Incremental Maintenance of Software Artifacts

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI. 02912
spr@cs.brown.edu

Abstract

We have built a software development tool, CLIME, that uses constraints implemented as database queries to ensure the consistency of the different artifacts of software development. This approach makes the environment responsible for detecting inconsistencies between software design, specifications, documentation, source code, and test cases without requiring any of these to be a primary representation. The tool works incrementally as the software is written and evolves without imposing a particular methodology or process. It includes a front end that lets the user explore and fix current inconsistencies. This paper describes the techniques underlying the tool, concentrating on the user interface and the incremental maintenance of constraints between these artifacts.

1. Introduction

Software is multidimensional. Software systems consist of a variety of artifacts such as specifications, design documents, source code, test cases, and documentation. Each of these dimensions describes only a limited part of the software — the actual system is their composite.

Software evolution is the process whereby software changes to meet changing requirements, systems, or user needs. A major problem in software development today occurs when the artifacts of a software system evolve at different rates. In some cases the source code will be updated for bug fixes or enhancements, but the specifications and design documents will not be modified to reflect these changes. In other cases the design will be changed to reflect a new feature before the code is updated. Test cases may be thorough for the initial system but, in the absence of a proper development methodology, tend to get overlooked with the addition of new features. Developers are familiar with the manner in which implementation changes take a long time to percolate to the documentation. Company documentation and coding standards often change, but old code is not brought up to date. 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.

To support consistent software evolution, one needs to ensure that all the artifacts remain consistent with one another. Moreover, one needs to do this while giving the programmer the freedom needed to develop the software in the best possible way. Some changes are best done at the source code level, for example performance enhancements. Other changes, such as new features, might best be done through design. Test cases can be written first following agile methodology, or written after the fact. Documentation is generally done when the coding is complete, but many programmers find it helpful to document the interface methods before the code is ever written in order to understand its functionality and needs. Different companies and different projects require different methodologies.

We have designed and built CLIME, a prototype tool that addresses these issues using a constraint-based mechanism for Java systems [24]. The tool is designed to tell the developer when artifacts become unsynchronized and to indicate what needs to be changed or updated to achieve system-wide consistency. The tool is methodology-independent. Our goal has been to assist the developer by providing the information needed to ensure consistency. We have been careful to ensure that the tool does not restrict developers but instead enhances development and evolution. Our hypothesis is that providing such information will lead to consistent and hence better software.

Our tool defines less rigid but still formal semantics for each type of artifact. Rather than using a common representation, we define the meaning of the artifact in terms of the constraints it imposes on other artifacts. This technique is surprisingly effective. For example, constraints can ensure consistency between UML class and interaction diagrams and the source, source code and documentation, test cases and source code, design patterns and either the design or the source. Constraints can also be used in the same way to check language or usage conventions and to check the development process, for example ensuring that files are tested before they are checked in.

The tool works in phases. It first extracts relevant information from each software artifact and stores it in a relational database. Next, it uses this stored information along with a description of the constraints among the artifacts to build the complete set of constraints for

the software system. Third, it uses the information in the database to test the validity of each of these constraints by mapping each constraint into a database query. Finally, it presents the results of these tests to the developers so that they can resolve any inconsistencies.

While [24] reports on the basic concepts behind the tool, this paper concentrates on the implementation aspects, particularly the user interface and incremental update. Having a good user interface is essential to providing a usable tool. Moreover, the key to making this approach practical is to ensure the information presented in the interface is accurate and timely. Even a limited software project with a limited set of artifacts such as CLIME itself (about 68,000 lines), involves over 10,000 constraints and a database containing 140Mb of data. Since most of the information does not change frequently, it is essential that the tool work incrementally, i.e. that the database is maintained incrementally and the set of constraints is updated and checked incrementally.

2. Related Work

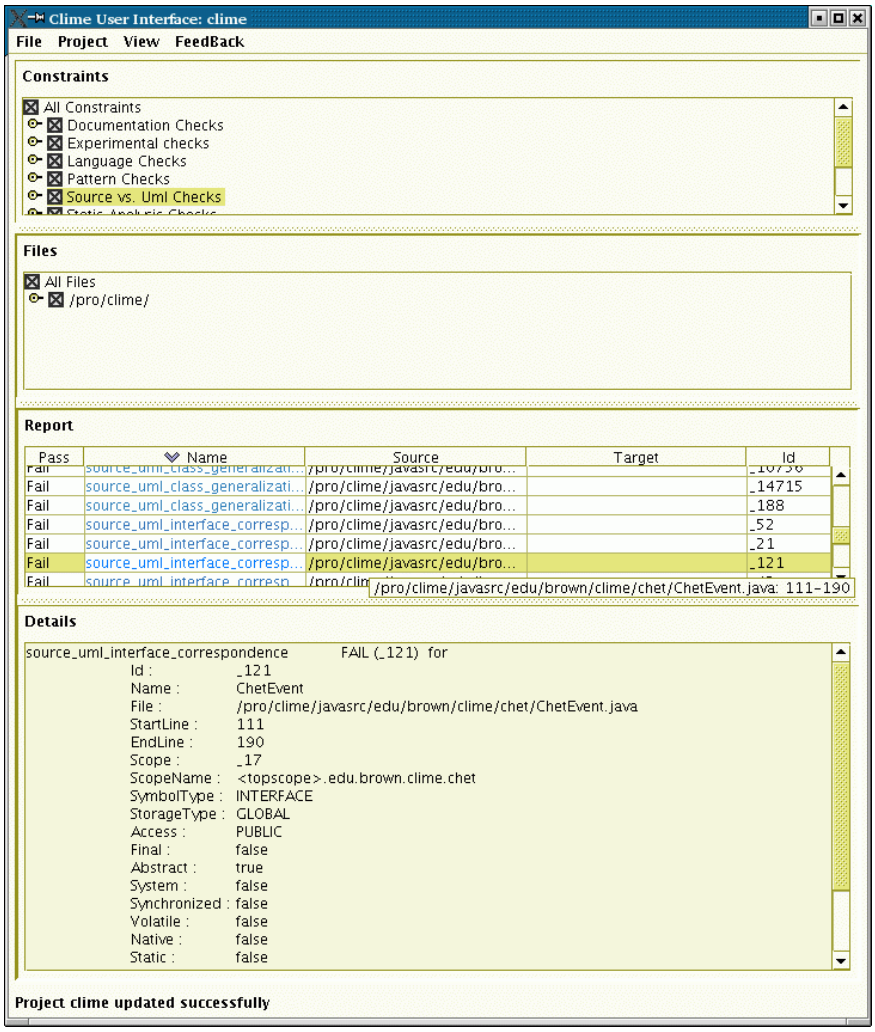
Conceptually, the simplest approach to ensuring the consistency of different aspects of software is to combine them all within a single programming language. Several environments such as Xerox Cedar Mesa environment [31] and Common Lisp [30] have combined documentation with code. These efforts led to literate programming [3,17] 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 [1] 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 [27].

Current work in this area attempts to build the program directly from the design using model-driven development and model-driven architectures [5,16,20,28]. Here the various UML views of the system are augmented with enough details to have code generate directly from the UML model. The system evolves by evolving the model. While there are potential problems with this approach, if it can be made to work it at least ensures that the code is consistent with some aspects of the formal design. It does not, however, address the whole problem of consistent software evolution in that it only addresses the UML design and the code and does not address all the other software artifacts, e.g. other design documents, documentation, design patterns, requirements, etc. Moreover, it constrains the developer to particular methodologies.

Consistency checking has been widely done on single software artifacts. Lint [26] and successors CCEL [6] and LCLint [8] perform static checking of programs. Style checkers such as Parasoft's tool suite or the *checkstyle* project perform style checking of programs. Systems such as ViewIntegra [7] and xlinkit [13] 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. Other systems check for the existence of design patterns in the code [2,12,19,23]. The Eclipse and NetBeans environments provided language style and usage checking including checking documentation against the source, most of which we have incorporated into our constraints. Ophelia attempts to provide a common framework for multiple tools, with integrators to translate between the representations and a traceability mechanism for defining explicit dependencies between artifacts [14,29]. Constraints have also been used to express language properties as in Minsky's work [21] and to specify interactions in the software process [15].

The closest work of these to ours is xlinkit [22] 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 constraints 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, configuration management, 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.



IGURE 1. Textual constraint presentation interface of CLIME.

3. The User Interface

Our tool is designed to provide the user with the information needed to maintain the consistency of different software artifacts. The objective here is to make it easy for the developer to identify and then fix inconsistencies.

In order to remain methodology independent, we have implemented the user interface for the tool as a standalone system as seen in Figure 1. The user interface provides the developer with the necessary facilities.

First, it provides a series of dialogs to let the developer define and edit the set of artifacts that compose the system, where to find them, and how to interpret them. This is a necessary starting point for finding inconsistencies.

Next, it provides facilities to update the consistency checks as files change in the project. The updates can either be done automatically in background or on demand by the user. Because the updates are incremental and reasonably fast, we have found the latter to be more useful since the developer generally knows when the artifacts have reached a stable state and doesn't want to see consistency checks that reflect the intermediate states.

The principle portion of the interface is designed to let the developer quickly browse over the various consistency checks. The top pane of the interface in Figure 1 lets the user select a set of relevant constraints. The constraints are arranged hierarchically. The second pane lets the user select the relevant files. The hierarchy here represents the directory hierarchy with singletons automatically combined with their parent. These two panes can be used in two ways. First, they can be used to selectively disable particular constraint types or files. Second, they can be used to quickly select a particular set of constraints or files that should be displayed.

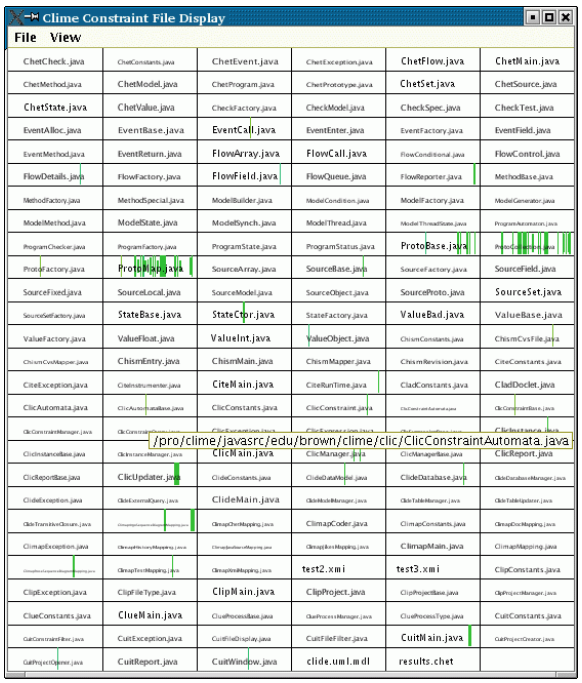


FIGURE 2. Graphical view of violated constraints in CLIME.

The third pane provides a tabular display of the constraints selected by the upper two panes. This can be sorted by constraint type, originating location, or violating location. Moreover, the user can select a particular constraint in this pane and have the detailed information concerning that constraint displayed in the bottom pane. The user can also double click on a constraint to bring up an appropriate editor to fix a constraint violation. Tool tips are provided to give the user additional, detailed information about the constraint or its location.

An alternative, graphical view of the constraints provided by the tool is shown in Figure 2. Here each artifact or file is displayed as a box and consistency violations relevant to that artifact are displayed as colored rectangles indicating the relative position in the file. The color coding indicates the type of violation and is tied back to the constraint display in the main window. Tool tips give the user information about the displayed constraints. Rectangles are nested so multiple constraints at the same position do not hide each other. This view lets the developer see at a glance which files have potential problems. Moreover, the two views are tied together. The developer can click on a displayed rectangle in the graphical view to get information on the particular constraint in the textual view. The constraints displayed in the graphical view are those that are selected by the constraint and file panes of the textual view. Finally, the currently selected constraint in the textual view is highlighted in the graphical view by making the corresponding rectangle brighter.

4. Overview of the Implementation

The overall tool is implemented using the components shown in Figure 3. These can be divided into two parts: the first part handles extracting the necessary information from artifacts while the second part uses this information to find, update, and display information about the constraints.

The *Project Manager* keeps track of the artifacts that compose the project and is able to detect, using the *Activity Monitor*, which have changed since the previous update.

The *Information Abstractors* gather as much information as possible from each of the artifacts that have changed and format this information in terms of relational tables, with each abstractor generating its own set of tables. Most of these are in two parts. The first part takes the artifact, isolates the information relevant to a particular dimension, and then generates an XML file with that information. The second part reads this XML file and then generates a second XML file consisting of commands describing what should be removed, inserted or updated.

The *Database Manager* is then responsible for incrementally updating the artifact information, attempting to minimize the changes to the database and maintaining identifying information across updates. In addition to updating the database, it generates an XML file describing what has changed in the database.

The *Constraint Manager* that is in charge of finding and updating constraints and storing the resultant constraint information in another database. It does this using a set of metaconstraints that define generic relationships, using these to find specific constraints, and then checking the constraints as the underlying data is changed. The *Presentation Manager* provides the user interface.

5. Extracting Information

Information is extracted from the various software artifacts using an array of tools each oriented to a particular type of artifact. Changes are detected and updated at the file level. That is, when the *Activity Monitor* detects that a particular artifact has changed, it causes all the information in that artifact to be updated in the system. The file level seems to be the most appropriate for updating for such information because it is easy to detect when a file changes and because it is easiest to write abstractors that work on a whole file at once. While it is most common for only a small amount of information in the file to actually change, detecting and abstracting only that information while maintaining consistency with the remainder of the database would have made all the extractors much

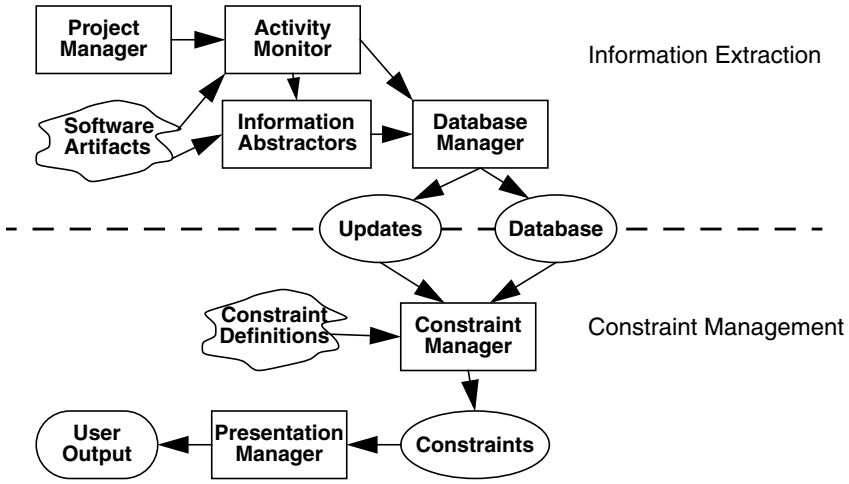


FIGURE 3. The architecture of the CLIME tool.

more complex and slower. We opted instead to put the burden of determining the actual incremental update in the *Database Manager* and to keep the abstractors fast and simple. This makes it easier to write abstractors and lets us globally optimize incremental updates.

The current set of abstractors includes:

- Symbol table information. This includes information about the symbol type, data type, access information, and location of each definition, the location and definition associated with each reference, and information about data types including the class hierarchy. It is generated by running a slightly modified version of IBM Jikes Java compiler that generates appropriate descriptions from the abstract syntax tree.
- Documentation information. This includes information about all Javadoc comments and the tags that they contain. It is generated by our own doclet (a applet-like plug-in for javadoc) which generates an XML description of the available documentation-related information.
- Semantic information. Language usage such as assignments inside conditionals, unnecessary duplication of constant strings, or unterminated switch cases are best detected from the abstract syntax trees. This is done as part of our Jikes compiler extension.
- UML class diagrams. This includes information about classes, attributes, operations, parameters, associations, and generalizations. It is extracted directly from the XMI (standard XML for UML) representation which is either the native representation of the UML tool or, in the case of Rational Rose, using a conversion package that generates XMI from the native representation.
- Test cases. We assume that the developer is using *Junit* [10], a common Java testing package. The information extractor reads the compiled Java class files using IBM's *JikesBT* package [18]. It finds all classes that are instances of test groupings and then identifies those functions that are actual test cases. It then patches the class files to capture flow information and runs *Junit* using the instrumented class files. The instrumentation calls routines that record each basic block entry, each call, as well as the entry and exit of the test cases. The result of running the instrumented code is an XML file for each test case that includes a description of the test case, the date and time it was run, whether it succeeded or failed, and coverage information for blocks, branches, functions, and calls.
- UML sequence diagrams. This includes information about the signature and class of call points as well as the method bodies and order in which they occur. As in the case of UML class diagrams, this information is extracted from an XMI representation.
- Configuration information. This includes information about all the past versions of each version-managed software artifact. The information that is recorded includes version history, author, descriptions, and change information. This information is obtained by requesting complete *CVS* log information [9] for each file.
- Static checks. We have a tool that does a full interprocedural flow analysis of a Java system and then checks if various dynamic contracts (such as proper use of Iterators or files) are maintained [25]. The tool also generates information about dead code, possible dereferencing of null pointers, and specification violations.

In addition to information that is abstracted directly from source artifacts, we found the need for additional data that was implicitly but not explicitly part of the software. Some of this data was needed to represent global information that is assumed by the developers, such as rules describing naming and lan-

guage usage conventions. Other information was needed because the set of formal artifacts used today is incomplete. Software development involves dimensions such as design patterns that are not directly represented by existing design tools or representations. We are able to define relations in the database that represent design patterns (and have done so for several of the patterns in Gamma, et al. [11] in our previous work [23]), but we need to manually specify the instances of these patterns that occur in each particular software system for the database. Because our constraint framework is sufficient for checking that these patterns actually exist in the software, we see our manually entered definitions as a placeholder for what will eventually be a useful tool that would let developers specify and maintain a design-pattern-based description of their system.

Dealing with a variety of artifacts forced us to confront the problem of maintaining consistent global names. Since we need to relate information in one artifact with that in another, we need to appropriately link equivalent references. In some cases, such as in UML diagrams, the programmer may provide only partial names, omitting the full package name and only providing the class name. However, in most cases, the abstraction tools have enough information to construct unique names for each package, class, method, field, scope, etc. We adopted a naming convention similar to that used in Java and required each abstraction tool to generate a field with this name for each appropriate entity. Our convention differs in that it includes scope names and can thus use names down to the scope level and is extended to include local variables and non-language constructs such as UML links. In addition, we added extra fields to various relations, such as source types, that contain abbreviated names that can be used for approximate matching.

The output of each of the abstractors is a set of commands to the *Database Manager.* The commands are of two forms. The first indicates that all data in a given database table meeting a criterion are to be deleted. This is used to remove all the old information that is associated with a particular artifact when that artifact changes. The second form indicates a new tuple to be added to a particular table. By organizing the data independently of the information source, this greatly simplifies the actual database manager. Moreover, identifying deleted and added information explicitly is necessary for incremental database update.

6. Storing and Updating Information

The *Database Manager* has four primary responsibilities. It first processes the commands from the information abstractors incrementally, adding and removing tuples in the database. Second, it manages unique identifiers in order to maintain linkages among tables and between the data and the constraints. Third, it maintains dynamic relations in the database. Lastly it generates a file describing what has changed in the database so that constraint processing can also be done incrementally.

The commands to the database manager describe sets of tuples to be added and removed from each relation. Typically, they indicate that all tuples that came from a particular artifact should be removed and then provide the new tuples for that artifact. This is true even if only a small change was made to the artifact. This presents two basic problems for an incremental framework. First, it means that the database manager would have to report a relatively large number of changes to the *Constraint Manager* even when only a small amount of data actually changes. Second, it makes tracking unique identifiers that relate information between artifacts more difficult. By solving these two problems, our database manager is able to handle incremental file-based updates efficiently.

The database manager does an intelligent update based on the information it is given. Instead of deleting tuples outright, it reads all the tuples that would otherwise be deleted. It then compares each tuple to be added against those scheduled for deletion. If the new tuple is already in the database, it ignores both the request to remove and insert it. If the tuple exists in the database but non-critical fields such as the line number have changed, it simply updates the changed fields. If the tuple is indeed new, it inserts it into the database. Finally, it removes all tuples that were not otherwise duplicated.

While maintaining the tuples in the relations, the *Database Manager* needs to manage the assignment of unique identifiers (UIDs). Here the database system requires that any relation defining a UID field also specify the set of fields that characterize the tuple and hence essentially define the UID. This set of fields is employed to ensure that the same UID is used to represent the same object through updates. For example, in the relation describing source definitions, the UID is characterized by the name, the scope, the data type, and the type of symbol for the definition. When a tuple with a UID field is to be added to the database, the database manager checks if there is an existing UID assigned to the set of characteristic fields. If so, it will reuse this UID; if not a new UID will be created.

The abstractors generate local UIDs as placeholders for the corresponding positions in the output tuples. The *Database Manager* also takes responsibility for replacing these local UIDs with the appropriate global ones based on the above analysis. Here it builds a mapping table as the queries are processed and then, when actually updating the database, replaces all local UIDs with the corresponding global ones. This lets the abstractors use local UIDs to represent links within the new data and to not have to worry about the global UID name space.

```
CONSTRAINT: source_uml_interface_correspondence
DESCRIPTION: Public interfaces in the source must appear in the UML

FORALL x IN SrcDefinition
    WHERE
        x.SymbolType == INTERFACE AND NOT x.System AND x.access == PUBLIC AND
            EXISTS z IN SrcScope
                WHERE
                    z.id == x.Scope AND z.ScopeType == PACKAGE
    CHECK
        EXISTS y IN in UmlClass
            WHERE
                y.ClassType == INTERFACE AND x.Name == y.TypeName
```

FIGURE 4. Sample metaconstraint definition.

Much of the information contained in software is hierarchical in nature, for example the class structure or scopes. Constraints based on such information often are interested not in the local hierarchy, but rather in the transitive closure of that hierarchy. Since transitive closure is not a normal database operation, our database manager automatically constructs and updates transitive relations as the database changes. Here we are able to define a closure relation for any particular database relation and have the new relation automatically recomputed when changes occur in the base relations. The manager currently maintains transitive closure relations for the class hierarchy, the scope hierarchy, and the static call graph. The database manager also provides for traditional query-based views. Currently, we use these to provide mappings from classes and methods to scopes in order to simplify constraint definition.

The final task of the database manager is to generate an XML description of what has changed in the database. The description identifies which tuples are inserted, deleted, and updated for each table of the database. For tables that have UIDs associated with each tuple the information reported is the UIDs of modified tuples. For tables without associated UIDs, the tool reports that the table changed.

7. Constraints and their Maintenance

Given data about the different dimensions of a software system, the next portion of our tool defines, manages, and presents the constraints that ensure the different artifacts remain consistent as the software evolves. The first step here involves defining what it means for two software artifacts to be consistent with one another. Typically, this will mean that a syntactic or semantic detail defined in one of the artifacts is represented appropriately in the other artifact. Our tool uses a constraint to reflect this association.

While it is not practical to have the developer explicitly define all the constraints that are needed to relate the various artifacts, it is possible to define rules whereby such constraints can be generated. These rules are what we call *metaconstraints*. Metaconstraints 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. This tuple is the source of the constraint. We require that any relation used as the source of the constraint have an associated UID field. This lets us easily identify the source for a constraint and to detect, based on the update file from the database manager, when we might have to check for new constraint instances (if new tuples are added to S), check the continued appropriateness of constraints (when tuples are updated in S), or remove existing constraints (when the source tuple for a constraint is removed from S).

The second part of the constraint definition, $\varphi(x)$, indicates the conditions under which the constraint is applicable, while the third part, $\Theta(x)$, is a qualified predicate that specifies the conditions the constraint must meet. Both of these are arbitrary predicates defined over tables in the database. Variables ranging over the tables can be defined using FORALL, EXISTS, NOTALL, NOTEXISTS, and UNIQUE, operators. Each such variable is meant to represent a tuple. The predicates can also include comparisons, string matching, arithmetic and string operators, and Boolean operations.

An example of a metaconstraint is shown in Figure 4. This constraint specifies that any public interface in the source code appear in the UML as an interface. The actual check looks at all definitions that are interfaces, not in system files, have public access, and are defined in a package scope. For each such definition it creates a constraint to check if there is a class in the UML diagram representing an interface with the same name as the source definition. The metaconstraint definitions for the actual system are coded as XML files that can be defined either globally or for a particular project.

Note that this constraint has to deal with possibly different naming conventions in UML and the source. This can be handled in several ways. The simplest is to define the matching criteria between names as part of the metaconstraint, for example by defining a regular expression that relates the name in the UML to the name in the source. The alternative we use is less general but more efficient. We have the information abstractors generate additional fields that represent

simplified names for appropriate items, for example the *TypeName* for the UML class and *MatchName* for source types. The query then only checks for matching of the simplified names.

The tool takes these metaconstraint definitions and uses them to generate the set of actual constraints for the software system and maintain this set as the software evolves. The actual constraints represent instances of the metaconstraints referring to particular items. For example,. the metaconstraint of Figure 4 would be mapped into a set of actual constraints, one for each public interface in the source. The tool needs to maintain the set of such constraint instances and their validity as the system evolves.

The tool also keeps track of the various metaconstraints, detects when they change, and then updates the corresponding constraints. This lets the programmer add new constraints or modify existing ones and still do incremental updates of the overall system. This is especially useful for project-specific constraints which tend to change periodically and generally for debugging constraint definitions.

Constraint maintenance is accomplished by mapping the metaconstraint formulas into SQL queries. In particular, the constraint manager is able to generate three 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 or when the metaconstraint definition has changed. Otherwise, the query is restricted to the set of modified UIDs since these are the potential candidates for new constraints. This provides for fast, incremental update of the set of constraints that are affected by a change.

The second type of query built by the constraint manager is used to generate the dependencies for the constraint. If the Θ expression uses an EXISTS operator, then the UID for the corresponding tuples that satisfy the expression are the elements that demonstrate the validity of the constraint. Similarly, if the Θ expression uses a FORALL operator, then the UID of any tuple that does not satisfy serves as a counterexample that demonstrates the failure of the constraint instance. The constraint manager will generate a set of queries for each actual constraint, one for each nested EXISTS or FORALL operator that is used in this way, to get the full set of UIDs upon which each particular constraint depends. These queries are generated for any constraint that has changed and are specific to a particular constraint instance.

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. One complication that arises is that some constraints are dependent on all tuples in a table. For example, if a constraint uses a FORALL operator in the Θ expression, then any change to the corresponding database table will require that the constraint be rechecked. To accommodate this, the constraint manager also keeps track of which tables each constraint is dependent upon. This information is determined statically by analyzing the metaconstraint formula.

The third type of generated query is used to update the status of constraints that might have changed. The set of all constraints that need to be checked is computed for each metaconstraint based on the update information passed from the database manager. Then individual queries are constructed to recheck the validity of each element of this set.

The translation of a metaconstraint into a query is relatively straightforward since the expressions used in the metaconstraint definitions parallel those in SQL. For example, the metaconstraint of Figure 4 generates the test query shown in Figure 5. While the queries that are generated look relatively complex, the database system is generally able to handle them efficiently using appropriate indices within the relations. For all the constraints we have defined, the generation query is evaluated using a single table scan so that the complexity of the checks is at worst linear. The update query is then done using indexed lookup on the relevant UIDs and runs in time that should only scale as the log of the size of the database.

The constraint manager keeps track of the set of constraint instances using a separate set of relations in the overall database. For each constraint instance, it keeps track of the metaconstraint, the UID of the source tuple for that constraint, the set of UIDs for each tuple that serves as positive or negative evidence for the constraint, the set of tables the constraint is dependent upon, and a flag indicating whether the constraint is currently valid or not. This information is updated incrementally based on the update files from the database manager and is done automatically whenever the database manager updates the database.

Our tool currently uses these predicate-based constraints to express sixty-eight global relationships among software artifacts as well as a number of project-specific relationships.

8. Experience and Future Work

We have used our constraint-based tool for its own development, for student projects, for developing support code for classes, for an Internet-scale computing project, for a software visualization system, and for a small set of development projects, mainly to validate

```
SELECT T_1.Id, 0
FROM SrcDefinition T_1
WHERE (T_1.Id IN ( '_50' ) AND
      ((T_1.SymbolType = 4) AND (NOT T_1.System) AND (T_1.Access = 1) AND
        EXISTS ( SELECT T_2.Id FROM SrcScope T_2 WHERE
              (((T_2.Id = T_1.Scope) AND (T_2.ScopeType = 6)) ) ) )
      AND NOT ( EXISTS (
            SELECT T_2.Id FROM UmlClass T_2 WHERE
             (((T_2.ClassType = 1) AND (T_1.Name = T_2.TypeName)) ) )  ) )
UNION
SELECT T_1.Id, 1
FROM SrcDefinition T_1
WHERE (T_1.Id IN ( '_50' ) AND
      ((T_1.SymbolType = 4) AND (NOT T_1.System) AND (T_1.Access = 1) AND
        EXISTS ( SELECT T_2.Id FROM SrcScope T_2 WHERE
              (((T_2.Id = T_1.Scope) AND (T_2.ScopeType = 6)) ) ) )
      AND EXISTS (
        SELECT T_2.Id FROM UmlClass T_2
         WHERE (((T_2.ClassType = 1) AND (T_1.Name = T_2.TypeName)) ) )  )
```

FIGURE 5. Example of an test-update query generated for the specification in Figure 4

the system and the approach. We have also run the system on a variety of open source projects to show that the system scales and generates meaningful checks. These include *Freecs* (a shareware chat program with 20,000 lines of code), *Egothor* (text search engine, 54,000 lines), Jalopy (Java pretty printer, 95,000), *Openjms* (implementation of the Java Messaging framework, 95,000), and *Ant* (170,000).

The system is used in its own development (68,000 lines of Java plus about 250,000 lines of external Java libraries). This system involves about 17,500 different constraints over 176 artifacts. While a full rebuild of the database takes around twenty minutes and involves over 9,000 queries, incremental updates when fewer than 20 files have changed typically take one to three minutes with a high degree of variance depending on exactly what changes. About one third of this time is spent updating the database by finding and analyzing changed files and then doing the incremental update, with about a third of this time being spent on recomputing the various transitive relationships that are stored in the database. The remainder is spent updating the constraints where the bulk of the time involves evaluating the SQL queries associated with the metaconstraints. A typical update at this level involves about 500 database queries, most of which are evaluated due to table updates rather than item updates.

This level of incremental performance is adequate and significantly better than non-incremental, but not exceptional. If we enable automatic updating, the updates occur in background and are not noticeable. However, if we want to update the front end immediately after changing files, there is a noticeable delay with large systems. We feel that much of this delay could be eliminated by being more intelligent about which constraints need to be rechecked when a table changes by associating a checking query with the table to detect if the changes are actually relevant and by minor modifications of some of the metaconstraint definitions to make the corresponding queries more efficient.

Our experience in using the system has shown it to be very helpful in identifying potential problems and inconsistencies. We have found and corrected numerous language and documentation problems in all the tested systems, problems that reflected inconsistencies between the code and either the language conventions or the documentation. It was particularly useful in ensuring the quality of course software before it was released to the class. For those systems where we have UML, we have been able to keep the UML synchronized with the source without having to completely regenerate it and hence lose the particular formatting and conventions we used in creating it initially. For systems where we used *junit*, it was helpful in identifying both coverage and what test cases needed to be rerun. Students found that it provided quick detection of careless errors, especially those dealing with language usage.

The current system was engineered for Java, but the techniques used and the overall approach should work for arbitrary languages and development environments. Adding additional artifacts to the framework involves writing an appropriate information extractor, a task that can generally be done by building on existing tools. For example, we use the Jikes compiler for extracting Java information, but in the past have extracted similar information for C and C++ using Sun's source browser database, the EGC compiler front ends, and minor modifications to *gcc*. One interesting future direction is applying the overall techniques to build a unified design model for a system by relating a variety of different design artifacts such as contracts, UML, user interface models, design patterns, architecture design languages, typestates, abstract state machines, and ownership types.

The system is available in buildable source form through our web site (www.cs.brown.edu/people/spr).

Overall, the system demonstrates that using an indirect mechanism based on simple, one-way constraints can be very effective way to integrate a variety of diverse software artifacts in a meaningful way. The techniques are extensible and powerful; they can handle a variety of different types of artifacts and provide a broad range of meaningful information to the developer.

Acknowledgements. This work was done with support from the National Science Foundation through grants CCF0218973, ACI9982266, CCR9988141. Manos Renieris and Shriram Krishnamurthi provided significant advise and feedback.

9. References

1. 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).

2. Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu, "Automatic code generation from design patterns," *IBM Systems Journal* Vol. **35**(2)(1996).

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. David Crocker, "Perfect developer: a tool for object-oriented formal specification and refinement," *Proc*. *Formal Methods Europe*, (2003).

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

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

8. 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).

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

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

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

12. Dennis Gruijs, "A framework of concepts for representing object-oriented design and design patterns in the context of tool support," Dept. of Computer Science INF-SCR-97-28, Utrecht University (August 1998).

13. 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).

14. Maciej Hapke, Andrzej Jaszkiewicz, Krzysztof Kowalczykiewicz, Dawid Weiss, and Piotr Zielniewicz, "OPHELIA: Open platform for distributed software development," *Proc Open Source International Conferece*, (2004).

15. Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich, "Intelligent assistance for software development and maintenance," *IEEE Software* Vol. **5**(3) pp. 40-45 (May 1988).

16. Alexander Knapp and Stephan Merz, "Model checking and code generation for UML state machines and collaborations," *Proc*. *5th Workshop in Tools for Software Design and Verification*, (2002).

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

18. 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).

19. Marco Meijers, "Tool support for object-oriented design patterns," Dept. of Computer Science INF-SCR-96-28, Utrecht University (August 1996).

20. Stephen J. Mellor, Anthony N. Clark, and Takao Futagami, "Model-driven development," *IEEE Software* Vol. **20**(5) pp. 14-18 (Sept 2003).

21. N. H. Minsky and D. Rozenshtein, "Law-governeed object-oriented systems," *Journal of Object-Oriented Programming* Vol. **1**(6) pp. 14-29 (March/April 1989).

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

23. Steven P. Reiss, "Working with patterns and code," *Proc*. *HICSS-33*, (January 2000).

24. Steven P. Reiss, "Constraining software evolution," *International Conference on Software Management*, pp. 162-171 (October 2002).

25. Steven P. Reiss, "Checking event-based specifications in Java systems," *Proc*. *SoftMC 2005*, (July 2005).

26. 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).

27. 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).

28. J. Smith, M. M. Kokar, and K. Baclawski, "Formal verification of UML diagrams: a first step towards code generation.," pp. 224-240 in *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists*, ed. Andy Evans, Robert France, Ana Moreira, and Bernhard Rumpe,Springer-Verlag (2001).

29. Mike Smith, Pauline Wilcox, Rick Dewar, and Dawid Weiss, "The Ophelia traceability layer," *2nd CCSE Workshop*, (March 2003).

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

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