypertextual programming is well-suited for use in combination with visual and other programming and interface construction techniques.

In the case of textual programming, having to learn the syntax and conventions of character-based programming languages constitutes a considerable problem for end-users. The importance of this problem cannot be overstated. Providing a hypertextual editor that embodies the syntax and language conventions, transforming them in navigational paths to be traversed by the end-user significantly reduces this burden.

However, in the case of end-users who have already learned a textual domain-specific language, there may be no clear advantage in starting to use a hypertextual editor for the same language.

5 Conclusions and Future Research

In this paper we introduced the concept of hypertextual programming. This programming technique represents the program code as hypertext [6], allowing the end-user to inspect and modify this code mainly by using navigation.

Millions of users navigate the World Wide Web. Hypertextual programming leverages this widely available end-user skill to facilitate the construction of computer programs.

The user is provided with an environment that allows interactive source code exploration through navigation. A well-designed environment could facilitate reading and understanding by providing various views of the source code at potentially different levels of abstraction and a consistent way to move between them.

In a hypertextual editor, the user interacts with interface elements in order to modify the program code. On any given node, a carefully chosen set of relevant editing controls allows program modification without overwhelming the user. When combined with DSLs, many of these interface components may represent concepts that are familiar to the user. This technique is expected to significantly reduce the learning effort needed to begin developing domain-specific programs.

We presented and discussed a concrete example of a hypertext editor, Benefit Catalog, both as validation and to illustrate this technique. On this application, end-users have been effectively developing, testing, debugging, maintaining, deploying and running complex programs for dynamic health care policy configuration without the intervention of professional programmers or IT staff.

Beyond syntax representation, various navigational contexts may be created in order to provide the user with a rich set of navigation paths that take into account the various tasks that form the software development process.

However, reaping the benefits described above requires well-designed hypertextual editors. This entails significant engineering challenges. Among other things, editors have to express the syntax of the underlying language through a concrete navigational and interface design to begin with. In this paper, we argue that many of the techniques used in web engineering, most noticeably design methodologies [8,9,10,11,12], can be helpful on this respect, leveraging years of academic research and real-world experience.

Hypertext has been used in programming before. We reviewed three representative examples [22,23,24]. In general, all of these tools and techniques assume that there are one or more underlying programming languages and use hypertext to rearrange and/or link the potentially different program sources with other documents and products of the software engineering process. In general, they use hypertext on an above-language level. As an exception, in the Smalltalk literate programming system that we reviewed [22], some of the Smalltalk syntax is expressed in the form of links. However, nodes still contain significant portions of textual code and the rendering of the program in the form of nodes and links is guided by the literate-programming goal of human readability. In this system, although navigation may play a role in the development process, it is not the primary means to edit the program code. Text editing is still a central part of the development process. Therefore, this system does not conform to our definition of hypertextual programming. More importantly, the user has to know Smalltalk in order to use this system. The need to learn a textual language is precisely what hypertextual programming tries to avoid.

We made a comparison with these tools mainly to clarify that their use of hypertext is different and addresses other problems related to software development. The side-by-side comparison should not be with these techniques, but mainly against visual and textual programming.

Hypertextual programming is different from visual programming. The first does not rely mainly on the expressive power of graphics to facilitate the development process; it rather relies on the organization of the source code as a set of nodes and an intuitive mechanism to move around these nodes: navigation.

It has been argued that visual programming may have scalability problems [5]. Hypertextual programming can help to mitigate the problem of dealing with a great number of visual primitives at one time by providing different views of the program code and a systematic mechanism to tie them up: navigational links. At the same time, hypertextual editors can benefit from the use of visual techniques as part of their interface.

We provided general guidelines to build hypertextual editors for textual, domain-specific languages. This suggests that hypertextual programming may also be useful in different areas (domain-scalability). In fact, we have used this technique in health-care insurance, small business rules definitions and programming email alerts for an academic system.

With textual programming, the user has to learn the syntax and conventions of character-based programming languages. This constitutes a significant problem that hypertextual programming may help to solve. Providing a hypertextual editor that embodies the syntax and language conventions, transforming them in navigational paths to be traversed by the end-user may significantly reduce this burden.

We discussed some basic correspondence principles that should exist between XML Schema [17] syntax elements and a navigational model that may serve as guides in the design process. Still, more formal and detailed methodologies for designing hypertextual editors could be developed in the future.

Moreover, a careful and detailed review of the use of navigational contexts for building hypertextual editors may be beneficial.

Several design patterns for hypertext in general have been developed [20] since they were first introduced in [21]. Design patterns that are specific to hypertextual editors may be needed. In our experience, building nodes that are overly small or providing an excessive number of editing controls on a single node are not desirable. However, some of these problems may be related to more fundamental limitations of this technique. The answer may lie in the fact that some languages may be more suitable than others for use with hypertextual programming.

Our definition requires the hypertext editor to be able to generate all possible finite instances of the language and it requires navigation to be the main inspection and editing mechanism. Although a more formal demonstration should be performed, one seemingly direct consequence is that all (or at least the main) editing tasks for the given language should be achievable through navigation.

This paper discusses mainly languages defined in XML Schema. The specifics of other grammars deserve further investigation.

Since a hypertextual development environment has a correspondence with the syntactic elements of the underlying code, it may be viewed simply as a mapping between the language syntax and the possible ways of representing these elements as a web-application (the formatting and layout could be specified separately with CSSs). We are currently designing a web-based hypertextual editor generation application for XML Schema-defined languages.

The development environments discussed on this paper are mainly web-applications. Other forms of hypertext should also be considered.

References

1. Martin, J.: An Information Systems Manifesto. Prentice-Hall, Englewood Cliffs (1984)
2. Cypher, A. (ed.): Watch What I Do: Programming by Demonstration. MIT Press, Cambridge (1993)
3. Lieberman, H., Paternò, F., Klann, M., Wulf, V.: End-User Development: An Emerging Paradigm. In: Lieberman, H., Paternò, F., Wulf, V. (eds.) End User Development, pp. 1–8. Springer, Netherlands (2006)
4. Jess, the Rule Engine for the Java Platform, `http://herzberg.ca.sandia.gov/`
5. Burnett, M.M., Baker, M.J., Bohus, C., Carlson, P., Yang, S., van Zee, P.: Scaling up Visual Programming Languages. IEEE Computer 28(3), 45–54 (1995)
6. Conklin, J.: Hypertext: an introduction and survey. Computer 20(9), 17–41 (1987)
7. Murugesan, S., Desphande, Y.: Web Engineering. Software Engineering and Web Application Development. LNCS-Hot Topics. Springer, New York (2001)
8. Schwabe, D., Rossi, G.: An Object Oriented Approach to Web-Based Application Design. Theory and Practice of Object Systems 4(4) (1998)

9. Fons, J., Pelechano, V., Albert, M., Pastor, O.: Development of Web applications from Web enhanced conceptual schemas. In: Song, I.-Y., Liddle, S.W., Ling, T.-W., Scheuermann, P. (eds.) ER 2003. LNCS, vol. 2813, pp. 232–245. Springer, Heidelberg (2003)
10. Ceri, S., Fraternali, P., Matera, M.: Conceptual modeling of data-intensive Web applications. IEEE Internet Computing 6(4), 20–30 (2002)
11. Knapp, A., Koch, N., Zhang, G., Hassler, H.M.: Modeling business processes in Web applications with ArgoUWE. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 69–83. Springer, Heidelberg (2004)
12. De Troyer, O.: Audience-driven Web design. In: Rossi, M., Siau, K. (eds.) Information Modeling in the New Millennium. IDEA Group Publishing, Hershey (2001)
13. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0., 3rd edn. W3C Recommendation (2004)
14. Ortiz-Chamorro, S.: Cytera. Rules Language Specification. Technical Report, CyteraSystems (2001)
15. Ortiz-Chamorro, S., Aquino, N., Rubin, R., Cernuzzi, L.: AtOOmix: un Lenguaje Extensible de Reglas de Negocios. In: Proceedings of CLEI 2008 (to appear) (2008)
16. Eclipse Home, `http://www.eclipse.org/`
17. Thompson, H.S., et al.: XML Schema Part 1: Structures, 2nd edn. W3C Recommendation (2004)
18. Biron, P.V., Malhotra, A.: XML Schema Part 2: Datatypes, 2nd edn. W3C Recommendation (2004)
19. Nielsen, J.: Designing Web Usability: The Practice of Simplicity. New Riders Publishing, Indianapolis (1999)
20. Hypermedia Design Patterns Repository, `http://www.designpattern.lu.unisi.ch/index.htm`
21. Rossi, G., Schwabe, D., Garrido, A.: Design reuse in hypermedia applications development. In: Proceedings of Hypertext 1997, pp. 57–66 (1997)
22. Østerbye, K.: Literate Smalltalk Programming Using Hypertext. IEEE Transactions on Software Engineering 21(2), 138–145 (1995)
23. Anderson, K.M., Taylor, R.N., Whitehead, E.J.: Chimera: hypermedia for heterogeneous software development enviroments. ACM Transactions on Information Systems 18(3), 211–245 (2000)
24. Garg, P.K., Scacchi, W.: ISHYS: Designing an Intelligent Software Hypertext System. IEEE Expert: Intelligent Systems and Their Applications 4(3), 52–63 (1989)
25. Shu, N.: Visual Programming. Van Nostrand Reinhold, New York (1988)