Design and Implementation of a UML-Based Design Repository*

Rudolf K. Keller, Jean-François Bédard, and Guy Saint-Denis

Département d'informatique et de recherche opérationnelle
Université de Montréal
C.P. 6128, succursale Centre-ville
Montréal (Québec) H3C 3J7, Canada
{keller, bedardje, stdenisg}@iro.umontreal.ca
http://www.iro.umontreal.ca/~{keller, bedardje, stdenisg}

Abstract. The aim of this paper is to present the SPOOL design repository, which is the foundation of the SPOOL software engineering environment. The SPOOL design repository is a practical implementation of the UML metamodel, and is used to store detailed design-level information that is extracted from the source code of industrial systems. Its internal mechanisms and related tools provide functionalities for querying data and observing dependencies between the components of the studied systems, facilitating core tasks conducted in reverse engineering, system comprehension, system analysis, and reengineering. This paper discusses the architecture, the schema, the mechanisms, and the implementation details of the repository, and examines the choice of the UML metamodel. Experiences conducted with large-scale systems are also presented, along with related work and future avenues in design repository research.

Keywords. Design repository, Unified Modeling Language, data interchange, reverse engineering, system analysis, system comprehension, reengineering, system visualization.

1 Introduction

In information systems engineering and software engineering alike, repositories play an important role. Repositories at the design level, henceforth referred to as *design repositories*, are key to capture and manage data in domains as diverse as corporate memory management [10], knowledge engineering [15], and software development and maintenance. In this paper, we report on our experience in designing and

* This research was supported by the SPOOL project organized by CSER (Consortium for Software Engineering Research) which is funded by Bell Canada, NSERC (Natural Sciences and Engineering Research Council of Canada), and NRC (National Research Council Canada).

K.R. Dittrich, A. Geppert, M.C. Norrie (Eds.): CAiSE 2001, LNCS 2068, pp. 448–464, 2001.

implementing a design repository in this latter domain, that is, in the realm of system comprehension, analysis, and evolution.

In order to understand, assess, and maintain software systems, it is essential to represent the analyzed systems at a high level of abstraction such as the analysis and design level. End user tools need access to this information, and thus, a design repository for storing the analyzed systems is required. Such a design repository should meet a number of requirements. First, it should be designed such that its schema will be resilient to change, adaptation, and extension, in order to address and accommodate easily new research projects. Second, it should preferably adopt a schema based on a standard metamodel, and offer extensibility mechanisms to cope with language-specific constructs. Third, in order to enable easy information interchange with other third-party tools, a flexible model interchange format should be used for importing and exporting purposes. Finally, the design of the repository should take into account scalability and performance considerations.

In the SPOOL project (*Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems*), a joint industry/university collaboration between the software quality assessment team of Bell Canada and the GELO group at Université de Montréal, we are investigating methods and tools for design composition [13] and for the comprehension and assessment of software design quality [14]. As part of the project, we have developed the SPOOL environment for reverse engineering, system comprehension, system analysis, and reengineering.

At the core of the SPOOL environment is the SPOOL design repository, which we designed with the four above-mentioned requirements in mind. The repository consists of the *repository schema* and the *physical data store*. The repository schema is an object-oriented class hierarchy that defines the structure and the behavior of the objects that are part of the reverse engineered *source code models*, the *abstract design components* that are to be identified from the source code, the *implemented design components*, and the *recovered and re-organized design models*. Moreover, the schema provides for more complex behavioral mechanisms that are applied throughout the schema classes, which includes uniform traversal of complex objects to retrieve contained objects, notification to the views on changes in the repository, and dependency accumulation to improve access performance to aggregated information. The schema of the design repository is based on an extended version of the *UML metamodel 1.1* [25]. We adopted the UML metamodel as it captures most of the schema requirements of the research activities of SPOOL. This extended UML metamodel (or SPOOL repository schema) is represented as a *Java 1.1* class hierarchy, in which the classes constitute the data of the *MVC-based* [3] SPOOL environment.

The object-oriented database of the SPOOL repository is implemented using *POET 6.0* [18]. It provides for data persistence, retrieval, consistency, and recovery. Using the precompiler of POET 6.0's *Java Tight Binding*, an object-oriented database representing the SPOOL repository is generated from the SPOOL schema. As POET 6.0 is *ODMG 3.0*-compliant [16], its substitution for another ODMG 3.0-compliant database management system would be accomplishable without major impact on the schema and the end user tools.

To deal with data interchange (importing source code into and exporting model information from the repository), *XMI* [17] technology is used. An import utility reads XMI-compliant files and maps the contained XML structures into objects of the repository's physical model. For exporting purposes, another utility traverses the objects of the repository and produces the corresponding XMI file. The adoption of standard technologies such as the UML and XMI enables easy information interchange between the SPOOL environment and other tools.

In the remainder of this paper, we first provide an overview of the SPOOL environment. Next, we describe the architecture of the SPOOL repository and detail its schema, discussing its top-level, core, relationship, behavior, and extension classes and relating it to the UML metamodel. Then, we describe two of the key mechanisms of the repository, that is, the traversal of complex objects and dependency management, and present one of the front-end user tools of the repository, the *SPOOL Design Browser*. Finally, we put our work into perspective, reporting on performance and interchange experiments, and discussing related and future work.

2 The SPOOL Environment

The SPOOL environment (Figure 1) uses a three-tier architecture to achieve a clear separation of concerns between the end user tools, the schema and the objects of the reverse engineered models, and the persistent datastore. The lowest tier consists of an *object-oriented database management system*, which provides the physical, persistent datastore for the reverse engineered source code models and the design information. The middle tier consists of the *repository schema*, which is an object-oriented schema of the reverse engineered models, comprising structure (classes, attributes, and relationships), behavior (access and manipulation functionality of objects), and mechanisms (higher-level functionality, such as complex object traversal, change notification, and dependency accumulation). We call these two lower tiers the SPOOL *design repository*. The upper tier consists of end user tools implementing domain-specific functionality based on the repository schema, i.e., *source code capturing*, and *visualization and analysis*.

In this section, we will describe the environment’s techniques and tools for source code capturing and data interchange, as well as for visualization and analysis.

2.1 Source Code Capturing and Data Interchange

Source code capturing is the first step within the reverse engineering process. Its goal is to extract an initial model from the source code. At this time, SPOOL supports C++ and uses *Datrix* [6] to *parse* C++ source code files. With the deployment of the Datrix Java parser, SPOOL will soon be able to extend its support for reverse engineering Java source code. Datrix provides complete information on the source code in form of an ASCII-based representation, the *Datrix/TA intermediate format*. The purpose of this intermediate representation is to make the Datrix output independent of the

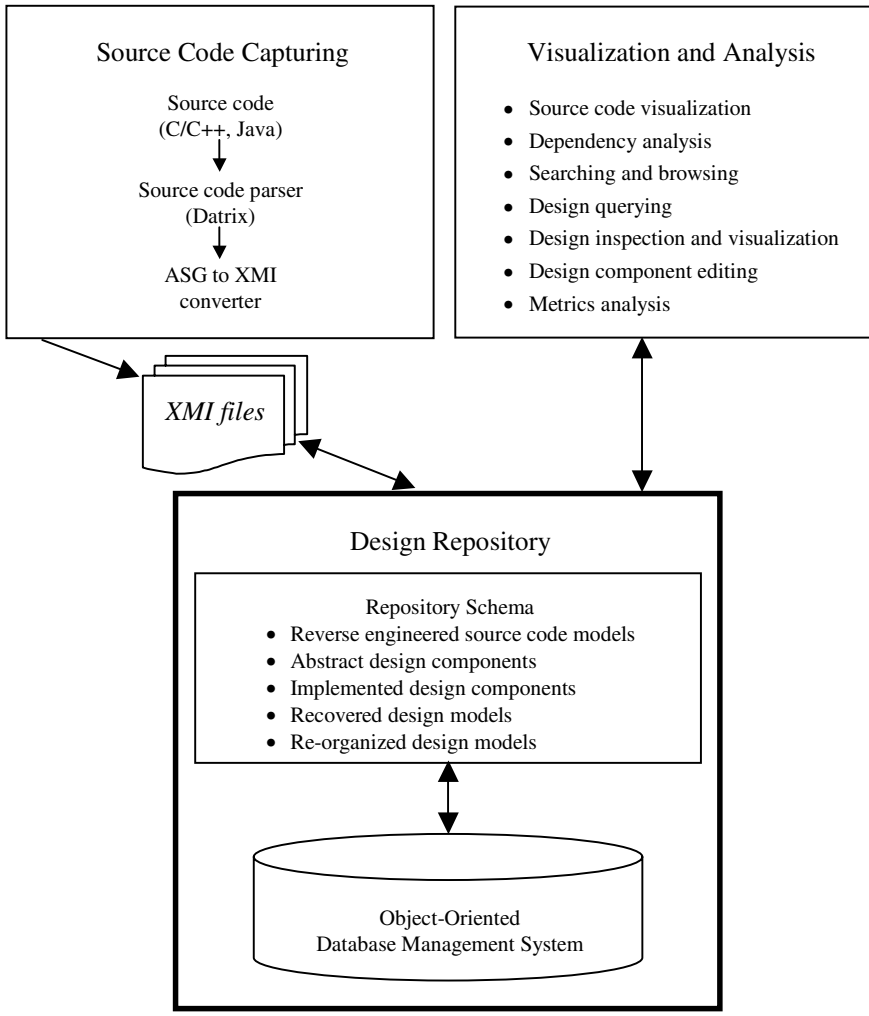

Fig. 1. Overview of the SPOOL environment

programming language being parsed. Moreover, it provides a data export mechanism to analysis and visualization tools, ranging from Bell Canada's suite of software comprehension tools to the SPOOL environment and to third-party source code comprehension tools. A *conversion* utility, built with *ANTLR* [1] and the *Datrix/TA* grammar description, assembles the nodes and arcs of the Datrix/TA source code representation files, applies some transformations to the resulting graphs (such as normalization of directory and file structures as well as addition of primitive data types) to map the Datrix/TA structures to those of the repository model, and generates XMI files. Thereafter, the resulting XMI files are read by an *import* utility, which leverages some components built for the *Argo* project [21] and which uses IBM's

xml4j XML parser [12]. The importer constructs the objects of an initial physical model in the SPOOL repository. Another utility is used to export the content of the repository, translating the structures of the internal schema into a resulting XMI-compliant file. At the current state of development, we capture and manage in the repository the source code information as listed in Table 1.

Table 1. Source code information managed in the SPOOL repository

1.	*Files* (name, directory).
2.	*Classifier – classes, structures, unions, anonymous unions, primitive types* (char, int, float, etc.), *enumerations* [name, file, visibility]. Class declarations are resolved to point to their definitions.
3.	*Generalization* relationships [superclass, subclass, visibility].
4.	*Attributes* [name, type, owner, visibility]. Global and static variables are stored in utility classes (as suggested by the UML), one associated to each file. Variable declarations are resolved to point to their definitions.
5.	*Operations* and *methods* [name, visibility, polymorphic, kind]. Methods are the implementations of operations. Free functions and operators are stored in *utility* classes (as suggested by the UML), one associated to each file. *Kind* stands for *constructor*, *destructor*, *standard*, or *operator*.
5.1	*Parameters* [name, type]. The type is a *classifier*.
5.2	*Return types* [name, type]. The type is a *classifier*.
5.3	*Call actions* [operation, sender, receiver]. The receiver points to the class to which a request (operation) is sent. The sender is the classifier that owns the method of the call action.
5.4	*Create actions*. These represent object instantiations.
5.5	*Variable use* within a method. This set contains all member attributes, parameters, and local attributes used by the method.
6.	*Friendship relationships* between classes and operations.
7.	*Class and function template instantiations*. These are stored as normal *classes* and as *operations* and *methods*, respectively.

2.2 Visualization and Analysis

The purpose of design representation is to provide for the interactive visualization and analysis of source code models, abstract design components, and implemented components. It is our contention that only the interplay among human cognition, automatic information matching and filtering, visual representations, and flexible visual transformations can lead to the all-important why behind the key design decisions in large-scale software systems. To date, we have implemented and integrated tools (for details, see [14]) for

- *Source code visualization*,
- Interactive and incremental *dependency analysis* (see Section 4.2),
- Design investigation by *searching and browsing*, based on both structure and full-text retrieval, using the *SPOOL Design Browser* (see Section 4.3),
- *Design querying* to classes that collaborate to solve a given problem,

- *Design inspection and visualization* within the context of the reverse engineered source code models,
- *Design component editing*, allowing for the interactive description of design components, and
- *Metrics analysis* to conduct quantitative analyses on desirable and undesirable source and design properties.

3 Repository Architecture and Schema

The major architectural design goal for the SPOOL repository was to make the schema resilient to change, adaptation, and extension, in order to address and accommodate easily new research projects. To achieve a high degree of flexibility, we decided to shield the implementation of the design repository completely from the client code that implements the tools for analysis and visualization. The retrieval and manipulation of objects in the design repository is accomplished via a hierarchy of public Java interfaces, and instantiations and initializations are implemented via an *Abstract Factory* [9].

The schema of the SPOOL repository is an object-oriented class hierarchy whose core structure is adopted from the UML metamodel. Being a metamodel for software analysis and design, the UML provides a well-thought foundation for SPOOL as a design comprehension environment. However, SPOOL reverse engineering starts with source code from which design information should be derived. This necessitates extensions to the UML metamodel in order to cover the programming language level as far as it is relevant for design recovery and analysis. In this section, we present the structure of the extended UML metamodel that serves as the schema of the SPOOL repository. This includes the top-level classes, the core classes, the relationship classes, the behavior classes, and the extension classes.

3.1 Top-Level Classes

The top-level classes of the SPOOL environment prescribe a key architectural design decision, which is based on the *Model/View/Controller (MVC)* paradigm of software engineering [3, 9]. MVC suggests a separation of the classes that implement the end user tools (the views) from the classes that define the underlying data (the models). This allows for both views and models to be reused independently. Furthermore, MVC provides for a change notification mechanism based on the *Observer* design pattern [9]. The Observer pattern allows tools, be they interactive analysis or background data processing tools, to react spontaneously to the changes of objects that are shared among several tools. In SPOOL, the classes *Element*, *ModelElement*, and *ViewElement* implement the functionality that breaks the SPOOL environment apart into a class hierarchy for end user tools (subclasses of *ViewElement*) and a class hierarchy for the repository (subclasses of *ModelElement*). The root class *Element*

prescribes the MVC based communication mechanism between ViewElements and ModelElements.

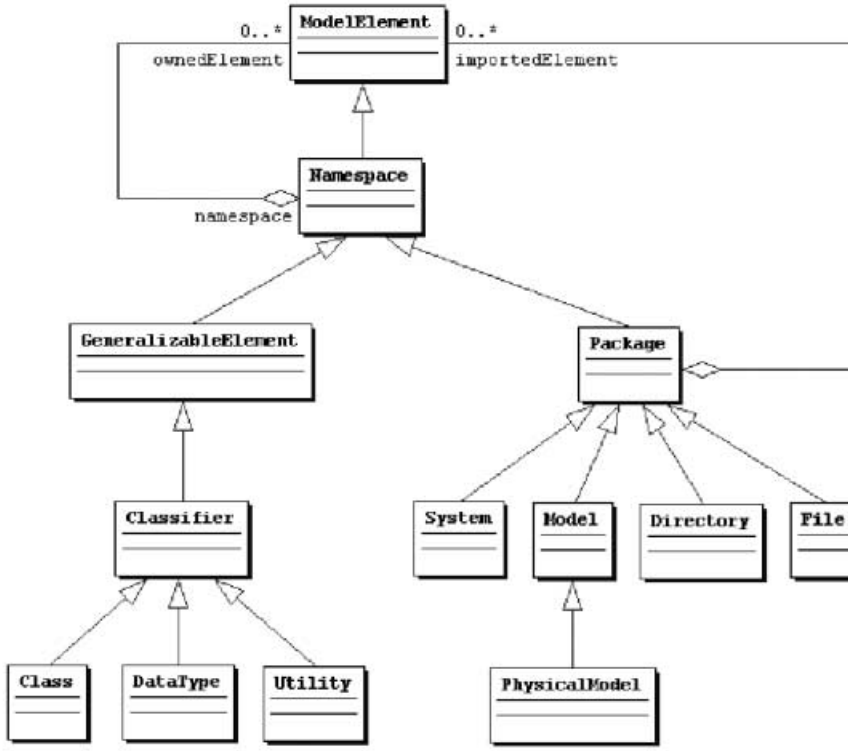

Fig. 2. SPOOL repository schema: Core classes

3.2 Core Classes

The core classes of the SPOOL repository schema adhere to a large extent to the classes defined in the core and model management packages of the UML metamodel. These classes define the basic structure and the containment hierarchy of the ModelElements managed in the repository (see Figure 2). At the center of the core classes is the *Namespace* class, which owns a collection of ModelElements. A *GeneralizableElement* defines the nodes involved in a generalization relationship, such as inheritance. A *Classifier* provides *Features*, which may be structural (*Attributes*) or behavioral (*Operations* and *Methods*) in nature. A *Package* is a means of clustering ModelElements.

3.3 Relationship Classes

"A relationship is a connection among model elements." [25] The UML introduces the notion of Relationship as a superclass of Generalization, Dependency, Flow, and Association for reasons of convenience, so that tools can refer to any connections among ModelElements based on the same supertype (for details, see [24]).

3.4 Behavior Classes

The behavior classes of the SPOOL repository implement the dynamics of the reverse engineered system. It is important to understand that the UML metamodel takes a forward engineering perspective and focuses on software analysis and design, rather than on the reverse engineering of source code. Therefore, the UML metamodel does not aim to encompass and unify programming language constructs.

The purpose of analysis and design is to specify what to do and how to do it, but it is the later stage of implementation in which the missing parts of a specification are filled to transform it into an executable system. However, the UML is comprehensive in that it provides a semantic foundation for the modeling of any specifics of a model. For example, the UML suggests State Machine diagrams (similar to Harel's Statechart formalism [11]) to specify the behavior of complex methods, operations, or classes. To cite another example, collaboration diagrams can be used to specify how different classes or certain parts of classes (that is, roles) have to interact with each other in order to solve a problem that transcends the boundaries of single classes.

In SPOOL, we look at a system from the opposite viewpoint, that is from the complete source code, and the goal is to derive these behavior specification models to get an improved understanding of the complex relationships among a system's constituents. For this purpose, we included in the SPOOL repository the key constructs of the behavior package of the UML metamodel, including the UML's Action and Collaboration classes. However, we modified certain parts to reduce space consumption and improve performance. For more information about the behavior classes of the SPOOL repository, see [24].

3.5 Extension Classes

The UML metamodel suggests two approaches to metamodel extension; one is based on the concept of TaggedValues and the other on the concept of Stereotypes. In SPOOL, we have only implemented the former approach since Stereotypes as defined in the UML metamodel would not scale to meet the performance requirements of the SPOOL repository.

4 Repository Mechanisms and Front-End

To be usable and reusable as the backend for a diverse set of interactive reverse engineering tools, the SPOOL repository implements a number of advanced mechanisms. The *traversal mechanism* defines how to retrieve objects of certain types from a complex object containment hierarchy. The *dependency mechanism* allows for compression of the vast amount of dependencies among ModelElements for fast retrieval and visualization. Finally, as a front-end to the repository and an interface to other SPOOL visualization tools, the *SPOOL Design Browser* is provided.

4.1 Traversal Mechanism

In SPOOL, the Namespace serves as a container for a group of ModelElements (Figure 2). Consequently, it defines methods that traverse complex object structures and retrieve ModelElements of a given type. For example, to identify all classes of a system, all files in all subdirectories of the system at hand must be checked for instances of the metatype Class. Unlike the objects in text-based repositories [8, 27], the objects in SPOOL's object-oriented database are typed and can be queried according to their types. SPOOL allows for the identification of the type of an object merely by using the Java *instanceof* operator or the reflective *isInstance* operation of the Java class *Class*. Hence, metaclass types can be provided as parameters to the retrieval methods of Namespace, which then recursively traverse the containment hierarchy of the namespace at hand and examine each ModelElement whether it is an instance of that type. If this is the case, the ModelElement is added to a return set, which is passed through the recursive traversal.

4.2 Accumulated Dependency Mechanism

An important requirement of the SPOOL repository is to provide information on dependencies between any pair of ModelElements within interactive response time. A straightforward approach to identify dependencies among ModelElements would be the traversal of the whole object structure at run-time. However, applied to reverse engineered software with directories that contain hundreds of files, this approach would require batch processing. A radically different approach would be to store each and every dependency among ModelElements as separate dependency objects, which would result in an unmanageable amount of dependency data. Hence, the solution that we adopted in SPOOL constitutes a trade-off between run-time efficiency and space consumption.

In SPOOL, we capture and accumulate dependencies at the level of Classifiers (for instance, classes, unions, or utilities). Accumulation refers to the fact that we store for each dependency its types together with the total number of primitive Connections on which each type is based. Given a pair of dependent Classifiers, we generate a so-

called *AccumulatedDependency* object, which captures this information for the dependencies in the two directions. To be able to identify dependencies between higher-level namespaces, such as directories, files, or packages, without much lag time, we store the union of all AccumulatedDependencies of all contained Classifiers of a given Namespace redundantly with the Namespace. Hence, if we want to identify, for example, dependencies between the directories *Directory1* and *Directory2*, we only need to iterate over the set of AccumulatedDependencies of *Directory1* and look up for each element of the set whether the ModelElement at the other end of the element at hand (that is, the one which is not contained in the Namespace under consideration) has as one of its parent namespaces *Directory2*.

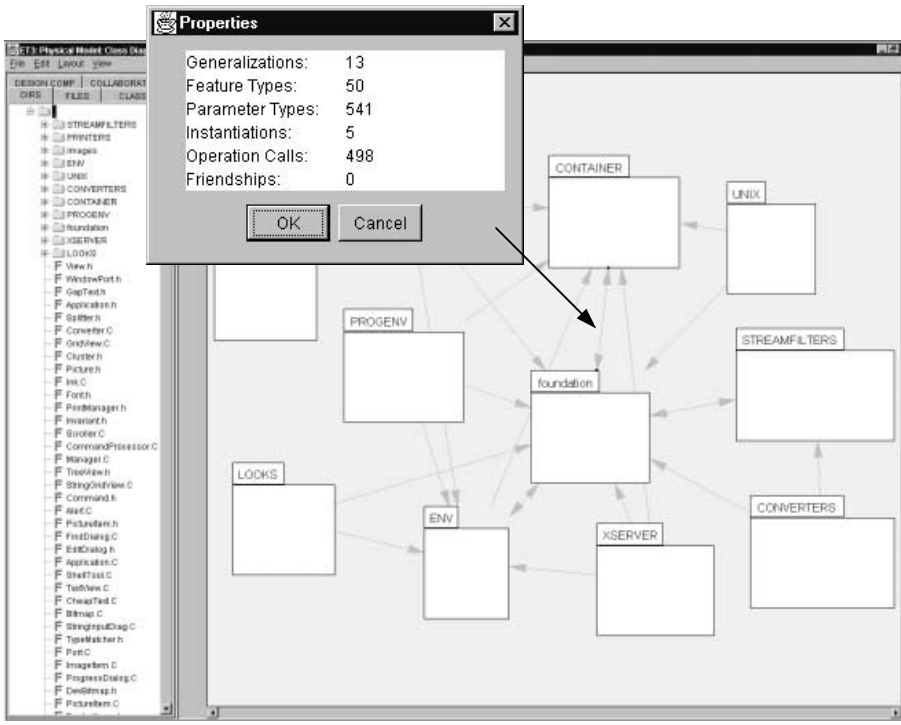

Fig. 3. SPOOL dependency diagram with dialog box for inspection of properties

Figure 3 shows a dependency diagram for the top-level directories of the system ET++ [26]. A property dialog box can be opened to inspect the nature of a specific dependency. In Figure 3, for instance, the dependency between the directories *CONTAINER* and *foundation* includes 13 generalization connections, 50 feature type connections (types of attributes and return types of operations and methods), 541 parameter type connections, 5 class instantiation connections (*CreateAction*), 498 operation call connections (*CallAction*), and 0 friendship connection. On demand, the dialog can also be invoked for each direction of a dependency. For more information about the accumulated dependency mechanism, see [24].

4.3 SPOOL Design Browser

The SPOOL environment provides a number of tools for design investigation and visualization. Among these is the *Design Browser*, which acts as a standard query engine to support design navigation. The SPOOL Design Browser offers predefined queries to the user, while making it possible to modify the set of predefined queries based on the traversal mechanism provided by the repository. It also manages the execution of queries, and displays the query results in a user-friendly way. For details on the Design Browser, refer to [22].

5 Experience and Perspectives

In this section, we first report on our experience with the SPOOL repository with respect to scalability and performance. Then, we discuss data interchange between the repository and external tools. Furthermore, we examine our choice of the UML metamodel and wrap up with a conclusion and a discussion of future work.

5.1 Scalability and Performance

Scalability and performance are critical for the success of source code investigation. Each step in the investigation process should be fast enough in order to avoid confusion and disorientation with the user, and the tools should be robust enough to accommodate industrial sized systems. In the following, we present two anecdotal experiments in which the performance of SPOOL queries was measured. The experiments were conducted on a 350MHz Pentium II machine with 256Mb of RAM running Windows NT 4.0. Two industrial C++ software systems were analyzed: *ET++ 3.0* [26], a well-known application framework, and *System A*, a large-scale system from the telecommunications domain provided by Bell Canada (for confidentiality reasons, we cannot disclose the real name of the system). The size metrics of these systems are shown in Table 2 (top section).

The first experiment consisted of measuring the times needed to execute a simple query which is predefined in the Design Browser but which is not directly supported by the repository schema, namely, the retrieval of all the *ModelElements* (directories, files, classes, C++ structures, C++ unions, C++ enumerations, operations, methods, and attributes) of a system. Table 2 (middle section) depicts the data for this query. The table shows that the first time the query is run, it takes longer (*Duration 1*) because, first, Poet needs to recreate the persistent objects that are stored on disk and, second, when loading a system, SPOOL caches some of the objects in internal hash tables. As soon as an element is “touched” by a query, it becomes available in memory, and the next time a query is accessing it, the execution is much faster (*Duration 2+*).

Table 2. Size metrics and durations of queries for two industrial systems

Size Metrics :	*ET++*	*System A*
Lines of code	70,796	472,824
Lines of pure comments	3,494	60,256
Blank lines	12,892	80,463
# of files (.C/.h)	485	1,900
# of classes (.C/.h)	722	3,103
# of generalizations	466	1,422
# of methods	6,255	17,634
# of attributes	4,460	1,928
Size of the system in the repository	19.3 MB	63.1 MB
# of *ModelElements*	20,868	47,834
Simple Query :	*ET++*	*System A*
Duration 1 (seconds)	22	47
Duration 2+ (seconds)	2	6
Template Method Query :	*ET++*	*System A*
# of occurrences found	371	364
Duration (seconds)	15	360

The above experiment shows that the retrieval of elements that are already referenced in the database is pretty fast. The execution of more complicated queries may take considerably longer. As a second experiment, we measured the time needed to retrieve all occurrences of the *Template Method* pattern [9] in the two systems. This query basically consists of the following five steps:

1. retrieve all classes in the system,
2. for each class, retrieve all methods,
3. for each method, retrieve all call actions,
4. for each call action, get the receivers,
5. for each receiver, look if the call action is defined in the same class and implemented in a subclass.

Table 2 (bottom section) shows the times needed to execute this query for the first time, assuming that all the *ModelElements* are already cached (a query that retrieves all *ModelElements* in the system was executed previously). These numbers are quite good considering that a considerable number of relations must be crossed in order to retrieve the desired information. The time needed to run a particular query may be higher, but these experiments suggest that only the complexities of the query and of the system are susceptible to increase execution time, whereas the access time to the *ModelElements* of the repository is relatively constant (mainly due to the use of hash tables).

5.2 Data Interchange

Before selecting XMI as the model interchange format for the SPOOL design repository, five interchange formats were considered and evaluated [23]. Advantages and disadvantages of the five formats were identified, where XMI came out as the strongest approach, mainly because it reuses existing solutions like the UML, XML, and MOF, has important industry support, and is generally complete. Other useful features that put XMI on top are partial or differential model exchanges, and general extension mechanisms. Working with XMI documents meant that we could benefit from readily available XML tools, components and expertise to develop our model importer, which was written in Java.

At the time of writing, we have completed our first experiments on exchanging XMI files with other software engineering tools such as *Rational Rose* [20]. We achieved good preliminary results, yet further experiments will be required, once more precise mappings between the supported programming languages and the respective repository schema constructs are available.

5.3 Discussion

The UML is hardly accepted in the reverse engineering community. Demeyer et al. have articulated some reasons for the "why not UML" [7]. We wholeheartedly agree that there is a lack of complete and precise mappings of programming languages to the UML. However, we consider this as a challenge for researchers, rather than a reason for abandoning the UML. With its Stereotype and TaggedValue extension mechanisms, the UML does provide constructs to capture the many details of source code written in different programming languages. The issue at hand is to define unambiguously how to map the various UML constructs to source code constructs and to provide tool support for the traceability in both directions. A second argument of Demeyer et al. against the UML is that it "does not include dependencies, such as invocations and accesses" [7]. This reflects a misconception in the software engineering community about the UML. All too often, the UML is looked at as a notation for structure diagrams only, and all other diagrams are rather neglected. Yet, the behavior package of the UML metamodel provides for a precise specification of the method internals. However, a critique against the UML may be that the behavioral package is too heavyweight to be directly applicable to software engineering. It is impossible to generate and store for each method a StateMachine object together with all its internal objects. In SPOOL, we implemented a shortcut solution for the representation of the bulk of the methods. We associated Actions directly to methods instead of generating StateMachines, which consist of Actions that are invoked by Messages. Refer to the UML for further details on the structure of StateMachines [2]. We do, however, allow StateMachines to be reverse engineered and stored for methods or classes of interest.

The UML has several advantages. First, the UML metamodel is well documented and based on well-established terminology. This is of great help to convey the

semantics of the different modeling constructs to tool developers. Second, the metamodel is designed for the domain of software design and analysis, which is at the core of forward engineering and which constitutes the target of the reverse engineering process. The UML introduces constructs at a high level of granularity, enabling the compression of the overwhelming amount of information that makes source code difficult to understand. Third, the UML metamodel is object-oriented, meaning that the structure, the basic access and manipulation behavior, and complex mechanisms can be separated from end user tools and encapsulated in the repository schema. Fourth, the UML defines a notation for the metamodel constructs, thus providing reverse engineering tool builders guidelines for the visual representation of the model elements. Finally, since the UML has gained much popularity in industry and academia alike, tools and utilities supporting the UML and related formalisms such as XMI are becoming readily available. This proves highly beneficial in projects such as ours.

Other research efforts in repository technology include the design of the *Software Information Base (SIB)* and prototype implementation, as described by Constantopoulos and Dörr [4], and Constantopoulos et al. [5]. The SIB, as a repository system, is used to store descriptions of software artefacts and relations between them. Requirement, design, and implementation descriptions provide application, system, and implementation views. These descriptions are linked by relationship objects that express attribution, aggregation, classification, generalization, correspondence, etc. between two or more software components. Links may express semantic or structural relationships, grouping of software artefact descriptions into larger functional units, and even user-defined or informal relationships for hypertext navigation or annotations. The representation language used in the SIB is *Telos* [15], a conceptual modelling language in the family of entity-relationship models with features for increasing its expressiveness. Finally, the SIB comes with a set of visual tools for querying and browsing, which allow the user to search for software component descriptions that match specific criteria expressed as queries, or to navigate through the repository's content in an exploratory way within a given subset of the SIB.

Even if the Software Information Base and the SPOOL design repository share apparent similarities in their architecture and functionality, the SIB is mainly intended for the storage of user-written descriptions of software artefacts residing outside the system (links can be made from the descriptions to the physical components on external storage). The main goal of the SIB is to act as a large encyclopedia of software components, may they be requirements specifications, design descriptions, or class implementations in a specific programming language. These components are classified using a well-defined scheme, in a way that system developers may rapidly browse the contents of the SIB to find the building blocks they need for composing a new system with the help of registered parts. In contrast, the SPOOL repository stores information extracted from source code to help software engineers conduct metrics analysis and investigate the properties of object-oriented systems, such as the presence of design pattern instances and the existence of dependencies between classes, files, or directories. While a prototype of the SIB storing extracted facts from

source code parsing has been developed, the retained facts remain of basic nature and serve as a start-up structure for the manual classification of classes, operations, and attributes. Furthermore, the SIB offers a mechanism to import artefact descriptions into its database; however, no export mechanism or data interchange facility is provided, assuming that information sharing with other tools is not seen as a primary objective. The SPOOL repository, in order to exchange information with other academic and commercial tools, comprises a stable and well-known internal datamodel and data interchange format. While both SPOOL and SIB were carefully designed with strict architectural and performance considerations, the main differences between them on the functionality side are best explained by their focus on reverse engineering and artefact reuse, respectively.

5.4 Conclusion and Future Work

In this paper, we presented the SPOOL design repository, the core part of the SPOOL environment. Based on the UML metamodel, its schema permits to store detailed information about the source code of systems, enabling users to conduct essential tasks of reverse engineering, system comprehension, system analysis, and reengineering. Its internal advanced mechanisms provide the core functionalities needed by the interactive visualization tools of the environment. XMI is used as model interchange format, easing information sharing between the SPOOL design repository and other software engineering environments. Our experience suggests that the choice of the UML and its metamodel was indeed a key factor in meeting the repository requirements stated at the outset of the project.

In our future work on the SPOOL design repository, we will aim to provide complete and precise mappings between the constructs of the UML-based SPOOL repository schema and the four programming languages C, C++, Java, and a proprietary language deployed by Bell Canada. We will also increase the information content of the SPOOL repository in respect to dynamic behavior. As discussed previously, a balance between space consumption and fast response time needs to be sought. One solution that we will investigate is parsing the source code of methods on the fly when querying, for example, control flow information. A third area of work will be to provide Web-based access to the repository, which will allow our project partners to remotely check in source code systems and immediately use SPOOL tools to query and visualize the repository content. Finally, we plan to investigate the UML-based repository approach beyond SPOOL in two other domains of our interest, that is, in corporate memory research [10], and in schema evolution [19].

Acknowledgments

We would like to thank the following organizations for providing us with licenses of their tools, thus assisting us in the development part of our research: *Bell Canada* for the source code parser *Datrix*, *Lucent Technologies* for their C++ source code

analyzer *GEN++* and the layout generators *Dot* and *Neato*, and *TakeFive Software* for their software development environment *SNiFF+*.

References

1. ANTLR, ANother Tool for Language Recognition, 2000. <http://www.antlr.org>.
2. Booch, G., Jacobson, I., and Rumbaugh, J. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
3. Buschmann, F., Meunier, R., Rohnert, H., Somerlad, P., and Stal, M. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley and Sons, 1996.
4. Constantopoulos, P., and Dörr, M. "Component Classification in the Software Information Base", Object-Oriented Software Composition, Oscar Nierstrasz and Dennis Tsichritzis (Eds.), pp. 177-200. Prentice Hall, 1995.
5. Constantopoulos, P., Jarke, M., Mylopoulos, J., and Vassiliou, Y. "The Software Information Base: A Server for Reuse." VLDB Journal 4(1):1-43, 1995.
6. Datrix homepage, 2000, Bell Canada. <http://www.iro.umontreal.ca/labs/gelo/datrix/>.
7. Demeyer, S., Ducasse, S., and Tichelaar, S. "Why unified is not universal: UML shortcomings for coping with round-trip engineering." In *Bernhard Rumpe, editor, Proceedings UML'99 (The Second International Conference on the Unified Modeling Language)*. Springer-Verlag, 1999. LNCS 1723.
8. Finnigan, P. J., Holt, R. C., Kalas, I., Kerr, S., Kontogiannis, K., Müller, H. A., Mylopoulos, J., Perelgut, S. G., Stanley, M., and Wong, K. "The software bookshelf." *IBM Systems Journal*, 36(4):564-593, 1997.
9. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
10. Gerbé, O., Keller, R. K., and Mineau, G. "Conceptual graphs for representing business processes in corporate memories." In *Proceedings of the Sixth International Conference on Conceptual Structures*, pages 401-415, Montpellier, France, August 1998.
11. Harel, D. "On visual formalisms." *Communications of the ACM*, 31(5):514-530, May 1988.
12. IBM-alphaWorks. XML Parser for Java, 2000. <http://www.alphaworks.ibm.com/tech/xml>.
13. Keller, R. K., and Schauer, R. "Design components: towards software composition at the design level." In *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, pages 302-310, April 1998.
14. Keller, R. K., Schauer, R., Robitaille, S., and Pagé, P. "Pattern-based reverse engineering of design components." In *Proceedings of the Twenty-First International Conference on Software Engineering*, pages 226-235, Los Angeles, CA, May 1999. IEEE.
15. Mylopoulos, J., Borgida, A., Jarke, M., and Koubarakis, M. "Telos: Representing Knowledge About Information Systems." *ACM Transactions on Information Systems*, Vol. 8, No. 4, October 1990, pages 325-362.
16. Object Data Management Group (ODMG), 2000. On-line at <http://www.odmg.com>.
17. OMG. "XML Metadata Interchange (XMI)", Document ad/98-10-05, October 1998. On-line at <ftp://ftp.omg.org/pub/docs/ad/98-10-05.pdf>.
18. Poet Java ODMG binding, on-line documentation. Poet Software Corporation, San Mateo, CA, 2000. On-line at <http://www.poet.com>.

19. Pons, A., and Keller, R. K. “Schema evolution in object databases by catalogs.” In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS'97)*, pages 368-376, Montréal, Québec, Canada, August 1997. IEEE.
20. Rational Software Corporation, 2000. On-line at <http://www.rational.com>.
21. Robbins, J. E., and Redmiles, D. F. “Software architecture critics in the Argo design environment.” *Knowledge-Based Systems*, (1):47-60, September 1998.
22. Robitaille, S., Schauer, R., and Keller, R. K. “Bridging program comprehension tools by design navigation.” In *Proceedings of the International Conference on Software Maintenance (ICSM'2000)*, pages 22-32, San Jose, CA, October 2000. IEEE.
23. Saint-Denis, G., Schauer, R., and Keller, R. K. “Selecting a Model Interchange Format: The SPOOL Case Study.” In *Proceedings of the Thirty-Third Annual Hawaii International Conference on System Sciences* (CD ROM, 10 pages). Maui, HI, January 2000.
24. Schauer, R., Keller, R. K., Laguë, B., Knapen, G., Robitaille, S., and Saint-Denis, G. “The SPOOL Design Repository: Architecture, Schema, and Mechanisms.” In *Hakan Erdogmus and Oryal Tanir, editors, Advances in Software Engineering. Topics in Evolution, Comprehension, and Evaluation*. Springer-Verlag, 2001. To appear.
25. UML. Documentation set version 1.1, 2000. On-line at <http://www.rational.com>.
26. Weinand, A., Gamma, A., and Marty, R. “Design and implementation of ET++, a seamless object-oriented application framework.” *Structured Programming*, 10(2):63-87, February 1989.
27. Wong, K., and Müller, H. (1998). Rigi user’s manual, version 5.4.4. University of Victoria, Victoria, Canada. On-line at <ftp://ftp.rigi.csc.uvic.ca/pub>.