

CLIME: An Environment for Constrained Evolution

Demonstration Description

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 are building 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. The demonstration will show 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 com-

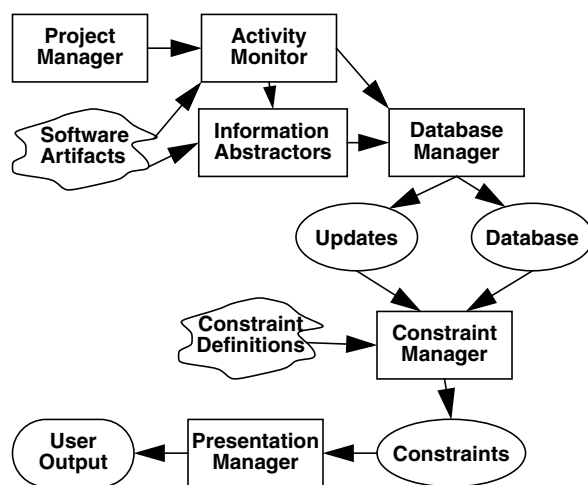


FIGURE 1. The architecture of the environment.

plete 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. Environment Architecture

The overall environment consists of the components shown in Figure 1¹. 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 *Project Manager* determines what files are part of the current project. This information is used by the *Activity Monitor* which determines when the files change, and then uses appropriate *Information Abstractors* to get relevant data about the corresponding software artifacts. This information is passed to the *Database Manager* which first determines what has actually changed and then updates the database according to those changes. We currently handle

1. Reiss, Steven P. "Constraining Software Evolution", *Proc. Intl. Conf. on Software Maintenance*, October, 2002, pp. 162-171.

source files, UML diagrams, style rules, design patterns, semantic information, test cases, and history information in this first part of the system.

The *Constraint Manager* is based on metaconstraints of 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, $\varphi(x)$ indicates the conditions under which the constraint is applicable, and $\Theta(x)$ is a qualified predicate that specifies the conditions the constraint must meet. It is passed the set of items that change in the database by the database manager, and uses the metaconstraints to generate appropriate SQL queries to create, update, and check the set of constraints that should be imposed on the system. This is done incrementally and can generally be done in background as the system changes.

Finally, the set of constraints are displayed to the user through the *Presentation Manager*. This tool gives the user the option of prioritizing the constraints and organizing the constraints as well as providing an understanding of what each constraint means and to what it applies.

3. 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.
- Configuration management checking to ensure that all files that are checked in through CVS have a corresponding log message.

4. 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.

5. Demonstration

We will 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.