Metamodel Search: Using XPath to Search Domain-Specific Models

Rajesh Sudarsan

Virginia Tech
Department of Computer Science
Blacksburg VA 24060, USA
sudarsar@vt.edu

Jeff Gray

University of Alabama at Birmingham
Department of Computer and Information Sciences
Birmingham AL 35294, USA
gray@cis.uab.edu

A common task that is often needed in many software development tools is the capability to search the artifact that is being created in a flexible and efficient manner. However, this capability is typically absent in meta-programmable modeling tools, which can be a serious disadvantage as the size of a model increases. As a remedy, we introduce a method to search domain models using XPath – a World Wide Web Consortium (W3C) standard that uses logical predicates to search an XML document. In this paper, an XPath search engine is described that traverses the internal representation of a modeling tool (rather than an XML document) and returns those model entities that match the XPath predicate expression. A set of search queries are demonstrated on a case study.

Keywords: XPath, domain-specific modeling, metamodeling, model search
ACM Classification: D.2 (Software Engineering), D.2.2 (Design Tools and Techniques)

1. INTRODUCTION

The theory of Information Retrieval (IR) (Rijsbergen, 1979) is focused on the science of data collection and management, in association with mechanisms that provide rapid access to the collected information. In comparison to data retrieval techniques, IR generally searches for items that partially match the given requirements and then selects the best item out of the selection. In many domains, the vastness of available information makes searching a difficult task because of the *noise* generated from false positive matches. In such cases, it is important to retain and organize the search results in such a way that rapid retrieval is possible. Thus, an efficient search tool often leads to a substantial increase in productivity regarding the retrieval of the correct information.

Search tools are a common feature in most application programs. For example, in the domain of text-based searching (e.g., find/replace utility in Microsoft Word) or web-based searching (e.g., Google keyword-based search engine), the advanced search option provides the user with the capability to add flexibility to the search query. The advanced options in a web-based search help

Manuscript received: 1 January 2006
Communicating Editor: Hassan Reza

the user to sift through noise and rule out false positives amid a mass of possible matches. Figure 1 illustrates the advanced options provided by the Google search engine (Brin and Page, 1998) and Figure 2 shows the screenshot of the find and replace utility in Microsoft Word. In each of these examples, the user of the application is provided with a form-based search dialog that allows options to be selected in pre-defined attributes of the search space (e.g., the file format in a Google search, or whether case sensitivity is to be used in a Word search).

New mechanisms for improving the search activity have the potential to influence multiple tools and applications. A recent article in *IEEE Computer* by Ramakrishnan (2005) discusses the current trends in web searching technologies that aim to provide a completely different online experience for a web user. In another article, Ferriera and Atkins (2005) show the importance of an input web search query that acts as an imprecise description of the user request and aids in locating relevant information. Searching also plays an important role in software development environments. Integrated development environments (IDEs), such as Eclipse and Visual Studio, provide basic mechanisms for searching through source code in order to identify places in the code that have specific properties.

The ability to search is particularly needed in software modeling tools, which may contain multiple levels of hierarchy among many connected entities. A generic modeling tool can be used to specify the characteristics of a software system at a high-level of abstraction. The properties of the system are preserved in the models and the interaction between different entities of the system is denoted using the models in the domain. As users modify the system to meet the changing requirements, the domain model also has to evolve to accommodate the corresponding changes. Thus, an efficient search tool is required that is capable of searching for models with specific properties. Unlike programming environments that are driven by the text of source code, most modeling tools do not support advanced search capabilities, such as regular expressions or wild cards for locating a particular kind of modeling element. Absence of advanced search capabilities can be overlooked in modeling tools if the search space is small. However, advanced automated support becomes necessary when the size of a model increases. From our own experience, as confirmed by others (Johann and Egyed, 2004), models can grow to contain several thousand elements. In modeling tools, when the number of modeling elements grows exponentially, an efficient, scalable search technique is required to match the attributes to locate the model elements.

This paper discusses the development of a scalable solution for adding improved flexibility to the search feature of a specific modeling tool. The solution provides a method to search efficiently for a modeled artifact using XPath (Simpson, 2002). The search predicate is specified using a query string written in XPath, and the output of the search is a list of all model entities that match the given predicate. The results of the search are rendered to the user at the proper abstraction level (i.e., the visual abstraction provided by the host modeling tool), rather than at the textual XML level. In summary, XPath is typically applied to XML documents, but our search tool uses the XPath language as a notation for specifying a search query within a domain-specific modeling tool.

This paper is organized as follows. Section 2 presents a brief overview of XPath and domain-specific modeling in order to set the context for the remainder of the paper. Section 3 gives a brief description of the problems encountered when searching is performed in modeling tools. The core of the paper is found in Section 4, which introduces the XMOS search plug-in. A case study is presented in Section 5. The contribution and future work are summarized in Section 6.

2. BACKGROUND

The two main concepts discussed in this paper are XPath (Simpson, 2002) and domain-specific modeling (Gray *et al*, 2006). This section provides a brief background on these two topics.

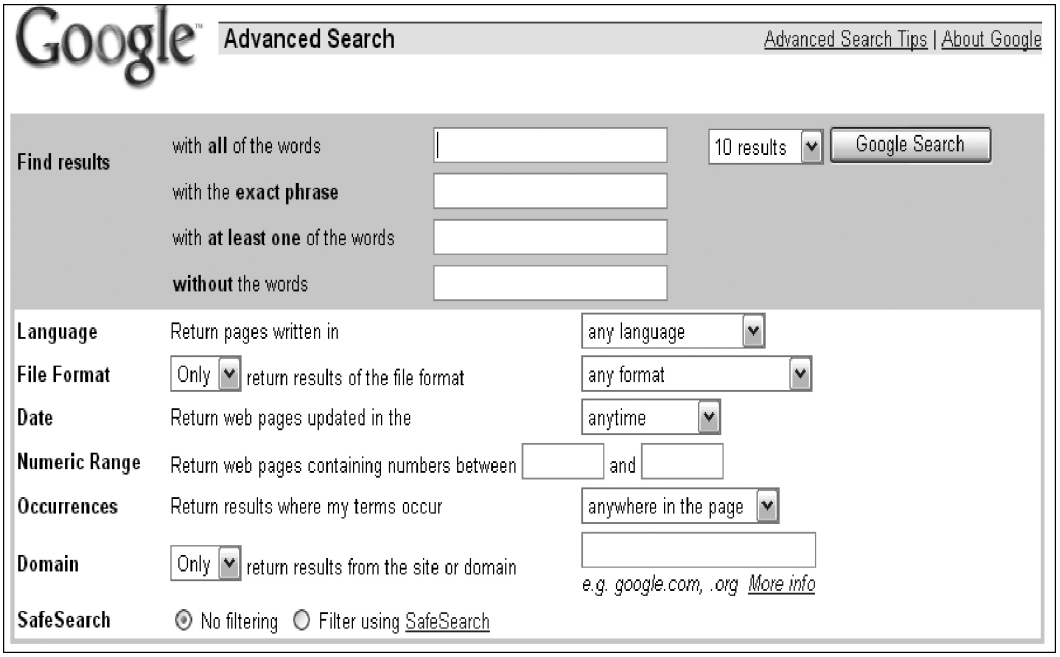

Figure 1: Advanced options in Google search engine

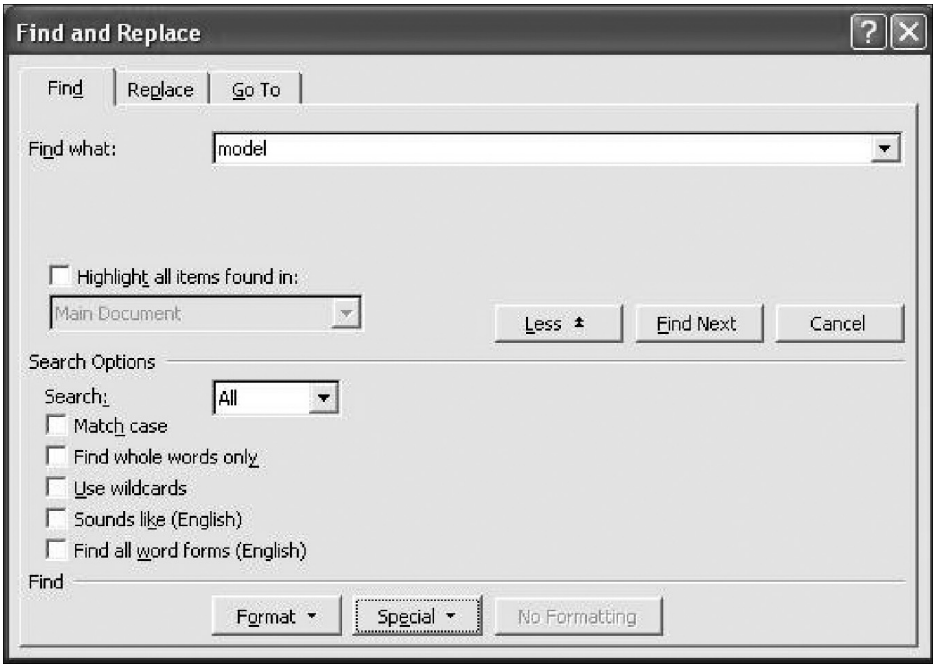

Figure 2: Find/replace utility in Microsoft Word

2.1 XML Path Language (XPath)

There are numerous tools and techniques that are based on the principle of searching using regular expressions, such as JQuery (McCormick and de Volder, 2004), XQuery (World Wide Web Consortium, 2005), and XPath (Simpson, 2002). JQuery is a flexible Java source code browser, but XQuery and XPath are query languages that are typically applied to XML files. The difference between XQuery and XPath lies in the fact that XQuery uses the structure of XML intelligently to express queries across all types of data that are represented in an XML document. XPath is a language for selecting parts of an XML document and is used in both extensible style-sheet language transformations (XSLT) (World Wide Web Consortium, 1999) and XPointer (Simpson, 2002).

The XPath query language is an industry standard for representing search expressions in hierarchical-based systems. XPath exploits the input tree structure by traversing either from the root node or any other context node, and selects nodes depending upon the return value of the matching predicate. The selection of nodes in an XML document is based on various properties (e.g., element type, attribute value, character content, and relative position). The basic expressions in XPath include string, numerals, Boolean, location paths, function calls, variable reference; the complex expressions include unions, filters, and relational expressions. The evaluation of an XPath expression produces an object, or a collection of objects, that exist in the associated XML document. These objects can be a node-set, Boolean, number, and string. Examples of XPath search expressions, as applied to domain-specific models, are provided in Section 5.2.

2.2 Domain-Specific Modeling with GME

The term *domain* generally denotes a group of entities that share the same characteristic or exhibit similar functionality. From a software engineering perspective, domain refers to a class of existing systems in a product-line architecture (PLA) (Bass, Clements, and Kazman, 2003; Parnas, 1976). All of the products developed using a specific architecture will share common properties of the product family in addition to the specific properties that make the product unique.

Models are an abstract representation of real-world systems or processes. Model Integrated Computing (MIC) (Sztipanovits and Karsai, 1997) employs domain-specific models (DSMs) to represent the software, its environment, and their relationship. With MIC, a modeling environment operates according to a metamodel, which contains a description of the entities, attributes, and relationships that are relevant in the domain (Karsai *et al*, 2004). With a domain-specific modeling environment, the intention of an end-user can be captured using idioms and icons that are familiar to the domain expert. This permits the description of the problem at a higher-level of abstraction from which other software artifacts (e.g., source code and simulation scripts) can be generated.

The Generic Modeling Environment (GME) (2005) is a meta-programmable tool that is based on the principles of MIC. The GME provides a universal design environment that can be configured over a wide range of domains. GME supports various concepts, such as hierarchy, multiple aspects, set, references and explicit constraints, for building complex DSMs (Lédeczi *et al*, 2001). The GME can be configured from a metamodel to work with any domain. Figure 3 illustrates the process for creating a modeling environment using GME and the MIC principles. On the left-hand side of the figure, a metamodel is specified using UML class diagrams and OCL constraints. This metamodel defines the syntax and static semantics of a modeling language (in this case, a metamodel for finite-state machines, FSMs). As indicated in the middle of the figure, end-users can then create DSMs (in this case, an automated teller machine based on FSM semantics) from the modeling language defined by the metamodel. The dynamic and executable semantics of a model are rendered by a model compiler, which is typically used to synthesize a model representation into some other

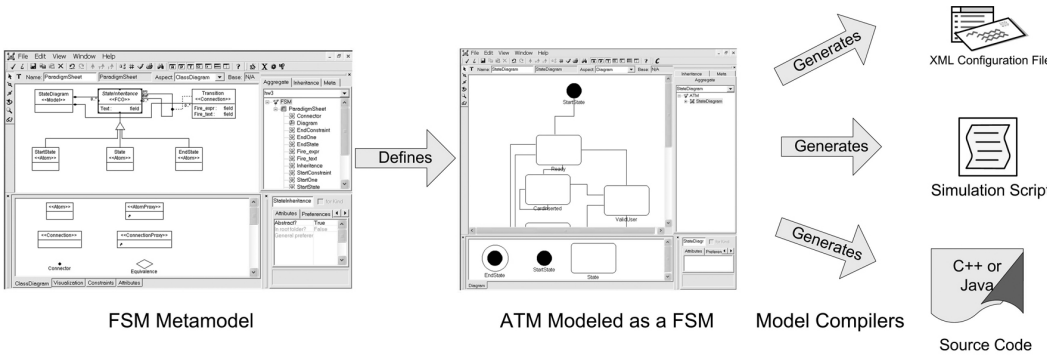

Figure 3: Metamodeling process with the GME

artifact. For example, model compilers may be created for a specific domain to generate source code, simulation scripts, or any other desired equivalent of the model representation (as shown in the right-hand side of Figure 3).

The GME allows third-party tools to be attached as plug-ins and invoked from within the modeling environment. The focus of the following section is a description of the core problems associated with searching in a modeling environment. As a solution to the model search problem, Section 4 presents a GME plug-in that permits search queries on models to be expressed in XPath.

3. LIMITATIONS AND CHALLENGES IN EXISTING MODEL SEARCH FACILITIES

Searching in a modeling tool involves comparisons between the user-provided search string and the property values of every modeling entity. Starting from GME v4, the search utility supports regular expressions as part of the search string. Users can specify input strings as a regular expression for Name, Role, Attributes, and Kind fields in the search window. Standard symbols such as "_", "-", ".", "*", "+", "(", ")", "[", "]", "$", and "^" are allowed in a valid regular expression. The search feature currently available in GME (shown in Figure 4) is capable of searching through the models in a simplistic manner by restricting the user to selecting one or more fixed checkboxes of element types. The number of attributes that can be matched is limited to those that correspond to Roles, Kinds, and Attributes; additional criteria beyond this fixed set are not permitted. Furthermore, the current search utility does not support composition of two or more search conditions to be expressed in a query. This leads to a serious disadvantage when the search domain contains thousands of entities. Another limitation in the original search utility is the lack of case-sensitive searching. These limitations are not specific to GME and are also evident in other metamodeling environments, such as MetaEdit++, which is a commercial tool from MetaCase (Kelly, Rossi, and Tolvanen, 2005).

In the search dialog shown in Figure 4, the user can specify a regular expression to search for basic element types like models, atoms, sets, and references. However, there are several limitations of the existing search utility, as listed below:

- The user cannot specify a hierarchical path in the input search string. An exact or partial name of the entity to be searched has to be given as input.
- The user cannot compose a query that has multiple attributes of a single entity (atom, model, set, or references) in the search string.
- The Attribute names and the value strings do not recognize names that contain spaces between them.

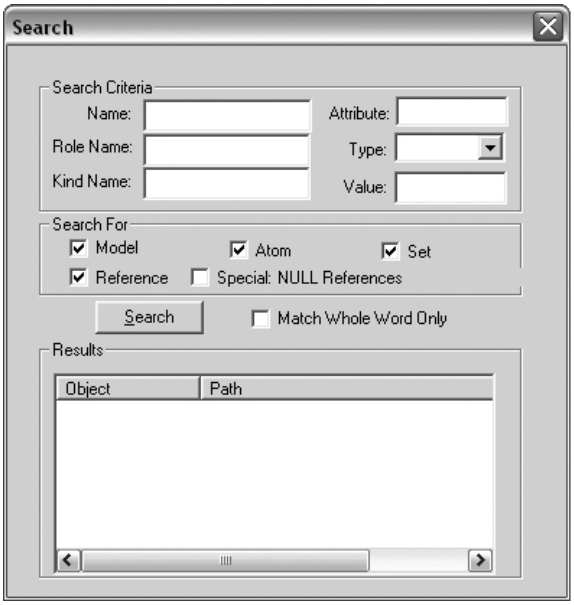

Figure 4: Existing search utility in GME

- The user cannot specify inequalities (such as < or >) as part of the search string, i.e., the value of the attribute specified in the search window has to be exactly equal to the attribute value of the element type being searched.

All of the above mentioned limitations reduce the flexibility in the search capability. To improve the flexibility of searching, the user may desire the ability to synthesize a new query by composing smaller sub-queries. This paper discusses the development of a new search plug-in for GME that uses an XPath predicate to specify the search criteria. XPath was chosen to be the input language for specifying the search predicates because it matches the hierarchical tree structure present in many GME models. XPath also provides the flexibility to compose new search queries by joining two or more sub-queries. A key challenge was to force XPath, which was designed to work with XML documents, to understand the internal representation of a model stored in the GME.

4. XPATH MODEL SEARCH (XMOS)

The XPath Model Search (XMOS) plug-in uses XPath predicates to define a search query on a domain-specific model. The plug-in can be used to locate the desired objects in a model that match the search criteria. The XPath query is specified from the entity names present in the domain's metamodel. The query predicates, when parsed using an XPath parser, output tokens that can be used for searching through a model.

The architecture of the XMOS plug-in is shown in Figure 5. The XPath Evaluator parses the input search predicate (an XPath expression) and provides a parse tree to the Interpreter module. The Interpreter module organizes the tree into searchable tokens and stores them in a customized data structure. The XMOS plug-in uses these tokens to search through the GME Builder Object list, which provides an application program interface (API) to the internal GME representation of a model. The final result produced by XMOS is a list of objects that match the search criteria. The

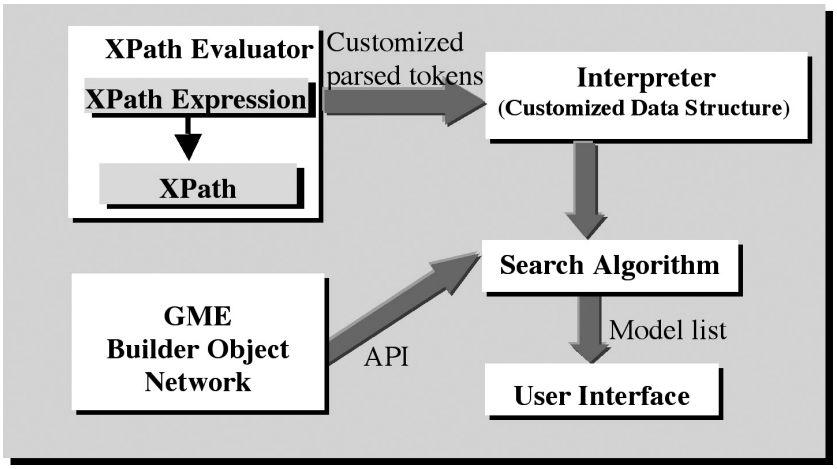

Figure 5: Architecture of the XMOS plug-in

user interface module displays the results of the search. Each of these modules is described briefly in the following sub-sections.

4.1 XPath Evaluator

The XPath Evaluator is divided into two main sub-modules: a module for receiving the search string from the user, and a module for evaluating the input expression for its syntactic correctness and subsequent generation of the appropriate parse tree. The input dialog box reads the search expression and passes it to the XPath parser. The XMOS XPath parser is based on the TinyXPath (TinyXPath, 2002) parser, which is an open source implementation. The parser supports most of the XPath expressions and conforms to the W3C XPath specification.

4.2 XPath Interpreter

The interpreter module uses tokens from the parser to drive the interpretation of the search expression. From the token list, every token is interpreted with respect to the previous token. For example, if the first token in the XPath expression is a slash (/), then the next expected node can be slash (/), star (*), or a node name. The first slash indicates that the searching starts from the root node. Thus, when a slash (/) is encountered as a second consecutive token, the tokens are combined to interpret it as a double-slash and the searching technique is set to recursive descent. If a star (*) is encountered as the second token, then the expression is interpreted to search for any kind of GME object. The node name is used to search for an object with the name that is a child of the root node. Example expressions showing these differences are provided in the case study of Section 5.

4.3 GME BON

To facilitate the automatic processing of model data, GME supports access to the underlying model information through the Builder Object Network (BON) API, which provides a powerful but easy to use interface for writing application programs that can be attached to GME as a plug-in. BON provides a network of C++ objects, where each object represents an entity in the GME model database. The C++ methods provide access to read/write object properties, attributes, and relations.

4.4 Search Module

The algorithm used for searching the objects in a GME model involves the use of BON objects in association with the XPath interpreter described in Section 4.2. Whenever a token is received, the process of searching is performed in a sequential manner. Depending upon the interpretation, corresponding BON objects are invoked and a search is performed for the particular token. For example, if the search token is a model name, then a BON `CBuilderModelList` object is queried to compare all the models returned by the reflective method `GetModels()`. A search is performed on this list to match the model name given in the search predicate. For the next token, the matched list of `CBuilderObjects` is searched for a match for the next token. By proceeding in a similar manner, the search is invoked for all the tokens given in the search predicate. For the first token, the input list encompasses all the `CBuilderObjects` in the project. The result of the search is the list of all `CBuilderObjects` returned after all the tokens have been processed.

As an example of the search process, if the search string is "//*" (which would return all elements in the model), the first token to be processed is slash (/). Interpretation of this token sets the starting point of the search as the root of the GME model tree. When the next token is also a slash, then the previous token and the current token are combined together and interpreted as double-slash (//). Interpretation of this token will set the search strategy as recursive descent. The next token star (*) acts as a wildcard character. This token, when interpreted with respect to the previous token (double-slash), results in the recursive search for all `CBuilder` objects present in the project starting from the root to the leaf. A summary of the search algorithm used in the plug-in is shown in Figure 6.

1. Read the input search string. Tokenize the string and store the tokens in a vector list.
2. While the token list is not completely traversed, do the following
   a. Get the next token from the list
   b Set the value of CURR_TOKEN as the current token.
   c. Interpret this token with respect to the previous token in PREV_TOKEN
   d. Query the GME Builder Object Network for the objects corresponding to the interpreted token.
   e. Store the list of matched GME objects, their location, and the type of object.
   f. Set PREV_TOKEN=CURR_TOKEN
3. Output the final list of matched GME objects to the user.

Figure 6: XPath model search algorithm

4.5 User Interface

The XMOS user interface is a dialog box that consists of an edit box for entering the XPath expression. It also has a list control box that is used to display the search results as a multi-tab output. The different tabs in the output window are the Object name, location path, location coordinates, and the type of the object (atom, model, reference, set, or connection). The path tab displays the path of the component from its level in the hierarchy to the topmost model. The Location tab gives the exact coordinates of the components within the window in which it is located. Figures 8 through 11, which are provided in the case study of Section 5, illustrate the XMOS search dialog interface.

5. CASE STUDY

As an example case study, the XMOS plug-in has been applied to models developed in the Embedded Systems Modeling Language (ESML), which is a modeling language designed by the

Vanderbilt/ISIS DARPA MoBIES team (ESML, 2005). Section 5.1 describes the models developed in ESML for denoting scenarios in the Boeing Bold Stroke Mission Computing Avionics Framework (Sharp, 1998). Two example search expressions on an ESML model are presented in Section 5.2.

5.1 Boeing Bold Stroke and the ESML

Bold Stroke is a product-line architecture written in C++ that was developed by Boeing in 1995 to support families of mission computing avionics applications for a variety of military aircraft (Sharp, 2000). Mission computing software such as Bold Stroke is responsible for controlling navigational sensors, weapon deployment sub-systems, and cockpit panel displays that are used by a fighter pilot. Bold Stroke is a multi-threaded, real-time system that has hard and soft deadlines. It is a very complex framework with several thousand components implemented in over a million lines of code.

The ESML is a domain-specific graphical modeling language in GME for specifying real-time mission computing embedded avionics applications, such as Bold Stroke. The goal of ESML is to address the issues arising in system integration, validation, verification, and testing of embedded systems. An example ESML model of the internal properties of a CORBA real-time event channel is shown in Figure 7 (this figure is provided as an example within the ESML distribution from Vanderbilt). It illustrates the components and interactions for a specific scenario within Bold Stroke. This scenario provides insight into a small subset of aspects that are related to weapon system avionics software development, and at the same time addresses issues of component distribution.

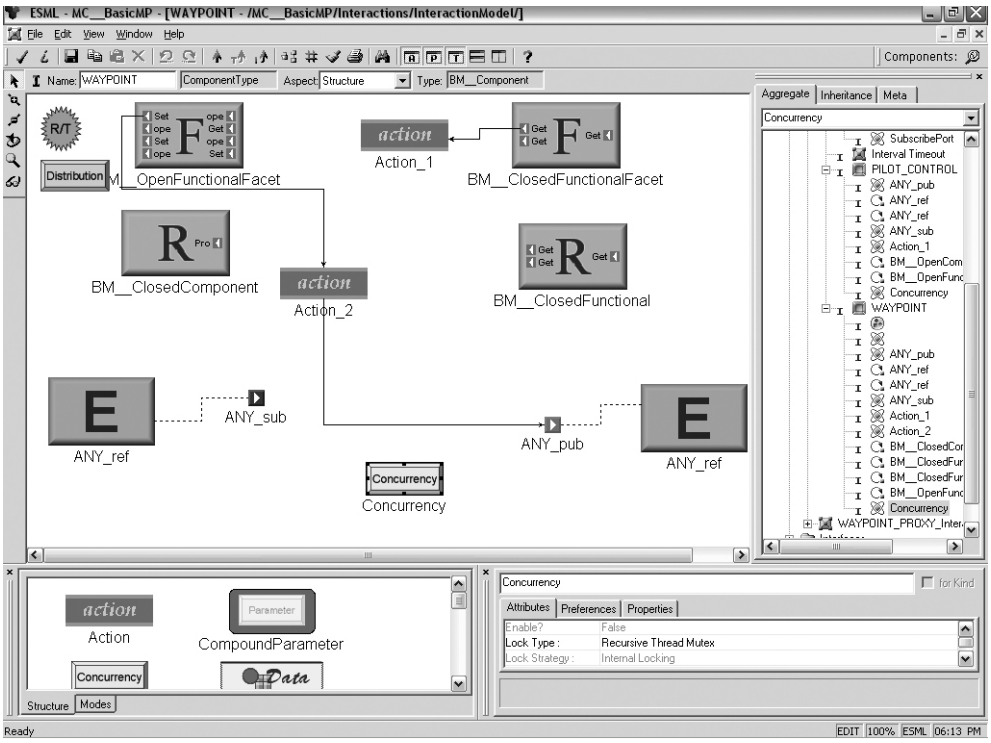

Figure 7: Bold Stroke component interaction in ESML

There are representative ESML models for all of the Bold Stroke usage scenarios that have been defined by Boeing.

From an ESML model, deployment and assembly files are generated that specify the usage scenarios for a specific product-line configuration of Bold Stroke. An ESML model may contain several hundred modeling elements that need to be configured to explore different design alternatives. Searching through a large number of hierarchical models (in order to change configuration properties) can be very time consuming without the aid of a flexible search feature.

5.2 Example search expressions

The XMOS plug-in was applied to several different test cases using the Bold Stroke ESML models. This sub-section presents four examples of the search test cases and the results.

Case 1: Lock Type and *Lock Strategy* are attributes of the concurrency atom that specify the concurrency mechanism in a model. The following XPath expression finds all concurrency elements in the scenario that have the attribute *Lock Type* set to *Null Lock*, and *Lock strategy* not set to *Recursive Thread Mutex* (result is shown in Figure 8):

//concurrency[@locktype= "Null Lock" and @lockstrategy != "Recursive Thread Mutex"]

Case 2: Data and *log* elements are used in an ESML model to store additional information. The following XPath expression finds all elements in which the *value* of the *data* element equals 5, and the *kind* attribute of the *log* element is set to *On Read* (result is shown in Figure 9):

//data[@value=5] and //log[@kind= "On Read"]

Case 3: In this case, the XPath expression is used to locate all elements in which the *value* of the *data* element equals 5, or the *kind* attribute of the *log* is set to *On* (result is shown in Figure 10).

//data[@value=5] or //log[@kind= "On"]

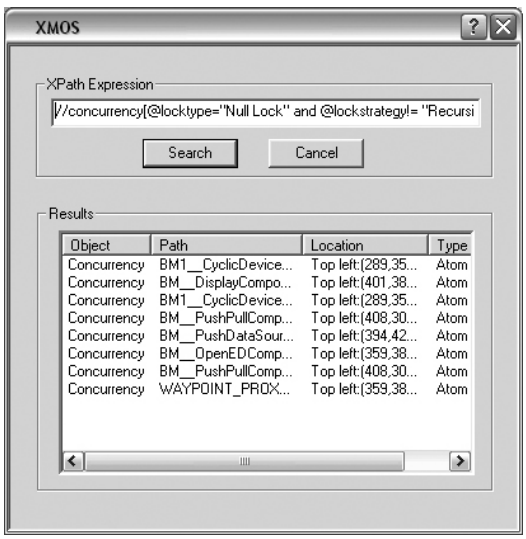

Figure 8: Result for case 1

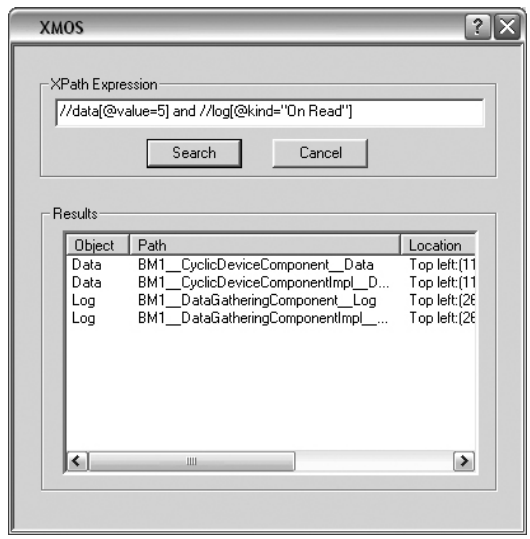

Figure 9: Result for case 2

Case 4: The *period* attribute is used with the *Timer* elements and the *datasize* attribute is used as an upper bound on the maximum data storage. The following XPath expression locates a union set of all components whose *period* is greater than *20000* and all components whose *data size* equals *32* (result shown in Figure 11).

//[@period>20000] and //*[@datasize=32]*

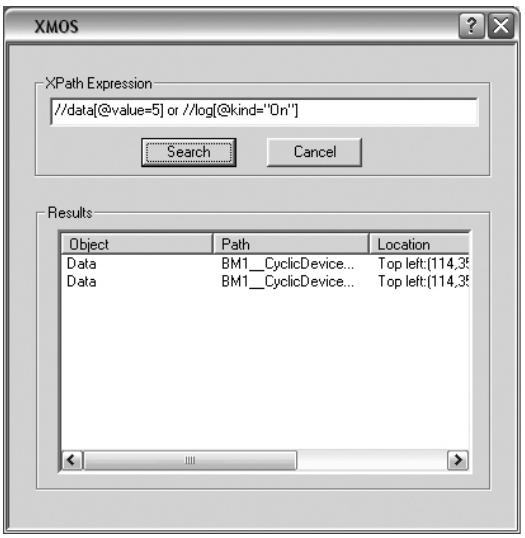

Figure 10: Result for case 3

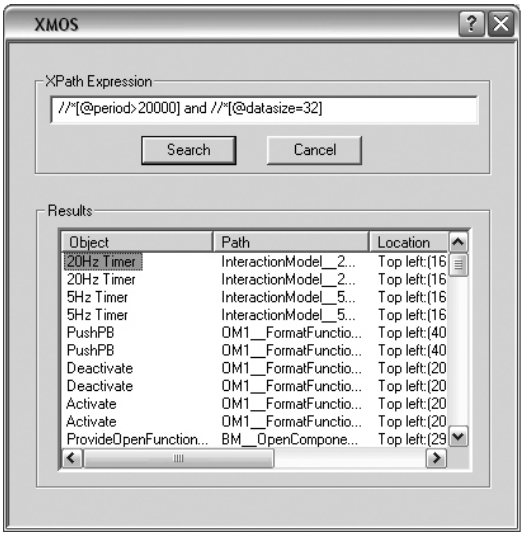

Figure 11: Result for case 4

5.3 Comparison with the existing search feature in GME

To determine the advantages offered by XMOS, we executed regular expressions that were similar to the previous case studies using the existing search utility in GME. The first query was synthesized to search for all *Concurrency* atoms whose attribute *Lock Strategy* is set to *External Locking*. This query is similar to the query used in test case 1, which searched for attribute values of model elements. The regular expression query does not match any results (shown in Figure 12 and 13). The regular expression *Concurrency** searches for all elements whose name starts with *Concurrency*.

A second query using the existing GME regular expression dialog was executed to locate all elements having an attribute *Multiplicity* whose value equals 4. The output on the left in Figure 14 shows three components that match the query. However, if the user modifies the query to search all elements that have an attribute *Multiplicity* whose value is greater than 4, then the output is null (see the output on the right in Figure 14). This shows the inability of the original GME search utility to support inequality expressions in the search query. As demonstrated in Section 5.2, XMOS offers a solution to these limitations.

6. RELATED WORK

To the best of our knowledge, there has been no prior work that has addressed the problem of searching domain-specific models efficiently. However, research is being conducted on efficient search tools for textual artifacts. The JQuery tool is based on the idea of textual referencing (McCormick and de Volder, 2004). The main functionality of JQuery enables identification of software artifacts in textual

source code. It is a query-based Java source code browser that allows a user to create different browser views depending upon the input query. A JQuery user can formulate independent logic queries and execute them against the source code at runtime. Each query, when executed, defines a unique browser

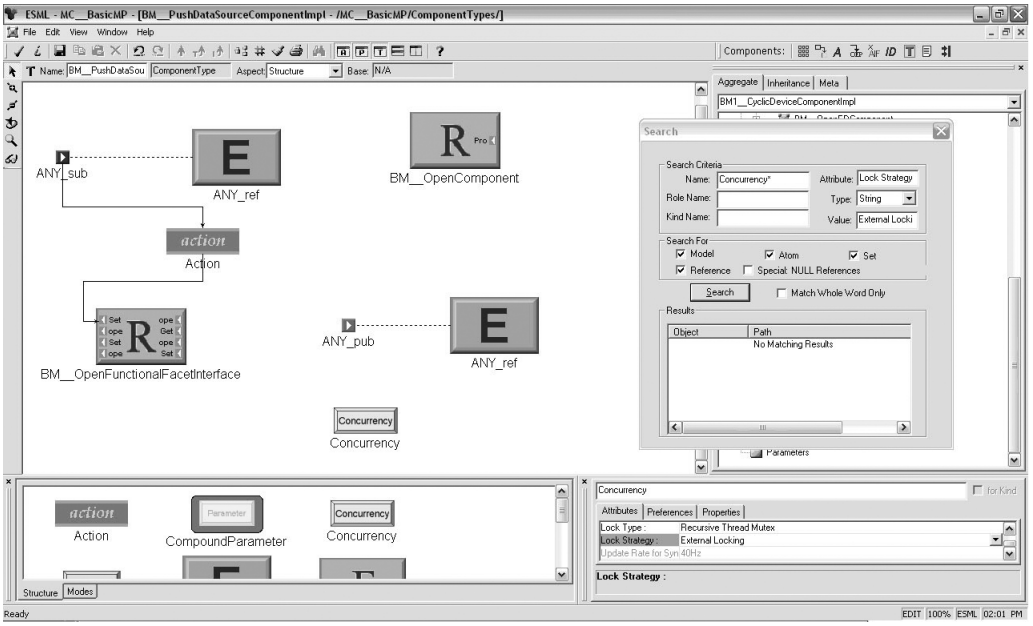

Figure 12: Example query with existing search utility

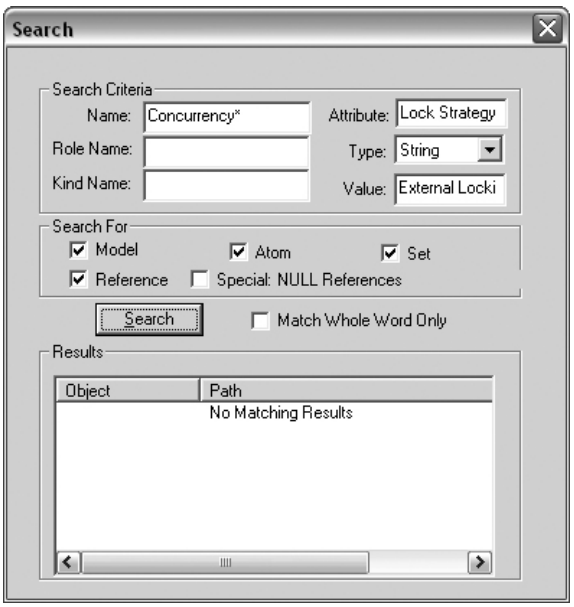

Figure 13: Test case query with GME regular expressions

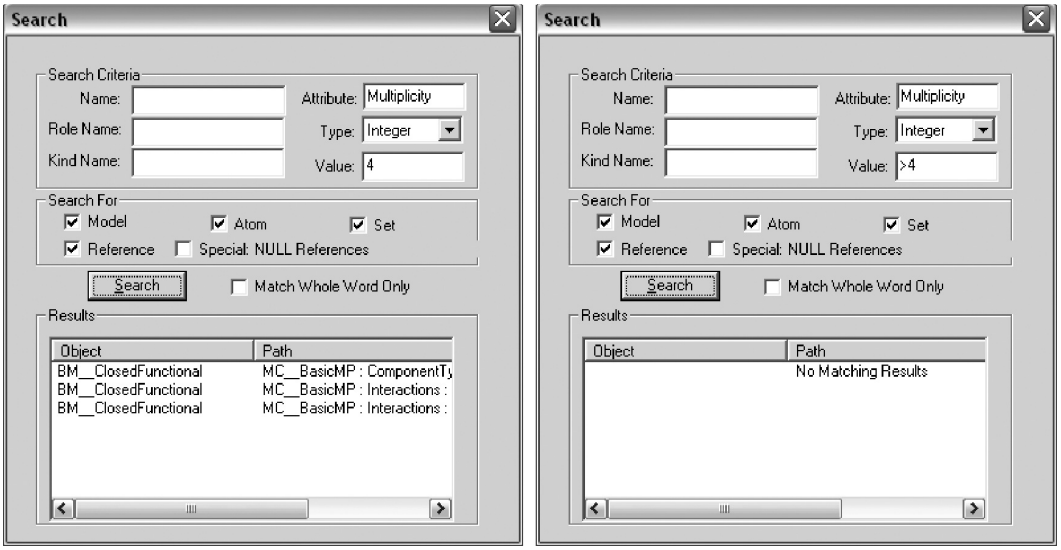

Figure 14: Example query showing limitations of existing GME search utility

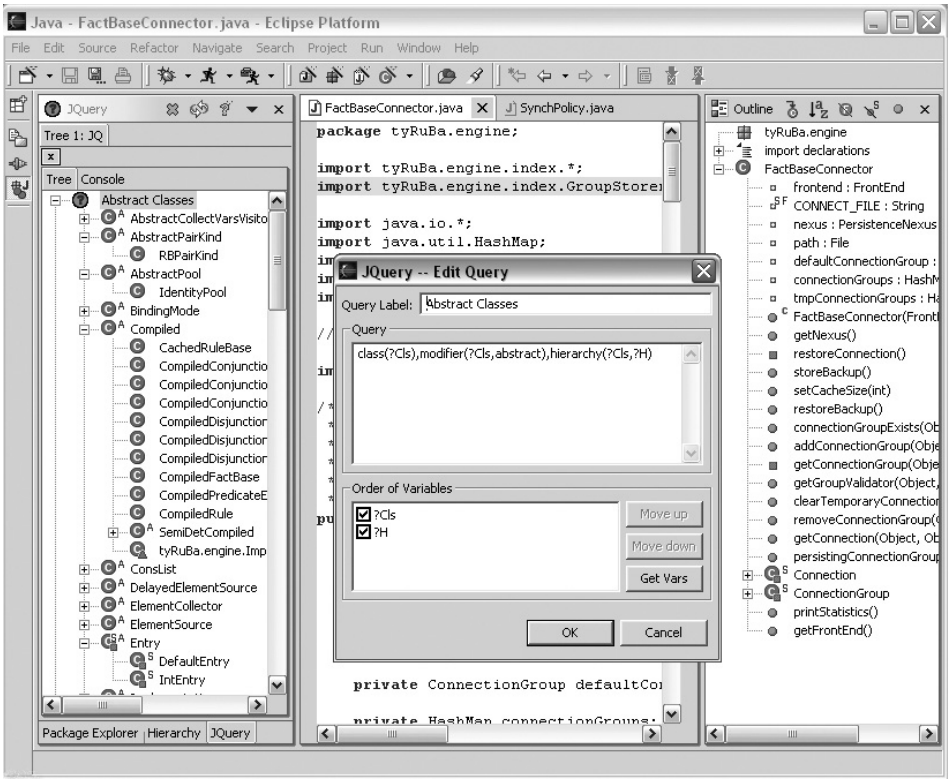

Figure 15: JQuery browser within Eclipse (McCormick and de Volder, 2004)

view for the source code under test. The user can either choose from a variety of existing queries or modify the existing queries to create new browser views. Figure 15 shows the view of a JQuery search session within Eclipse. As can be seen in the search dialog, a general similarity exists between the XMOS dialog and JQuery. The primary difference is the types of software artifacts that are targeted (i.e., JQuery is focused on textual source code, but XMOS' objective is to provide searching facilities for visual hierarchical models). The mismatch in abstraction layers suggests that it would be difficult for JQuery to be adapted to the concerns of domain-specific modeling.

With respect to searching visual objects, Kurlander and Bier (1988) initially devised a basic technique for searching and replacing graphical objects. Later, Kurlander and Fiener (1992) extended the idea of graphical search and replace by allowing users to perform a constraint-based search and replace action on the objects. This work was focused within the context of graphical image editors, which is a separate domain from the software modeling focus of this paper.

7. CONCLUSION

The lack of flexibility in the previous GME search utility necessitated the development of an efficient algorithm and plug-in to enable customized searching of domain-specific models. The XMOS plug-in exploits the hierarchical structure of GME to search for artifacts that match an XPath predicate expression. In very large models, there is often a need to search for a particular set of modeling entities to change their properties. XMOS provides the flexibility needed to locate these entities based on attribute values. There remain several limitations and areas that we have identified for future work. These include the following:

- A desired feature that is needed in XMOS would allow a direct link between the results and the actual location of the models in the domain, such that when a user clicks on the link on the output screen, a new window will be opened to show the exact location of the matched modeling elements. The implementation of this feature will enable a model engineer to locate a desired model entity more quickly by minimizing the amount of mouse clicks required.
- The XPath Evaluator will be extended to include a more complex expression for searching. This is merely an implementation effort that will extend XMOS' ability to handle additional XPath primitives that are currently not supported (e.g., collection operations and primitive functions such as max, min, size). The addition of such primitives would further distinguish the power offered by XMOS over that of the regular expressions offered in the standard GME search dialog.
- We believe that XMOS can be generalized for other modeling toolkits to add flexibility to the searching technique. To enable XMOS within other metamodeling tools, the implementation that hooks into the host modeling tool API needs to be abstracted into a set of classes that can be specialized based on the features provided by the API.

Additional information about XMOS, including a plug-in download page, can be found at: http://www.cis.uab.edu/info/alumni/sudarsar/xmos/

REFERENCES

BASS, L., CLEMENTS, P. and KAZMAN, R. (2003): *Software Architecture in Practice*. Addison-Wesley, Boston, MA, USA.

BRIN, S. and PAGE, L. (1998): The anatomy of a large-scale hypertextual web search engine. *Proceedings of 7th International WWW Conference*, Brisbane, Australia, 7:107–117.

ESML. (2005): The embedded systems modeling language. http://repo.isis.vanderbilt.edu/tools/get_tool?ESML. Accessed 25-May-2006.

FERREIRA, A. and ATKINSON, J. (2005): Intelligent search agents using web-driven natural-language explanatory dialogs. *IEEE Computer*, 38(10): 44–52.

GME. (2005): The generic modeling environment. http://www.isis.vanderbilt.edu/projects/gme. Accessed 25-May-2006.
GRAY, J., TOLVANEN, J-P., KELLY, S., GOKHALE, A., NEEMA, S. and SPRINKLE, J. (2006): Domain-specific modeling, *CRC Handbook on Dynamic System Modeling*, CRC Press.
JOHANN, S. and EGYED, A. (2004): Instant and incremental transformation of models, *Proceedings of the Conference on Automated Software Engineering*, Linz, Austria, 362–365.
KARSAI, G., MAROTI, M., LÉDECZI, A., GRAY, J. and SZTIPANOVITS, J. (2004): Composition and cloning in modeling and metamodeling. *IEEE Transactions on Control System Technology* (special issue on Computer Automated Multi-Paradigm Modeling), 12(2): 263–278.
KELLY, S., ROSSI, M. and TOLVANEN, J. (2005): What is needed in a MetaCASE environment? *Journal of Enterprise Modeling and Information Systems Architectures*, 1(1): 25–35.
KURLANDER, D. and BIER, E.A. (1988): Graphical search and replace. *Proceedings of the 15th annual conference on Computer graphics and interactive techniques SIGGRAPH '88*, 22(4):113–120.
KURLANDER, D. and FEINER, S. (1992): Interactive constraint-based search and replace. *Proceedings of the SIGCHI conference on Human factors in computing systems*, Monterey, California, 609–618.
LÉDECZI, A., BAKAY, A., MAROTI, M., VOLGYESI, P., NORDSTROM, G., SPRINKLE, J. and KARSAI, G. (2001): Composing domain-specific design environments. *IEEE Computer*, 34(11): 44–51.
McCORMICK, E. and VOLDER, K.D. (2004): JQuery: Finding your way through tangled code. *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, Vancouver, BC, Canada, 19: 9–10
PARNAS, D.L. (1976): On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(3):1–9.
RAMAKRISHNAN, N. (2005): Frontiers of search. *IEEE Computer*, 38(10): 26–27.
RIJSBERGEN, C.J.V. (1979): *Information Retrieval*, Butterworths, London.
SHARP, D. (1998): Reducing avionics software cost through component based product-line development. *Proceedings of 17th AIAA/IEEE/SAE Digital Avionics Systems Conference*, Bellevue, WA, 2:G32/1–G32/8.
SHARP, D. (2000): Component-based product line development of Avionics Software. *First Software Product Lines Conference (SPLC-1)*, Denver, Colorado, 353–369.
SIMPSON, J. (2002): *XPath and XPointer*. O'Reilly Publications, San Francisco, CA, USA.
SZTIPANOVITS, J. and KARSAI, G. (1997): Model-integrated computing, *IEEE Computer*, 30(4):10–12.
TINYXPATH. (2002): A tiny C++ XPath processor. http://tinyxpath.sourceforge.net. Updated 31-Dec-2003. Accessed 25-May-2006.
WORLD WIDE WEB CONSORTIUM. (1999): XSL Transformations (XSLT). W3C recommendation. http://www.w3.org/TR/xslt. Accessed 25-May-2006.
WORLD WIDE WEB CONSORTIUM. (2005): XQuery 1.0: An XML query language. W3C Candidate Recommendation. http://www.w3.org/TR/xquery. Updated 3-November-2005.

BIOGRAPHICAL NOTES

Rajesh Sudarsan

Rajesh Sudarsan is a second year PhD student in the Department of Computer Science at Virginia Tech. He received his Bachelors degree in Computer Engineering from University of Mumbai, India and M.S. in Computer Science from the University of Alabama at Birmingham. Rajesh's research is primarily focused in the area of high performance computing (HPC) and dynamic parallel scheduling. More information about his research can be found at http://people.cs.vt.edu/~sudarsar.

Jeff Gray

Jeff Gray is an Assistant Professor in the Department of Computer and Information Sciences at the University of Alabama at Birmingham (UAB) where he directs the Software Composition and Modeling (SoftCom) laboratory. He received the Ph.D. in Computer Science from Vanderbilt University, where he also served as a research assistant at the Institute for Software Integrated Systems. Jeff's primary research areas are model-driven engineering, aspect-oriented programming, and generative programming. His research is currently supported by NSF, with recent funding from DARPA and an IBM Eclipse grant. Jeff currently serves as the chair of the Alabama IEEE Computer Society. More information about his research and publications can be found at http://www.gray-area.org