Transactions of the Illinois State Academy of Science
(2007), Volume 100, #1, pp. 71-80

received 12/21/05
accepted 11/26/06

A Case Study in Reuse: An XML-Editing Component and Contract Editor

Laurence Leff and Mina A
Western Illinois University
Macomb IL 61455

I. INTRODUCTION AND OVERVIEW

Many decades ago, Bill Joy developed the first full-screen editor for UNIX, **vi**. Users could edit files on ASCII terminals by moving the cursor up, down, backwards and forwards with the **h**, **j**, **k**, **l** keys (the home keys) and press other sequences to delete lines, words, sentences and insert text. On a slow connection on an ASCII terminal, a few lines might be displayed; if one enjoyed a relatively high speed connection, one would be able to edit using a full screen. Note that this was before "WIMP" interfaces, such as the Xerox Star, Macintosh, and Windows, were widely available.

This worked using the "escape codes" on the terminal. When a computer program sent a sequence of characters to the terminal, it might move the cursor, clear the screen, delete a line or ring the bell. Unfortunately, this was different for each brand of terminal. So Bill Joy "abstracted" the problem. A file named **termcap** contained the sequences needed for each operation. The users would set an environment variable to indicate which terminal they were using. **vi** would read the **termcap** file and issue the correct sequence of escape codes to implement the user's editing request. (Gaughan, 2003) This was called optimal cursor movement, as the logic in vi will determine the sequence of escape codes with the fewest, or close to the fewest, number of characters needed to change the display.

Ken Arnold created a library called **curses** by "simply lifting nearly intact" the routines to implement "optimal cursor movement" from Bill Joy's editor. (Arnold), (Arnold and Amir). This was used for many full-screen programs, most notably, an "adventure game" called Rogue (Wichman, 1997). Wichman indicated that his group, which originally implemented Rogue, simply used it after "it made the rounds to other Universities." Some web sites, however, including (die.net) (Foldoc, 1993), indicate that **curses** was developed specifically to support games.

In this article, we report an analogous sequence, over two decades later. Mr. Go Eguchi developed a Graphical User Interface (GUI) to create rule-base for transforming XML files (Eguchi and Leff, 2002) (Tong, et. al., 2005). Those works define two expert systems that would examine XML files. It would match this information against the XML appearing in the left-hand side of the rule. These expert systems would then generate the XML designated by the right-hand side of the rule, transferring information from the input. The standard (Leff, 2002) also defined additions to the XML so that when the expert system is run, information that was in the input file being matched is transferred into the output XML. The goal of the editor was to allow the rule-writer or user to create the XML defining the transformations to occur.

That editor's user interface had two panes, a menu bar and three sets of buttons. The user, or rule writer, develops the XML rule-base in the right-hand pane. The rule-writer loads samples of the XML to be transformed in the left-hand pane. They use the left-hand set of arrow buttons to move within that XML. Then, using the right-hand set of arrow buttons, the rule-writer moves within the rule-base to indicate which rule to create or modify. In the middle set of buttons, there is a "Move From Sample" button. The user presses this to set up the example. Then, the user has to modify the XML moved to the left hand set of the rules to add the special markup. The rule writer moves within the rule-base (again with the right set of buttons) and inserts the special markup to indicate how to recognize the inputs that are considered similar. Lastly, the user loads a sample of the XML to be produced when the input is recognized, and similarly, they move some of the XML to the "right hand side" of the rule. The reader of this paper does not have to be concerned with the above detail; only to realize that Mr. Eguchi implemented code to allow transformation of XML for the specific purpose of implementing the above-described system, and he embedded that within his system without planning for any reusability.

The first author extracted this into a Java Swing "component" for general use. Swing is a package, provided by Sun, which programmers use to develop GUIs using the Java programming language. The items on one of these implemented GUIs which includes other panels, edit boxes, buttons, sliders, and menus are all components (Walrath and Campione, 1998) derived from the class **Component** (Java, 2004). The programmer sets up a "containment hierarchy" in their GUI; for example, a main screen might contain several panels, some of which may have buttons or edit boxes.

The XML-Editing **Component** which is the subject of this report contains a panel in which XML is displayed with a set of four arrow buttons; the user uses these to navigate the XML. One element or "tag" is always highlighted in red. That is, when the user clicks the left arrow, the containing tag is highlighted; when the user clicks the right arrow, the highlight changes from the current tag to the immediate child on which the mouse is selected. The Up and Down arrows move the cursor from the current tag to the subsequent, or previous tag at the same level.

The programmer creates the XML editing component using its constructor and adds it to the frame or another panel using the **add** method of the frame. This is precisely how they would add an ordinary built-in edit box or button. Then they can use **setParser** to load an XML file, which is displayed. The programmer can retrieve a **parser** object. This allows the user to add or otherwise manipulate elements. It also supports the operation **getPointer** so the programmer can get the node that the user has highlighted. For example, assume the user selected an element that represents an "if," and selected a button to add a "then" part. The logic for a button would use the **getPointer** method to get a pointer to this element and then use the XML manipulation methods that are part of the packages that come with the Java programming language to add the new information.

The senior author developed an XML standard for legal contracts (Leff, 2000). It was proposed to the Legal XML standards organization. A legal contract can be viewed as a series of obligations. This is a use of deontic logic which includes the modelling of obligations, prohibitions, and permissions (Boulmakoul, 2002). In a simple contract to purchase pizza, the pizza company has an obligation to deliver the agreed-upon number of boxes of and type of pizza to the purchaser. Conversely, the purchaser has an obligation to pay the amount. Some of these contracts may be obligations which may be conditional (Boulmakoul, 2002). For example, the purchaser might state that the pizza is only to be delivered if it was not raining. One application of deontic contract languages and logic is the monitoring of contract compliance, particularly in complex situations such as quality-of-service agreements (Neal, et. al., 2003). The Distributed Systems Technology Centre at the University of Queensland has developed contract monitoring systems based upon Business Contract Language (Neal, et. al., 2003) (Meyer, 2005).

The first author used the component in a contract editor. It allowed the user to create the contract interactively. She used the **getPointer** method to determine which clause that the user selected. Then, the user could issue commands to add new conditional elements to that clause.

II. SETTING: SOFTWARE REUSE

The idea of reusing programs, software knowledge and software “artifacts” such as documentation or test plans was mentioned in the famous 1968 NATO Software Engineering Conference that is considered the birth place of software engineering and is the subject of several books and articles, as reviewed in (Krueger, 1992). Several reports show that large organizations that develop software make good use of reuse techniques and report productivity gains (Lim, 1994) (Mili, et. al., 2001) (Poulin, 1997). (Krueger, 1992) recognizes that “many programmers” copy contiguous blocks of code. In many cases, no effort is made to repackage the software copied so it can be be easily used a third or later time. The programmer simply copies the code to solve the immediate problem (Caldiera and Basili, 1991). Arnold’s work, (Pant, et. al., 1996), and this case report cover cases where the code is extracted for the purpose of reuse. (Pant, et. al., 1996) had developers implement three systems: a matrix class, a name and phone manager, and an inventory manager. Then, the programmers identified potentially reusable classes and prepared a class library of thirteen reusable components. They found that generalization increased the average number of lines of code per module from 226 to 264 lines, with similar increases in other measures of complexity. They found that the time to perform the generalization was 55% of the original development effort.

(Laubile and Visaggio, 1997) applied a concept called “program slicing” to help identify reusable software. However, the senior author was unable to find a report that this technique extracted a reusable software module from previously-written code.

III. SETTING: XML AND THE CONTRACT STANDARD

XML has angle brackets and tags that makes it superficially similar to HTML—both come from Standard Generalized Markup Language (SGML). See Figure Three at the end of the article for an example of XML. However, each type of XML document uses its own tags, often defined by a standards organization. For example, Common Business Language (xcbl.org, 2004), RosettaNet, and the Open Application Group develop standards for purchase orders. Thus, many different organizations needing to purchase supplies could use a common XML, even if they use different software to manage their accounts receivable, accounts payable, and delivery management (Cooke, 2001). These tags describe and mark up the text contained within them. Techniques such as DOM processing allow a programmer to implement software to easily find the tags representing the desired information item and retrieve it (Maruyama et. al., 1999).

	Functions	Mr. Eguchi's Program	My Contract Editor	Hypothetical Editor for MathML
Commonality	The traversing buttons that allow user to move around	User can move across both the sample XML document and the rule-based XML document being formed	User moves to select a clause to receive a new **Then** or **Implication**	User moves to select an expression to be modified or evaluated
	Visually shows the selected and non-selected part of the XML clearly	Part to be moved from sample document is displayed in red. Part to be edited in rule-based document is also marked in red	Anything that is newly added or selected is displayed in red. New party, clause to receive, new **Then** or **Implication**	Anything that is added or selected is displayed in red.
Variability	The editing buttons that allow user to create or manipulate the XML document	Replace text with variables etc.	Add a new party, clause to receive, new **Then** or **Implication**	Evaluate a sub expression. Modify subexpression by introducing a number, variable, operator, indentifie structure etc

LegalXML's aim is to standardize XML used for exchange among the entities participating in the legal system; it is a collection of Technical Committees. In 2002, it joined with the Organization for the Advancement of Structured Information Standards (OASIS) (Mountain, D., 2003). One of the Technical Committees is the eContracts Technical Committee. The contract proposal mentioned in Section I. was submitted originally to Legal XML in August 2000 and it is now on the OASIS web-site in the eContracts Technical Committee documents section. The hope is that different organizations would be able to exchange contracts as they are negotiating them, but would need to use the same schema (Meyer, 2005). In addition, XML would allow the reuse of contract clauses or boiler plate, called "precedent documents," and the exchange of information with and between systems developed by contract management vendors.

IV. REUSING THE XML NAVIGATION AND DISPLAY CODE

As mentioned in Section I, we started with Mr. Go Eguchi's XML editor which had two panes. And, we created a reusable Java Swing Component, **ComponentForXml**. In creating a reusable class for a domain, one must locate a set of applications that have commonality, but identify those areas that would vary for each application (Frakes and Isoda, 1994) (Fayad, et. al., 1999).

This is best explained for **ComponentForXML**, by identifying two different application domains. One is an editor for contracts, for which it was successfully used. The other is a hypothetical editor for MathML. MathML is a W3C standard that can be embedded in web pages for the display of mathematical equations and to allow exchange of information between mathematical programs, e. g., between symbolic mathematics systems such as Mathematica, Macsyma and Maple (Topping, 1998).

As mentioned in Section I, Mr. Go Eguchi developed a Graphical User Interface for XML rules whose main file was **GuiForXml.java**. This contained the front end that interacted with a rule-writer for XML transformation. **GuiForXml.java** contained 1833 lines of code (LOC). Of these, 334 lines represented interacting with the user to traverse the XML and displaying in red the part selected. **GuiForXML** also created and displayed the panels to contain the XML and the arrows and supported two DOM trees (as **DomParserPart**) for the sample XML and the rule-based file being created.

Thus, the first author simplified this to only one panel and one DOM Tree for **ComponentForXML**. The 334 lines were revised slightly giving 322 lines in this component representing reused code.

Three smaller class files were used with no change, **DomParserPart** (281 LOC), **ApplicationPanel** (47 LOC), and **ALine.java** (22 LOC). The first contained the recursive routine that converted the XML, represented as a DOM tree, into an array of ALine objects. The other two represented the panel on which the user saw the representation of the XML and a single line, as displayed in the panel, respectively.

However, in order to make a reusable **Component**, several facilities needed to be generalized for the programmer. Consistent with standard practice in developing object-oriented code, we implemented a "getter" method to retrieve the **DomParserPart**. Also, the first author added three other methods. The programmer uses these to implement loading a new file into the display area. For example, the programmer would use this when coding the action to be performed when the user does "File" and "Open" from the menu. (Mr. Eguchi hard coded the menu and these functions into his program, so we set up the code to allow the programmer using the component to access this capability.) These four methods were 90 LOC added.

The **DomParserPart** also supported an important method, **getPointer** which got a pointer in the DOM tree to the XML that the user selected and currently sees highlighted in red. The **DOMParserPart** also had a "getter" method to retrieve the DOM model representing the XML argument. From this, the programmer would invoke methods (Maruyama, et. al., 1999) (Java, 2004) for such purposes as adding an attribute to it or adding a tag below it. This was used in the contract editor, described in other sections, to add an implication to the currently selected clause.

Thus, we reused 672 LOC from Mr. Eguchi's rule editor and added 90 LOC which is an 11% addition to make a reusable class. This is low compared to that reported by (Pant et. al., 1996) to make code implemented for a specific purpose into a reusable object.

Unfortunately, we did not keep a record of the number of hours spent by Mr. Eguchi or the first author of this article. Both completed Masters degrees at Western Illinois University doing the work as a final project or thesis, but neither had a prior degree in a computer-related field. However, the first author of this paper started her project very early in her career at WIU. Thus, even if they had records of the number of hours spent on their respective projects, they would not be comparable. Thus, all we can observe regarding time is that the first author wrote the code that was reused as well as an application and the second author made it into a reusable component and used it in a different application in a time frame consistent with a Masters project or Masters thesis.

V. THE CONTRACT EDITOR

As mentioned in Section I, the first author implemented an editor for legal contracts that are represented in XML based upon a suggested standard for the OASIS eContracts Technical Committee.

A legal contract contains a list of the parties to the contract (Harrop, 2004). Corresponding to this, our contract editor has buttons to prepare the XML containing a list of the parties and their addresses. Each party has a unique PartyID by which it is referenced in the clauses.

Then, the contract editor's user creates clauses using the "Add Clause" button. A dialogue pops up, and the user enters the Clause ID (which must be unique) and one of the PartyID's for the "Clause From" and "Clause To" These would indicate the party who owed the obligation and the party to whom they owed the obligation.

Many clauses just contain text such as "deliver ten boxes of pizza." However, most interesting are the implications. They indicate that the obligation only exists when something else happens. Sometimes, this is text. For example, if the pizza was for a picnic, the parties might agree that the pizza would only be delivered it was not raining on that day. Alternatively, one obligation only is in force if a prior action to which the parties agreed occurs. For example, the buyer is only obligated to make a payment after the product was delivered.

The user will often use the contract editor to replace an ordinary clause, created as indicated above, by an implication. Here (Figure One), the user used the arrow buttons to traverse to the clause containing "Deliver Ten Boxes of Pizza." Then, the user selected **Add Implication** and the dialogue you see popped up. The user entered the text of the condition that it not be raining.

Now, the user must indicate what would happen if it was not raining on the indicated day. In this case (Figure Two), that means that Acme Pizza would deliver the pizza to the first party. The user traversed to the "Then" part of the clause and selected **Add Clause to Then**. In the resulting dialogue, the user provides a new clause id and selects the parties from whom and to whom the obligation is owed. They also enter the text of the clause, which in this case is a repeat of what was already in the clause.

Then, the user will create a new implication in a similar manner. However, the user will select Clause ID C002 in the "Add Implication" dialogue instead of entering "Condition Text." In the "Add Clause to Then," the user would enter text for the obligation which would be to pay money, and the "Clause From" would be P001 for the buyer. The resulting XML Contract after all of these steps is shown in Figure Three.

VI. RESULTS AND CONCLUSIONS

The senior author did a search for "code scavenging." Then, a citation search was done on the relevant articles—which articles cited them. This was done in both citeseer.IST, the Scientific Literature Digital Library started by the NEC, now run by Pennsylvania State University and sponsored by Microsoft Research, the National Science Foundation, NASA and the ACM. That author also searched for citations to Arnold's work on **curses**, as well as (Pant, et. al., 1996). Thus, although (Kruger, 1992) reported that code scavenging was widely used in practice, (Pant, et. al., 1996) was the only article found that reported on specific experience extracting a reusable component, class or library subroutine from preexisting code not intended for reuse. Thus, this modest contribution to the literature.

(Meyer, 2005) discusses the difficulties in authoring contracts in XML. (Lauritsen, 1992) reports little analysis or empirical analysis of what happens when one authors a legal document. We acknowledge concerns[1] that it might not be easy for individuals to work with an XML representation. (An alternative would be a natural language representation of the contract that would be updated interactively as the user activated certain buttons.) However, we note Van Enger's work on representation methods for tax legislation. The system that the users most preferred turned out to be the one in which they made the most errors in comprehension (van Engers, et. al., 2002). The best user interface for the structured creation of contracts can only be resolved by empirical work, which is "further work" not only for this editor but for all systems and ideas for generating legal discourse, documents and representation (Lauritsen, 1992)[2].

So, we make this modest contribution by demonstrating one more interface style for deontic logic in specific, and XML manipulation for contracts. It is now available as an open source project, http://sourceforge.net/projects/amina.

[1] Some came from the members of the first author's Masters Thesis Committee. Others came when a version of this work was submitted and rejected by a conference.

[2] This was confirmed by later research by the senior author and by a personal communication from Peter Meyer in 2005.

REFERENCES

Arnold, Kenneth C. R. C., “Screen Updating and Cursor Movement Optimization: A Library Package” Computer Science Division, Department of Electrical Engineering and Computer Science, http://www.ocf.berkeley.edu/Library/Computer/UnixProgMan/curses/Curses.doc

Arnold, K., Amir, Elan, “Screen Updating and Cursor Movement Optimization: A Library Package” http://docs.freebsd.org/44doc/psd/19.curses/paper.pdf

Boulmakoul, A., “Integrated Approach to Contract Lifecycle” Procedings of the Ninth Workshop of the HP OpenView University Association June 11-13th 2002 http://www.hpovua.org/PUBLICATIONS/PROCEEDINGS/9-HPOVUAWS/

Caldiera, G. and Basili, V. R., “Identify and Qualifying Reusable Software Components” IEEE Computer, Vol. 24, No. 2. pages 61-70 February 1991.

Cooke, J. A. “XML Comes to Logistics” Logistics Management and Distribution Report Vol 40 No. 11 pages 57-62, November 2001.

(die.net) On-Line Dictionary

Eguchi, G. and Leff, L. L. “Rule-based XML: Rules About XML in XML to Support Litigation Regarding Contracts” Artificial Intelligence and Law Vol 10: pages 283-294, 2002.

Fayad, M. E., Schmidt, D. C., Johnson, R. E., Building Application Frameworks: Object-Oriented Foundations of Framework Design, John Wiley and Sons, New York, 1999.

Foldoc, Rogue Free On-Line Dictionary of Computing, 1993. http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=rogue

Frakes, W. and Isoda, S., “Systematic Reuse” IEEE Software Vol. 11 No. 5 pages 15-19, September 1994.

Gaughan “A Short Introduction to Curses” March 22, 2003. http://talideon.com/ notes/dcom3/curses.pdf

Harrop, J., “eContracts Structure Markup – Preliminary Report: eContracts Structure sub-committee” July 19, 2004. http://www.oasis-open.org/committees/download.php/13191/ pdf00001.pdf

Java, “Java 2 Platform Standard Edition 5.0 API Specification”, 2004. http://java.sun.com/j2se/1.5.0/docs/api/

Krueger, C. W., “Software Reuse” ACM Computing Surveys, Vol. 24, No. 2., pages 131-183, June 1992.

Laubile, F., Visaggio, G. “Extracting Reusable Functions by Flow-Graph Based Program Slicing” IEEE Transactions on Software Engineering Vol. 23, No. 4, 246-259, April 1997.

Lauritsen, “Technology Report: Building Legal Practice Systems with Today’s Commercial Authoring Tools” Artificial Intelligence and Law Vol 1, pages 87-102, 1992.

Leff, L. Legal XML: Unofficial Note: Suggested Contract Standard, 2000, http://www.oasis-open.org/apps/org/workgroup/legalxml-econtracts/documents.php

Leff, L. Standardization Rule-Based Transformation, and Generation of Rule-Based Processing, Transformation, and Generation of Legal XML Documents, Document Number WD_100XX_2001_05_05, Now at http://www.wiu.edu/users/mflll/WD.html

Lim, W., “Effects of Reuse on Quality, Productivity and Economics,” IEEE Software, Vol. 11, No. 5, Pages 23-30, Sep 1994.

Maruyama, Tamura, K., Uramoto N., XML and Java: Developing Web Applications, Addison-Wesley, Reading, MA 1999.

Meyer, P., “Requirements for Technical Specification: OASIS Legal XML eContracts TC” May 2005. http://www.oasis-open.org/committees/download.php/12799/econtracts.pdf

Mili, H., Mili, A., Yacoub, S., Addy, E., Reuse-Based Software Engineering: Techniques, Organizations, and Measurement, Wiley-Interscience, 2001.

Mountain, D., “XML E-Contracts: Documents that Describe Themselves” International Journal of Law and Information Technology, Vol. 11 No. 3 pages 274-285, 2003.

Neal, S., Cole, J., Linington, P. F., Milosevic, Z., Gibson, S., Kulkarni, S. “EDOC: Proceedings of the Seventh International Conference on Enterprise Distributed Object Computing” IEEE Computer Society Press, page 50-60, 2003

OASIS, “OASIS Legal XML” www.legalxml.org

Pant, Y., Henderson-Sellers, B., Verner, J. M., “Generalization of Object-Oriented Components for Reuse: measurement of Effort and Size Changes” Journal of Object-Oriented Programming, Vol 9 No. 2. Pages 19-31 and 41, May 1996.

Poulin, J. Measuring Software Reuse: Principles, Practices and Ecopnomic Models, Addison-Wesley, Reading MA, 1997.

Tong, T., Eguchi, G. Cheon, J., Callahan, J., Leff, L., "Rules About XML in XML" submitted to Expert Systems with Applications, 2005.

Topping, P. "MathML Requirements" May 14, 1998. http://www.w3.org/Math/W3CDocs/mathmlrequ.html

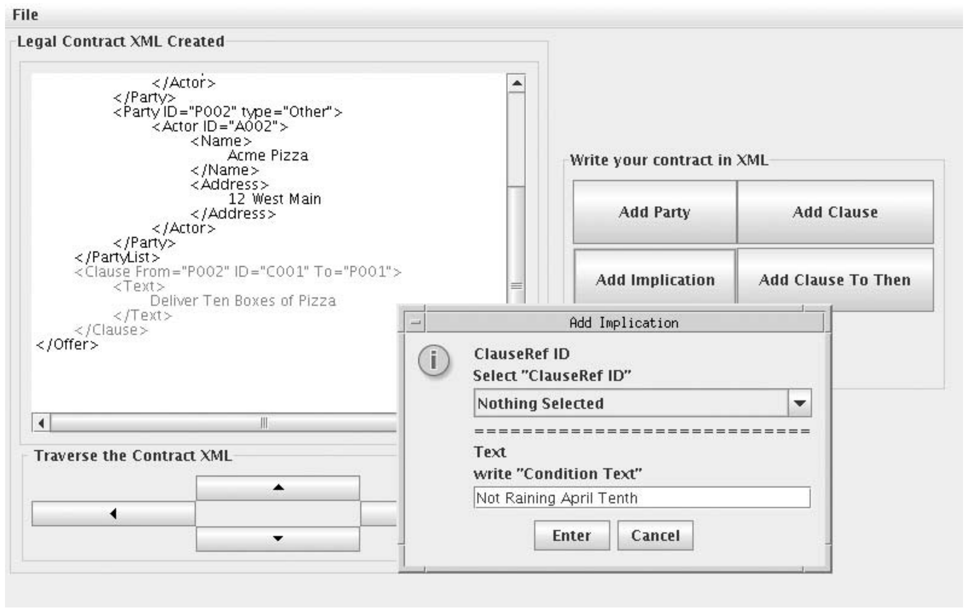
van Engers, T., van Driel, L., Boekenoogen, M., "The Effect of Formal Representation Formats on the Quality of Legal Decision Making" in Legal Knowledge and Information Systems. Jurix 2002: The Fifteenth Annual Conference, (Bench-Capon, T. J. M., Daskalopulu, A., Winkels, R. G. F. editor) pages 63-71.

Walrath, K., Campione, M., The JFC Swing Tutorial, Addison-Wesley, Boston, 1993.

Wichman, G. R., "A Brief History of Rogue" 1997 http://www.wichman.org/roguehistory.html

Xcbl.org, XML Common Business Library, 2004, www.xcbl.org

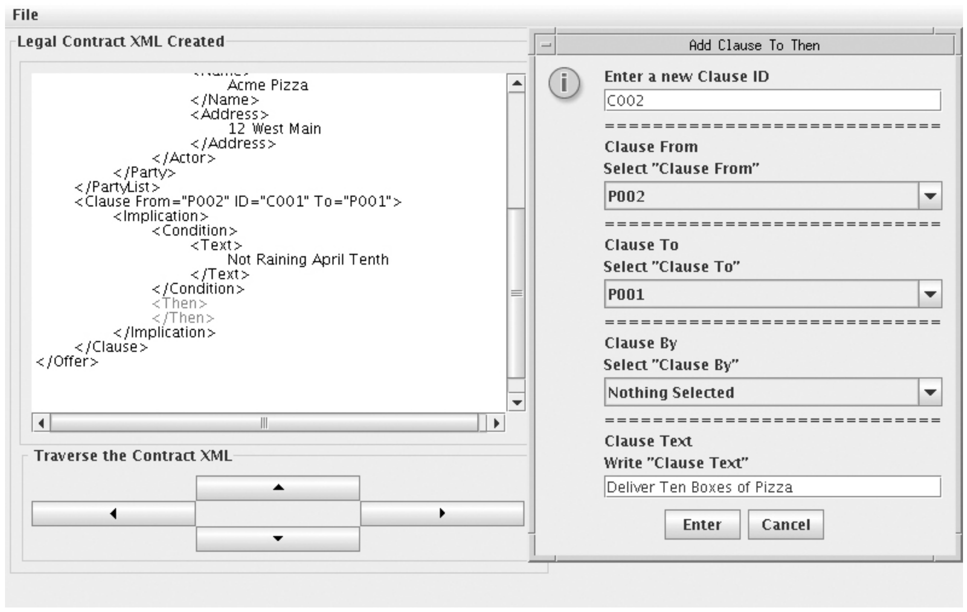
Figure 1. Adding an Implication to Replace a Clause



```
            </Actor>
        </Party>
        <Party ID="P002" type="Other">
            <Actor ID="A002">
                <Name>
                    Acme Pizza
                </Name>
                <Address>
                    12 West Main
                </Address>
            </Actor>
        </Party>
    </PartyList>
    <Clause From="P002" ID="C001" To="P001">
        <Text>
            Deliver Ten Boxes of Pizza
        </Text>
    </Clause>
</Offer>
```

ClauseRef ID
Select "ClauseRef ID"
Nothing Selected
============================
Text
write "Condition Text"
Not Raining April Tenth

Figure 2. Add Clause to Then Dialogue.



```
                    Acme Pizza
                </Name>
                <Address>
                    12 West Main
                </Address>
            </Actor>
        </Party>
    </PartyList>
    <Clause From="P002" ID="C001" To="P001">
        <Implication>
            <Condition>
                <Text>
                    Not Raining April Tenth
                </Text>
            </Condition>
            <Then>
            </Then>
        </Implication>
    </Clause>
</Offer>
```

Enter a new Clause ID
C002
============================
Clause From
Select "Clause From"
P002
============================
Clause To
Select "Clause To"
P001
============================
Clause By
Select "Clause By"
Nothing Selected
============================
Clause Text
Write "Clause Text"
Deliver Ten Boxes of Pizza