Generalized Selective XML Markup of Source Code Using Agile Parsing

James R. Cordy

School of Computing, Queen's University
Kingston, Ontario, Canada K7L 3N6
cordy@cs.queensu.ca

Abstract

Increasingly focus in the software comprehension community is shifting from representing the results of analysis in the graph and database domain to reflecting insights directly into source. The obvious modern representation for this reflection is XML markup. In the simplest case, XML markup of the abstract syntax tree itself can be represented in source, although the result is wordy, overly detailed and cumbersome to deal with. A more realistic solution is to use island or multi-weight parsing to mark up the AST in only those sections of source of interest to the current task.

In this paper we outline a method for extending and generalizing the partial markup idea to minimize source markup not only by marking only sections, but by selectively marking up the source with only a subset of the AST nodes relevant to each particular task as well. By exploiting agile parsing, this idea is further extended to allow for task-directed selective markup as a natural extension of selective syntactic markup.

1. Introduction

Attachment to source is increasingly one of the most important problems in program comprehension [1,2,3]. If real programmers are to take action based on the inferences and insights made possible by design recovery and analysis, the results of these insights must be presented in a way that attaches them to the source code itself. Moreover, as program comprehension technology matures, the possibility of automating appropriate reprogramming of source becomes a more and more tantalizing.

Both of these observations imply that representation of design information, including structure, semantics and business knowledge, need to be somehow attached directly to source. XML [4] provides an industry standard source markup technology that has the potential to provide us with an appropriate bridge. Using a partial XML markup of the parse tree in source code, McArthur [3] has demonstrated some of this potential. In this paper we extend and generalize the partial markup idea to minimize and focus source markup by selectively marking up in the source only those AST nodes that are relevant to each particular task. By exploiting the ideas of agile parsing, we generalize this idea to contextually and semantically sensitive selective partial AST markup as well.

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).□

2. Agile Parsing

Agile parsing [5,6] refers to the ability to use a customized version of the input language grammar for each particular analysis and transformation task. Based on a standard "base grammar" for the input language, agile parsing provides the ability to "override" nonterminal definitions on a per-task basis to modify the parse to yield an AST that makes the source analysis or transformation more efficient and convenient.

Although in theory different parsers can be generated for each task using traditional parser generator technology, agile parsing is most conveniently supported using an interpretive parser that supports execution time grammar definition and modification such as that provided by TXL [7,8]. Because TXL's parser interprets grammars directly at analysis and transformation execution time, it easily supports agile parsing.

The TXL language provides several features designed to support agile parsing. Each TXL program begins with a "base grammar" for the input language, a standard general purpose grammar typically based on the language's standard reference grammar. The basic agile parsing feature is the nonterminal "override", which allows a given nonterminal of the base grammar to be replaced with a definition more appropriate to the task at hand. Overrides are written in TXL using the "redefine" statement [Figure 1(a)]. The semantics of an override is that the effective grammar for the tool is the original base grammar with the definition of the overridden nonterminal replaced by given redefinition, yielding a different custom AST intended to make the task easier.

More sophisticated overrides can use extensions of the existing nonterminal form by referring to it using the the "..." notation [Figure 1(b)]. In a TXL redefine, "..." refers to the definition of the overridden nonterminal form before it was extended by the redefine. The "..." can be used in a post-extension, in which additional alternatives for the nonterminal are added after the originals by the redefine, or as a pre-extension, in which additional grammatical forms come first and take precedence over the original forms. Extension overrides

```
include "Cpp.Grammar"

redefine function_definition
   [function_header]
   [opt exception_specification]
   [function_body]
end redefine

define function_header
   [opt decl_specifiers]
   [function_declarator]
   [opt ctor_initializer]
end define
```

(a) Replacing a nonterminal in TXL.

*The **redefine** statement gives a new definition for the redefined nonterminal which replaces the original in the base grammar to yield a different parse. In this case, the new nonterminal [function_header] is introduced to capture the entire header line of C++ functions in one piece.*

```
include "Java.Grammar"

redefine expression
     ...
  |  [xmltag] [expression] [xmlendt
end redefine

redefine method_call
     [jdbc_call]
  |  ...
end redefine
```

(b) Extending a nonterminal in TXL.

*The **"..."** notation in a **redefine** statement refers to the original syntactic forms of the redefined nonterminal in order to allow extension of the nonterminal to other forms. In cases of ambiguity, the order of alternatives determines the parse. If the "..." appears first in an extension, then the old forms are preferred. If it appears last, then the new forms take precedence, as with [jdbc_call] above.*

Figure 1. Basic TXL Support for Agile Parsing.

often exploit TXL's ordered ambiguity resolution. New forms introduced by the redefine can ambiguously overlap existing forms, with the semantics that forms are tried in the order specified in the effective definition, with first forms taking precedence over later forms, yielding a well-defined deterministic parse.

Other features of TXL supporting agile parsing include general nonterminal polymorphism (*[any]*), nonterminal type query (*[typeof]*), programmed syntactic AST extraction (*[^]*) and transformation-time reparsing of transformed elements (*[reparse]*). Grammar reflection (modifying the grammar on-the-fly as the result of transformation or analysis) is theoretically possible and supported by the TXL engine but not yet directly available in TXL.

3. An XML-based TXL

FreeTXL [9] is a new implementation of the TXL language based on XML from the ground up. Because FreeTXL uses XML directly to represent parse trees, full XML markup of the AST in source is a simple flag and need not be programmed. For example, the result of the *–Dparse* ("Dump parse") flag in FreeTXL is a version of the input source fully XML-marked with the input parse [Figure 2].

Similarly, because the result of every FreeTXL transformation is a new parse tree internally represented using XML, the parse tree of the result of a TXL transformation can also be directly yielded using the *–Dfinal* ("Dump final parse") flag, which yields a version of the transformed output source that is fully XML-marked with the output parse of the source.

However, while for some purposes it may be useful, a fully

```
TXL 10.2d (18.9.02) (c)1988-2002 Queen's
University at Kingston
Compiling Txl/cpp.Txl ...
Parsing Examples/groff.cpp ...

----- Input Parse Tree -----
<program>
 <declaration_list><repeat_declaration>
   <declaration>
    <preprocessor>
     <literal text="#include"/>
     <literal text="<"/>
     <filename>
      <file_identifier><id text="stdio"/>
      </file_identifier>
      <repeat_dot_slash_identifier>
       <dot_slash_identifier>
        <opt_'.><lit_'.><literal text="."/>
        </lit_'.></opt_'.>
        <opt_'/><empty/></opt_'/>
        <file_identifier><id text="h"/>
        </file_identifier>
       </dot_slash_identifier>
       <repeat_dot_slash_identifier>
        <empty/>
        <empty/>
       </repeat_dot_slash_identifier>
      </repeat_dot_slash_identifier>
     </filename>
     <literal text=">"/>
    </preprocessor>
   </declaration>
   <repeat_declaration>
    <declaration>
     <preprocessor>
      <literal text="#include"/>
      <literal text="<"/>
      <filename>
       <file_identifier><id text="string"/>
       </file_identifier>
```

Figure 2. Example output of the *–Dparse* FreeTXL flag for the first line and a half of a C++ program.

XML-marked source parse is a huge and not very practical source representation [2]. More interesting is the question of how to minimize the XML-markup to focus on only those AST nodes that are relevant to the next task we have in mind [3]. For example, if we are interested in the call structure of the program, only those nodes representing method declarations and method calls may be appropriate.

4. Programmed AST Markup

If we are to gain control of AST markup in TXL, we must first address the issue in the programmed side of the language, not in the debugging dumps of the FreeTXL processor. Basically, we need to code a TXL transformation that transforms an input program to its own XML-marked up source AST. In order to do this, we can make use of the agile parsing [5,6] features of the language to define a generic set of XML markup syntactic forms [Figure 3(a)]. We can then write a simple polymorphic rule to visit every node in the input AST, query its node type and then use polymorphic transformation to replace it by its own XML-marked up form [Figure 3(b)].

This is actually trickier than it sounds, which is why the rule set in Figure 3 is not quite as simple as one might expect. Because TXL rules continue transforming to a fixed point, it is necessary to limit the scope of the visit rule to avoid looking inside already marked-up nodes ("**skipping** [xml_node]"), otherwise the transformation would never halt. However, by skipping marked-up nodes, the rule never visits any internal nodes. This is corrected using a recursive rule invocation in the XML markup function itself ("[toXml Node]") to explicitly

```
% TXL generic transform of input program
% into XML parse tree

% This line is the ONLY language dependency
% in this program!
include "Cpp.Grm"

% Polymorphic XML markup grammar
define xmlnode
    [xmltag] [any] [endxmltag]
end define

define xmltag
    < [id] >
end define

define endxmltag
    </ [id] >
end define

% Main rule to get us started
function main
    % Need a generic null node to seed mark
    construct Null [id]  _
    deconstruct * [any] Null
        NullNode [any]
    % XML markup the whole program
    replace [program]
        P [program]
    by
        P [toXml NullNode]
end function
```

(a) Grammatical forms and main function of the generic XML parse tree markup program

The base grammar is that of the input language, in this case C++. The [xmlnode] definition is the root of a separate polymorphic grammar that allows markup of any nonterminal.

```
% Visit each node in the parse of the input
rule toXml SameNode [any]
    % Do not mark up already marked up node
    skipping [xmlnode]
    % Otherwise visit every parse tree node
    replace $ [any]
        Node [any]
    % Don't recursively re-visit a node
    deconstruct not Node
        SameNode
    by
        Node [tagWithType]
end rule

% Mark a parse tree node with its type in X
function tagWithType
    replace [any]
        Node [any]
    % Get the type of the node
    construct Type [id]
        _ [typeof Node]
    % Construct an XML markup of it, and
    % recursively visit its children
    construct Xml [xmlnode]
      <Type> Node [toXml Node] </Type>
    % Make it generic so we can replace it
    deconstruct * [any] Xml
        XmlNode [any]
    by
        XmlNode
end function
```

(b) Transformation rules of the parse tree markup program

The [toXml] rule visits every parse tree node, skipping those we have already marked. The [tagWithType] function constructs the XML markup of the node with its type name, and recursively invokes [toXml] to tag inner nodes.

Figure 3. Generic TXL program to mark up input source with its own AST as XML tags.

visit the subnodes of each node we mark up. The result is a simple, language-independent generic XML AST markup program that yields XML output isomorphic to FreeTXL's parse tree dump (although not as pretty). The difference is that now that we have control of the markup program, we can use it as a basis for exploring refinements of AST source markup.

5. Island Parsing

One approach to focussing AST markup can be at least partly addressed using Moonen's "island parsing" technique [10] to mark up with XML only those "islands" in the source that are relevant to the task at hand. This technique vastly reduces the size of the marked up source by avoiding markup of most of it. Using agile parsing, island parsing can be coded directly in TXL [Figure 4]. The main nonterminal of the base grammar (*[program]*) is overridden using a redefine to say that the input consists of the "islands", nonterminals of interest selected from the nonterminal set of the base grammar, and the "water", uninterpreted sequences of input text that lie between. TXL transformation and analysis rules can then be applied to only the islands, with the water being skipped.

In particular, if the desired output is simply XML-marked up islands with no markup elsewhere, we can simply add the island grammar overrides to our AST markup program, and then change the main rule to invoke *[toXml]* on islands only [Figure 4]. The result is output that is fully marked up in the islands but unmarked elsewhere.

Agile parsing allows each tool to use its own definition of a different island grammar appropriate to its task, all based on the same base grammar. Because the base grammar itself is never changed, no grammar maintenance problems are introduced by these variants, and there is no need to maintain a set of alternative but similar grammars.

Complex multi-level island and lake structures can easily be encoded by introducing other nonterminals from the base grammar as first alternatives for uninterpreted elements in the definitions for "water". Because first forms take precedence in the parse, these embedded islands will always be recognized by the parser rather than being discarded as "water".

It should be pointed out that in TXL there is no need to use island parsing to mark up islands. The same effect can be achieved using no grammar overrides at all simply by removing the island overrides and targeting the main rule shown in Figure 4 to *[expression]* rather than *[island]*. This technique has the added benefit that only contextually valid *[expression]*s will be marked up, rather than simply any sequence of input items that happens to look like one. Moreover, because TXL can parse valid input just as fast as it can slough lexemes, there is no efficiency penalty to doing things this way. However, this method ignores the robust parsing benefits of island parsing and thus may not be suitable in all cases.

```
% This line is the ONLY language dependenc
% in this program!
include "Cpp.Grm"

% Island grammar for C++ expressions only
redefine program
    [repeat island_or_water]
end redefine

define island_or_water
    [island] | [water]
end define

define island
    [expression]
end define

define water
    [token] | [key]
end define

% Main rule to get us started
rule main
    % Need a generic null node to seed mar
    construct Null [id]  _
    deconstruct * [any] Null
        NullNode [any]
    % XML markup all islands only
    skipping [xmlnode]
    replace $ [island]
        I [island]
    by
        I [toXml NullNode]
end rule
```

Figure 4. Modifications to the generic XML parse tree markup program to island parse expressions only.

The base grammar is still C++, but an agile parsing override changes the main nonterminal [program] to parse [expression] islands only, sloughing off everything else as raw lexemes. The main rule is then modified to target islands rather than the whole input. The rest of the TXL program remains unchanged.

6. Selective AST Markup

Given the heavyweight XML representation of full AST's, even island markup of AST's only reduces the problem, it does not solve it. McArthur [3] has suggested that this can be addressed using partial AST markup. By gaining control over marked up sections using the "unparsed" notion, McArthur's method yields XML markup that is significantly more focussed and lightweight. Ideally, we should have a partial markup that has only those AST nonterminals of interest to each particular task marked up with XML, for example simply *[expression]* but not *[factor], [term], [primary], [reference],* and so on that it derives, and not *[statement, [method], [class]* and so on that derive it. In this way our output can be minimally marked up to exactly suit the task at hand.

Selective markup of syntactic forms using agile parsing in TXL uses a technique similar to the island markup technique described in the previous section. Beginning with our original

```
% The interesting nonterminals for the tas
% this line goes at the beginning of the m
% rule to make it easy to change
export InterestingTypes [repeat id]
    'expression 'function_declarator

% Mark a parse tree node with its type in
% - but only if it's an interesting node
function tagWithType
    replace [any]
        Node [any]
    % Get the type of the node
    construct Type [id]
        _ [typeof Node]
    % Check it's one of our interesting on
    import InterestingTypes
    deconstruct * [id] InterestingTypes
        Type
    % Construct an XML markup of it, and
    % recursively visit its children
    construct Xml [xmlnode]
      <Type> Node [toXml Node] </Type>
    % Make it generic so we can replace it
    deconstruct * [any] Xml
        XmlNode [any]
    by
        XmlNode
end function
```

Figure 5. Modifications to the generic XML parse tree markup program to selectively mark up *[expression]* and *[function_declarator]* only.

The list of interesting nonterminals for the task is given by the global InterestingTypes list. The [toXml] markup function then simply checks that each node is in the interesting set before marking it up with XML.

---

generic markup program of Figure 3, first a set of nonterminal names to be marked is specified as a sequence of identifiers [Figure 5]. The XML markup rule *[toXML]* is then modified to look specifically for nodes whose type name matches one of these interesting nonterminal names. As each is found, it is marked with XML. The result is a version of the input with only those nonterminal nodes marked with XML tags [Figure 6].

This selective markup technique marks precisely those parts of the source we are interested in while completely avoiding the overhead of full XML markup. The result is a lightweight, efficient marked-up source highlighting only those AST nodes we are really interested in.

7. Refining Markup to Task Using Agile Parsing

However, using agile parsing we can do even better. Using grammar overrides, we can actually modify the grammar to use a different parse more convenient for our task. By exploiting TXL's ordered ambiguity resolution, we are free to add grammar overrides that specify very precisely the exact form of the items we are interested in for a particular task. For example, if we are interested only in method calls to the Java JDBC library, why mark up all method calls?

Figure 7 shows a version of the selective markup program in which the Java language base grammar has been modified to parse JDBC method calls in preference to general method calls. Because the overriding redefine gives the JDBC forms as the first alternative, the parser will always find these first and parse each particular method call as general method call only if it cannot be parsed as a JDBC call.

This is a very simplified demonstration of a much more powerful technique. By exploiting the ordered ambiguity of TXL's agile parser, we can focus the selective markup very precisely in this way. Any structure whose characteristics can be described using context free form can be selectively marked with XML using this method, without any modification to the language base grammar or the parser.

8. Semantically Refining Markup Using Transformation Rules

Of course, XML markup need not be the first or only thing that a TXL program does. By exploiting ambiguity in the grammar, we can also focus the markup by writing analysis rules that change the nonterminal type of a subtree based on its contextual or semantic properties. For example, if we want to identify only those JDBC calls that are embedded in a particular class or that reference a particular host variable, we can write a TXL rule to find such instances and change their nonterminal type, and then focus the XML selective markup rules to mark up exactly those instances based on the changed nonterminal type.

Figure 8 shows an example of this technique. In this case we are interested only in those JDBC calls that are conditional - that is, those that are guarded by an if or while statement. The grammar overrides in this case allow two separate and ambiguously identical alternatives for JDBC calls - *[jdbc_call]* and *[guarded_jdbc_call]*. Because *[jdbc_call]* is given as the first alternative, the parser will always parse any JDBC calls in the input source as *[jdbc_call],* and never as *[guarded_jdbc_call]*. However, because both alternatives are allowed, TXL will consider parse trees using either to be well-formed, allowing transformation rules to introduce *[guarded_jdbc_call]*s in place of *[jdbc_call]*s in the parse.

The rule *[identifyGuardedJdbcCalls]* does exactly that. Taking advantage of agile parsing to identify *[guarded_statement]*s, the rule changes the type of any *[jdbc_call]*s embedded in them to [guarded_jdbc_call]. Once this is done, the selective AST markup can be targeted to *[guarded_jdbc_call]* to introduce the XML markup of every guarded call to JDBC in the source output.

While this example is a simple and contrived contextual analysis, it does serve to demonstrate the general technique. The result of any source analysis that can be coded in TXL, no matter how complex, can be reflected into output source as a selective AST markup using this paradigm.

```
void <function_declarator>possible_command::build_argv()</function_declarator>
{
    int len=args.length();
    int argc=1;
    char*p=0;
    if(<expression>len>0</expression>){
        <expression>p=&args[<expression>0</expression>]</expression>;
        for(int i=0;<expression>i<len</expression>;<expression>i++</expression>)
            if(<expression>p[<expression>i</expression>]=='\0'</expression>)
                <expression>argc++</expression>;
    }
    <expression>argv=new char*[<expression>argc+1</expression>]</expression>;
    <expression>argv[<expression>0</expression>]=name</expression>;
    for(int i=1;<expression>i<argc</expression>;<expression>i++</expression>){
        <expression>argv[<expression>i</expression>]=p</expression>;
        <expression>p=strchr(p,'\0')+1</expression>;
    }
    <expression>argv[<expression>argc</expression>]=0</expression>;
}
```

Figure 6. Extract from example output of the selective AST markup program of Figure 5.

```
% Use the same markup program with Java
include "Java.Grm"

% Use parser to identify JDBC calls
% (simplified for demonstration purposes)
redefine method_call
       [jdbc_call]
    |  ...
end redefine

define jdbc_call
    [jdbc_name] [arguments]
end define

define jdbc_name
    'createStatement | 'prepareStatement
  | 'executeUpdate | 'executeQuery | 'getR
end define


% This time the only interesting things ar
% JDBC calls - this line goes in the main
export InterestingTypes [repeat id]
    'jdbc_call
```

Figure 7. Modifications to the selective generic XML parse tree markup program using agile parsing to identify and selectively mark up *[jdbc_call]* only.

The base grammar in this case is Java. The Java base grammar has been overridden to prefer parsing method calls as [jdbc_call] when the name of the called method is one of the standard JDBC operations. The selective [tagWithType] rule remains as in Figure 5 and the rest of the TXL program is unchanged.

9. Practicality, Performance and Scalability

Three questions of practicality arise concerning the techniques outlined in this paper. The first is the question of applicability to other languages. Although we have only used C++ and Java in the examples in this paper, the only dependence on source language is the base grammar and our use of it in analysis rules. The technique itself is language independent, and the TXL program of Figure 3 will work exactly as is for any other language simply by replacing the include statement for the base grammar with an include for the base grammar of any other language.

The second issue is the question of output size. If selective markup is to be a practical technique, the resulting marked up XML documents must be minimally larger than the original source. This measure is certainly met by the selective AST markup technique. The fully XML marked up AST of the 800 line standard open source *groff.cpp* program is 1.07 Mb, or 55 times larger than the original 19 kb source code. By island marking the AST's of only the expressions in the program as shown in Figure 4 we still get 307 kb or 15 times the original. However, by selectively marking up only the *[expression]* and *[function_declarator]* nonterminals as shown in Figure 5, the result is only 30.4 kb or 1.57 times the original source size.

The third and possibly most important issue is the question of efficiency and scalability. One of the reasons for the TXL maxim *"let the parser do the work"* is the speed of the TXL parser. On an 800 MHz PowerPC, the FreeTXL parser takes 0.16 seconds of CPU time to parse and pretty print the 800 line standard open source *groff.cpp* program, and 0.97 seconds to dump its 2.5 Mb XML parse tree. The programmed generic XML source-to-parse tree transform of Figure 3 takes 2.44 seconds to do the same thing using source transformation rules.

The island parsing version of Figure 4 takes 0.94 seconds to mark up expression islands only, or 1.27 seconds if expression islands are marked using the standard full parse instead. The generic selective markup program of Figure 5 takes 0.52 seconds to selectively mark up expressions and function declarators as shown in Figure 6.

Experiments show that performance of all of these programs is linear in the length of the input source. For example, the selective markup that took 0.52 seconds to process the 800 lines of *groff.cpp* takes 5.39 seconds to mark up a similar 8,000 line C++ program that is 10 times larger, and 55.09 seconds to process an 80,000 line C++ program that is 100 times larger.

10. Conclusion

Structural markup of source code using XML is a powerful and permanent representation with a wide range of uses in software analysis systems. While full markup of AST nodes in source is useful, it is too large and cumbersome to be practical and efficient for large scale tasks. McArthur [3] has demonstrated that partial markup may be the solution to these limitations.

In this paper we have shown how we can generalize and focus the idea of partial markup using agile parsing. By making possible selective markup of only the nonterminals of interest,

```
% Use the same markup program with Java
include "Java.Grm"

% Use parser to identify JDBC calls
% (simplified for demonstration purposes)
redefine method_call
       [jdbc_call]
    |  [guarded_jdbc_call]
    |  ...
end redefine

define jdbc_call
    [jdbc_name] [arguments]
end define

define guarded_jdbc_call
    [jdbc_name] [arguments]
end define

define jdbc_name
    'createStatement | 'prepareStatement
  | 'executeUpdate | 'executeQuery | 'getR
end define

% Use parser to identify guarded statement
redefine if_statement
    'if '( [expression] ')
        [guarded_statement]
    [opt else_clause]
end redefine

redefine else_clause
    'else
        [guarded_statement]
end redefine

redefine while_statement
    'while '( [expression] ')
        [guarded_statement]
end redefine

define guarded_statement
    [statement]
end define

% This time interesting things are guarded
% JDBC calls - this line goes in the main
export InterestingTypes [repeat id]
    'jdbc_call

% Visit each guarded statement
rule identifyGuardedJdbcCalls
    replace $ [guarded_statement]
      GS [guarded_statement]
    by
        GS [retypeJdbcCalls]
end rule

% Retype all embedded [jdbc_call] nodes
% to [guarded_jdbc_call]
rule retypeJdbcCalls
    replace [method_call]
        JDBC [jdbc_call]
    deconstruct JDBC
        Name [jdbc_name] Args [arguments]
    construct GJDBC [guarded_jdbc_call]
        Name Args
    by
        GJDBC
end rule
```

Figure 8. Modifications to the selective generic XML parse tree markup program using agile parsing to semantically selectively mark up *[guarded_jdbc_call]*s only.

The Java base grammar has been overridden to prefer parsing method calls as [jdbc_call] when the name of the called method is one of the standard JDBC operations. The ambiguously identical [guarded_jdbc_call] type is used as an indicator for the result of the analysis done by the [identifyGuardedJdbcCalls] rule. Identification of guarded contexts is done using agile parsing to override the definitions of [if_statement], [else_clause] and [while_statement] to use [guarded_statement] in place of [statement]. To complete the program, the selective [tagWithType] rule remains as in Figure 5, the main rule is modified to invoke [identifyGuardedJdbcCalls] before [toXml], and the rest of the TXL program is unchanged.

agile parsing allows for precise refinement of XML markup to the task, without the need for any redundant tags. By exploiting ambiguity, we can use transformation rules to introduce semantics into the markup, allowing for a lightweight but effective XML source representation of the results of complex and interesting software analyses. Measurements show that the technique is efficient, practical and scalable to industrially-sized source programs.

In this paper we have intentionally elided a number of minor technical details in order to keep examples small and maintain focus. For example, the output markup shown in Figure 6 is technically invalid XML because we have not shown the rules to perform the lexical translation of < and > to *<* and *>* which must be a part of any real XML markup transformation. We have also not shown the necessary formatting cues for pretty printing of the XML output. These details do not affect the applicability or performance of the techniques described.

References.

[1] J.R. Cordy, K.A. Schneider, T.R. Dean and A.J. Malton, "HSML: Design Directed Source Code Hot Spots", *Proc. IWPC 2001 - 9th Int. Workshop on Program Comprehension*, Toronto, Canada (May 2001), pp. 145-154.

[2] J.F. Power and B.A. Malloy, "Program Annotation in XML: a Parse-tree Based Approach", *Proc. WCRE 2002 - 9th Working Conference on Reverse Engineering,* Richmond, Virginia (October 2002), pp. 190-198.

[3] G. McArthur, J. Mylopoulos and S.K.K. Ng, "An Extensible Tool for Source Code Representation Using XML", *Proc. WCRE 2002 - 9th Working Conference on Reverse Engineering,* Richmond, Virginia (October 2002), pp. 199-208..

[4] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0", *http://www.w3.org/TR/1998/REC-xml-19980210.pdf* (February 1998).

[5] T.R. Dean, J.R. Cordy, A.J. Malton and K.A. Schneider, "Grammar Programming in TXL", *Proc. SCAM'02 - IEEE 2nd International Workshop on Source Code Analysis and Manipulation,* Montréal (October 2002), pp. 93-102.

[6] T.R. Dean, J.R. Cordy, A.J. Malton and K.A. Schneider, "Agile Parsing in TXL", *submitted to J. Automated Software Engineering Special Issue on Source Code Analysis and Manipulation,* expected 2003.

[7] J.R. Cordy, C.D. Halpern and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects", *Computer Languages* 16,1 (January 1991), pp. 97-107.

[8] J.R. Cordy, T.R. Dean, A.J. Malton and K.A. Schneider, "Software Engineering by Source Transformation - Experience with TXL", *Proc. SCAM'01 - IEEE 1st International Workshop on Source Code Analysis and Manipulation*, Florence (November 2001), pp. 168-178.

[9] TXL Project, "The TXL Programming Language, Version 10.2", *http://www.txl.ca/docs/TXL102LangRef.pdf* (April 2002).

[10] Leon Moonen, "Generating Robust Parsers Using Island Grammars", *Proc. 8th Working Conference on Reverse Engineering*, Stuttgart (October 2001), pp. 13–22.