Building a Constraint-Based Software Environment

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. This paper describes the techniques that underlie the environment, concentrating on those that deal with the diversity of artifacts the environment supports and on the definition and incremental maintenance of constraints between these artifacts.

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 source code will be updated to include all the necessary changes, but the specifications and design documents are often not modified to reflect these changes. 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. Any user and developer is familiar with the manner in which documentation becomes out-of-date, and how implementation changes take a long time to percolate to the documentation. 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. For example, it provides constraints that specify that:

- every class, field, and method that appears in the UML class diagram for the system much also appear in the source code;
- every public class appears in a UML class diagram that includes all its public methods and fields;
- every public method of a public class has associated documentation;
- all procedures (or all basic blocks or all branches) are covered by at least one test case;
- the sequence of calls shown in a UML interaction diagram is realizable in the code;
- design patterns that are part of the design exist in the code and persist through changes; and
- the program obeys a set of specified naming and language usage conventions.

This environment provides several capabilities. First, it extracts relevant information from each of the software artifacts. A piece of information is relevant either because it is required to identify potential constraints (e.g. the existence of a class in a UML diagram), or because it is needed to test and verify constraints (e.g. the name and symbol type of each defined symbol). 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.

Unlike code, which is extremely precise and highly amenable to well-understood analyses, many of the dimensions we consider are not defined precisely. As a result, our environment is harder to describe formally, and it is forced to use more heuristics than one based purely on, say, program analysis. Nevertheless, programmers use all these dimensions and rely on them as systems evolve, so we have a responsibility to support them through the program's life-

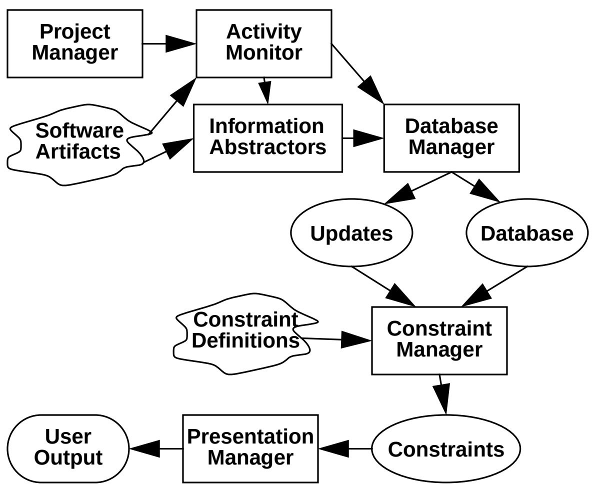


FIGURE 1. The architecture of the environment.

span. Hopefully, with better environmental support and thereby greater use, some of these other dimensions will also acquire more precise formulations.

In the next section of this paper we outline the structure of the environment and identify the key issues that have arisen in its construction. The following sections look at these issues in detail and describe our current solutions. We conclude by describing the current state of the system and our experiences with it.

2. Environment Overview: The Challenges

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.

Extracting information about programs is a non-trivial task, which is made harder still when tackling numerous dimensions, some of which are less precisely specified than others. Moreover, some contain information that spans multiple software dimensions. For example, source files contain syntactic information, semantic information, and documentation while UML files contain information from each of the different types of UML diagrams. Rather than using the artifacts directly as the basis for constraint generation and checking, the environment extracts information from the artifacts, converts it into a standardized representation, and stores it in a relational database.

The information is extracted through a set of information abstractor tools. Most of these are in two parts. The first part takes the artifact, isolates the information that is relevant to a particular dimension, and then generates an XML file containing that information. The second part reads this XML file and then generates a second XML file consisting of commands to the database manager describing what tuples in what relations should be removed, inserted or updated.

There were several problems that had to be solved in order to make this part of the environment work. These included:

- What are the artifacts associated with a particular software project? To address this question we needed to provide a project manager, which we describe in section 3.
- What information is required from the artifacts? This required an understanding of the often vague semantics associated with some software artifacts as well as a good intuition for what types of constraints would be most effective. We discuss this in section 4.
- How to extract this information efficiently? Each dimension of the software presents its own problems but also suggests potential solutions. The environment provides a framework in which new extractors are easy to add and where multiple extractors can operate on a single kind of artifact. We explain this in section 4.
- How to extract the information unobtrusively? We felt it was essential that the system operate in parallel with the developer while imposing no additional effort. This led us to include an activity monitor in the environment that detects when artifacts have changed and automatically finds and runs the appropriate abstractors and then the database manager. Section 5. discusses this portion of the environment.
- How to relate information across different abstractors or different dimensions? This problem is two-fold. It first necessitates a consistent means of naming and identifying items. Second, it requires the database manager to track and maintain identities for tuples in the database. This is discussed in section 4.

Having gathered information and filed it in the database, the environment must generate, maintain, and display the set of constraints that ensue. This work is done by two components, a constraint manager that is in charge of finding and updating constraints and storing the resultant constraint information in another database, and a presentation manager that uses this second database to provide appropriate feedback to the developer.

Again, there were several key problems that needed to be addressed in order to make this work efficiently and effectively. These included:

- How to define constraints? We chose to define a notion of metaconstraints that are instantiated based on objects in the database. The form of these constraints was dictated by this requirement and the need to associate a particular location in a source artifact with the constraint if it is violated. We found that most constraints could be defined using formulas that related information in the database. Constraints that affected behavior, however, were more complex. These issues are described in section 6.

- How to detect and check constraints? Given a particular form for a metaconstraint, we had to be able to quickly find the appropriate constraints and check whether they were satisfied. For formula-based metaconstraints, this is done by mapping the formula into a set of SQL queries. For behavioral constraints, this is accompanied by additional checks. This is also discussed in section 6.
- How to manage constraints incrementally? A complex software system can have large numbers of constraints. It was thus essential for scalability of the environment to be able to only check and update constraints that might be affected by the small set of changes that the developer makes as the system evolves. This was accomplished by having the database manager generate a file describing the updates to the database and through the use of unique identifiers (UIDs) for database items. The incremental update logic is described in section 5.
- What information should be presented to the developer about the constraints and how should it be presented? We have been experimenting with both a standalone approach where a new tool is used to present the information to the developer and an integrated approach where we use existing mechanisms of a software development environment. This is described in section 7.

3. The Project Manager

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).

To facilitate these definitions, the project manager provides for project hierarchies, whereby common definitions of files to include or exclude can be specified in a separate project that can then inherited by other project definitions. A special project, generic, contains global definitions and is inherited by all projects. Finally, the project manager provides facilities for associating environment variables

```
<PROJECT NAME="generic">
<FILES>
        <EXCLUDE PATTERN="@/bBACKUP"/>
        <EXCLUDE PATTERN="@/bCONTROL"/>
        <EXCLUDE PATTERN="@/bBUFFERS" />
        <EXCLUDE PATTERN="@/RCS" />
        <EXCLUDE PATTERN="@/CVS" />
        <EXCLUDE PATTERN="@/.*" />
</FILES>
</PROJECT>

<PROJECT NAME="clime" OWNER="spr" GROUP="ivy">
<FILES>
        <USE DIRECTORY="/pro/clime/javasrc/edu/brown/clime" />
        <USE DIRECTORY="/pro/clime/uml" />
        <EXCLUDE DIRECTORY=".../clime/ClicPlugin" />
</FILES>
<ENVIRONMENT>
        <SET NAME="CLASSPATH" VALUE="...bloom/java" />
        <SET NAME="SOURCEPATH" VALUE="/pro/clime/javasrc" />
        <SET NAME="BROWN_CLIME_ROOT" VALUE="/pro" />
        <SET NAME="BROWN_CLIME_TEA" VALUE="/pro/tea" />
        <SET NAME="BROWN_CLIME_ARCH" VALUE="sol" />
</ENVIRONMENT>
</PROJECT>
```

FIGURE 2. Sample project definitions.

with a project. This is a simple but important detail, since tools need access to information such as the Java class path. Two sample project description files are shown in Figure 2. The first is the generic project defining particular file patterns to exclude. The second is the description of the project for our constraint-based environment itself.

While XML files are a convenient framework for the project manager, they are not always convenient for the user. To this end, we have implemented a separate front end for generating and editing project files and have created a module that takes information from IBM's Eclipse environment for Java and automatically generates or updates a project file.

4. Artifact Analysis and Abstraction

The abstraction process for software dimensions generally involves two stages. The first stage is dependent on the type of artifact and analysis. It 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. We wanted to ensure that the framework could handle a wide variety of different artifacts and dimensions, and we needed to make sure that the necessary information could be extracted quickly and automatically.

To this end, we have created a variety of abstractors. These include:

- 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 that is invoked by the standard javadoc program for each source file and which generates an XML description of all the available documentation-related information.
- Semantic information. Control flow graphs emphasizing calls are generated for each method in the source code using the Soot Java Optimization Framework [23]. Each node in the graph represents a call point and contains the signature and class of the calling and called method.
- 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.
- 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.
- Test cases. We assume that the developer is using *Junit* **[7]**, a common Java testing package. The information extractor reads the compiled Java class files using IBM's *JikesBT* package [12]. 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, and coverage information for blocks, branches, functions, and calls.
- History 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 (assuming that the CVS version management system [5] is used) for each file.

In addition to information that is abstracted directly from source artifacts, we found the need to have additional information that was not directly reflected in the artifacts that comprise the software. Some of this information was needed to represent global information that is assumed by the developers, such as style rules describing naming conventions as well as language usage rules such as every field must be read and written at least once or that all data fields should be private.

Other information was needed because the set of formal artifacts that are used today are incomplete. The software development activity includes information such as design patterns that are not currently 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. [6] and our previous work [17]), 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.

Another example of an incomplete set of artifacts is the description of usage conventions for groups of methods. Part of the design or specifications of a software system is a description of how a class or library should be used. Though there have been efforts, both old [20] and new [2], to validate software against such specifications, usage information is usually specified through informal comments and documentation, and cannot be easily captured in a tool. Our constraint mechanism lets us formally define such constraints and then check them as the software evolves. However, we need to provide the instantiations of the conditions manually instead of extracting them from existing artifacts. Currently we use this approach to specify and check conditions such as "any method that returns an instance of class C (or any of its subclasses) needs to return a newly created instance".

Dealing with a variety of artifacts forced us to confront naming. Since we wanted to relate information in one artifact with that in another, we needed to ensure that we could 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.

5. Database Management

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: an activity monitor, a command generator and a database manager. We describe each of these in turn.

The *activity monito*r 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 commands with respect to the database. The commands are of two forms. The first indicates that all data in a given database table that meet a criterion are to be deleted. This is typically used to remove all the old information that is associated with a particular artifact when that artifact changes. The second form of command indicates a new tuple to be add to a particular table. By organizing the data independently of the information source, this package greatly simplifies the actual database manager.

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. Finally, it needs to generate a file describing what has changed in the database so that later constraint processing can be done incrementally.

The commands to the database manager describe sets of tuples to be added and removed from each relation. Typically, they would indicate that all tuples that came from a particular artifact should be removed and then provide all 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 the incremental framework that the database manager fits in. First, it means that the database manager would have to report a relatively large number of changes to the later constraint manager even when only a small amount of data actually changed. Second, it makes tracking unique identifiers more difficult.

To avoid these difficulties, the database manager attempts to intelligently update based on the information it has. 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 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). Unique identifiers are associated with tuples in most of the tables of the database. They are used to provide links between tables (such as the link between a symbol reference and its definition) and form the basis for later constraint management, where they are used to associate a particular constraint instance with the tuples in the database that caused it to be created and validated.

To support unique identifier management, the database system requires that any relation containing a UID field also specify the set of fields that characterize each tuple. This set of fields is then used 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 processing for UIDs takes advantage of the previously described incremental processing: it makes the assumption that any UID that would be reused will come from the set of tuples that are being deleted from the relation. This assumes that a UID will only be reused if the file it originally came from and it now derives from are the same. For Java, this is generally true for most symbols. Where it is not implicitly true, we ensure that the UID also depends on the original file.

A second part of UID management handled by the database involves tracking UIDs that are assigned by the various information abstractors and mapping these into global UIDs for the database. For example, the symbol table information abstractor assigns each definition that it generates a new local UID and links each reference to its definition using it. These local identifiers are unique only with respect to the particular data file that is generated, are not unique globally, and do not correspond to existing or future global UIDs. The database manager handles local UIDs by tracking which fields contain UIDs and mapping the local UIDs, where given, into global UIDs that are generated using the previously cited rules for UID reuse.

The third 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. This is only done for tables that have UIDs associated with each tuple and the information that is reported is just the UID of the corresponding tuple. For tables that do not have associated UIDs, the only information to report is whether the table was changed.

6. Defining Metaconstraints

Given data about all the different dimensions of a software system, the next portion of the environment defines, manages, and presents the constraints that ensure the different software artifacts remain consistent as the software evolves. The first part of this task 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 environment 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*. The environment imposes several conditions on how metaconstraints can be defined. These include:

- It must be easy to identify the source of the constraint. If a constraint is violated, it is essential that the developer be told where and in what artifact the problem arises. For example, if a method is not covered by any test case, the developer will want to know the identity and location of the method.
- It must be easy to identify conflicts to a constraint. For example, if a UML class diagram defines a method as public but the actual method is protected, the developer will want to know both the location of the method in the class diagram and the location of the method in the source code.
- It must be possible to mange the set of constraints incrementally. As the database changes, the system will need to manage the set of constraints, creating new constraints and removing old ones as necessary. This should be possible without having to recompute the full set of constraints or to check conditions for all constraints, and it should be based on the update files generated by the database manager.
- It must be possible to check the validity of the set of constraints incrementally. Similarly, checking the validity of the full set of constraints will probably be too costly. Instead, the system should be able to determine, based again on the update file from the database manager, what constraints need to be checked and then to only recheck those.

To satisfy these requirements, we started with 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. 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

```
<CONSTRAINT TYPE="QUERY" NAME="source_correspond" >
  <EXPR OP="FORALL" VAR="x" TABLE="SrcDefinition">
    <EXPR OP="AND">
      <EXPR OP="EQL">
        <EXPR OP="FIELD" FIELD="SymbolType" VAR="x" />
        <EXPR OP="INT" VALUE="4" />
      </EXPR>
      <EXPR OP="NOT">
        <EXPR OP="FIELD" FIELD="System" VAR="x" />
      </EXPR>
      <EXPR OP="EXISTS" VAR="z" TABLE="SrcScope">
        <EXPR OP="AND">
          <EXPR OP="EQL">
            <EXPR OP="FIELD" FIELD="Id" VAR="z" />
            <EXPR OP="FIELD" FIELD="Scope" VAR="x" />
          </EXPR>
          <EXPR OP="EQL">
            <EXPR OP="FIELD" FIELD="ScopeType" VAR="z" />
            <EXPR OP="INT" VALUE="6" />
          </EXPR>
        </EXPR>
      </EXPR>
    </EXPR>
    <EXPR OP="EXISTS" VAR="y" TABLE="UmlClass">
      <EXPR OP="AND">
        <EXPR OP="EQL">
          <EXPR OP="FIELD" FIELD="ClassType" VAR="y" />
          <EXPR OP="INT" VALUE="1" />
        </EXPR>
        <EXPR OP="EQL">
          <EXPR OP="FIELD" FIELD="Name" VAR="x" />
          <EXPR OP="FIELD" FIELD="TypeName" VAR="y" />
        </EXPR>
      </EXPR>
    </EXPR>
  </EXPR>
</CONSTRAINT>
```

FIGURE 3. Sample metaconstraint definition.

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 (when new tuples are added to the relation S) or to 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 the 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. Fields of that tuple can then be accessed via a FIELD operator. The predicates can also include comparisons, string matching, arithmetic and string operators, and Boolean operations. The metaconstraint definitions are provided by XML files that can be defined either globally or for a particular project. An example of such a definition is shown in Figure 3.

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. 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 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 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 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 environment uses these predicate-based constraints to express a wide variety of different relationships among software artifacts. These 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.
- 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 pattern. This was done to illustrate that patterns could be represented and is limited because of the lack of tools to create and reflect pattern instances in software.
- 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.

While such predicate-based constraints are quite powerful, they are generally not sufficient for specifying and checking conditions that are related to behavior. To cover these, we are experimenting with different specification techniques and different ways of checking whether different software artifacts are indeed consistent.

One example of this involves extending our support for UML to include sequence diagrams. The UML specification [8] states that sequence diagrams depict the order in which messages are passed between groups of objects collaborating in a particular behavior.

Sequence diagrams thus provide a powerful tool for exposing the interactions between objects. This strength is particularly applicable to the visualization of design patterns. For instance, the Observer Pattern [6] creates a relationship within a group of objects such that when one changes (the Subject) the others are notified (the Observers).

Figure 4 shows the Observer Pattern in the form of a sequence diagram. Time progresses down the page. Each

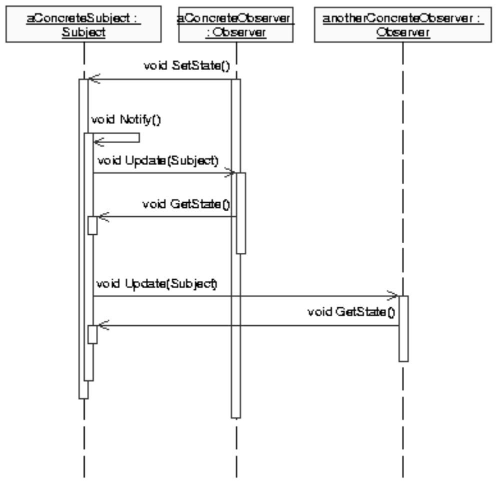


FIGURE 4. Sequence diagram for the Observer pattern.

```
<TUPLE>
  <FDATA NAME='ModelFile' VALUE='.../rose/observer.xml' />
  <FDATA NAME='Index' VALUE='0' />
  <FDATA NAME='Id' VALUE='G.14' />
  <FDATA NAME='NameOfCall' VALUE='void SetState()' />
  <FDATA NAME='ClassOfCall' VALUE='Subject' />
  <FDATA NAME='NameOfCaller' VALUE='*' />
  <FDATA NAME='ClassOfCaller' VALUE='Observer' />
</TUPLE>
```

FIGURE 5. A sample tuple from the sequence diagram of Figure 4.

box represents a particular object. The object names are optional and for our purposes are ignored. The vertical lines underneath each object are called lifelines and represent the duration of that object's existence. Messages are represented by arrows between lifelines. As a matter of convention, we place a message's signature on the side nearest to the object that it is called from

In the context of our system, each message in the diagram is represented as a tuple as in Figure 5. This tuple represents the first call in the Observer Pattern sequence diagram of Figure 4. The element with name 'Index' represents the order that the message occurs with respect to diagram's other messages.

The thin rectangles, each superimposed on a lifeline, are called *activations* or *foci of control*. Each focus of control represents the duration of a message's execution. Further, multiple messages may originate from a single focus of control, representing that they were called by the message that the focus of control represents.

We decided to utilize nested foci of control so that the message body that each message is called from can be precisely represented. Those foci of control that are not explicitly named are flagged as wildcards. Because a sequence diagram is not intended to depict *all* messages in an execution, only those that contribute toward a particular goal are shown.

We discovered that predicate-based constraints are not sufficient to verify the behavioral information expressed by sequence diagrams. Because the first message in the diagram represents a call point in an unknown message body, we check all message bodies in the source code for call points of its signature and class. For each such call point that we find, we verify that the call points represented by the rest of the messages of unknown origin in the sequence diagram occur after it. Finally we verify that each of the identified call points begins the proper sequence of messages represented in the diagram.

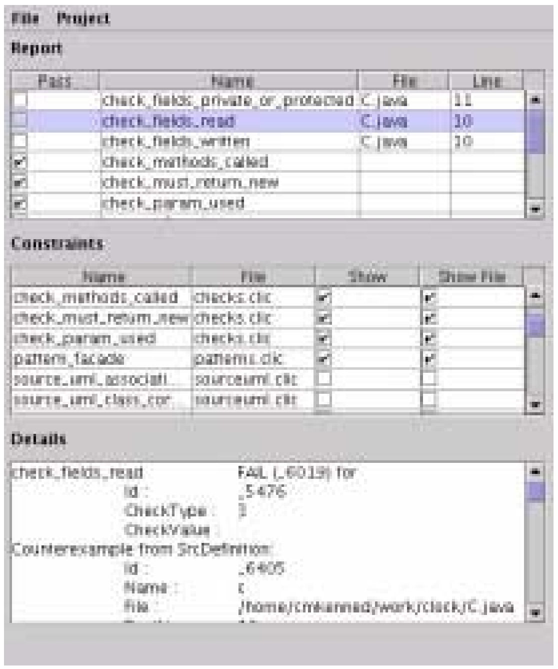
7. Presenting Constraints

The constraint manager generates the set of actual constraints along with enough information (based on UIDs) so that a front end can identify the source and related information for each constraint. The next part of our environment involves taking this information and presenting it to the user.

The overall environment is structured so that we can take multiple approaches to presentation. Each presentation manager can access the database of constraints to get information about what is currently consistent or inconsistent about the software artifacts. Moreover, each presentation manager can access the abstraction database to get enough detailed information about the UIDs associated with each constraint so that appropriate information can be presented to the user.

We currently have implemented two approaches to presenting the current set of constraints. The first is a standalone system while the second uses the facilities of an existing programming environment as seen in Figure 6. The standalone system generates the project description file, or can work with a description generated elsewhere. Updates are performed when a project is opened for viewing or at the request of the user. All constraints for the project are displayed and can be sorted by file, constraint name, or pass or failure. The user can choose to hide or show any constraints to keep from cluttering the view with constraints in which they are not interested. For example, a user may hide all UML constraints for a project that has no UML files. Emacs is opened to display any failures associated with a file and line number.

Along with the standalone system, we also created a plugin for IBM's Eclipse Java environment. This plugin creates a project description file from the Eclipse project description, requiring no input from the user. Updates occur automatically whenever the project is compiled. By default Eclipse uses an incremental compiler, which means



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.clc	✓	✓
check_must_return_new	checks.clc	✓	✓
check_param_used	checks.clc	✓	✓
pattern_facade	patterns.clc	✓	✓
source_uml_associati...	sourceuml.clc	☐	☐
source_uml_class_cor...	sourceuml.clc	☐	☐

Details

```
check_fields_read              FAIL (_6019) for
             Id :              _5476
             CheckType :       3
             CheckValue :
Counterexample from SrcDefinition:
             Id :              _6405
             Name :            c
             File :            /home/cmkanned/work/clock/C.java
```

FIGURE 6. Standalone constraint interface.

compiles are quite frequent. Because of the time required for an update, updates are noticeably slow. If incremental compiling is turned off in favor of less frequent full builds, however, there are no such problems. Failed constraints are displayed in Eclipse's task list alongside items like compiler errors and warnings. The list of constraints is also displayed in its own window in order to allow the user to hide constraints. The user can also input a description of each constraint to make the constraints shown in the task list more informative. Constraint names and hidden constraints are stored across Eclipse sessions.

8. 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. Our primary concerns have been whether the approach represented by the environment will report actual inconsistencies, whether the necessary information gathering will be both accurate and unobtrusive, whether constraints can be managed and checked incrementally during active software development, and whether the approach scales to handle large software systems.

Our first experiment with the environment involved developing a simple game program using UML and Java. The program consists of about 2500 lines of Java and 25 classes. We alternatately wrote and evolved the game program by generating new UML designs and new source code. In both cases the environment pointed out the differences and led us to ensure the code and the UML diagrams evolved consistently.

Our second experiment involved using the environment on itself (about 25,000 lines of Java) during development. Our principle goal in this instance was to ensure that the system was both accurate and unobtrusive. Our experience was that we did not even notice when the updates were occurring. Moreover, our 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 Our experience here was mixed. The environment worked, generating the appropriate data and constraints. However it took considerable time to generate the initial data. The database that was generated is about 140Mb in size and has about 15,000 constraints. Setting up the database took about an hour, while generating the constraints took about 6 hours. Most of the setup time was spent in reading and writing the XML data files containing information for the database. Most of the constraint time was spent evaluating the query for one particular metaconstraint which apparently was optimized poorly by the underlying database system.

The times here are slower than ideal, but they also represent an extreme case: importing an large legacy project into the environment. We expect the common case to be a user who employs the environment from the very beginning, in which case the environment only makes incremental updates at each stage. This would limit both the size of the data files that are generated and read and the complexity and number of database queries that need to be generated to handle the constraints. Moreover, it is possible to modify the environment for an initial load of the system to avoid the incremental mechanism (and hence most of the XML files), and to rearrange the constraint that is generating the complex query so it is optimized correctly so that even the larger system could be installed in an hour or two. We are currently working on improving the performance here and on extending the environment to handle IBM's Eclipse environment for Java which contains 1,200,000 lines of code.

9. 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 [22] and Common Lisp [19] have combined documentation with code. These efforts led to literate programming [4,11] 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 [3] 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 [18].

A number of current and past approaches to software development have attempted to provide a comprehensive, language-based environment. Garden [14-16] and Escalante [13] tried to do this for visual languages; the proposed DARPA prototyping language tried to do this for specifications; and intentional programming [1] tries to combine multiple approaches in a single textual framework. These approaches tend to focus on different programming dimensions, generally ignoring specifications and design as well as documentation. The same remarks apply to programming techniques such as aspect-oriented programming [9,10] or multi-dimensional separation of concerns [21]. In the end, we believe that approaches that attempt to embed dimensions of concern in the program will never scale to handle the multitude of dimensions that programmers must reconcile. The different dimensions require different notations and do not interact hierarchically or cleanly. Moreover, the set of software dimensions grows constantly.

10. Conclusion and Future Work

We have described a new software environment that tries to detect inconsistencies between numerous artifacts used in the development cycle. It does so by translating different notations into a common constraint language. The environment is abstract enough to support the addition of new notations. It constantly monitors changes, efficiently updates constraints, and reports violations when artifacts become inconsistent. Preliminary experiments show that the environment both provides useful feedback and does so efficiently. Maintaining the consistency of the various dimensions of software is an important step on the road to giving multiple design artifacts equal billing.

There are many natural directions for future work: supporting more dimensions, strengthening the tool support, using a richer semantic constraint model, and improving efficiency to support large legacy systems.

11. References

1. William Aitken, Brian Dickens, Paul Kwiatkowski, Oege de Moor, David Richter, and Charles Simonyi, *Transformation in Intentional Programming*. sep 1997.
2. Thomas Ball and Sriram K. Rajamani, “The SLAM project: debugging system software via static analysis,” *ACM Principles of Programming Languages*, pp. 1-3 (January 2002).
3. Don Batory, David Brant, Michael Gibson, and Michael Nolen, “ExCIS: an integration of domain-specific languages and feature-oriented programming,” in *New Visions for Software Design and Productivity*: *Research and Applications*, (dec 2001).
4. 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).
5. Karl Fogel, *Open Source Development with CVS*, CoriolisOpen Press (1999).
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley (1995).
7. E. Gamma and K. Beck, “Test infected: Programmers love writing tests,” *http://www.junit.org*, (1998).
8. Object Management Group, *OMG Unified Modeling Language Specification* (www.omg.org) (September 2001).
9. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, “Aspect-Oriented Programming,” in *European Conference on Object-Oriented Programming*, (jun 1997).
10. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, “An Overview of AspectJ,” in *European Conference on Object-Oriented Programming*, (2001).
11. Donald E. Knuth, “Literate programming,” *The Computer Journal* Vol. **27**(2) pp. 97-111 (1984).
12. 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).
13. Jeffrey D. McWhirter and Gary J. Nutt, “Escalante: an environment for the rapid construction of visual language applications,” U. Colorado Boulder, CU-CS-692-93 (1993).
14. Steven P. Reiss, “Working in the Garden environment for conceptual programming,” *IEEE Software* Vol. **4**(6) pp. 16-27 (November 1987).
15. Steven P. Reiss, “A Conceptual Programming Environment,” *9th Intl. Conf. on Software Engineering*, (March 1987).
16. Steven P. Reiss, “An object-oriented framework for graphical programming,” pp. 189-218 in *Research directions in object-oriented programming*, ed. Peter Wegner,MIT Press (1987).
17. Steven P. Reiss, “Working with patterns and code,” *Proc. HICSS-33*, (January 2000).
18. 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).
19. Guy Lewis Steele, Jr., *Common Lisp*: *the Language*, Digital Press, Bedford, MA (1990).
20. 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).
21. Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton Jr., “N degrees of separation: multidimensional separation of concerns,” *Proceedings of the 21st Intl. Conf. Software Engineering*, pp. 107-119 (1999).
22. Warren Teitelman, “A tour through Cedar,” *IEEE Software* Vol. **1**(2) pp. 44-73 (April 1984).
23. 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).