CLIME: An Environment for Constrained Evolution

Demonstration Proposal

Steven P. Reiss, Christina M. Kennedy, Tom Wooldridge, Shriram Krishnamurthi
Department of Computer Science
Brown University
Providence, RI 02912
{spr,cmkenned,twooldri,sk}@cs.brown.edu

Abstract

We have built a software development environment that uses constraints to ensure the consistency of the different artifacts associated with software. This approach to software development makes the environment responsible for detecting most inconsistencies between software design, specifications, documentation, source code, and test cases. The environment provides facilities to ensure that these various dimensions remain consistent as the software is written and evolves. The environment works with the wide variety of artifacts typically associated with a large software system. It handles both the static and dynamic aspects of software. Moreover, it works incrementally so that consistency information is readily available to the developer as the system changes. We propose to demonstrate this environment and its capabilities.

1. Introduction

Software is multidimensional. Software systems consist of a wide variety of artifacts such as specifications, design diagrams and descriptions, source code, test cases, and documentation. Each of these dimensions describes only a limited part of the software — the actual system is properly the combination of all the artifacts.

Software evolution is the process whereby software changes to meet changing requirements, systems, or user needs. A major problem with software today is that the different artifacts of a software system tend to evolve at different rates. The result is that developers learn not to trust and thus not to use anything other than the source code, making software less reliable and much more difficult to understand and evolve.

We are in the process of developing a software development environment that addresses these issues using a constraint-based mechanism. The environment defines and analyzes the consistency of constraints on the software system, including ones that span different dimensions.

This environment provides several capabilities. First, it automatically extracts relevant information from each of the software artifacts. Second, the environment stores and maintains this information in a database, doing incremental updates automatically as the software changes. Third, the environment uses this information along with a description of the types of constraints to be generated to build the complete set of constraints for the software system. Fourth, it uses the information in the database to incrementally test the validity of these constraints. Finally, it provides facilities for presenting the results of these tests to the developers so that they may take steps to resolve inconsistencies.

2. Related Work

Conceptually, the simplest approach to ensuring the consistency of different aspects of software is to combine them all within a single programming language. Several environments such as Xerox Cedar Mesa environment [20] and Common Lisp [18] have combined documentation with code. These efforts led to literate programming [3,12] and, more recently, the use of *javadoc* and its corresponding conventions. Environments like Visual Studio combine code and user interface design. Proponents of UML propose writing complete systems within its framework, thus making it a programming language that combines design with code. Batory [2] lifts this idea to the level of modules that encapsulate code, documentation and other dimensions; however, these must all compose through the same mechanism. This it not only very restrictive, it is unclear how, for instance, to compose text the same way we compose code. Other recent work looks at the impact of evolution of code but ignores the other dimensions [17].

There are a number of systems that check the consistency of single aspect of software. Lint [16] and successors such as CCEL [5] and LCLint [7] perform static checking of programs. Style checkers such as Parasoft's tool suite or the checkstyle project perform style and convention checking of programs. Systems such as ViewIntegra [6] and xlinkit [11] have been used to check the consistency of UML diagrams. There are also a broad range of tools for doing test coverage, languages such as Eiffel that include checkable specifications in the code, and systems such as Flavors [4] do static checking of external specifications.

The closest work of these to ours is the xlinkit approach [14] as applied to software engineering. Xlinkit provides the general ability to check the consistency of multiple XML documents. XML documents can either be specified directly or can be derived from other artifacts. The constraints use a set-based XML query language based on XPath and XLink. The current system is able to handle very large documents using a disk-based representation and is able to do limited incremental checking of con-

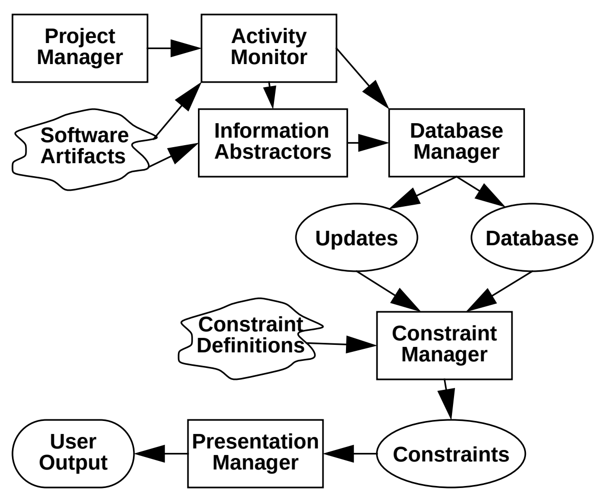


FIGURE 1. The architecture of the environment.

straints by looking at what portions of the XML tree have changed.

Our efforts differ in several respects. First, rather than using XML and XPath, we use a relational framework and SQL queries. This provides a more powerful query language and eliminates the need to treat large documents separate from small ones. Second, our system does incremental update of both the internal representation and the constraints and does incremental constraint update at the constraint level rather than the rule level and thus can be used continuously throughout the development process. Xlinkit would require that XML files be generated for any changes and does incremental update of constraints at the rule level. Third, our system handles a broader range or software artifacts, both static and dynamic. For example, we handle test cases and coverage, UML interaction diagrams, and behavioral specifications. Finally, our system works within existing programming environments, existing programming tools, and existing methodologies, and does so without requiring any action from the programmer.

3. Environment Architecture

The overall environment consists of the components shown in Figure 1. The components can be broken into two parts: the first part manages extracting the necessary information from the source artifacts while the second part uses this information to find, update, and display information about the constraints.

The first problem that we had to deal with was to identify the artifacts of a particular software system. In our environment this is done by the project manager component of Figure 1. This component lets the developer define a project using an XML file that describes the components. It assumes that artifacts are represented by files in the file system. The project description file then provides either a list of particular files or a list of directories to search for files. It also lets the developer specify which subdirectories or files should be excluded from the search (e.g., version information and editor backups are candidates for exclusion).

Once we know what the artifacts are, we need to process them. This involves identifying the particular information that is needed to describe and understand a specific software dimension and then extracting that information from the artifact and putting it into a form that can be easily understood by later tools. The environment handles such abstractions for a wide variety of different artifacts. The current abstractors include:

- Symbol table information from compiler analysis.
- Documentation information from a JavaDoc doclet.
- Semantic information derived from Soot Java Optimization Framework [21].
- UML class and sequence diagram information derived from the XMI representation or from Rational Rose's mdl files.
- Test cases and test coverage information derived from *Junit* **[10]**, and a run time instrumenter based on IBM's *JikesBT* package [13].
- History information derived from the CVS version management system [8].

In addition to information that is abstracted directly from source artifacts, we allow additional information that is not directly reflected in the artifacts. Some of this information represents global information such as style and language usage rules. Other information represents design patterns (and have done so for several of the patterns in Gamma, et al. [9] and our previous work [15]), where we specify the instances of patterns that occur in the system. We also allow specifications of usage conventions for classes or packages. Though there have been efforts, both old [19] and new [1], to validate software against such specifications, usage information is usually specified through informal comments and documentation, and cannot be easily captured in a tool. We allow extended automaton-based specifications to be defined and then checked within the system.

The database management component of the environment is responsible for taking the information that is generated by the various abstractors and inserting it in the database. It consists of three modules shown in Figure 1: an activity monitor, a command generator and a database manager. The activity monitor runs periodically to detect which software artifacts have been modified by the developer. When it detects such modifications, it runs all the appropriate information abstractors, and collects the names of the resultant data files. The command generator consumes the information in these data files and packages it as a set of additions and deletions to w the database.

The database manager itself has three primary responsibilities. It first needs to process the commands that are provided by the command generator, adding and removing tuples in the database. Second, it needs to manage unique identifiers so that references in previous constraints are maintained and incremental constraint checking is possible. Finally, it needs to generate a file describing what has changed in the database for incremental update.

Our environment differentiates between metaconstraints and constraints. Metaconstraints are rules for generating specific constraints. These have the form $\forall(x \in S)\varphi(x)\Theta(x)$. Here S is a relation in the database and x represents a tuple of that relation, φ(x) indicates the conditions under which the constraint is applicable, and Θ(x) is a qualified predicate the specifies the conditions the constraint must meet.

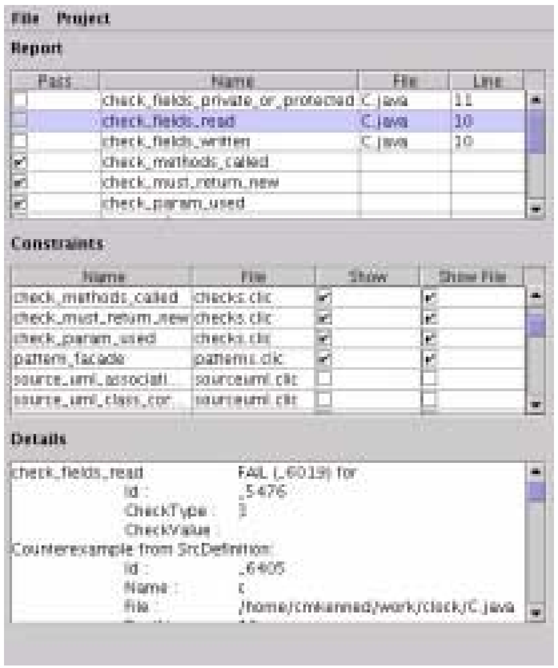
These formulas are translated into SQL queries. In particular, the constraint manager is able to generate two types of queries from each formula. The first is designed to generate the set of UIDs that correspond to particular instances of a metaconstraint along with a Boolean value indicating whether the constraint holds or not. This query can be issued over the whole database or only for a particular set of UIDs. The query is issued over the whole database when the constraint set is initially created and when the query definition file has changed. Otherwise, the query is restricted to the set of UIDs that have been added or changed for the source table. The second type of query that is built by the constraint manager from the metaconstraint definition is used to generate the dependencies for the constraint. These dependencies are used in two ways. First, they are used to report information to the developer about why a constraint may or may not hold. Second, they are used by the constraint manager to determine when a particular constraint instance needs to be rechecked after an update to a set of software artifacts. The information about the individual constraints in then stored in the database.

The last part of the environment, the presentation manager, provides information about the constraints and their state to the user. We currently provide two interactive interfaces here, one that is a standalone interface for browsing over the constraint information as seen in Figure 2. and the other which is a plugin to IBM's Eclipse environment.

4. Consistency Checking

The above architecture can be used to check a broad range of consistency conditions among software artifacts. The current conditions that are checked include:

- Constraints from UML class diagrams on the source code that indicate that every class has a corresponding source class, that every interface matches a source interface, that class and interface generalizations match the actual class hierarchy, that UML operations correspond to methods, that UML attributes correspond to fields, and that UML associations are reflected in the source.
- Constraints from the source code on UML class diagrams that ensure that all public classes and interfaces appear in the UML model, that all generalizations among public classes and interfaces are reflected by UML generalizations, that public methods and fields of public classes are reflected in the UML, and the associations based on fields are reported in the UML.
- The sequence of calls in a UML interaction diagram is realizable by the code.
- Constraints from test cases on the source code that ensure that a test case that covers a particular method has been run since the method was last modified.
- Constraints from documentation on the source code that ensure that any specification of parameters, see also links, and throws clauses correspond to the current source code.
- Constraints from the source code on documentation that ensure that all public methods of public classes or interfaces are documented.
- Naming conventions for classes, interfaces, methods, fields, local variables and constants.
- Pattern constraints for the Facade and Singleton patterns.
- Language usage conventions including the fact the all parameters not starting with an underscore are actually used, that all fields are read and written at least once, that all methods are called at least once, and that data fields are either private or protected.
- Test consistency checking that ensure that all procedures are covered by at least one test case which has been run since the procedure was last changed.
- Dynamic specifications of class usage such as Java Iterators must always call *hasNext* before calling *next* and files that are opened must be closed.



Report

Pass	Name	File	Line
☐	check_fields_private_or_protected	C.java	11
☐	check_fields_read	C.java	10
☐	check_fields_written	C.java	10
✓	check_methods_called		
✓	check_must_return_new		
✓	check_param_used		

Constraints

Name	File	Show	Show File
check_methods_called	checks.clic	✓	✓
check_must_return_new	checks.clic	✓	✓
check_param_used	checks.clic	✓	✓
pattern_facade	patterns.clic	✓	✓
source_uml_associati...	sourceuml.clic	☐	☐
source_uml_class_cor...	sourceuml.clic	☐	☐

Details

```
check_fields_read            FAIL (_6019) for
              Id :           _5476
              CheckType :    1
              CheckValue :
Counterexample from SrcDefinition:
              Id :           _6405
              Name :         C
              File :         /home/cmkanned/work/clack/C.java
```

FIGURE 2. Standalone constraint interface.

- Configuration management checking to ensure that all files that are checked in through CVS have a corresponding log message.

5. Experience

We have been using our constraint-based environment both on itself and for a small set of development projects, mainly to validate that the underlying systems work and that the approach is a viable one.

The system is currently used in its own development (40,000 lines of Java plus about 250,000 lines of external Java libraries). Our experience has been that we did not even notice when the updates were occurring. Moreover, periodic checks of the violated constraints showed them to be accurate and helpful in finding potential (or in some cases real) problems with the system. This experiment also validated our approach to incremental update of both the abstraction information and the constraints.

Finally, we ran experiments to see how and whether the approach can scale to larger systems. Specifically, we defined a project for the SOOT package for Java code analysis and optimization and generated the set of abstraction information and constraints. SOOT has approximately 200,000 lines of Java source The generated database here is about 140Mb in size and has about 15,000 constraints. While the initial setup of this database took about an hour, incremental updates have been extremely fast, i.e. well under a minute.

6. Demonstration

We plan to demonstrate various aspects of the system. The demonstration will take the viewer through the definition of a constraint through to seeing where and how the constraint is violated in the system. In doing this we will show how the system does incremental constraint analysis and how it is able to pinpoint the source of violations.

The demonstration will show both static and dynamic constraints, show our modified test coverage tool, and illustrate how both the standalone and the Eclipse-based user interface can be used.

Finally, the demonstration will illustrate the performance of the environment both in terms of setting up the initial set of constraints and for doing incremental update of that set.

7. References

1. Thomas Ball and Sriram K. Rajamani, “The SLAM project: debugging system software via static analysis,” *ACM Principles of Programming Languages*, pp. 1-3 $K ballslam (January 2002).

2. Don Batory, David Brant, Michael Gibson, and Michael Nolen, “ExCIS: an integration of domain-specific languages and feature-oriented programming,” in *Workshops on New Visions for Software Design and Productivity*: *Research and Applications*, (dec 2001).

3. Bart Childs, “Literate programming, a practitioner’s view,” *TUGboat, Proceedings of the 1991 annual meeting of the Tex User’s Group* Vol. **12**(3) pp. 1001-1008 (1991).

4. J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil, “FLAVERS: A finite state verification technique for software systems,” *IBM Systems Journal* Vol. **41**(1) pp. 140-165 (2002).

5. Carolyn K. Duby, Scott Meyers, and Steven P. Reiss, “CCEL: a metalanguage for C++,” *Proc. Second Usenix C++ Conference*, (August 1992).

6. Alexander Egyed, “Scalable consistency checking between diagrams -- the ViewIntegra approach,” *16th IEEE Intl Conf on Automated Software Engineering*, (November 2001).

7. David Evans, John Guttag, James Horning, and Yang Meng Tan, “LCLint: a tool for using specifications to check code,” *Software Engineering Notes* Vol. **19**(5) pp. 87-96 (December 1994).

8. Karl Fogel, *Open Source Development with CVS*, CoriolisOpen Press (1999).

9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley (1995).

10. E. Gamma and K. Beck, “Test infected: Programmers love writing tests,” *http://www.junit.org*, (1998).

11. Clare Gryce, Anthony Finkelstein, and Christian Nentwich, “Lightweight checking for UML based software development,” *2002 Workshop on Consisteny Problems in UML-based Software Development*, (2002).

12. Donald E. Knuth, “Literate programming,” *The Computer Journal* Vol. **27**(2) pp. 97-111 (1984).

13. Chris Laffra, Doug Lorch, Dave Streeter, Frank Tip, and John Field, “What is Jikes Bytecode Toolkit,” *http://www.alphaworks.ibm.com/tech/jikesbt*, (March 2000).

14. Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein, “xlinkit: A consistncy checking and smart link generation service,” *ACM Transaction on Internet Technology*, (To appear).

15. Steven P. Reiss, “Working with patterns and code,” *Proc. HICSS-33*, (January 2000).

16. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, “The C programming language,” *Bell Systems Tech. J.* Vol. **57**(6) pp. 1991-2020 (1978).

17. Barbara G. Ryder and Frank Tip, “Change impact analysis for object-oriented programs,” *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 46-53 (June 2001).

18. Guy Lewis Steele, Jr., *Common Lisp*: *the Language*, Digital Press, Bedford, MA (1990).

19. Robert E. Strom and Shaula Yemini, “Typestate: a programming language concept for enhancing software reliability,” *IEEE Transactions on Software Engineering* Vol. **12**(1) pp. 157-171 (1986).

20. Warren Teitelman, “A tour through Cedar,” *IEEE Software* Vol. **1**(2) pp. 44-73 (April 1984).

21. Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co, “Soot - a Java optimization framework,” *Proceedings of CASCON 1999*, pp. 125 -135 (1999).