Facilitating Software Evolution Research with Kenyon

Jennifer Bevan[1], E. James Whitehead[1], Jr., Sunghun Kim[1], and Michael Godfrey[2]

[1]Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA, USA
01-831-459-1227

{jbevan, ejw, hunkim}@cs.ucsc.edu

[2]School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
01-519-888-4567

migod@uwaterloo.ca

ABSTRACT

Software evolution research inherently has several resource-intensive logistical constraints. Archived project artifacts, such as those found in source code repositories and bug tracking systems, are the principal source of input data. Analysis-specific facts, such as commit metadata or the location of design patterns within the code, must be extracted for each change or configuration of interest. The results of this resource-intensive "fact extraction" phase must be stored efficiently, for later use by more experimental types of research tasks, such as algorithm or model refinement. In order to perform any type of software evolution research, each of these logistical issues must be addressed and an implementation to manage it created. In this paper, we introduce Kenyon, a system designed to facilitate software evolution research by providing a common set of solutions to these common logistical problems. We have used Kenyon for processing source code data from 12 systems of varying sizes and domains, archived in 3 different types of software configuration management systems. We present our experiences using Kenyon with these systems, and also describe Kenyon's usage by students in a graduate seminar class.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering.*

General Terms

Management, Measurement, Documentation, Design, Reliability.

Keywords

Software evolution, software stratigraphy, software configuration management

1. INTRODUCTION

Software evolution research has investigated many different issues, from modeling program development [3] to using the archived development history to recommend likely members of change-impact sets [18, 20]. Many of these research analyses sample the archived project artifacts, such as source code configurations or modification request forms, at specific points in time, either by date or by semantic label, over a significant portion of the entire project history. Depending upon the type of analysis being performed, different types of "facts" may be extracted from each configuration, or from the data associated with the difference between a pair of configurations (e.g., source code repository commit data). These facts, which comprise the raw data on which the evolution analysis operates, are commonly stored once and used during multiple analysis passes.

The method of obtaining the input data, preprocessing it to extract the relevant facts, and storing the results has traditionally been re-implemented for each software evolution research system created. Due to the time cost of implementation, researchers are forced to make tradeoffs in the type of data they can analyze, such as committing to a single programming language or supporting a single software configuration management (SCM) system. For example, the tradeoff of only supporting CVS [5] is commonly considered acceptable [7, 8, 10], given the large number of CVS-archived projects available through SourceForge. The primary threat to validity with such systems, however, stems from the fact that most industrial systems do not use CVS, and therefore the applicability of the analysis results to such systems is sometimes questioned [20].

These types of tradeoffs, and the practice of implementing new solutions to these common problems in each analysis system, lead to difficulties when attempting to apply the results of one analysis technique to improve or refine the results of another. We believe that these difficulties stem from the process of mapping the set of data structures and their associated assumptions used by one researcher to the set of data structures used by another. It is also not always true that one analysis system can use the final results of another system; instead, the partial results from an earlier point in the processing may be the optimal point of composition or reuse. For example, both Ying et al. and Zimmerman use association rule mining to provide recommendation sets on artifacts that are likely to need changing given an initial source code modification [18, 20]. If a different analysis engine wants to reuse the computed association sets for a different purpose, they are not necessarily easily available or reusable. We believe a shared set of data structures for common software evolution representation problems would improve researchers' ability to compare and leverage results by improving such data availability and minimizing (or removing) the need to map one structure to another.

In this paper, we present Kenyon, a system inspired by our desire to improve the results of our instability analysis system (IVA) [4] with the origin analysis performed by Beagle [11]. While examining the Beagle data flow, we noticed the similarities between its early preprocessing stages and our own. In fact, Zimmerman and Weissgerber identified very similar preprocessing tasks as those "performed by most analyses" [19]. As a result, we redesigned a significant portion of our existing software, including some of the Beagle code, to become an independent subsystem, and defined a set of responsibilities and services that it would provide. This is Kenyon.

Kenyon supports software evolution research systems that perform "stratigraphy", the analysis of a series of related layers (strata) that comprise a time-based software development history. For source code, the strata are the configurations within the historical record archived by an SCM system, with possible associations to data in other, time-variant sources such as bug-tracking systems. Kenyon provides automated configuration retrieval from an SCM system onto the local filesystem, invokes analysis-specific "fact extractors" on each retrieved configuration, and saves the extracted facts into a relational database using an object/relational mapping (ORM) system. These results are then accessible to any number of later analysis techniques implemented by external *evolution analysis* systems. While Kenyon does not mandate that these independent evolution analysis systems report results in a specific format, it does provide reusable data structures via the ORM system that are expected to provide a common basis for interpreting and reusing the results that are provided.

Our aim in creating Kenyon is to reduce the start-up time associated with software evolution research by providing a common framework within which new analysis methods can use any of the supported SCM systems and any of the supported data types. Additionally, by providing a sufficiently flexible structure for storing extracted facts, the process of composing or comparing results is much simpler. As a means of testing our progress towards these goals, we provided Kenyon to the students in a 10-week graduate seminar on software evolution, during which they were required to complete a software evolution-based research project. Concurrently with this seminar we used Kenyon to analyze one 500 KLOC industrial system and nine different open-source systems up to 298 KLOC, to better test and model its scalability.

The rest of the paper is structured as follows. Section 2 discusses the requirements associated with any infrastructure tool intended to support software evolution research. Section 3 presents the structure and data flow of Kenyon, and describes how each of the requirements are addressed. Section 4 describes in greater detail our experiences using Kenyon in the classroom, on the open-source systems, and on the industrial system. It also describes the adaptations to Kenyon made in order to meet the needs exposed during this time. Section 5 discusses related work, and Section 6 discusses planned future work for Kenyon.

2. EVOLUTION INFRASTRUCTURE REQUIREMENTS

Any infrastructure tool that proposes to assist software evolution research must allow researchers to focus on their research-specific interests. This goal implies that the tool should reduce manual labor costs as much as possible. The key such benefit that an evolution assistance tool could provide is automated per-configuration fact extraction processing: the current most labor-intensive tasks in software evolution research are those of interacting with various data archives (e.g. SCM systems), identifying and extracting semantically consistent configurations, and analysis-tool invocation. A secondary benefit is a reduction in "time-to-research", because research-specific systems would not need to reinvent solutions to the common logistical problems; the time saved may even be used to perform more in-depth analyses than otherwise possible within fixed time limitations. An acceptable evolution assistance tool must support these benefits without limiting the types of evolution research possible on the preprocessing results; for example, requiring the "data cleaning" [19] phase as used by association rule mining techniques [18, 20] would remove data required by IVA [4]. These benefits and constraints imply several requirements on a software evolution infrastructure tool. We outline these requirements below, and discuss the extent to which they impact the acceptability of such as system.

Req. 1: Automated configuration retrieval. A significant part of the labor associated with per-configuration fact extraction is based on interacting with the systems that archive the configuration data. Given a specification of the times for which configurations should be retrieved, a set of constraints that define the membership of the retrieved configurations (e.g. files in directory "foo", or not ending in ".doc"), and a set of data sources, the infrastructure tool must automatically, without any further manual assistance, obtain a time series of configurations from the archiving systems.

Req. 2: Allow user control on configuration times.

Req. 2.a: Allow limits on timespan processed. Software evolution systems do not always need to analyze the entire history of a system to produce interesting results. An evolution infrastructure tool must therefore accept a timespan specification that explicitly limits the earliest and latest configuration times to be processed.

Req. 2.b: Allow user-specified minimal processing interval. The frequency at which configurations need to be retrieved is also a function of the evolution analysis to be performed and the software project being analyzed. An infrastructure tool must also accept a "sampling rate"-type specification (such as "once per day") that selects a subset of configurations to be processed from the set of configurations available within a given timespan.

Req. 3: Support for different software configuration management systems. Software configuration management (SCM) have several implementations with widely varying capabilities, such as CVS [5] and ClearCase [14]. An infrastructure tool must have the ability to interact with several commonly-used such systems, and the ability to easily add support for others.

Req. 3.a: Retrieve consistent source code configurations. We consider the smallest unit of change in the state of a source code repository that is of interest to software evolution researchers to be a "logical change" [19], interpreted at the level of a single, user-issued "commit" command. Non-transaction based systems such as CVS alter the state of the repository each time a change to a given file is stored, instead of once per such logical change. Configurations from such systems must be retrieved such that only the effects of logical changes are visible.

Req. 3.b: Access archived metadata associated with each logical change. If a data source records metadata associated with archived data, such as the author and log message for a given SCM commit, the infrastructure tool must be able to access it and make it available to analysis tools.

Req. 4: Support multiple data input sources. The necessary data to perform software evolution analysis is not always stored in a single data source. While different types of data are certainly expected to be stored in different types of systems (such as bug tracking data and version history data), sometimes the same type of data may be spread across different systems. For example, if two libraries are co-evolving and interdependent, but are archived in separate SCM systems, an evolution infrastructure tool must be able to access both version histories in a consistent and seamless manner.

Req. 5: Allow incremental processing. A software evolution infrastructure tool must allow researchers to both “catch up” to the present time and to “keep up” with ongoing development. Given the computational costs that still apply to per-configuration processing, previous processing results must be able to be easily integrated with results from new processing.

Req. 6: Support a broad variety of user-defined fact extraction tools. Automated per-configuration processing is only useful when the user has the ability to select arbitrary fact extraction tools that provide the data relevant to the specific research being performed. Given the time cost of retrieving each configuration from the archiving systems, an infrastructure tool must therefore support the invocation of a user-specified, heterogeneous set of fact extraction tools on each retrieved configuration.

Req. 7: Support processing of multiple types of data in multiple languages. A software evolution infrastructure tool must not inherently limit the types of systems whose evolution may be analyzed. While it may place the burden for language-specific analysis on the user-defined processing tools, it must not make it impossible to accommodate systems with unfamiliar or mixed languages (e.g. modeling or programming languages) or data types (e.g. source code, design documents).

Req. 8: Provide efficient, accessible, and optional storage of extracted facts. The facts extracted during per-configuration processing may be stored in several different ways, from XML files to relational databases. Because software evolution research is likely to analyze the time series of per-configuration results, a storage method that allows efficient access to these results along the time dimension must be provided. Evolution researchers must also be allowed to decide to not use the provided storage method, as it might not be immediately compatible with their existing analysis systems.

Req. 9: Scalability.

Req. 9.a: Computational scalability. The infrastructure tool must not require significantly more memory or CPU resources to automatically process a series of configurations than that required for processing of a single configuration. This allows an arbitrary number of configurations to be automatically processed.

Req. 9.b: Data access scalability. The infrastructure tool must not require that all facts extracted from a given configuration be loaded into memory when access to a subset of these facts is desired. This allows multi-configuration analysis tools to minimize their own memory usage.

Req. 9.c: Support parallel batch processing. Even though a tool that supports automatic per-configuration processing can dramatically reduce the labor costs of software evolution research, the computational costs can still be significant. The results from processing different timespans in parallel must be able to be merged in the storage system, to further reduce the time-to-research.

Req. 10: Availability. Any system that intends to facilitate software evolution research must be easily available to researchers (ideally through a web-based download), run on several common processing platforms, and provide effective and helpful documentation to its intended audience.

3. KENYON ARCHITECTURE

Kenyon currently fulfills all of the requirements described in Section 2, with the exception of supporting multiple data input sources (Req. 4). In this section we present a more detailed look at Kenyon’s design and how it addresses each of the functional requirements.

3.1 Usage Overview

Kenyon is designed as an asynchronous, minimally interactive application. It is configured via a user-supplied processing configuration file: a text “properties” file that names the data sources, configuration selection guides, and third-party tools to be invoked on each configuration (Req. 6). Kenyon is also normally configured with an ORM-specific properties file that names the database to which the preprocessed data should be stored (Req. 8). Kenyon “samples” a data source (i.e. SCM system) at a specified time interval, such as once per second or twice per day, between a start date and an end date, which may be set to “last” for incremental processing (Req. 2).

Kenyon’s data source sampling algorithm is driven by the expected needs of its users. Software evolution research is primarily concerned with the effects of “logical changes” [19]. We interpret a logical change at the lowest archived level: that associated with a user-issued “commit” command. We use the term “configuration” as a set of files defined by a set of inclusion constraints; Kenyon configurations represent the state of the repository along a particular branch (variant) at a given timestamp. We perceive these configurations as analogous to the geologic strata that form a fossil record, and give each a unique *configuration specification* comprised of the project identifier, branch identifier, and timestamp. The set of “interesting” configuration specifications are those that reflect the result of applying a single logical change. When sampling the data source, Kenyon retrieves configurations only for these “interesting” configuration specifications, and uses the user-specified time interval to ensure a minimum time between successive retrieved configurations.

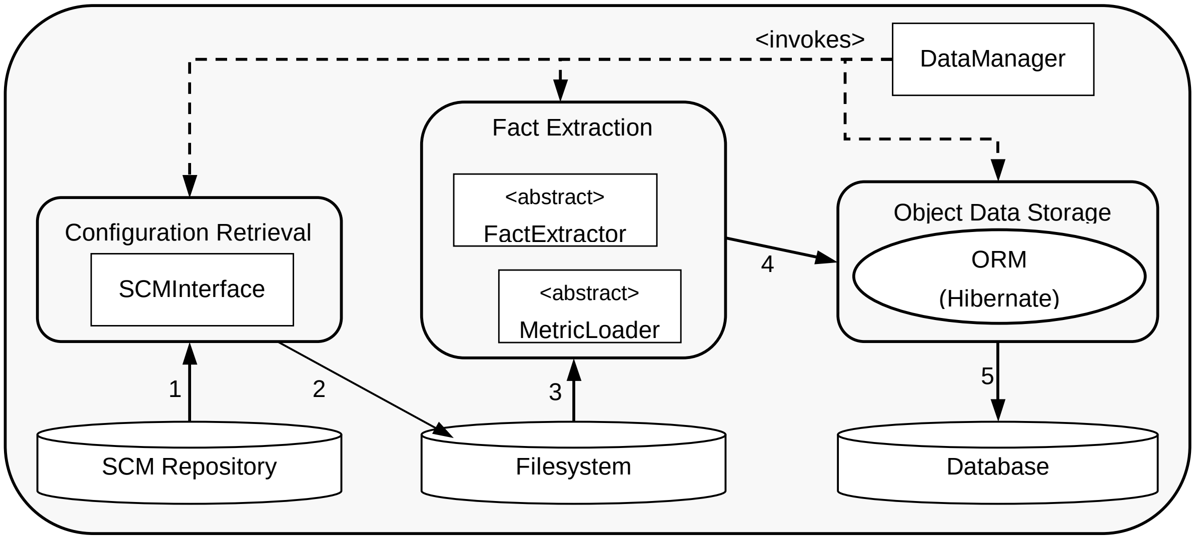

Figure 1. High-level data flow architecture of Kenyon. The numbers on the solid arrows indicate the processing order.

Some SCM systems, such as CVS, do not archive files changed by a single user-issued "commit" command as a single, logical change. In the case of CVS, files that should be considered as being part of the same commit "transaction" are actually stored in the repository with different timestamps. When Kenyon retrieves data from such systems, it applies a sliding-window "transaction recovery" algorithm [19] to regroup these files into a single logical change (Req. 3.a). Kenyon uses the set of times at which at least one transaction completed (i.e. the latest time at which a file in a given commit transaction was written to the repository) to define the set of logical change-based configurations. When retrieving a configuration from such a system, Kenyon ensures that any file that was already saved to the repository at the specification timestamp, but that is also part of an uncompleted (or, "ongoing") transaction, is retrieved as it existed before that ongoing transaction began.

Kenyon supports incremental updates to an existing processed data set (Req. 5). It can therefore be used to process a development history, and then keep up with ongoing development using, for example, nightly updates. It can also be used in a successive-refinement mode, where a large time interval is initially used to identify areas of particular interest. What constitutes "particular interest" is of course analysis-specific. One scenario where successive refinement is useful occurs when analysis quality is dependent upon the amount of change that occurs between two compared configurations. For example, it may be that as the number of logical changes incorporated between two compared configurations increases, so does the difficulty of producing a correct result. If the analysis provides a result-quality "confidence value", then an automated successive refinement process could be configured.

For example, if Kenyon preprocessing is performed at an initial interval (such as once per week), the analysis tool could run over those results, generating a set of confidence values for each successive pair of configurations. For each pair where the confidence value is below a certain threshold, Kenyon could be rerun at a smaller time interval (such as once per day or once per hour). This process could iterate until either the smallest time interval possible is reached (once per second) or the confidence value exceeds the threshold. Successive refinement allows Kenyon users to avoid processing all data at small time intervals, which reduces part of the computational cost of performing software evolution analyses (Reqs. 1,2,5).

3.2 Data Flow Architecture

The high-level Kenyon data flow architecture is shown in Figure 1. The DataManager class is the execution entry point; it reads the configuration files and invokes the configuration retrieval, fact extraction, and object storage methods. The SCMInterface class isolates Kenyon from the implementations associated with each concrete SCM subclass (Req. 3): at this point, Kenyon supports the CVS, Subversion, and ClearCase SCM systems. It also supports a "filesystem" implementation that is intended for use when access to the SCM repository is not available but a series of pre-downloaded configurations (such as system releases) are. The FactExtractor and MetricLoader abstract classes are the API points for research-specific tool invocation extensions.

Kenyon retrieves each configuration to be processed and places it in the local filesystem. We do this because program analysis tools commonly support a filesystem input source and rarely support direct SCM interaction. The DataManager class then invokes the series of concrete FactExtractor subclasses specified by the user in the processing configuration file. These subclasses are the means by which external, analysis-specific, fact extraction tools interface with Kenyon. While we expect most users to provide FactExtractor subclasses, Kenyon does come with two: one that invokes and loads the data from Grammatech's CodeSurfer[1], and one that invokes part of the SWAGKIT[2] pipeline and reads the resulting TA-formatted files.

Kenyon draws a distinction between programs that produce a graph representation of the configuration under analysis (such as

[1] www.grammatech.com/products/codesurfer

[2] www.swag.uwaterloo.ca/tools.html

a call graph or a containment graph) and those that calculate metrics that may be associated with the configuration or some of its entities (such as total number of lines of code, or average number of files per directory). The former programs are considered to be fact extractors, and the latter are termed "metric loaders". The rationale behind this distinction is that metrics may be calculated by different systems on different program entities, and a common system representation is necessary to merge the results of each individual metric loader. We therefore expect metric loaders to attribute the graph (or elements therein) produced by a fact extractor; each such graph is expected to be an internally consistent system representation.

The user-supplied Kenyon processing configuration file specifies the metric loaders that will be invoked by each fact extractor, via a concrete MetricLoader subclass. For performance reasons, Kenyon ensures that each metric loader only calculates its results once, although it may load its results onto many different graphs. As with FactExtractor subclasses, we expect most users to provide their own MetricLoader subclasses, although Kenyon does provide one that invokes and loads the data produced by UnderstandForC++[3].

Kenyon saves the results from each processed configuration to a database (Req. 8). As part of our goal to improve the ability of third-party analysis systems to reuse Kenyon data structures, we decided to use an object-relational mapping (ORM) system to help automate the storage to, and retrieval of Java objects from, the database (Reqs. 8,9.b). This affords us some added SQL dialect isolation, allows automatically generated object-based database schemas to be used, and simplifies the process of merging of results processed either in parallel or incrementally (Reqs. 5,9.c). Kenyon's current ORM system is Hibernate 2.1.6 [2], which has been helpful in some respects although it has had some scalability problems. Hibernate allows Kenyon to use XDoclet [17] tags, embedded within javadoc-type comments, to annotate the source code with the information necessary to automatically generate the object-relational mapping files. These in turn are used to generate the database schemas. Hibernate also provides HQL, an object-based query language that translates the queries into SQL based on the dialect specified in the Hibernate configuration file.

Kenyon does provide some prepackaged Hibernate convenience queries for obtaining the set of analyzed projects, the configuration specifications for which data is available in a given project, and the fact extractor-specific result for a given specification. We have found that, in practice, our own analysis systems use handcrafted HQL, optimized for the specific query goal, for almost any purpose other than these.

Software evolution researchers are not required to use Hibernate, because they can still access the Kenyon-processed data using SQL. The benefit from an ORM system primarily comes from the concept of "persistent objects", which means that a Java object instance can be retrieved from the database in exactly the same state that it was persisted to the database. This retrieval mechanism simplifies the source code necessary to load results stored either by Kenyon or by a third-party analysis tool. The data flow for this type of scenario is shown in Figure 2, where solid lines indicate data storage and dashed lines indicate data access.

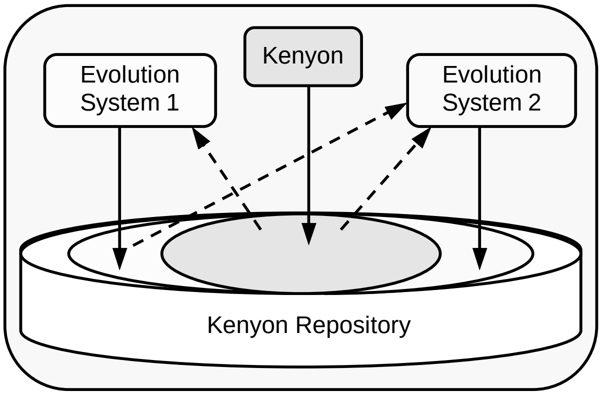

Figure 2. Data flow between Kenyon and two software evolution tools, one of which uses the results of the other.

Kenyon does not currently place any restrictions on the database used for storing the preprocessed data, beyond the requirement that a JDBC driver be available. Database administration issues, such as write permissions, are also not managed in Kenyon. Kenyon does make its tables "immutable" to external analysis systems via Hibernate, but this only restricts deletions and modifications, not insertions. We expect to be refining our database requirements as we evaluate collaborative evolution analysis environments in the future.

3.3 Persisted Data Structures

The goal of our persisted data model is to facilitate result comparison and reuse across different evolution analysis systems. We balance the expressiveness of a flexible schema with the comparability of a rigid schema by adopting a model that provides flexibility only in the portions of the schema associated with analysis-specific results, and that is rigid everywhere else. We maximize the comparability of the flexible, analysis-specific, schema by using an expressive and rigid graph-based framework and requiring the use of a descriptive graph schema data structure to provide information on the specific organization of a given graph. Certainly, "log-based" evolution analyses do not require a graph representation to convey results [7, 8, 10, 18, 20], but they do not preclude it. On the other hand, evolution research based in program analysis [4, 11-13] commonly use some form of graph-based representation (such as a call graph, or via an entity identification scheme rooted in a containment hierarchy), without which it is very difficult to compare results. The Kenyon graph structure can not only express several different standard graph exchange formats such as GXL or TA, but is in fact more expressive than either of these; as one example, it allows multiple ordered values to be associated with a single attribute.

Kenyon stores all results generated from the per-configuration fact extractor processing, as well as analysis-independent data such as the configuration specifications processed and the project to which the per-configuration results should be assigned. The association relationships between most of the Kenyon-persisted classes are shown in Figure 3. The Project and ConfigSpec classes act as the principle identifiers used to retrieve a specific ConfigData instance, which contains a set of per-configuration

[3] www.scitools.com

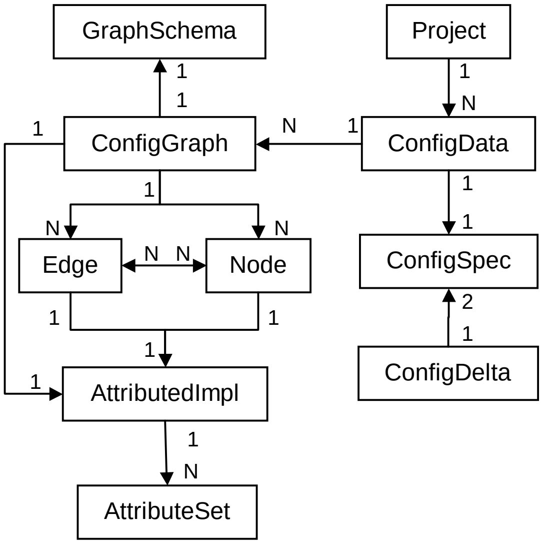

Figure 3. Association relationships between persisted Kenyon classes.

ConfigGraph results. Each ConfigGraph is identified by the name of the FactExtractor subclass that created it, and references the GraphSchema subclass used during its construction. This is the mechanism by which ConfigGraphs provide a description of their structure (such as naming scheme and containment hierarchy) to evolution analysis systems (Req. 7). ConfigGraphs are comprised of nodes, edges, and any attributes assigned to these graph entities; these classes are naturally persisted as well.

To further isolate research-specific fact extractor tools from the specific SCM requirements, differences between successive configurations are calculated automatically by Kenyon, stored as ConfigDelta instances, and associated with two configuration specifications. Because ConfigDeltas are generated from data within the software configuration management system, the granularity of the data stored is not required to be uniform; however, the minimal data granularity identifies which lines in which files were added, deleted, or modified. In the case where Kenyon is being invoked on a timespan that overlaps previously processed data (as would be done with a successive refinement approach to data processing), no configurations that were previously processed are reprocessed. ConfigDeltas are re-computed whenever the time difference between successive configurations is smaller than previously stored.

Software evolution analysis systems are not required use Hibernate to store their results. We do feel that making software evolution results available in this way will improve the ease with which researchers can compare or reuse the results of others. To this end, Kenyon provides an abstract, Hibernate-persistable, “EvolutionPath” class that associates a set of nodes in one ConfigGraph to a set of nodes in another ConfigGraph. This class can be used as a building block for composing or persisting more complex software evolution results.

3.4 Scalability

Certain performance issues should be given consideration when creating a FactExtractor subclass. The per-configuration running time of Kenyon is the total of the fact extractor run time plus the time to load the resulting graph into memory and store it to the database. The larger the graph, the more time it takes to perform these latter two steps. Our goal has been to ensure that Kenyon is scalable, within the limitation that any given pairing between a fact extractor and a system to be analyzed will dramatically impact the performance numbers (Req. 9.a).

Memory usage is also an issue: Kenyon cannot enforce that third-party fact extractors remove all references to the created ConfigGraph or its components. If a fact extractor does not do this, the garbage collector will not be able to reclaim the memory used. Analyzing the memory usage of Kenyon while it is running over a limited number of configurations with a new fact extractor is a very useful secondary validation phase.

4. KENYON IN PRACTICE

We have recently concluded concurrent projects in which Kenyon was used in three very different settings. The first setting was in the lab, where a Kenyon developer used Kenyon to process a 500 KLOC industrial system archived in ClearCase for later analysis by IVA [4]. In a second lab setting, another Kenyon developer processed several different open-source projects up to 298 KLOC using Kenyon, for later analysis with SignatureFE [13]. The third setting was in the classroom, where students in a 10-week graduate seminar class were expected to perform some type of software evolution analysis as a project. Independent of these projects, developers of a third-party code-clone evolution analysis system [12] also used Kenyon. The next sections describe these projects and the lessons learned from them.

4.1 System X

We applied Kenyon to “System X” as a preprocessing phase before invoking IVA, an evolution analysis tool that identifies historically high-maintenance, statically-dependent, code regions [4]. System X is approximately 500 KLOC (commented) of mixed C/C++ source code. Its developer made six months of its ClearCase history available to us. We used CodeSurfer as our fact extractor, and created ConfigGraphs at a mixed line/procedure level of granularity. The owners of System X specified that we were to process the configurations at each of a list of ClearCase labels. This use of Kenyon represented the first time we had applied a commercial fact extraction tool to a mixed-language system of this size. We expected to get a better sense of how well our support of third-party per-configuration analysis tools performed, and to also discover any scalability problems that we had not anticipated. We did encounter several such challenges, which allowed us to assess how well we were meeting the requirements for an evolution infrastructure tool, as well as to further refine and improve those requirements.

Our first significant challenge with System X arose as we realized that while we had the available history for the project, we did not have the history of the project’s “environment”: a set of co-evolving libraries that defined several macros, procedures, and global variables used by System X. The environment’s version history was archived in a separate SCM system, to which we did not have access. We therefore needed to manually align specific releases of this environment with the ClearCase history. This situation resulted in the creation of our “filesystem” SCM subclass, wherein a set of pre-existing directories are assigned configuration specifications in an XML file. We realized that we would eventually need to extend Kenyon to access data from multiple SCM systems to produce a consistent configuration for projects like System X (Req. 4), but for this specific project we instead opted to create a set of configurations, combined with the appropriate environment, on the filesystem. This decision gave us the advantage of being able to run Kenyon on the data from multiple machines, some of which were not able to directly mount ClearCase views due to access restrictions.

The second challenge with System X arose from a realization that graph node names, as produced by CodeSurfer, were not normalized to remove semantically meaningless whitespace, such as found in the parameter lists in C++. This caused name-based associations (performed by IVA) between nodes in different configurations to fail. This problem couldn't be solved within Kenyon due to its origins in the fact extraction process; however, we did modify the Kenyon-provided CodeSurfer FactExtractor subclass to perform the normalization (Reqs. 6,7).

Our last significant challenge when processing System X with Kenyon stemmed from a scalability issue with our use of Hibernate (Req. 9a). Kenyon had been taking advantage of a cascaded "save" operation, in which a single save call would persist all associated data structures to the database. To this point, we had not found a problem with this process. The System X graphs, however, averaged approximately 650MB in memory each, and saving them using Hibernate directly was causing OutOfMemory errors. The core reason for the memory growth was the lack of a "write-and-discard" operation in Hibernate; instead, Hibernate replaces newly saved objects with "persistent" versions that contain the added state necessary to perform caching and dirty-object checking. The persistent versions of the graph nodes alone added approximately 600MB to the Kenyon memory usage during a save operation. To avoid this memory growth, we modified Kenyon to directly save each ConfigGraph using JDBC calls, and continued to use Hibernate to save to all of the other objects associated with the ConfigData. We would prefer to maintain SQL-dialect independence through an improved ORM system; however, this solution does solve the immediate problem.

Our running time for Kenyon/CodeSurfer on a single configuration of System X was approximately 1 hour on a computer with a 2.8GHz processor and 3GB of physical memory. It reached a steady-state memory usage of approximately 1.8GB. CodeSurfer contributed 30 minutes to this total, producing per-configuration graphs containing both containment and dependence relations with approximately 135,000 nodes and 344,000 edges. The remaining 30 minutes involved reading the CodeSurfer-produced graph file from disk, calculating a ConfigDelta directly from a ClearCase-provided delta, and saving the results into the database. We expect to gain some reduction in the Kenyon-specific time from optimizations such as tuning our database transactions per graph ratio.

Because Kenyon was primarily extracted from existing IVA code, the data structure reuse level between the programs was quite high. Even so, the process of applying a familiar evolution analysis tool to Kenyon data caused us to realize several API-level weaknesses between Kenyon and IVA in the early releases. These weaknesses were primarily related to the functionality provide by the GraphSchema class and to the expected boundary between which data should be persisted by Kenyon and which by IVA. We resolved most these issues before the graduate seminar class started to use Kenyon.

Our experience using Kenyon as the preprocessing stage for IVA with System X provided us with several key validations. First, it proved that the separation of analysis-independent, logistical issues (Kenyon) and the analysis-specific research issues (IVA) was possible. It also showed that this separation provided a significant benefit; only minimal reprocessing was needed when correcting an IVA error. We determined that not only could Kenyon perform automatic per-configuration processing with a third-party fact extraction tool, but that it could do so with reasonable performance and at high scale.

4.2 SignatureFE

We also used Kenyon during an ongoing signature change analysis project on a total of nine different open-source systems of up to 298 KLOC in size [13]. The fact extractor used during this process, SignatureFE, is entirely separate from Kenyon, and was created by a Kenyon developer that had not been involved in creating either of the provided FactExtractor subclasses. The systems analyzed using SignatureFE can be found in Table 1, where "revisions" indicates the number of configurations extracted by Kenyon directly from the SCM repository. In some SignatureFE analyses, a set of releases was processed instead, via Kenyon's filesystem (FS) SCM interface subclass.

Table 1. Projects analyzed by SignatureFE using Kenyon.

Project	SCM	# revisions (or releases)
Apache Portable Runtime (APR)	SVN	5990 revisions
Apache HTTP 1.3 (Apache 1.3)	SVN	7747 revisions
Apache HTTP 2.0 (Apache 2)	CVS	3877 revisions
APR utility (APU)	SVN	1353 revisions
Subversion	SVN	5886 revisions
CVS	CVS	2873 revisions
Linux Kernel 2.5 (Linux)	FS	75 releases
GCC	FS	15 releases
GCC	CVS	3012 revisions
Sendmail	FS	37 releases
Subversion (SVN)	SVN	6029 revisions

The most significant findings from this project related to our expectations on the ease of access using HQL to the Kenyon data (Req. 8). Creating queries to retrieve, for example, only those Procedure nodes that had a specific metric value over a certain amount was still more difficult than we had hoped.

The reason for this difficulty is that Hibernate 2.x does not directly support mapping or referencing collections of collections. Kenyon, on the other hand, uses cascading collections within its ConfigGraph class, as shown in Figure 3. Hibernate actually forced the creation of a distinct AttributeSet in order to create a mapping for the AttributedImpl class. HQL has a similar weakness when working with collections of collections, especially when lazy initialization is used to avoid loading an entire graph. While in most cases the HQL for a given query is simpler than the equivalent query in SQL, in this case it is not. Our best approach for this query with the current version of Hibernate is performed in memory, using an iterative approach that avoids loading the entire graph.

We still believe that our use of an ORM solution will improve the ability of evolution analysis tools to reuse Kenyon data structures. In most cases, evolution analysis tools will still need to handcraft queries, whether they be performed in the database or not, specific to their interests. We are planning a two-stage approach to managing these types of problems in future versions of Kenyon. The first involves upgrading to Hibernate 3; this appears worthwhile due to improvements in its query and mapping capabilities. During this time we will review the existing Hibernate mappings and try to design new mappings that would be easier to query. The second phase will likely involve migrating Kenyon to use an ORM framework instead of using a specific ORM solution.

Our experience using Kenyon with SignatureFE provided us with several more key validation points. Kenyon correctly handled a large number of revisions from each of several different open-source systems of varying sizes. It also allowed a third-party fact extractor to be used without any modifications to Kenyon. Another benefit of this project was a better understanding of the API-level questions that might be raised by third-party researchers when creating a FactExtractor subclass.

4.3 Kenyon in the Classroom

To test Kenyon's ability to reduce the overhead associated with performing software evolution research, we made Kenyon available to the students in a 10-week graduate seminar class on software evolution for use with their projects. We expected the students to pick projects in which they would use the already-processed data in a Kenyon repository as the input for their analyses. Instead, most of the students chose projects where they created their own fact extractors and performed their own analyses.

Table 2 shows the projects analyzed by students in the class. Of the 6 students, 1 used Kenyon fully, 2 only used the SCM configuration retrieval and fact extractor invocation portions, 2 were not able to use Kenyon at its then-current set of capabilities, and 1 chose a project in which the use of Kenyon was not applicable.

Table 2. Projects analyzed by students in seminar class.

Student	Project	SCM	Kenyon?
1	Hibernate	N/A	No
1	Eclipse	N/A	No
2	Kenyon	SVN	Yes / no database
3	<test program>	SVN	Yes/ no database
4	Recoder	CVS	Yes
5	<proprietary>	Perforce	No

The reasons given for not using Kenyon fell into two main categories: support and data storage flexibility (Reqs. 4,6,7). Kenyon currently supports only the CVS, Subversion, and ClearCase SCM systems. Student 5 selected a project for analysis that was archived in Perforce; Kenyon was therefore not an option for him. Kenyon also does not yet provide a built-in Java containment model. While in this case it did not prove to be a major issue, the primary benefits of such containment schemas are in language-specific graph construction and convenience query methods. Student 4 created his own graph schema for his project, and therefore was able to use Kenyon completely.

The approaches used by the remaining students were not compatible with Kenyon for the second category of reasons: data storage flexibility. Specifically, Kenyon 1.3 did not have support for persisting configuration-based results generated by analyzing the abstract syntax tree (AST) of a program. Students using such approaches had their fact extractors write their results to the filesystem instead.

This realization changed our perception of what types of results a fact extractor might produce (Req. 8). For example, Student 1 used a third-party tool that analyzes abstract syntax trees to identify code clones within source code. The results generated are analogous to proper subgraphs of the full system graph, as opposed to our expectation of the full system graph itself. To better support this and other types of configuration-specific results, we developed a new "ConfigSummary" interface and added an association from ConfigGraph to any class that implements this interface. We also created an inheritable Hibernate mapping for this interface. Student 1 could now, for example, create a ConfigSummary subclass that contained a collection of SubGraph (a Kenyon-provided class) instances that mapped nodes belonging to a given clone identifier onto a ConfigGraph that represented the containment structure of the system. This data could be automatically persisted with Hibernate, which would give the student the advantage of reusing the Kenyon data structures during evolution analysis of the extracted code clones.

4.4 Clone Genealogy Extraction

M. Kim et al. at the Univ. of Washington have developed a tool that extracts clone "genealogies" as part of their research on clone evolution [12]. They analyze multiple versions of a program where each version corresponds to a commit transaction, because they are interested in understanding how code clones evolve in finer granularity than releases.

Table 3. Projects analyzed for clone genealogy.

Project	SCM	# transactions
carol	CVS	164 revisions
dnsjava	CVS	905 revisions

Kim et al. were the first external adopters of Kenyon, and at the time were primarily interested in it for the automated configuration retrieval from CVS. They had previously analyzed only system releases, and wanted to decrease the amount of time required to perform the same analysis in a finer granularity. To speed up their process, they used Kenyon only to automatically retrieve each configuration that corresponded to a repository "check-in", or commit. Table 3 shows the projects for which configurations were retrieved in this manner.

It was through their early support that we realized the need to fully support a "no database" mode of operation, especially as our persisted-classes data model was still rapidly evolving. Given our addition of the ConfigSummary class, we believe that this clone genealogy tool can now adopt Kenyon more fully.

4.5 Beagle

Beagle performs "entity mapping", where a system entity in one configuration is associated to another entity in a different configuration. Beagle focuses on associating file and procedure entities that were not wholly added or deleted between two system releases, but instead were “renamed, moved, or otherwise changed” [11]. This process is named “origin analysis”, and can be used to improve the results of software evolution analysis by allowing an entity’s history to be followed across these types of changes.

Six months ago we performed a preliminary integration between a previous version of Beagle and an early version of Kenyon. We were pleased with the results with respect to code reuse: we were able to reduce the Beagle source code size by 18% even without performing a complete replacement of compatible data structures. Instead, we mapped the Beagle data structures used by the analysis portions of the system to Kenyon data structures, and used Kenyon classes from those integration points down to the database access level. We then used this experience to better refine the Kenyon data model. We would expect the current version of Beagle to attain a similar level of code reduction, because the core set of facts required for origin analysis has not significantly changed. This experience furthered our goal of minimizing the effort necessary to reuse results between disparate evolution analysis systems via a shared set of data structures.

5. RELATED WORK

While many systems have focused on mining version control systems for software evolution analysis, only a few stand out as having addressed some of the requirements that a software evolution infrastructure tool must support, as discussed in Section 2. All of these tools incorporate some research-specific design decisions that keep them from being general-purpose infrastructure tools, and they each address similar issues in unique ways.

Minero, created by Alonso, Devanbu, and Gertz [1], and Bloof, created by Draheim and Pekacki [8], are most like Kenyon in their separation of the logistical issues of mining software repositories from research-specific issues. Several key differences, however, stand out. Minero uses database schemas that are tied to the type of data source analyzed, and requires the built-in capabilities of a commercial database to improve the searchability of the database contents. Bloof allows users to define custom evolution metrics using the data from CVS log files as input, but their database schema is based on a minimal common subset of commit metadata (e.g. author, lines changed, etc.) archived in version control systems. Conversely, Kenyon does not place significant requirements on the capabilities of its underlying database, and avoids schema restrictions by using an object-relational mapping system that maximizes the benefits of both flexible and rigid schemas. Furthermore, Bloof is not intended to incorporate program analysis results, and this is reflected in its design. It is unclear if Minero’s design will easily extend to include program analysis results.

The other systems of note all deal with the issue of inferring relationships between different types of historical data, such as that archived in version control systems, bug-tracking data, email archives, and so forth. Hipikat, created by Cubranic and Murphy [7], infers associations between the data archived within these different sources to create an “implicit group memory”, and uses the resulting data to recommend relevant artifacts for a given task. German and Mockus’ softChange system [10] analyzes these data sources to identify “software trails” for later analysis and visualization. Neither of these systems treats its preprocessing subsystem as an independent system usable for other analysis techniques. Both support only CVS as their SCM repository. Fischer, Pinzger, and Gall created a system to populate a “Release History Database” (RHDB) [9] that associates bug tracking data with version control data; this system is more closely related to Kenyon than the other two because the results of the association are stored for later, unspecified, evolution analysis. The types of evolution analysis that can be performed on the RHDB-stored data are, of course, limited to using data from CVS log files. None of these systems can easily incorporate third-party per-configuration analysis tools into their design. Kenyon, on the other hand, is designed to support multiple types of data sources.

Lastly, OSSmole [6] shares a similar approach as Kenyon, although it is intended for project-level analysis instead of program-level analysis. It collects and archives project data, and third-party analysis results based on project data, for open-source software systems such as found on SourceForge.

6. FUTURE WORK

Kenyon has three main tasks: automated SCM configuration and associated data retrieval, fact extractor invocation, and data storage and access. Our next steps in developing each of these areas are outlined below:

- ***SCM/Data Support.*** While we have already demonstrated the flexibility of our SCM interface, we do intend to add support for Perforce, and other commonly used SCM systems, as needed. Our next phase of development for the data retrieval side of Kenyon is the ability to access multiple SCM repositories and to associate data from systems such as Bugzilla to the retrieved configurations. We expect that the majority of this effort will be spent in managing the association of specific states in the alternate data sources to specific time-based configurations in a primary data source. We hope to incorporate several of the association inference techniques developed for Hipikat [7], softChange [10], and RHDB [9].

  As part of this effort, we plan to research a more comprehensive data model for the types of, and relationships between, the data that comprise a single configuration. We also plan to extend our SCM model to allow for single projects with dependent components archived separate repositories (e.g. CVS “modules”), the names and locations of which may also evolve over time.

- ***Fact Extractor Support.*** We have shown that the invocation of third-party fact extractors from Kenyon is already well supported. Our data model for extractor results, on the other hand, did need to be modified and refined as we gained experience from applying different types of fact extractors. Our current efforts are focused on applying the lessons learned from the graduate seminar class, specifically with respect to language support and “partial result” support, such as those generated by AST analyzers or code clone detectors. Our future work in language support includes the adoption of the Java containment model created by Godfrey, as well as finalizing the newly created Recoder[4] fact extractor subclass.

[4] recoder.sourceforge.net

- ***Data Storage/Access Support.*** We plan to adopt Hibernate 3 as soon as it stabilizes. The anticipated performance and mapping improvements alone make it worth the effort to convert, even if support for collections of collections remains weak. The adoption of an ORM framework, instead of requiring a specific implementation, is desirable but not a high priority.

7. CONCLUSIONS

Kenyon was created in order to facilitate both the creation of new software evolution analysis tools and the sharing of data between such tools. We have provided requirements that any software evolution infrastructure tool should support. To date, Kenyon comes closest to simultaneously supporting all these requirements. It provides a flexible, extensible, and scalable infrastructure that addresses the common logistical software evolution issues of configuration retrieval, fact extractor invocation, and database storage and access. Kenyon makes it possible to separate these common concerns from evolution research concerns by allowing analysis-specific fact extraction tools to be invoked on each of a set of automatically retrieved historical configurations.

Kenyon supports many different types of stratigraphic software evolution research, from code feature evolution analyses [12, 13] to dependence graph-based maintenance history analysis [4]. It can also serve as an automated data collection infrastructure for development-assisting recommendation systems such as those created by Ying et al. and Zimmerman et al. [18, 20]. It has been used with five heterogeneous third-party fact extractor tools, three different software configuration management systems, and has processed both industrial and open-source systems up to 500 KLOC in size. We have tested Kenyon's ability to provide commonly usable data structures during preliminary integrations with two distinct tools [4, 11]. We look forward to improving our support of software evolution research as other evolution analysis systems look to adopt Kenyon.

8. ACKNOWLEDGMENTS

Thanks to Lijie Zou at the University of Waterloo for her insight into Beagle, to Miryung Kim at the University of Washington for her feedback while using Kenyon, and to the students of the 2005 winter-quarter UCSC CS290G course. Thanks also to Paul Anderson at Grammatech for his help with Kenyon's support and use of CodeSurfer. Kenyon was originally a subsystem of IVA, a software evolution analysis tool currently funded by NSF Grant CCR-01234603 and a Cooperative Agreement with NASA Ames Research Center. Kenyon and its associated documentation are available for download from its project web page at www.cse.ucsc.edu/research/labs/grase/kenyon/.

9. REFERENCES

[1] Alonso, O., Devanbu, P., and Gertz, M., "Database Techniques for the Analysis and Exploration of Software Repositories," In MSR '04 [15], pp. 37-41.

[2] Bauer, C. and King, G., *Hibernate In Action, Practical Object/Relational Mapping*: Manning Publications, 2004.

[3] Belady, L. A. and Lehman, M. M., "A Model of Large Program Development," *IBM Systems Journal*, vol. 15(3), 1976, pp. 225-252.

[4] Bevan, J. and Whitehead Jr., E. J., "Identification of Software Instabilities," Proc. of Working Conference on Reverse Engineering (WCRE '03), Victoria, BC, Canada, Nov. 2003, pp. 134-143.

[5] Cederqvist, P., "Version Management with CVS" http://www.cvshome.org/docs/manual/

[6] Conklin, M., Howison, J., and Crowston, K., "Collaboration Using OSSmole: A Repository of FLOSS Data and Analyses," In MSR '05 [16], pp. 116-120.

[7] Cubranic, D. and Murphy, G. C., "Hipikat: Recommending Pertinent Software Development Artifacts," Proc. of 25th Int'l Conference on Software Engineering (ICSE '03), Portland, OR., May, 2003, pp. 408-418.

[8] Draheim, D. and Pekacki, L., "Process-Centric Analytical Processing of Version Control Data," Proc. of Int'l Workshop on Principles of Software Evolution (IWPSE '03), Helsinki, Finland, Sept., 2003, pp. 131-136.

[9] Fischer, M., Pinzger, M., and Gall, H., "Populating a Release History Database from Version Control and Bug Tracking Systems," Proc. of Int'l Conference on Software Maintenance (ICSM '03), Sept. 2003, pp. 23-32.

[10] German, D., "Mining CVS Repositories, the softChange experience," In MSR '04 [15], pp. 17-21.

[11] [Godfrey, M. and Zou, L., "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," *IEEE Transactions on Software Engineering*, vol. 31(2), Feb. 2005, pp. 166-181.

[12] Kim, M., Sazawal, V., Notkin, D., and Murphy, G. C., "An Empirical Study of Code Clone Genealogies," to appear, in Proc. of Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '05), Lisbon, Portugal, Sept. 2005.

[13] Kim, S., Whitehead Jr., E. J., and Bevan, J., "Analysis of Signature Change Patterns," In MSR '05 [16], pp. 64-68.

[14] Leblang, D., "The CM Challenge: Configuration Management that Works," in *Configuration Management*, W. F. Tichy, Ed.: John Wiley & Sons, 1994, pp. 1-38.

[15] *Proc. 1st Int'l Workshop on Mining Software Repositories (MSR '04)*, Edinburgh, Scotland, U.K., May 25, 2004. ACM.

[16] *Proc. 2nd Int'l Workshop on Mining Software Repositories (MSR '05)*, St. Louis, MO, USA, May 17, 2005. ACM.

[17] Walls, C. and Richards, N., *XDoclet In Action*: Manning Publications, 2003.

[18] Ying, A. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C., "Predicting Source Code Changes by Mining Change History," *IEEE Transactions of Software Engineering*, vol. 30(9), Sep. 2004, pp. 574-586.

[19] Zimmerman, T. and Weissgerber, P., "Preprocessing CVS Data for Fine-Grained Analysis," In MSR '04 [15], pp. 2-6.

[20] Zimmerman, T., Weissgerber, P., Diehl, S., and Zeller, A., "Mining Version Histories to Guide Software Changes," Proc. of Int'l Conference on Software Engineering (ICSE '04), Edinburgh, Scotland, UK, May 2004, pp. 563-572.