Proc. IWPC 2001 - IEEE 9th International Workshop on Program Comprehension, Toronto, May 2001, pp. 145-154

HSML: Design Directed Source Code Hot Spots

James R. Cordy* Kevin A. Schneider† Thomas R. Dean* Andrew J. Malton*
Legasys Corporation, Kingston, Ontario, Canada
{cordy,kas,dean,malton}@cs.queensu.ca

Abstract

HSML, the Hot Spot Markup Language, is an ultra-high level executable specification language designed for concisely specifying source code hot spots of all kinds. Each HSML rule specifies the abstract syntactic class of the items to be marked as hot using a nonterminal of the target language grammar, and the semantic conditions under which such items are to be marked using an algebraic expression on the design properties of the item. Conditions can include restrictions on abstract syntactic structure (patterns), design recovered semantic properties (queries on the design database), and semantic properties induced by other markup rules. HSML has been used in industrial practice to specify source code hot spots for the Year 2000 and a wide range of other application maintenance tasks on systems implemented in Cobol, PL/I and RPG. In this paper we introduce the basic concepts of HSML and demonstrate its use in real software maintenance tasks.

1. Background

Design recovery [1], the reverse engineering of a design database or graph from source code artifacts, is now a common and accepted technique in program comprehension. Systems such as Rigi [2], the Software Bookshelf [3], Kontogiannis [4], Cremer [5] and many others now routinely extract a design graph from source code files. Design analysis [6] explores the recovered design database to discover properties such as architectural well-formedness [7]. Such analyses are often reported as new relationships or modified design graphs using tools such as PROGRES [8] and GROK [9].

While analyses reported as high-level graphs, tables or diagrams are often well understood by application architects and project leaders, the activities required in response to such discoveries are invariably involved with changes to the actual source code, to be carried out by line programmers. Thus it would seem desirable to use a representation of the results of an analysis that has a direct attachment to actual lines of source.

This important fact has previously been recognized in the area of performance tuning. Performance hot spots [10] are small sections of source code that are labeled as "hot" because of a high level of execution time or memory activity that is observed for them in an execution profile. Represented as a markup or elision [11] of the source code of the application, hot spots focus the tuning programmer's effort on exactly those sections of code that actually affect performance.

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).
** Author's current address: Department of Computing & Information Science, Queen's University, Kingston, Ontario, Canada K7L 3N6.*
† Author's current address: Department of Computer Science, University of Saskatchewan, 57 Campus Drive, Saskatoon, Saskatchewan, Canada S7N 5A9.

2. Maintenance Hot Spots

Maintenance hot spots are a generalization of performance hot spots to any kind of design or source code analysis activity. Sections of source code are labeled as hot because a design or source code analysis looking for sensitivity to a particular maintenance issue, such as the Year 2000 problem, the expansion of credit card account numbers, or a change to interest computation laws, has identified them as potentially relevant.

By representing the results of such analyses as source code elisions [11], we make the results of the analysis accessible to all members of the programming team. This has many advantages: it focuses the effort of the line programmers' maintenance activity on exactly those sections of code which may be affected; it assists managers by providing a checklist of code sections to be examined and modified; and it provides a test strategy by explicitly enumerating the code sections that need to be covered.

LS/2000 [12] used the concept of maintenance hot spots to assist in the Year 2000 conversion of over three billion lines of Cobol, PL/I and RPG source code. Using a general design recovery process followed by custom design analysis and hot spotting processes for the Year 2000 problem, LS/2000 produced hot spot reports for every module of an application that had any potential Year 2000 risks embedded in it. Figure 1 shows an example LS/2000 hot spot report for one module of a 1,000 module Cobol application.

Clients of LS/2000 reported a 30-40 fold increase in Year 2000 conversion productivity with use of hot spot reports. Time to examine and convert a source code module of a few thousand lines of source was reduced from a few hours to less than five minutes, and accuracy of conversion before testing was increased from about 75% to over 99%.

3. HSML

HSML, the Hot Spot Markup Language, is an ultra-high level executable specification language designed for concisely specifying source code hot spots of all kinds. An HSML run takes as input an HSML rule set, a normalized source code module to be hot spotted and the design database for the

```
Program: XYEGPROG

Line  Program Source Line                                                  HS Src File
----  -------------------                                                  -- --------
12    001200      16  JULIAN-DATE.                                            XXCOPYJL
13    001300          20  JULIAN-YR  PIC  9(2).                               XXCOPYJL
14    001400          20  JULIAN-DAY PIC  9(3).                               XXCOPYJL

15    001500              24  FISCAL-MMDDYY.                                  XXCOPYFS
16    001510* MONTH/DAY MAY BE USED AS GROUP OR SEPARATELY                    XXCOPYFS
17    001600                  28  FISCAL-DATE.                                XXCOPYFS
18    001700                      32  FISCAL-MO    PIC 9(2).                  XXCOPYFS
19    001800                      32  FISCAL-DAY   PIC 9(2).                  XXCOPYFS
20    001900                  28  FISCAL-YR        PIC 9(2).                  XXCOPYFS

26    002600      16  FISCAL-DATE-JULIAN        PIC S9(5) COMP-3.             XXCOPYDJ

52    005300              24  WS-FISCAL-DATE-JULIAN     PIC S9(5) COMP-3.     XYEGPROG

236                IF FISCAL-DATE-JULIAN IS NOT GREATER THAN               <- XYEGPROG
237                         WS-FISCAL-DATE-JULIAN                          <- XYEGPROG
240                   PERFORM FISCAL-DATE-LESS.                               XYEGPROG

28    003000* WHEN WE HIT THE SENTINEL, IT'S TIME TO LEAVE                    XXCOPYPG
29    003100    IF  FISCAL-PROC-FLD IS EQUAL TO ZEROS                         XXCOPYPG
30    003110          AND FISCAL-MMDDYY IS EQUAL TO ZEROS                  <- XXCOPYPG
31    003500       PERFORM FISCAL-ZEXIT                                       XXCOPYPG
32    003600* OTHERWISE IT'S NEXT YEAR NOW                                    XXCOPYPG
33    003700    ELSE                                                          XXCOPYPG
34    003800       ADD 1 TO JULIAN-YR                                      <- XXCOPYPG
35    003820       PERFORM FISCAL-PROCESS.                                    XXCOPYPG
```

Figure 1. Example LS/2000 Year 2000 Hot Spot Report.

Numbers on the left are actual source code line numbers within the source files involved. The source file each line is from appears on the right. Year 2000 risks (hot spots) are marked with an arrow <- on the right. Other lines are context, such as the declarations of variables mentioned in hot spots, that allow the report to be understood independently of the rest of the source.

application. The output of the run is the normalized source code module with hot spots marked up using XML-like markup brackets, and (optionally) a set of new relationships for the marked up entities to be added to the design database.

The input source code has been normalized by stripping comments and other lexical noise, inlining include files, expanding macros, and disambiguating all names by global unique naming of all program and data entities. The input design database is the result of a design recovery and analysis process on the normalized source code of the entire application of which the module is a part. Unique naming forms the link between the items declared and referred to in the source and the design relationships involving them in the recovered design database. Because the entire design database is available when marking up each individual source module, relationships that transcend module boundaries can be used in specifying markup criteria.

HSML is normally run in the context of the LS/AMT architecture (Figure 2), which provides source normalization, unique naming, design recovery, version integration and reporting.

3.1. HSML Rules

Each HSML rule specifies the class of items to be marked as hot, specified as a nonterminal of the source language's abstract syntax, and the conditions under which such items are to be marked, specified using an algebraic expression on the design properties of the item and the entities contained within it. Conditions can include restrictions on abstract syntactic structure (patterns), design recovered semantic properties (queries on the design database or design graph), and semantic properties induced by other markup rules.

HSML rules take the form:

```
HOTSPOT_NAME = [nonterminal_name] constraints  ;
```

where *HOTSPOT_NAME* is an identifier that names the particular kind of hot spot, *nonterminal_name* is a nonterminal of the target language's abstract syntax that identifies the kind of thing to be marked as hot, and the optional *constraints* specify the conditions under which those nonterminals should be marked as hot.

The simplest HSML rules simply mark items of a particular abstract syntactic class. Example:

```
% Mark up all CICS statements in Cobol programs
CICS_STMT = [cics_statement];
```

The identifier to the left of the equals sign is the name of the markup, in this case CICS_STMT, which will be used in the markup brackets and as the name of the associated markup relationship. Nonterminals of the abstract syntax, in this case

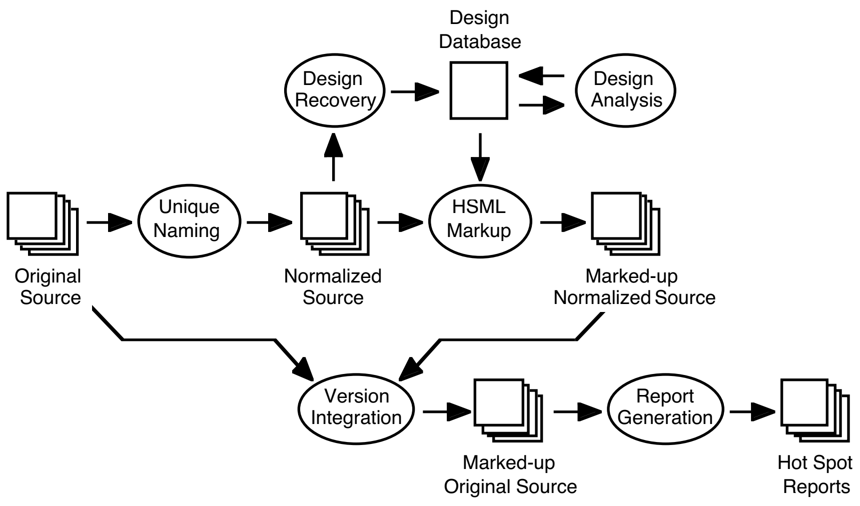

Figure 2. The LS/AMT Process Architecture.

HSML normally runs in the LS/AMT process environment. LS/AMT provides Unique Naming, which disambiguates references to similarly named items; Design Recovery and Analysis, which builds and augments the design database with recovered, inferred or externally documented design information; Version Integration, which reintegrates original lexical information such as comments and formatting and into the marked-up source; and Reporting, which creates and displays hot spot reports in web and printed form.

[cics_statement], are always referred to using square brackets, as in TXL [13].

When run on a normalized source module, this rule will mark as hot all uses of Cobol CICS statements, yielding a new normalized source with embedded markup brackets:

```
          ...
          MOVE X TO Y.
{CICS_STMT
          EXEC CICS.
             READ ...
             ...
          END-EXEC.
}CICS_STMT
          ...
```

The output of the markup will then be fed to the LS/AMT version integrator to reflect the markup into the original raw source. The LS/AMT reporter then elides unmarked source to produce the hot spot reports.

3.2. HSML Constraints

Constraints specify the conditions under which items are to be marked as hot. A constraint consists of a *property operator* and a *property*. Property operators are < ("which contains"), * ("which has a first") and > ("which is contained in"). Properties can be structural properties, pattern properties, design properties or markup properties.

Examples.

< [array_reference]	*constrains to only those items that contain an array reference*
* Date	*constrains to only those items whose first contained entity (i.e.,unique name) has a Date fact in the design database*

Constraints can be combined using negation (not), conjunction (and) and disjunction (or).

! constraint	*constrains to items that do not meet the given constraint*
(constraint , constraint , ...)	*constrains to items that meet every one of the given constraints*
(constraint | constraint | ...)	*constrains to items that meet one or more of the given constraints*

3.3. Structural Properties

Structural properties describe constraints on the syntactic structure of or contained in the items to be marked as hot. Syntactic properties are expressed using the nonterminals of the abstract syntax.

Example 1. Identify as hot all IF statements containing a nested if statement. The rule can be read as "mark as NESTED_IF all [if_statement]'s that are properly contained in (>) another [if_statement]."

```
% Mark all nested if statements as hot
NESTED_IF = [if_statement] > [if_statement];
```

Example 2. Identify as hot all comparisons to a literal value. The rule can be read as "mark as LIT_COMPARE all [comparison]'s that properly contain (<) a [comparand] which is entirely (*) a [literal]."

```
% Mark all comparisons to literal values as hot
LIT_COMPARE = [comparison] < [comparand] * [literal];
```

3.4. Pattern Properties

Pattern properties describe constraints on the textual representation of items to be marked as hot. Patterns are text strings or regular expressions that the text of the item must match or contain.

grep ("regexp")	*constrains to items whose text matches (*) or contains (<) a match for the given regular expression*
grepid ("regexp")	*constrains to items whose first identifier (*) or which contain any identifier ( ) whose original text matches the given regular expression*
pattern ("text")	*constrains to items whose text has the exact same parsed structure (*) as the given text*

Example 1. Identify as hot all Cobol declarations of data fields at level 05. The rule can be read as "mark as LEVEL_5 all [declaration]'s containing a [level_number] that exactly matches (*) the pattern '5' or '05'."

```
% Mark all declarations at level 5
LEVEL_5 =
     [declaration] < [level_number] * pattern ("5" l"05" );
```

Example 2. Identify as hot all declarations of items whose name begins or ends with the letters "YY". The rule can be read as "mark as YY_DECL all [declaration]'s whose first (i.e., declared) [name] has an identifier that begins or ends with 'YY'."

```
% Mark all YY declarations
YY_DECL =
     [declaration] * [name] * grepid  ("^YY" l"YY$" );
```

3.5. Design Properties

Design properties constrain the markup by a query on the design database using the first (*) or any contained (<) unique [name] of an item as the key. Design properties are denoted by the name of a design relationship (i.e., fact or edge). By convention design fact names begin with a capital letter.

Factname	*constrains to items for which there is a Factname fact in the design database*
Factname (attr)	*constrains to items for which there is a Factname fact with attribute attr in the design database*

Design relationships include facts design recovered from source, such as *Calls*, *Uses*, *Contains* and *FieldSize* facts, as well as relationships derived from business type analysis or other information, such as *Money*, *Date* and *AccountNumber* facts. Relationships may be unary, binary or n-ary. For binary and n-ary relationships, the first entity [name] is normally the primary query key.

Example 1. Categorize statements as potentially dangerous or harmless based on whether they reference any [name] that has a *Date* fact in the design database.

```
% Mark statements that reference any name with a
% Date fact as DANGEROUS_STMT
DANGEROUS_STMT = [statement] < Date;

% Mark statements that don't reference any name with
% a  Date fact as HARMLESS_STMT
HARMLESS_STMT = [statement] !< Date;
```

Example 2. Find declarations of money variables, and all expressions involving money.

```
% Mark money declarations
MONEY_DECLARATION = [declaration] * Money;

% Mark all expressions involving any money variables
MONEY_EXPRESSION = [expression] < Money;
```

Example 3. Mark up all birth date variable declarations. The rule can be read as "mark as BIRTHDATE the [declaration]s of all [name]s with a *Date* fact whose identifier contains the substring 'BRTH' or 'BIRTH'."

```
% Mark birth dates
BIRTHDATE = [declaration]
     * [name] * Date * grepid ("BRTH" l "BIRTH") ;
```

3.6. Markup Properties

Markup properties constrain new items to be marked up to be those already marked up (*) or nested within the scope of items marked up (>) with a given markup tag. Markup properties are denoted by *hasmarkup.*

hasmarkup (MARKUPNAME)
*constrains to items already marked up (*) or contained within something marked up (<) as MARKUPNAME*

Example 1. Mark all arithmetic operators in money expressions. This rule assumes that the MONEY_EXPRESSION rule has preceded it. The rule can be read as "mark up as MONEY_OP all [arithmetic_operator]s appearing in MONEY_EXPRESSION markups."

```
% Mark up all arithmetic operators in
% money computations
MONEY_OP = [arithmetic_operator]
      > hasmarkup (MONEY_EXPRESSION);
```

Example 2. Identify the entire record declaration that embeds variables previously identified as money declarations. The rule can be read as "mark up as RECORD_CONTEXT every [declaration] that contains a [declaration] already marked as MONEY_DECLARATION."

```
% Mark all declarations that contain a sub-declaration
% with a MONEY_DECLARATION markup
RECORD_CONTEXT = [declaration] < [declaration]
      * hasmarkup (MONEY_DECLARATION) ;
```

3.7. Induced Relationships

Each markup induces new facts about all of the entities ([name]s) contained in the markup which can be queried in later markups using the *markup* property. The markup property constrains the rule to items whose first (*) or which contains any (<) [name] that already appears in the given markup somewhere.

markup — *constrains to items whose name appears in a markup somewhere in the result*

markup (MARKUPNAME) — *constrains to items whose name appears inside a MARKUPNAME markup somewhere in the result*

Example. Find the declarations of all items appearing inside any markup we have made. We can read the rule as "mark as INTERESTING_DECLARATION any declaration whose first (i.e. declared) [name] appears in a markup somewhere." Of course, this rule assumes that one or more other markup rules precede it!

```
% Mark up the declarations of all entities mentioned in
% any other markup
INTERESTING_DECLARATION =
                              [declaration] * markup;
```

Induced relationships can be exported from an HSML run as new design relationships added to the design database. In this way the results of HSML runs can be stored in the design database for use in subsequent markups or other design analysis.

3.8. Clustering

Cluster properties allow closure of items to be marked across binary or n-ary design relationships such as *Move* (X,Y) (i.e., X is assigned to Y) or *Compare* (X,Y) (i.e., X is compared to Y).

cluster (Factname, MARKUPNAME) — *constrains to items related by a (chain of) Factname relationships to an item that appears in a MARKUPNAME markup*

cluster (Factname) — *constrains to items that are related by a (chain of) Factname relationships to an item that appears inside any markup*

Example. Mark all variables with FlightNumber facts and all of the variables that they transitively interact with. The second rule can be read as "mark as FLIGHT_NUMBER_CLUSTER all [name]s which are related to any [name] appearing inside a FLIGHT_NUMBER by a *Move* or *Compare* relationship."

```
% Mark all flight numbers
FLIGHT_NUMBER = [name] *  FlightNumber;

% And all related variables
FLIGHT_NUMBER_CLUSTER = [name]
      * cluster (Move | Compare, FLIGHT_NUMBER);
```

3.9. Transitive Closure

Markup rules can be automatically recursively reapplied to transitively close a cluster. Transitive closure is denoted by *= in place of = in the rule specification.

Example. Transitively close the flight number cluster shown above. The following rule is run after the rules in the previous example. The form *= means that the rule will be automatically re-run on its own result after each application of the rule, until no new candidates for markup are found.

```
% Transitively close the flight number cluster
FLIGHT_NUMBER_CLUSTER *=  [name]
      * cluster (Move | Compare,
                  FLIGHT_NUMBER_CLUSTER);
```

4. Examples

4.1. LS/2000

Because HSML grew out of our experience with the Year 2000 problem, an obvious first test was its ability to compactly express the Year 2000 hot spot markup rules of the LS/2000 system. In LS/2000, markup is implemented by a custom markup program consisting of about five thousand lines of TXL code. Figure 3 shows the HSML specification of LS/2000, which uses only 16 rules.

In order to validate that the HSML version was indeed expressing the same markups, a regression test suite consisting of a Cobol application of about 3 million source lines, a PL/I application of about 300,000 lines and an RPG application of about 100,000 lines was assembled. For each application, a set of hot spot reports was generated using the original LS/2000 system and then a new set was generated by running the HSML specification on the same inputs. The total time for each run was

```
% HSML Spec for LS/2000 Year 2000 Hot Spots
% J.R. Cordy, Legasys Corporation, October 1998

% Keys containing a date are hot.
HOT_KEY = [key_identifier] < Date ;

% Date fields with literal values are hot.  This captures 88 values as well as initial values.
HOT_FIELD = [declaration] * Date < [value_clause] < [literal] * interesting ;

% Files with a hot key are themselves hot.
HOT_FILE = [file_declaration] < hasmarkup (HOT_KEY) ;

% Certain kinds of date fields with Z pictures are hot.
HOT_PICTURE_FIELD = [declaration]
    * Date ("YY","YYMM","YYMMDD","YYNNN", "FYY","FYYMM","FYYMMDD","FYYNNN")
    * Pic ("ZZ", "ZZZ", "ZZZZ", "ZZZZZ", "ZZZZZZ", "ZZZZZZZ") ;

% Arithmetic statements with dates are hot.  Subsumes COMPUTE, ADD, SUBTRACT, etc.
HOT_ARITHMETIC = [arithmetic_statement] * Date ;

% Date inequalities are hot.  Exactly the same as LS/2000 - we could be more precise if we want.
DATE_INEQUALITY = [comparison] * Date < [inequality] ;

% Literal comparisons to dates are hot.  The "interesting" property screens out boring literals.
COMPARE_HOT_LITERAL = [comparison] * Date < [literal] * interesting ;

% Literal moves to dates are hot.
MOVE_LITERAL_DATE = [move_statement] * Date < [literal] * interesting ;

% INSPECT, STRING and UNSTRING statements on dates are hot.
INSPECT_DATE = [stringop_statement] * Date ;

% File descriptions of hot files are hot.
HOT_FILE = [file_description_entry] < hasmarkup (HOT_FILE) ;

% Nontrivial arithmetic expressions involving dates are hot.
HOT_ARITHMETIC_EXPRESSION =
    [arithmetic_expression] ( < [arithmetic_operator], < [arithmetic_primary] * Date ) ;

% Subscript expressions involving dates are hot.
HOT_SUBSCRIPT = [subscript_expression] < Date ;

% Declaration context - slightly more precise than LS/2000 - marks innermost group only.
HOT_USED_FIELD = [declaration] * Date * markup ;
HOT_DATA_CONTEXT = [declaration] * Date < [declaration] * hasmarkup ;

% Statement context - slightly more precise than LS/2000 - marks innermost context only.
HOT_STATEMENT_CONTEXT = [statement] < [comparison] * hasmarkup ;
HOT_STATEMENT_CONTEXT = [statement] < hasmarkup (HOT_KEY) ;
```

Figure 3. HSML Specification of LS/2000 Year 2000 Risk Analysis for Cobol.

measured and the two sets of hot spot reports were compared for differences.

Although we expected that it might be somewhat less efficient than the custom programmed markup engine used in LS/2000, the HSML version was actually measured to be about 10% faster on average. The generated hot spot reports were virtually identical, with two exceptions. The HSML specification of Figure 3 turned out to be slightly more aggressive in its hot spotting of interactions of literal values with dates, leading to a one or two extra hot spots not previously identified by LS/2000. The HSML specification also uncovered a small bug in the Cobol version of LS/2000 which had missed three uses of dates as sort keys in the regression set due to an apparent typographical error in its search patterns. Given the relative sizes of the two hot spotting specifications (16 lines of HSML vs. a 5,000 line TXL program), it is not surprising that such an error might more easily slip by in the latter. Of course, the HSML specification was also written with a more mature understanding of the problem, so one might expect it to be more accurate in any case.

4.2. Error Handling Analysis

One of the first real tests of HSML was a problem posed by a client with an application consisting of about a million lines of PL/I code. In this case the application was known to be unstable

```
% HSML Spec for Error Condition Backtracing in PL/I
% J.R. Cordy, Legasys Corporation, July 1999

% This general hot spot markup specification hot spots all references to a set of interesting things.
% The conditions that guard interesting things, the statements that cause these conditions, the
% procedures containing those statements, and the calls to them are also hot spotted. The effect is to
% highlight all interesting things and the conditions that directly or indirectly affect them.

% PART I - What's interesting?

% Interesting things - this time, they are things whose names contain ABEND, ABND, ERR
INTERESTING = [name] * grep ("ERR" | "ABND" | "ABEND") ;

% Statements and declarations containing interesting things
INTERESTING_STATEMENT = [statement] < hasmarkup (INTERESTING) ;

% PART II - Conditions that guard interesting things

% Conditions under which interesting things are executed
IF_CONTEXT = [if_statement] * hasmarkup (INTERESTING_STATEMENT) ;
IF_CONTEXT_CONDITION = [if_condition] > hasmarkup (IF_CONTEXT) ;

% Assignments to variables of those conditions
IF_CONTEXT_CONDITION_ASSIGNMENT = [assignment_statement] * markup (IF_CONTEXT_CONDITION);
IF_CONTEXT_CONDITION_ASSIGNMENT = [call_statement] * markup (IF_CONTEXT_CONDITION) ;

% Conditions under which those assignments are made
IF_CONTEXT_CONDITION_ASSIGNMENT_IF_CONTEXT =
   [if_statement] * hasmarkup (IF_CONTEXT_CONDITION_ASSIGNMENT) ;

% PART III - Calls to routines with interesting things, and conditions under which those calls are made

% Routines that are interesting or that contain interesting things
PROC_CONTEXT = [procedure_declaration]
    ( !* grep ("(MAIN)"), ( * hasmarkup (INTERESTING) | < hasmarkup (INTERESTING_STATEMENT))) ;
PROC_CONTEXT_NAME = [label] > hasmarkup (PROC_CONTEXT) ;

% Calls to those routines
PROC_CONTEXT_CALL = [call_statement] < [name] * markup (PROC_CONTEXT_NAME) ;

% Conditions under which those routines are called
PROC_CONTEXT_CALL_IF_CONTEXT = [if_statement] * hasmarkup (PROC_CONTEXT_CALL) ;
PROC_CONTEXT_CALL_IF_CONTEXT_CONDITION =
   [if_condition] > hasmarkup (PROC_CONTEXT_CALL_IF_CONTEXT) ;

% Assignments to variables of those conditions
PROC_CONTEXT_CALL_IF_CONTEXT_CONDITION_ASSIGNMENT =
    [assignment_statement] * markup (PROC_CONTEXT_CALL_IF_CONTEXT_CONDITION) ;

% Conditions under which those assignments are made
PROC_CONTEXT_CALL_IF_CONTEXT_CONDITION_ASSIGNMENT_IF_CONTEXT =
   [if_statement] * hasmarkup (PROC_CONTEXT_CALL_IF_CONTEXT_CONDITION_ASSIGNMENT) ;
```

Figure 4. HSML Specification of Error Backtrace Analysis for PL/I.

in the presence of erroneous input, but the programmers were finding it very difficult to determine the causes of the instability because the code used a programming style that deferred all error reporting to the end of a run and did not distinguish between different classes of errors.

Figure 4 shows the HSML specification written to attack this problem. It was known that most variables involved with error handling used a predictable naming convention, involving names containing the substrings ERR, ABND and ABEND. This was used as the "seed" of the specification. The specification then hot spots all statements and declarations that use these seed names.

Part 2 of the specification highlights all IF statements that guard any of the interesting statements, and identifies the condition expressions of these IFs. It then goes on to mark all assignments to variables that are used in these conditions, effectively identifying the ways in which these conditions can be caused.

Finally, part 3 of the specification identifies any internal routines that enclose these IF statements, and goes on to highlight all calls to these routines, the IFs and conditions that guard these calls, and finally any assignments that change the variables used in these second order conditions. The result is a hot spot report that effectively traces the conditions under which any error condition or abend (i.e., exception) can be raised.

This specification was authored and run in about two days, demonstrating how rapidly new problems can be attacked using HSML. It's interesting to note that this HSML specification does not use any design facts at all - in a sense, it does its own design recovery from scratch, using the design information uncovered by previous rules to drive the hot spots of later rules.

4.3. Other Applications

HSML has also been used in projects to find and normalize rollover code generated by different Y2K tools in the same source code, to find and trace the flow of flight numbers through the source code of an airline management system, to trace back through complex computations the data entries stored in an archival data warehouse in order to validate the archived data, to identify the external interfaces and types of transactions in complex interactive systems, and for several other design-directed source code analysis tasks.

5. Implementation Issues

Implementation of HSML poses many challenges. Because it requires access to abstract syntactic structure, it seems appropriate to implement HSML in a language like TXL [13] that already works with parse trees. However, HSML also requires access to the design database, and TXL's symbolic nature tends to make database access awkward and inefficient. Moreover, the obvious strategy of translating HSML rules to TXL programs would require the TXL compiler to be distributed with HSML.

In the end HSML was implemented as a generic interpreter for the HSML notation written in TXL. In order to allow database access, a new TXL database module was designed that encodes Entity-Relationship databases as AVL trees to allow reasonably efficient queries in a natural way. Because TXL is

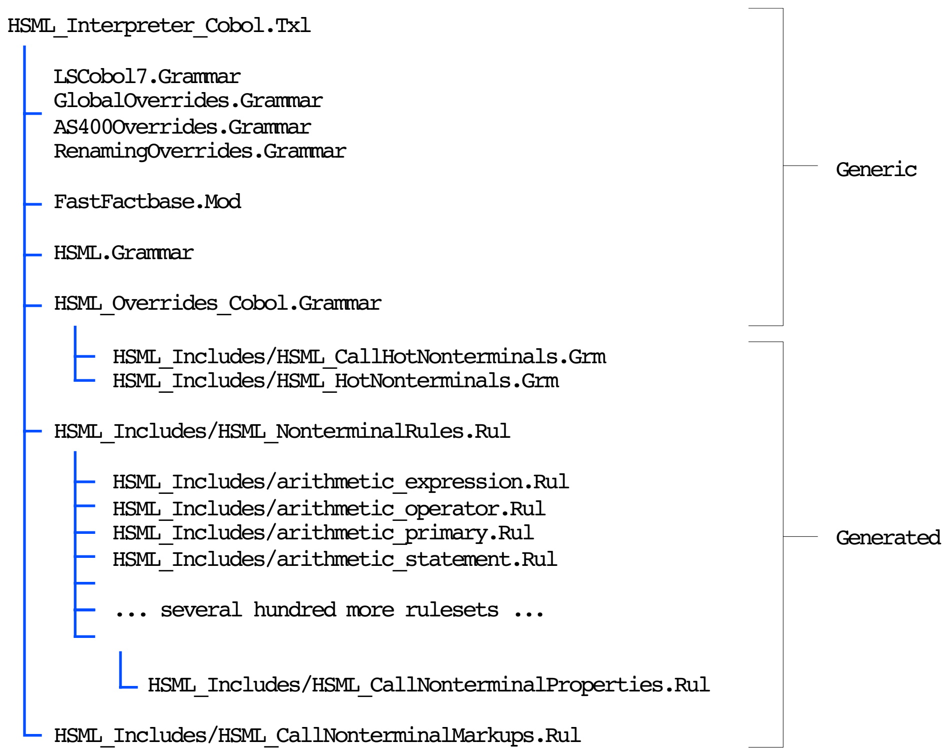
HSML_Interpreter_Cobol.Txl
LSCobol7.Grammar
GlobalOverrides.Grammar
AS400Overrides.Grammar
RenamingOverrides.Grammar
FastFactbase.Mod
HSML.Grammar
HSML_Overrides_Cobol.Grammar
HSML_Includes/HSML_CallHotNonterminals.Grm
HSML_Includes/HSML_HotNonterminals.Grm
HSML_Includes/HSML_NonterminalRules.Rul
HSML_Includes/arithmetic_expression.Rul
HSML_Includes/arithmetic_operator.Rul
HSML_Includes/arithmetic_primary.Rul
HSML_Includes/arithmetic_statement.Rul
... several hundred more rulesets ...
HSML_Includes/HSML_CallNonterminalProperties.Rul
HSML_Includes/HSML_CallNonterminalMarkups.Rul
Generic
Generated

Figure 5. Implementation of an HSML interpreter using TXL.

A generic HSML interpreter consisting of the target language grammar (LSCobol7.Grammar etc.), the database interface (FastFactbase.Mod), the HSML grammar (HSML.Grammar) and a skeletal rule decoder (HSML_Cobol_Interpreter.Txl) is augmented with a template-generated set of TXL rules for each nonterminal of the target language abstract syntax. The TXL compiler is then used to make a standalone HSML interpreter for the target language that is ready to mark up any nonterminal.

statically typed, it was necessary to generate a separate set of TXL functions and rules to implement markup for each nonterminal in the grammar of each target language (Figure 5). Since the Cobol reference grammar involves more than 800 nonterminals and the grammars of other languages are similarly large, this implementation strategy resulted in truly enormous TXL programs with thousands of TXL functions and rules, each of which is "speculative" in that it may or may not ever be used in the HSML specifications we actually run.

To our surprise this implementation method has proven not only practical but reasonably efficient as well. Since TXL rules and functions are guarded by pattern matches, and since the set of nonterminals actually mentioned in each HSML specification can be enumerated as it is run, we can arrange that the speculative TXL rules and functions fail quickly if they are unused in a particular run. With care, we can arrange that this failure be limited to a single integer comparison in a guarding TXL pattern.

HSML could be similarly implemented using other grammar-based source code analysis and manipulation tools such as Gentle [14] and NewYacc [15]. However, there are advantages to using TXL in place of Yacc-based tools that affect the usability and generality of HSML. Since TXL supports general context free grammars and does not impose LL, LR or LALR restrictions, the grammar used for each target language can be the user-level reference syntax for the language rather than a compiler-oriented "implementation" grammar. This allows HSML specifications to be coded using the abstract set of language concepts originally designed for users of the language, and avoids differences in interpretation due to parser restrictions. Of course, the down side if this argument is that TXL has its own grammar notation and does not easily import Yacc-style grammars, making it a more onerous task to add a new target language. TXL also has a standard technique for extending grammars to be "robust", which allows HSML to successfully process a wide range of variants and dialects of each target language without failing on syntax errors, even for inputs that are badly malformed or are missing macros and include files.

6. Summary

HSML, the Hot Spot Markup Language, is a concise executable specification language for specifying source code maintenance hot spots of all kinds. Using the abstract syntax tree to specify structural properties and queries on the design database to uncover semantic properties, HSML has been shown to be a practical tool for source code mining searches of many kinds.

At present HSML must be run in the LS/AMT environment shown in Figure 2. It is clear that it would be desirable to make HSML more widely accessible to other researchers by freeing it of this requirement. In light of the recent interest in using XML [16] in the reverse engineering community, it would seem a good idea to develop an HSML implementation to process XML-based program and design representations such as GXL [17].

HSML is not an easy notation to learn. It seems clear that it is not one that could be effectively used in the field by the average industrial programmer. For this reason, we believe that ideally HSML's capabilities should be embedded in a by-example authoring environment in which HSML specifications are inferred from actual source code samples rather than authored directly by hand.

While HSML has been used in practical industrial work, it is still very much a research prototype. As the range of applications expands, we continue to learn more about the HSML paradigm and to redesign HSML and its implementation in response.

It is important to note that this paper is not the first to introduce the idea of source code maintenance hot spots, or to point out that attachment to source code is an important aspect of software analysis and reengineering. Lethbridge and Singer [18] have empirically observed the need for tools to explore source code, a philosophy that inherently underlies HSML. Program slicing [19] is a well established analysis technique for "hot spotting" sections of source code that may influence a variable or other program entity during execution. CQML [20] is a quite general and flexible language for posing source code queries. And TuringTool [11] uses algebraic combinations of source code elisions to achieve results somewhat similar to HSML hot spots.

HSML adds to these ideas a concise, language independent executable formal specification language, the ability to take external design information into account in source code queries, and the representation of results as hot spot reports in original source text.

References.

[1] T.J. Biggerstaff, "Design recovery for maintenance and reuse", *IEEE Computer* 22,7 (July 1989), pp. 36-49.

[2] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. "Programmable reverse engineering", *International Journal of Software Engineering and Knowledge Engineering* 4,4 (December 1994), pp. 501-520.

[3] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K.Kontogiannis, H. A. Müller, J. Mylopoulos, and S. G. Perelgut, "The Software Bookshelf", *IBM Systems Journal* 36,4 (1997), pp. 564-593.

[4] K. Sartipi, K. Kontogiannis and F. Mavaddat, "A Pattern Matching Framework for Software Architecture Recovery and Restructuring", *Proc. IWPC 2000, 8th International Workshop on Program Comprehensio*n, Limerick, Ireland (June 2000), pp. 37-47.

[5] K. Cremer, "A Tool Supporting the Re-Design of Legacy Applications", *Proc. 2nd Euromicro Conference on Software Maintenance & Reengineering* (1998), pp. 142-148.

[6] L. Feijs, R. Krikhaar and R. van Ommering, "A Relational Approach to Software Architecture Analysis", *Software Practice and Experience* 28,4 (April 1998), pp. 371-400.

[7] H. Fahmy and R.C. Holt, "Software Architecture Transformations", *Proc. ICSM 2000, International Conference on Software Maintenanc*e, San Jose (October 2000).

[8] A. Schürr, A. Winter and A. Zündorf, "Visual Programming with Graph Rewriting Systems", *Proc. 11th IEEE Symposium on Visual Language*s, Darmstadt, Germany (Sept. 1995), pp. 326-333.

[9] R.C. Holt, "Binary Relational Algebra Applied to Software Architecture", Technical Report CSRI-345, Computer Systems Research Institute, University of Toronto (March 1996).

[10] "VTune(TM) Performance Analyzer V4.5", Intel Corporation (2000).

[11] J.R. Cordy, N.L. Eliot and M.G. Robertson, "TuringTool: A user interface to aid in the software maintenance task", *IEEE Transactions on Software Engineering* 16,3 (March 1990), pp. 294-301.

[12] J.R. Cordy, "The DRI Legasys Group LS/2000 Technical Guide to the Year 2000", Technical Report ED5-97, Legasys Corp., Kingston, and IBM Corp., Toronto (April 1997).

[13] 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.

[14] Friedrich W. Schroer, *The GENTLE Compiler Construction System*, R. Oldenbourg, Munich and Vienna, 1997.

[15] James J. Purtilo and John R. Callahan, "Parse-Tree Annotations", *Communications of the ACM* 32,12 (December 1989), pp. 1467-1477.

[16] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, "Extensible markup language (XML) 1.0", W3C Recommendation REC-xml19980210 (February 1998).

[17] R.C. Holt, A. Winter and A. Schürr, "GXL: Toward A Standard Exchange Format", *Proc. WCRE 2000 Working Conference on Reverse Engineerin*g, Brisbane, Australia (November 2000).

[18] T. Lethbridge and J. Singer, "Understanding Software Maintenance Tools: Some Empirical Research", *Proc. Workshop on Empirical Studies of Software Maintenance (WESS 97)*, Bari, Italy (October 1997), pp. 157-162.

[19] M. Weiser, "Program Slicing", *IEEE Transactions on Software Engineering* 10,4 (July 1984), pp. 352-357.

[20] *Code Query and Manipulation Language*, Reasoning Inc., Mountain View, California.