International Journal of Software Engineering
and Knowledge Engineering
Vol. 16, No. 1 (2006) 1–25

World Scientific
www.worldscientific.com

ANALYSIS OF META-PROGRAMS: AN EXAMPLE

STAN JARZABEK*,‡, HONGYU ZHANG†,§, SHEN RU*,
VU TUNG LAM* and SUN ZHENXIN*

*School of Computing, National University of Singapore,
Lower Kent Ridge Road, Singapore 117543
†School of Computer Science and Information Technology,
RMIT University, Melbourne 3001, Australia
‡stan@comp.nus.edu.sg
§hongyu@cs.rmit.edu.au

Meta-programs are generic, incomplete, adaptable programs that are instantiated at construction time to meet specific requirements. Templates and generative techniques are examples of meta-programming techniques. Understanding of meta-programs is more difficult than understanding of concrete, executable programs. Static and dynamic analysis methods have been applied to ease understanding of programs — can similar methods be used for meta-programs? In our projects, we build meta-programs with a meta-programming technique called XVCL. Meta-programs in XVCL are organized into a hierarchy of meta-components from which the XVCL processor generates concrete, executable programs that meet specific requirements. We developed an automated system that analyzes XVCL meta-programs, and presents developers with information that helps them work with meta-programs more effectively. Our system conducts both static and dynamic analysis of a meta-program. An integral part of our solution is a query language, FQL in which we formulate questions about meta-program properties. An FQL query processor automatically answers a class of queries. The analysis method described in the paper is specific to XVCL. However, the principle of our approach can be applied to other meta-programming systems. We believe readers interested in meta-programming in general will find some of the lessons from our experiment interesting and useful.

Keywords: Meta-programs; program understanding; program analysis; program queries; debugging.

1. Introduction

Meta-programs are generic, incomplete, adaptable programs. Custom programs, derived from a meta-program, may differ in requirements, design decisions, platforms, etc. Programs instrumented with macros [1,2], transformational systems and generative techniques [3] are examples of meta-programming techniques.[a]

[a]Static meta-programming systems transform a program before compilation and load time [3]. Other forms of meta-programming modify programs during execution (e.g., using reflection). In this paper, we are concerned with static meta-programming only.

In our projects, we build meta-programs with XVCL (XML-based Variant Configuration Language) [4,5]. XVCL is a public domain meta-language, method and tool for enhanced maintainability and reusability. In addition to unrestrictive parameterization and a range of mechanisms to handle changes, XVCL supports meta-level partitioning of programs into generic, adaptable meta-components called x-frames. x-frames are inter-related and organized into a hierarchical structure called an x-framework. An x-framework is a meta-program from which the XVCL Processor generates concrete, executable programs that meet specific requirements. In that sense, an XVCL meta-program may be considered an implementation of the product line architecture concept [6].

Meta-programming systems have the potential to enhance conventional programming techniques, helping programmers to write more generic, adaptable, and changeable programs. The reader can find a discussion of meta-programming systems (e.g., meta-programming with C++ templates) with many examples of their application in software engineering in [3]. Application of XVCL to meet software engineering goals is described in [7,8,9] and further sources can be found at our web site [5].

Flexible manipulation of programs and genericity, central in software reuse, are the strength of meta-programming. However, meta-programming techniques come at a price: High maintenance cost is a living proof that understanding of concrete programs is already complex enough. A meta-program — being a generic representation of a class of programs — is bound to be more difficult to describe and understand than a concrete program. We believe that methods and tools for understanding and debugging of meta-programs are essential for wider acceptance of meta-programming as an effective approach to software development. We believe that lack of such methods and tools is one of the factors that does not allow us to fully exploit the potentials of meta-programming techniques in software engineering context.

Static and dynamic analysis methods have been applied to ease understanding and maintenance of programs: Could similar methods aid in understanding of meta-programs? Given the lack of fundamentals unifying a wide variety of meta-programming techniques, and relative immaturity of meta-programming concepts, it is difficult to address this question in general.

In our earlier work, we developed a Program Query Language (PQL) [10] for static analysis of conventional programs (e.g., written in COBOL or Java). The aim of PQL system was to acquire information about properties of programs that could be automatically computed in the course of static program analysis. This information was then presented to a programmer, in order to help him/her understand high-level properties of programs, related to various maintenance tasks. In PQL, a programmer expressed queries in terms of conceptual models of program design information. The information required to answer program queries was computed by a front-end and stored in a Program Knowledge Base (PKB). A PQL query evaluator

could automatically answer queries consulting the PKB for relevant information.

In this paper, we applied PQL concepts to aid in understanding of x-frameworks, i.e., meta-programs written in XVCL. We developed an automated system that analyzes an x-framework, and presents developers with information that helps them work with an x-framework in a more effective way. Our system conducts both static and dynamic analysis of an x-framework. An integral part of our solution is an x-framework query language, called FQL, in which we can formulate questions about x-framework properties. An FQL query processor automatically answers queries. We believe information that can be acquired in a question-answer session can help developers understand higher-level x-framework properties. FQL queries are meant to bridge the gap between detailed information that can be directly observed at the x-framework level, and higher-level properties of an x-framework that are directly related to various tasks developers need to accomplish when enhancing or reusing an x-framework.

The design of FQL follows the approach of PQL [10]: We start by building models describing conceptual entities of XVCL (such as an x-frame, a meta-variable or an XVCL command) and the relationships among those entities that matter to developers working with x-frameworks. FQL is a PQL-like language that allows us to formulate questions about x-frameworks in terms of those models.

An important finding from our study is that static program analysis techniques alone have limited value for understanding of meta-programs. Interesting and useful queries require partial processing (that is instantiation) of a meta-program. Therefore, we incorporated elements of dynamic analysis into our x-framework analysis system.

The details of the analysis method described in the paper are specific to XVCL. However, the principle of our approach can be applied to other meta-programming systems. We believe readers interested in meta-programming in general will find some of the lessons from our experiment interesting and useful.

2. Background and Related Work

Genericity and changeability are two important program qualities that underlie effective reuse and maintenance. Developers address these qualities with a range of programming language features and design techniques such as inheritance, generics [11], information hiding [12], design patterns [13], architectural approaches [6] or mechanisms supported by component platforms (such as J2EE™ or .NET™). However, often it becomes difficult to achieve genericity and changeability with conventional methods, without compromising other design goals that developers must meet [7,8]. Meta-level mechanisms offer parameterization (and sometimes generation) capabilities that are not constrained by an underlying programming language or component platform. When generic design becomes difficult with conventional methods, meta-level techniques may offer a workable solution [8,14].

Macros (e.g., C preprocessor CPP [1] or m4 [2]) are the oldest form of meta-

programming. Most of the macro systems are merely implementation level mechanisms for managing changes (or variability in the context of reuse [6]). Failing to address change at analysis and design levels [15], macros never evolved towards full-fledged "design for change" methods [16]. Programs instrumented with macros tend to be difficult to understand and test [15]. Among the most important deficiencies of macros, is weak parameterization and lack of a meta-level decomposition mechanism. Lack of proper rules for parameter propagation (and overriding) across program units does not allow programmers to formally link and control chains of inter-related program transformations. Despite these problems and new powerful features of modern programming languages, macros are still often used by programmers [17]. It was generally believed that functions, and later classes and components, were more promising than macros for reuse. In fact, it has never been an either-or situation, as meta-level processing (of which macros are a simple example) addresses different engineering goals than functions/classes/components: The prime goal of functions/classes/components is to define executable units of a program runtime architecture.

Frame technology [16] is an example of a fully developed and highly successful meta-programming system which originated from the concepts of macros, but then evolved into a comprehensive mechanism for managing changes over years of software evolution and to facilitate systematic reuse. Frame technology has been extensively applied in industry to manage variants and evolve multi-million-line, COBOL-based, information systems. While designing a frame architecture is not trivial, subsequent complexity reductions and productivity gains are substantial. These gains are due to the flexibility of the resulting architectures and their evolvability over time.

A potential role of generative techniques (meta-programming, in particular) in solving software engineering goals have been described in [3]. The authors also indicated problems with understanding and debugging meta-programs as an important factor that limits engineering benefits and hinders wide adoption of such techniques. Static and dynamic analysis methods have been applied to ease understanding of programs [10] but application of such methods to meta-programs has not been explored yet. Among existing approaches, only m4 [2] offers some meta-level debugging features.

3. An Overview of XVCL

3.1. *What is XVCL?*

XVCL [18,5] that works with any other software technology and adds value to it in terms of enhanced genericity and changeability. First, we develop a program with one of the programming languages and with conventional design techniques, to achieve proper program modularization and required runtime properties such as performance or reliability. Then, we apply XVCL on top of a program to facilitate change and/or to inject extra levels of genericity into it.

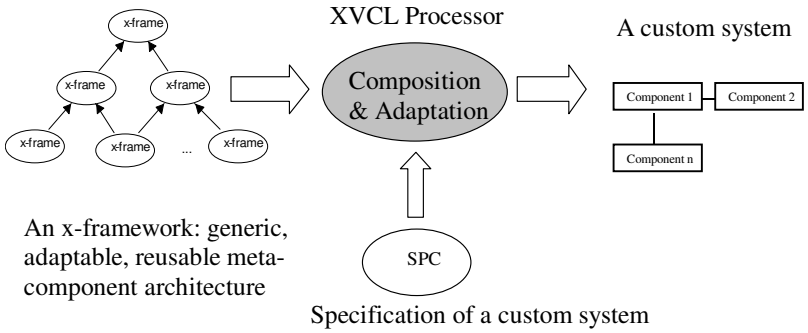

Fig. 1. XVCL at work.

XVCL partitions a program into meta-components. XVCL meta-components are called x-frames. Meta-level partitioning is independent of (therefore, does not conflict with) decomposition into program modules (e.g., classes, functions, higher-level components and sub-systems). x-frames form a hierarchically structured architecture, called an x-framework, and are instrumented with XVCL commands (Fig. 1). Decomposition along x-frame boundaries, hierarchical organization of x-frames and unrestrictive parameterization of x-frames with XVCL commands are the main XVCL mechanisms to facilitate changeability and genericity. An x-framework is carefully designed into layers to enhance reuse. It is also "normalized" to unify recurring patterns of program similarities which reduces conceptual complexity of a program as perceived by a programmer.

XVCL uses "composition with adaptation" rules to generate a specific program from an x-framework. Rules of XVCL are simple but sufficient to manage a wide range of program variants at analysis, design and code levels, from a compact base of meta-components.

XVCL is based on frame concepts. Being a modern and versatile version of Bassett's frames [16], the underlying principles of the XVCL have been thoroughly tested in practice. Unlike original frames, XVCL blends with contemporary programming paradigms and complements other design techniques.

3.2. *How does XVCL work?*

XVCL works on the principle of constructing custom systems by composing generic, reusable x-frames, after possible adaptations. Any location or structure in an x-frame can be a designated variation point, available for adaptation by ancestor x-frames. This "composition with adaptation" process turns x-frames into concrete components of the custom system we wish to build. Program generation rules are 100% transparent to a programmer, who can fine-tune and re-generate code without losing prior customizations. x-frames can evolve as needed without ever forcing retrofits. Usually, from a small number of x-frames we can generate many concrete

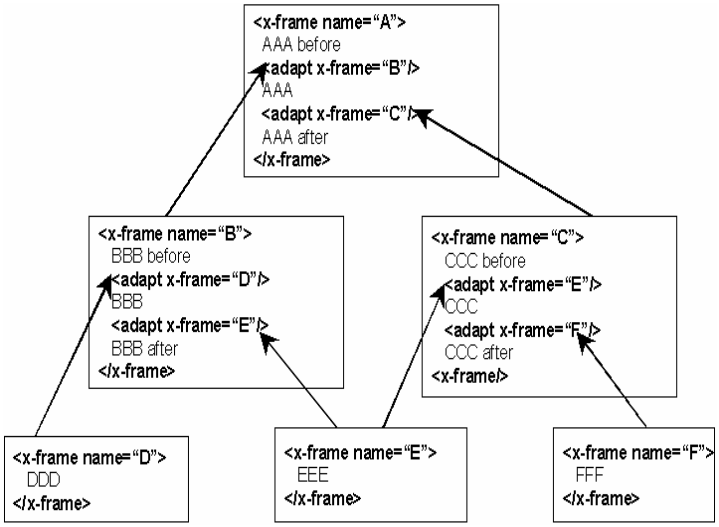

Fig. 2. An example of an x-framework and SPC.

components that differ in various characteristics such as functional requirements or design decisions.

An x-frame is an XML [19] file with components' code or architectural elements (such as groups of components or interfaces), instrumented with XVCL commands for ease of change and evolution. XVCL commands, designed as XML tags, allow the composition of the x-frames, selection of pre-defined options based on certain conditions, etc. Meta-variables and expressions provide a parameterization mechanism. Values of meta-variables are propagated across x-frames and meta-variable scoping rules enhance genericity and reuse.

Customization of an x-framework is supported by the XVCL Processor. The XVCL Processor traverses an x-framework, interprets XVCL commands embedded in visited x-frames, and emits a custom program into one or more files. In our example, the output is emitted to a single file. The Processor's traversal order is dictated by <adapt> commands embedded in x-frames. The <adapt> command tells the Processor to customize and include the specified x-frame. This customization process of the x-framework is directed by instructions contained in the specification x-frame, called SPC for short.

We shall now illustrate the customization process. In the example of Fig. 2, x-frame A is SPC and x-frames B, C, D and E form the x-framework:

- x-frame A adapts x-frames B and C,
- x-frame B adapts D and E,
- x-frame C adapts D and F.

The faded text starting with capital letters (e.g., AAA, BBB, etc.) in boxes of Fig. 2 represents any program code contained in the respective x-frame.

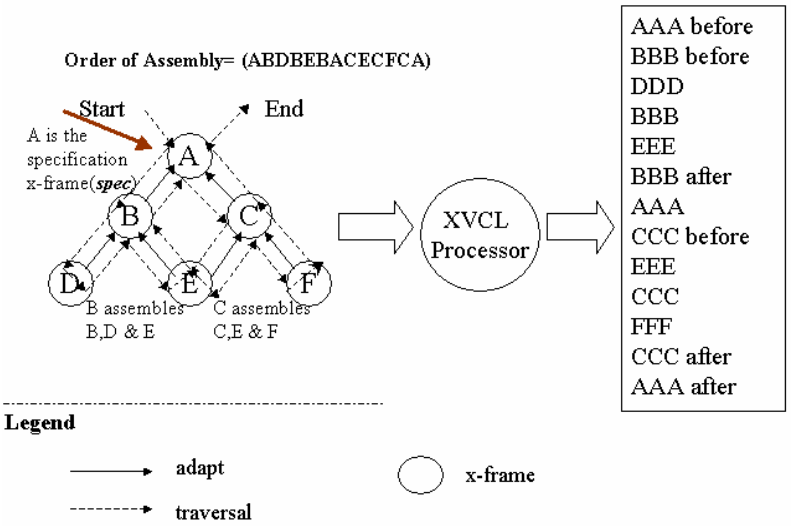

Fig. 3. An example of x-framework processing.

Figure 3 shows the traversal path of the XVCL Processor for the example shown in Fig. 2. The Processor starts with the top-most x-frame, in our case it is an x-frame A. The Processor reads the contents of x-frame A from the top to the bottom, emitting any program code and interpreting any XVCL commands encountered on the way. Having emitted "AAA before", the Processor encounters command <adapt x-frame = "B">, suspends processing of x-frame A, and starts reading x-frame B. Having emitted "BBB before", the Processor encounters command <adapt x-frame = "D">, suspends processing of x-frame B, and starts reading x-frame D. Once the processing of a given x-frame is completed, the Processor returns to interpret the parent x-frame, from the point where the processing was suspended.

The complete result of processing the x-framework of Fig. 2 is shown on the right-hand side of Fig. 3. What we did not show in the above example, is that <adapt> command may also specify customization commands. In such a case, the specified x-frame is customized before it is processed. We shall discuss this in more detail below.

3.3. *Essential XVCL commands*

We shall now describe a subset of XVCL that is sufficient for the purpose of this paper. We refer the reader to XVCL web site [5] for complete description of XVCL.

<x-frame> command:
<x-frame name = "*name*">
x-frame body: mixture of code and XVCL commands
</x-frame>

An x-frame body contains the program code instrumented for ease of changing with XVCL commands. Attribute ***name*** defines the name of x-frame.

<adapt> command:
<adapt x-frame =“*name*”>
adapt-body: mixture of <insert>, <insert-before>, <insert-after> commands
</adapt>
(or)
<adapt x-frame =“*name*”/>

The <adapt> command instructs the processor to:

- adapt the x-frame defined in “x-frame” attribute and apply the commands listed in the adapt-body,
- include the adapted x-frame into the current x-frame,
- resume processing of the current x-frame.

The x-frame ***name*** may be a character string specifying the name of a concrete x-frame (e.g., “A”) or a reference to a meta-variable whose value specifies x-frame name. Meta-variable reference has a form “@V” (explained later in more details).

The adapt-body may contain a mixture of <insert>, <insert-before> and <insert-after> command. Customization commands may affect any x-frame reached in a sequence of <adapt> commands from a given one. For example, in x-framework of Fig. 1, customization commands specified in an <adapt x-frame=“editor”> may refer not only to x-frame *editor*, but also to x-frames *menu* and *toolbar*.

<break> command:
<break name = “*break-name*”>
break-body
</break>

The <break> command marks a point at which an x-frame can be adapted by ancestor x-frames. The **break-body** defines the default code that may be replaced or extended by <insert>, <insert-before> and <insert-after> commands matching a given <break>. This matching is done by the ***break-name*** which is defined as either character string or meta-variable reference.

<insert> command:
<insert break =“*break-name*”>
insert-body
</insert>

The <insert> command replaces the **break-body** of matching <break> commands with **insert-body**. The matching <break> commands are found below

<insert> in the same x-frame and in all its descendents. The <insert-before> command inserts the **insert-body** before the matching <break> commands. The <insert-after> command inserts the **insert-body** after the matching <break> commands.

The **insert-body** may contain a mixture of code and XVCL commands.

<set> command:

<set var = "*var-name*" value ="*value*"/>

The <set> command is used to define a single-value meta-variable. The <set> command assigns a ***value*** defined in ***value*** attribute to single-value meta-variable ***var-name*** defined in ***var*** attribute. Value is either a string or a reference to a meta-variable.

<set-multi> command:

<set-multi var ="*var-name*" value ="*value1, value2, . . .*"/>

The <set-multi> command is used to define a multi-value meta-variable. The <set-multi> command assigns multiple values (*value1*, *value2*, . . .) define in "value" attribute to a multi-value meta-variable name *var-name* define in ***var*** attribute.

<value-of> command

<value-of expr ="*variable-ref*"/>

The value of the "*variable-ref*" is evaluated and the result is inserted in place of the <value-of> command. Meta-variable reference can be direct or indirect such as:

"@v" — value of meta-variable v (direct reference)
"@@v" — value-of(value-of (v)) (indirect reference)
"@. . . @v" — multi-level indirect reference

Meta-variable scoping rules:

Meta-variable scoping rules are the same for both single-value and multi-value meta-variables. While many x-frames may include <set> commands for the same meta-variable, only <set> commands from one x-frame take effect during each run of XVCL processor through x-frames. The <set> command(s) in the ancestor x-frame takes precedence over <set> commands in its descendent x-frames. That is, once x-frame X sets the value of meta-variable v, <set> commands that define the same meta-variable v in descendent x-frames (if any) visited by the processor will not take effect. However, the subsequent <set> commands in x-frame X can reset the value of meta-variable v. Meta-variable v becomes undefined as soon as the processor returns the processing to the parent x-frame that adapts x-frame X.

Meta-variables become undefined as soon as the processing level rises above the x-frame that effectively set meta-variable values. This makes it possible for other

x-frames to set and use the same meta-variables and prevents the interference among meta-variables used in two different sub-trees in the x-frame hierarchy.

The above scoping rule has important implication on reuse. Lower level x-frames must be generic so that they can be reused in many systems. Such x-frames define default values of meta-variables in their respective <set> commands. However, ancestor x-frames often need to adapt lower level x-frames for reuse in different contexts. Some of the adaptations are done by setting values of meta-variables. Therefore, ancestor x-frames must have a power to override defaults defined in lower level x-frames. Meta-variable scoping rules in XVCL reflect the above thinking.

Definition of <select> command

<select option =“*var-name*”>
select-body: may contain options listed below
</select>
select-body:
<option-undefined> (optional)
option-body
</option-undefined>
<option value = “*value*”> (0 or more)
option-body
</option>
<otherwise> (optional)
option-body
</otherwise>

The <select> command selects from a set of options based on meta-variable “*var-name*” as follows:

<option-undefined> is processed, if the meta-variable “*var-name*” is undefined,
<option> is processed, if value of “*var-name*” matches <option>’s “value”,
<otherwise> is processed, if none of the <option>’s “*value*” is matched.

The option-body may contain a mixture of textual content and XVCL commands.

Definition of <while> command:

<while using-items-in=“*multi-var*”>
while-body
</while>

The <while> command iterates over while-body using the values of ***multi-var*** defined in ***using-items-in*** attribute. The ith iteration uses ith value of the multi-valued meta-variable “multi-var”. Inside the while-body, ***multi-var*** with the ith value can be used as single-value meta-variable. The while-body may contain a mixture of code and XVCL commands.

Description of how we use XVCL to design generic systems would occupy a large part of this paper and would repeat what we have already described in other papers. We refer the reader to the examples of a generic editor [20] and generic design of a buffer class library [8]. Any of those two studies explains the concepts with enough details to get an understanding of how XVCL meta-level constructs work in practice. The reader can also find full documentation (including code) for the two studies at our web site [5]).

4. Understanding an x-Framework

Using XVCL mechanisms described in the last section, we create meta-level structures on top of conventional programs. Here, we are concerned with understanding the properties of meta-level structures rather than properties of actual programs.

An x-framework is analyzed for understanding when it is developed, evolved and reused. In any of those situations, we must understand the internal structure of selected x-frames, as well relationships among x-frames. Here are some examples of scenarios in which we need to understand an x-framework and the information that can be useful to accomplish those scenarios:

Scenario 1: Adding a new x-frame X into an x-framework during x-framework development or evolution.

Information relevant to Scenario 1:

1. Which x-frames can potentially adapt (reuse) x-frame X, directly or indirectly?
2. Which x-frames can be adapted from x-frame X, directly or indirectly?
3. In what ways is an x-frame X related to other x-frames? For example:
   (a) Which meta-variables set in other x-frames are referred to in x-frame X?
   (b) Which meta-variables set in x-frame X are referred to in other x-frames?
4. How general x-frame X should be? Which XVCL commands and meta-variables allow us to realize different types of variability in x-frame X?

Scenario 2: Modifying an x-frame X.

Information relevant to Scenario 2:

5. What is the impact of modifying x-frame X on x-frames adapted from it?
6. What is the impact of modifying x-frame X on x-frames that adapt it?
7. Suppose we modify <set> command that assigns value to a meta-variable x — which references to x can be affected?
8. Suppose we modify <select> command — which commands assign value to a meta-variable that controls a given <select>?

Scenario 3: Reusing an x-frame X

Information relevant to Scenario 3:

9. What are the possible ways to customize x-frame X for reuse?
10. What are the roles of meta-variables that are set or referred to in x-frame X?

11. Which variabilities are addressed in x-frame X and which XVCL commands realize these variabilities?

Scenario 4: Analyzing reuse statistics

Information relevant to Scenario 4:

12. The number of x-frames that can be adapted in an x-framework.
13. The number of x-frames that can be adapted for a given SPC (that is, for a specific run of the XVCL Processor over the x-framework).
14. How many times has a given x-frame been adapted for a given set of SPCs?

We discussed only a small sample of high-level properties of x-frameworks that must be understood in various x-framework usage scenarios. Some of those properties can be directly inferred from an x-framework. But to check other properties — information available from an x-framework must be complemented by the information from external sources such as documentation or expert knowledge.

In the following sections, we describe an x-framework analysis system that collects information in the course of both static and dynamic analysis of an x-framework. By querying this information, developers can easier check x-framework properties such as we discussed above.

5. An Approach to Automating x-Framework Analysis

The rationale for our approach is that much of the information related to high-level x-framework properties can be derived from the x-framework. This information can be further transformed and filtered to provide a useful help for developers in understanding an x-framework. We applied concepts from the earlier project on static program analysis [10] and arrived at our automated x-framework analysis system as follows:

1. We created a conceptual meta-level information model that can be directly computed by automatic static and dynamic analysis of an x-framework.
2. We formulated queries related to x-framework properties. These queries could be automatically answered based on the meta-level information model.
3. We designed an x-frame query language, FQL for short, to specify information (in the form of queries) that could help us infer high-level x-framework properties, such as those discussed in the last section.
4. We defined an x-Framework Knowledge Base (FKB), a repository to store the x-framework information according to a schema defined by the conceptual meta-level information model.
5. We designed a front-end to conduct static and dynamic analysis of the x-framework and to load the information into the FKB.
6. We implemented an interpreter to evaluate queries written in FQL, and display the results.

Here, we give examples of queries that FQL is meant to express and our automated analysis system is meant to answer:

1. Select all the <adapt> commands from all x-frames.
2. Select meta-variables that are modified in x-frame X.
3. Select x-frames that modify value of meta-variable V.
4. Select x-frames that adapt x-frame X.
5. Select XVCL select structures that adapt x-frame X.
6. Select x-frames adapted directly or indirectly from XVCL command at line 10 in x-frame X.
7. Select x-frames containing break point named BP.
8. Select x-frames that adapt x-frame X with modification at break point BP.
9. Select x-frames that set meta-variable v before adapting x-frame X.
10. Select x-frames that use meta-variable v in some <adapt> command.
11. Select x-frames that are adapted directly or indirectly from x-frame X and modify meta-variable V, and make selection (in XVCL <select> command) based on meta-variable v.

Clearly, the answers to the above queries can only partially contribute to checking high-level properties discussed in the last section. However, answering even simple queries for large x-frameworks is difficult and error-prone, as one has to inspect multiple x-frames and emulate XVCL processing to find the result. Therefore, we designed an automated system to support query answering.

6. A Conceptual Meta-Level Information Model for x-Frameworks

We use class diagrams to describe our models. Figure 4 depicts major XVCL commands and their important attributes.

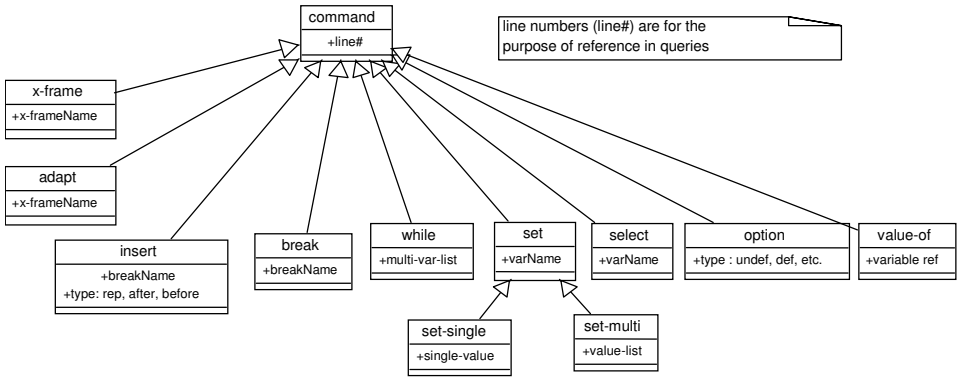

Fig. 4. XVCL command model.

The following relationships among XVCL commands are of interest for querying purpose:

Relationship *Follows* (Fig. 5) depicts the sequential order of commands in an x-frame. Relationship *Follows** is a transitive closure of *Follows*. Relationship *Contains* models direct nesting among commands. Relationship *Contains** models direct or indirect nesting among commands.

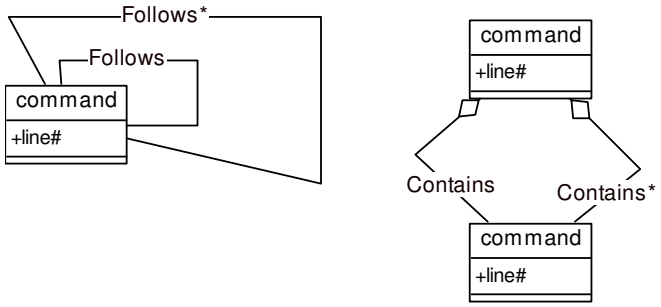

Fig. 5. XVCL command structure model.

Relationship Adapts (x, y) (Fig. 6) holds if x-frame x adapts x-frame y in a processing context under consideration. Notice that x-frame x may contain a command <`adapt x-frame ="y"`> but still this command may not be executed. Relationship Adapts* is a transitive closure of Adapt. Relationships Next and Next* model processing control flow processing among any XVCL commands: Next (c, d) holds if commands c and d are in the same x-frame and command d is executed after command c. Relationship Next* (c, d) holds if command d is executed after command c, independently of whether commands c and d are in the same or in different x-frames.

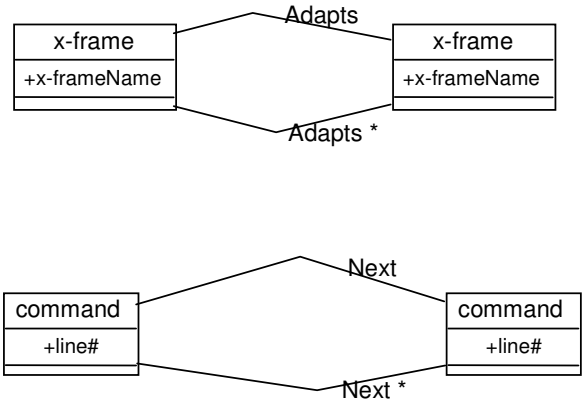

Fig. 6. Command processing order model.

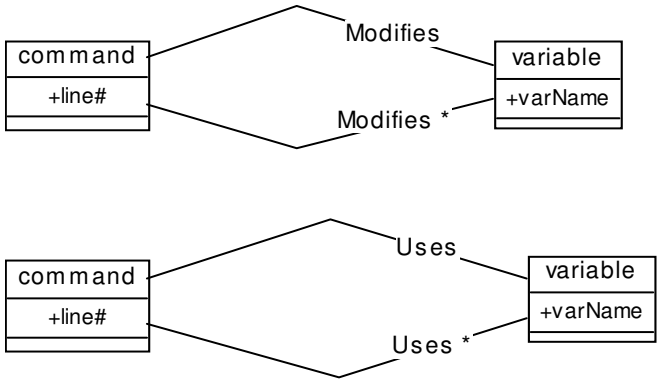

Fig. 7. XVCL meta-variable usage model.

Relationship Modifies (x, v) (Fig. 7) holds if command x directly sets the value of meta-variable v. Relationship Modifies* (x, v) holds if Modifies (x, v) or there exist y such that Adapts* (x, y) and Modifies (y, v). Relationships Uses and Uses* are defined in a similar way.

Relationships Affects and Affects* (Fig. 8) model data flow among meta-variable definition and reference points. Relationship Affects (s, ref) holds if value of meta-variable v assigned in <set> command s can be actually used in reference to this meta-variable ref. Relationship Affects* (s, ref) holds if there is a chain of meta-variable definitions and references, starting at <set> command s and ending at meta-variable reference ref, such that s affects (directly or indirectly) ref.

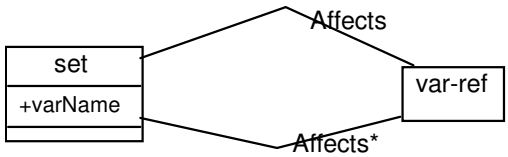

Fig. 8. Data flow model.

7. Querying x-Frameworks in FQL

FQL queries are expressed in terms of x-frame information model described in the previous section. In queries, the reader will see references to model entities (such as command, x-frame, adapt, etc.), attributes (such as x-frame.x-frameName or variable.varName) and entity relationships (such as Adapts (x-frame, x-frame)). Evaluation of a query yields a list of specific instances of model entities that match the query, for example, x-frame named "X", XVCL command found at line number 35 or meta-variable named "V".

In a query, after keyword **Select**, we list model entities we are interested to find or keyword BOOLEAN, in case the result is true or false. We further constrain the results by writing (optional) conditions that the results should satisfy. These conditions include **from**, **with**, and **such that** clauses. In *FQL*, all the keywords are in bold font.

We first explain queries in the simplest form, with **Select** not constrained by any conditions:

Select x-frame

Meaning: this query returns as a result all the x-frames in the x-framework.

Select x-frame.x-frameName

Meaning: returns as a result names of all x-frames in the x-framework

Select <adapt, select>

Meaning: query result may be a tuple. Here the result will contain all possible combinations of all the <adapt> and <select> commands.

The above query examples did not include any conditions. However, in most situations, we wish to select only specific elements, for example, <set> commands that modify a certain meta-variable. We specify properties of elements to be selected in conditions that follow **Select**. Conditions are expressed in terms of:

(a) entity attribute values and constants (**with** clause),
(b) participation of entities in relationships (**such that** clause).

In addition, if we wish to restrict the scope of searching to specific x-frames, we can do so by listing names of those x-frames in the **from** clause. All clauses are optional Clauses **such that** and **with** may occur many times in the same query.

There is an implicit **and** operator between clauses — that means a query result must satisfy the conditions specified in all the clauses.

Declarations introduce synonyms that refer to model entities. Synonyms can be used in the remaining part of the query to mean a corresponding entity. So we can write:

x-frame x;

Select x **such that** Modifies (x, "V") — here we introduce synonym x for x-frame and constraint the result with condition.

A typical query has the following format: Select ... from ... such that ... with ...

Below, we show query examples:

Q1. Select all the adapt commands from all x-frames
Select adapt

Q2. Select all the <adapt> commands from x-frame "CAD"
x-frame x; adapt a;
Select adapt **from** x **with** x.x-frameName="CAD"

or simply: **Select** a **from** "CAD"

Q3. Select all the <insert> commands directly nested within <adapt> commands
Select insert **such that** Contains (adapt, insert)

Q4. Select meta-variables whose values are modified in x-frame X and used in x-frame Y variable v; x-frame x, y;
Select v **such that** Modifies (x, v) **and** Uses (y, v) **with** x.x-frameName = "X" **and** y.x-frameName = "Y"

Q5. Select x-frames that modify value of meta-variable "COLOR"
Select x-frame **such that** Modifies (x-frame, "COLOR")

Q6. Select x-frames that adapt x-frame X directly or indirectly
Select x-frame **such that** Adapts* (x-frame, "X")

Q7. Select all references to meta-variable v affected by <set> command at line number 20 in x-frame "CAD"
var-ref ref;
Select ref **from** "CAD" **such that** Affects ("CAD".20, ref)

8. An x-Framework Analysis System

As XVCL is based on XML [19], XML query languages such as XQL [21], XPath [22] or XQuery [23] can be used to answer certain queries. We started prototyping FQL query evaluator using an XML query language XQL [21] and a public domain XML parser, JAXP from Sun Inc. [24]. We used JAXP parser for parsing XVCL files and extracting information about x-framework. The extracted information included the x-frame structure, adaptation hierarchy of x-frames, breakpoint settings, etc. We stored the extracted information in the XML format. We translated queries written in FQL into equivalent queries written in XQL and then used the XQL query engine to evaluate queries. In this prototype solution, we could address only queries that could be answered by simple search for XML tags representing XVCL commands. However, answering any non-trivial FQL query involves more than that. Some queries need information about relationships among different types of XVCL commands (e.g., as modeled by relationships Follows and Contains). But most importantly, many useful queries cannot be answered based on information computed in the course of static analysis of an x-framework, but require processing of an x-framework.

Consider the following simple query:

Select x-frame **such that** Adapts (x-frame, "X")

To answer this query, it is not enough to search for x-frames containing command <adapt x-frame = "X"/>. We must also consider commands <adapt x-frame = "@V"/> and check possible values of meta-variable V. For this, we have to interpret the x-framework. Meta-variables are heavily used to parameterize x-frameworks: x-frame names in <adapt> or break names in <insert> are most often expressed

in terms of meta-variable references. Also, <value-of> is commonly used to represent class and method names in generic way. Such parameterization is an important technique to achieve reuse. Since XML query languages could not address the above situations, we decided to implement our own x-framework analyzer that conducted both static and dynamic analysis of an x-framework. Our aim was to answer accurately any query written in FQL in terms of models described in Sec. 6.

In our previous project, we designed a Program Query Language, PQL for short, for analysis of static properties of programs written in conventional programming languages [10]. We implemented a generic PQL query evaluator to facilitate reuse of evaluator's components across various source languages. We achieved language-independence by parameterizing the query evaluator with conceptual models of program information such as that described in Sec. 6. To implement a query evaluator for a new source language, we must implement a parser that analyzes source programs, computes program information according to the models, and stores it in a meaningful form in the Program Knowledge Bases (PKB). The PQL query evaluator is driven by tables that store the conceptual model schema, so that changes of the conceptual model do not affect most of the evaluator's components.

The above model-based design of PQL allowed us to easily adapt the generic PQL query evaluator to work with FQL, a query language for x-frameworks.

Figure 9 shows the main components of our x-framework analysis system: An x-Framework Knowledge Base (FKB) stores all the information used for query evaluation, computed by both Static Analyzer (SA) and Dynamic Analyzer (DA). The SA parses an x-framework and stores the information explicitly represented

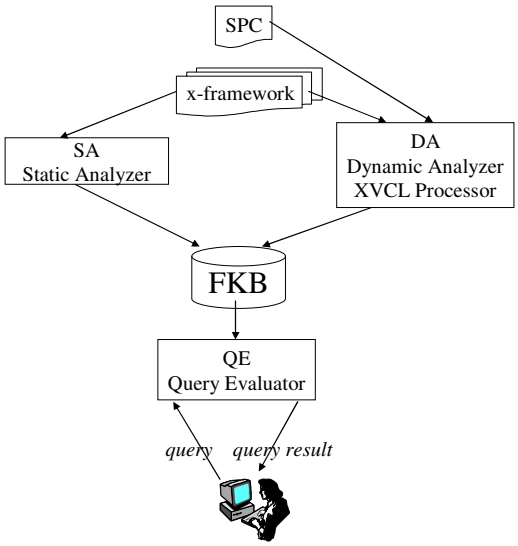

Fig. 9. An overview of the x-framework analysis system.

by XVCL commands embedded in x-frames. The DA processes the x-framework — that is why it must be fed with an SPC, in addition to the x-framework. SPC defines values for meta-variables and other settings that enable processing of the x-framework. It was natural to build DA on top of the XVCL Processor. Our XVCL Processor is implemented in Java™ using a public domain XML parser, JAXP from Sun.

To build the DA, we instrumented the XVCL Processor with extra code to collect x-framework information, according to conceptual models described in Sec. 6. We attached events to XVCL commands and implemented listener interfaces to handle such events. At the time of processing, all the relevant meta-variables already have specific values assigned to them, therefore the queries that require dynamic processing of an x-framework can be answered.

Once SA and DA have completed an x-framework analysis, all the information required to answer queries has already been stored in the FKB. A programmer can now enter a query. The Query Evaluator (QE) first validates the query, checking if all the references to XVCL commands, their attributes and relationships conform to the conceptual meta-level information model. Then, the QE evaluates the query, fetching the information from the FKB via APIs.

Efficient query evaluation is an important concern in the design of the FKB. We experimented with the following four implementation options for the FKB:

1. Data structures such as tables, graphs and trees, stored in memory and/or files. The analyzers SA and DA used API to feed information about an x-framework during analysis, and to access it for query evaluation. These APIs provided abstraction over the physical representation of data structures which allowed us to easily change the representations for better performance or to simplify query evaluation.
2. Mainstream database servers MySQL™ 4.1 and MS SQL™ Server 2000 SP3. We defined FKB schema in an improved 3NF form to reduce redundancy. The information collected during analysis of an x-framework was inserted into database with optimized SQL statements. Our Query Evaluator translated FQL queries into SQL for evaluation by the database query engine.
3. A file-based relational database Microsoft Access 2003™, with FKB schema and SQL used in the same way as above.
4. In-memory relational database HSQLDB [25], an open source in-memory relational database written in Java. HSQLDB runs totally inside memory, eliminating the need for frequent, time-consuming disk access operations.

We conducted experiments with the above media on the same hardware platform and with various data sets (that is, x-frameworks and queries). HSQLDB performance was the best in most cases, except for trivial-size data sets. HSQLDB provides a lightweight Java SQL Engine that accommodates entire data in memory, to minimize disc access during query evaluation. However, the HSQLDB has upper bound for database size (400 Mb for version 1.7.3). As FKB size is typically

x-framework size	Number of XVCL commands	Time to build FKB (in seconds)
Small	238	1.434
Medium	1,108	2.656
Large	3,720	13.359

Fig. 10. Time to build FKB for three x-frameworks.

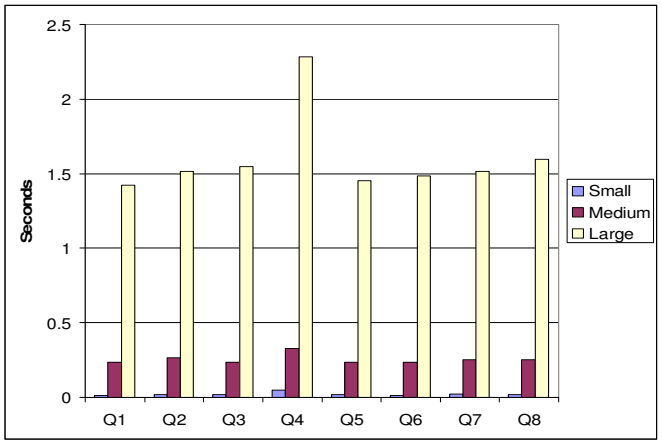

Fig. 11. Query evaluation time for three x-frameworks.

between few KB and 20 MB, this limitation was acceptable to us and we selected HSQLDB as a medium for the FKB in our project.

We evaluated the performance of our Query Evaluator for HSQLDB implementation of FKB on a Pentium IV 2.6 GHz, 1 GB RAM, Windows XP SP2 platform. We measured time to conduct static and dynamic analysis of x-frameworks, and to build FKBs. In Fig. 10, we show the elapsed time to build FKB for small, medium and large x-frameworks. As the analysis time depends mostly on the number of XVCL commands to be processed, we characterized the size of an x-framework in terms of the number of XVCL commands contained in the x-frames rather than in terms of lines of code. Figure 11 shows the elapsed time to evaluate the eight queries (given in Appendix B) for FKBs built for each of the three x-frameworks.

Both FKB build time and query processing time radically increase as the size of an x-framework grows. Query processing time has also to do with the complexity of a query and special characteristics of an x-framework (such as the number of <adapt> commands with many <insert> commands). We found the performance acceptable and in this implementation we did not perform any specialized optimizations during FQL-to-SQL translation, relying on the SQL engine to optimize queries.

9. Conclusions

Meta-programs are generic, incomplete, adaptable programs. Flexible manipulation of programs and genericity, central in software reuse, are the strength of meta-programming. But a meta-program — being a generic representation of a class of programs — is bound to be more difficult to describe and understand than a concrete program. In this paper, we addressed the problem of analysis of meta-programs for understanding, focusing on a specific meta-programming system called XVCL. First, we considered static program analysis methods, developed in analysis of conventional programs. Having found some limitations of such methods in the context of analysis of meta-programs, we also considered dynamic analysis methods. We developed an automated system that analyzes an x-framework (i.e., a meta-program in XVCL), and presents developers with information that helps them work with an x-framework in a more effective way. Our system conducts both static and dynamic analysis of an x-framework. An integral part of our solution is an x-framework query language, called FQL, in which we can formulate questions about x-framework properties. An FQL query processor automatically answers queries. We designed FQL based on our earlier work on PQL, a query language for conventional programs [10]. In the paper, we described FQL as well as the design of our x-framework analysis system.

We believe information that can be acquired in a question-answer session can help developers understand higher-level x-framework properties. FQL queries are meant to bridge the gap between detailed information that can be directly observed at the x-framework level, and higher-level properties of an x-framework that are directly related to various tasks developers need accomplish when enhancing or reusing an x-framework. We developed requirements for a class of queries to be supported by FQL based on our own work with XVCL and inputs from our partners who applied similar systems on the large, industrial scale [16]. However, we cannot yet provide an extensive empirical evidence of how well our analysis tool can support challenging tasks of x-framework development, evolution and reuse. Our tool is still immature and can be used only in our internal projects. While our initial experiences are positive, we are also aware of many ways in which we can improve our system.

Our immediate goal is to integrate the x-framework analysis system into a Smart Graphical Editor (SGE), an x-framework visualization tool developed in our lab. The intention is to display query results using SGE's advanced presentation features, and also to allow developers to seamlessly ask questions about an x-framework while browsing it in SGE windows. Further, we plan to integrate FQL into the XVCL debugger. All these tools will be components of an XVCL Workbench, an IDE for XVCL meta-programming, being developed in our lab. We also plan to incorporate specialized optimizations during FQL-to-SQL translation to reduce the time to evaluate complex queries for large x-frameworks. As a long-term plan, we would like to discover higher-level conceptual models of x-frameworks that would

allow us to better link automatically computed information to typical scenarios for x-framework development, evolution and reuse.

We believe meta-programming techniques have much potential to improve software development practice. Problems of designing generic, adaptable, easy to maintain and reuse programs are difficult to address with conventional programming approaches. Some of these problems can be tackled easier at the meta-level, where we are not restricted by the rules of the implementation language. While extending programs in useful ways, meta-level structures remain fully integrated with the base code. Such solutions are easier to adopt in the industrial practice than powerful abstractions that are disconnected from code. We believe that methods and tools for understanding and debugging of meta-programs are essential for wider acceptance of meta-programming as an effective approach to software development.

Acknowledgments

This work was supported by NUS Research Grant R-252-000-211-112.

References

1. CPP GNU C preprocessor manual, http://gcc.gnu.org/onlinedocs/cpp/
2. M4 http://www.gnu.org/software/m4/manual/m4.html
3. K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, Reading, MA, 2000.
4. T. W. Wong, S. Jarzabek, M. S. Soe, R. Shen and H. Zhang, XML Implementation of frame processor, in *Proc. Symposium on Software Reusability, SSR'01*, Toronto, Canada, May 2001, pp. 164–172.
5. XVCL (XML-based Variant Configuration Language) method and tool for managing software changes during evolution and reuse, http://fxvcl.sourceforge.net, 2005.
6. P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002.
7. H. A. Basit, D. C. Rajapakse and S. Jarzabek, Beyond templates: A study of clones in the STL and some general implications, in *Proc. Int. Conf. Software Engineering, ICSE'05*, St. Louis, MI, May 2005, pp. 451–445.
8. S. Jarzabek and L. Shubiao, Eliminating redundancies with a "composition with adaptation" meta-programming technique, in *Proc. ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, pp. 237–246.
9. U. Pettersson and S. Jarzabek, Industrial experience with building a web portal product line using a lightweight, reactive approach, accepted for ESEC-FSE'05, *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, Lisbon, 2005.
10. S. Jarzabek, Design of flexible static program analyzers with PQL, *IEEE Trans. on Software Engineering*, March 1998, pp. 197–215.
11. R. Garcia *et al.*, A comparative study of language support for generic programming, in *Proc. 18th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*, 2003, pp. 115–134.
12. D. Parnas, On the criteria to be used in decomposing software into modules, *Communications of the ACM* **15**(12) (1972) 1053–1058.

13. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns — Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
14. C. Marcos, M. Campo and A. Pirotte, Rectifying design patterns as meta-level constructs, *Electronic Journal of SADIO* **2**(1) (1999) 17–29.
15. A. Karhinen, A. Ran and T. Tallgren, Configuring designs for reuse, in *Proc. Int. Conf. Software Engineering, ICSE '97*, Boston, MA, pp. 701–710.
16. P. Bassett, *Framing Software Reuse — Lessons from Real World*, Yourdon Press, Prentice Hall, 1997.
17. M. Ernst, G. Badros and D. Notkin, An empirical analysis of C preprocessor use, *IEEE Trans. on Software Engineering*, December 2002, pp. 1146–1170.
18. S. Jarzabek, P. Basset, H. Zhang and W. Zhang, XVCL: XML-based variant configuration language, in *Proc. Int. Conf. on Software Engineering, ICSE '03*, May 2003, Portland, pp. 810–811.
19. XML Extensible Markup Language http://java.sun.com/xml/
20. M. S. Soe, H. Zhang and S. Jarzabek, XVCL: A tutorial, in *Proc. 14th Int. Conf. on Software Engineering and Knowledge Engineering, SEKE '02*, ACM Press, Italy, 2002, pp. 341–349.
21. XQL XML Query Language http://www.w3.org/TandS/QL/QL98/pp/xql.html
22. XPath XML Path Language http://www.w3.org/TR/xpath
23. XQuery, http://www.w3.org/TR/xquery
24. Sun Microsystems. SUN Java Technology and XML. http://java.sun.com/xml/
25. HSQLDB http://www.hsqldb.org/

Appendix A. FQL Grammar

Meta symbols:
[a] — optional (0 or 1 occurrences of a)
a* — repetition 0 or more times of a
a+ — repetition 1 or more times of a

Lexical rules:
LETTER : A-Z | a-z — capital or small letter
DIGIT : 0-9
IDENT : LETTER (LETTER | DIGIT | '-' | '_')*
INTEGER ::= DIGIT+
STRING ::= ' "' (LETTER | DIGIT)* ' "'
CONST ::= STRING | INTEGER

Auxiliary grammar rules:
tuple ::= elem | '<' elem (',' elem)* '>'
elem ::= synonym | attrRef
synonym, attrName: IDENT
entRef ::= synonym | '_' | STRING | lineRef
lineRef ::= INTEGER | STRING.INTEGER | synonym.INTEGER

Comment on line numbers: entRef are references to XVCL entities defined in models. The entRef's appear as relationship arguments in conditions in **such that** clauses. Whenever it makes sense (according to models) and does not lead to

ambiguities, line number (lineRef) can be used as arguments of relationships to denote a corresponding x-structure.

Examples:
"CAD".10 means line# 10 in x-frame named CAD
xf.10 means line# 10 in x-frame denoted by synonym xf

Grammar rules:
select-cl ::= declaration* **Select** result [from-cl] (with-cl | suchthat-cl | pattern-cl)*
declaration ::= IDENT synonym (',' synonym)* ';'
result ::= tuple | 'BOOLEAN'
from-cl ::= **from** x-frame-list [**down**]
x-frame-list ::= x-frame-ref [',' x-frame-ref]
x-frame-ref ::= STRING | synonym
synonym-list ::= synonym [',' synonym]

Comment on from clause: **from** clause specifies x-frames to be searched. x-frames are referenced (x-frame-ref) by name (STRING) or by synonym of type x-frame. The **from** clause can be further qualified by **down**. Qualifier **down** means we should start searching at a given x-frame and then search all the x-frames down from there (along the adapt link) in the x-framework

with-cl ::= **with** attrCond
suchthat-cl ::= **such that** relCond
attrCond ::= attrCompare ('**and**' attrCompare)*
attrCompare ::= attrRef COMP-OPER ref
COMP-OPER ::= '=' | '<' | '>' | '≠'
attrRef ::= synonym '.' attrName
ref ::= CONST | attrRef
relCond ::= relRef ('**and**' relRef)*
relRef ::= all the relationships in the XVCL models

Appendix B. Queries Used for Performance Evaluation

1. Select all the adapt commands from all x-frames
   adapt a;
   Select a
2. Select all the <insert> commands directly nested within <adapt> commands
   insert i; adapt a;
   Select i **such that** Contains (a, i)
3. Select variables whose values are modified in x-frame X
   variable v; x-frame x, command c;
   Select v **such that** Modifies (c, v) and Contains (x, c) **with** x.name='X'
4. Select x-frames that modify value of variable COLOR
   x-frame x; variable v; command c;

Select x **such that** Modifies (c, v) and Contains (x, c) **with** v.name='COLOR'

5. Select x-frames that adapt x-frame X

x-frame x1; x-frame x2;
Select x1 **such that** Adapts (x1, x2) **with** x2.name='X'

6. Select x-frames adapted directly or indirectly from XVCL command at line 10 in x-frame X

x-frame x1; x-frame x2; command c;
Select x1 **such that** Adapts* (c, x1) **with** c.line_no=10 **such that** Contains (x2, c) **with** x2.name='X'

7. Select x-frames containing a break point B

break b; x-frame x;
Select x **such that** Contains (x, b)

8. Select x-frames that set variable v before adapting x-frame X

x-frame x1; variable v; x-frame x2; set s; adapt a;
Select x1 **such that** Contains (x1, s) **and** Modifies (s, v) **and** Contains (x1, a) **and** Adapts (a, x2) **with** x2.name='X' **such that** Follows (a, s)