# Constraining Software Evolution

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912
spr@cs.brown.edu, 401-863-7641, FAX: 401-863-6757

**Abstract**

*Software is multidimensional but the tools that support it are not. The lack of tool support causes the software artifacts representing different dimensions to evolve independently and inconsistently. In order to support the evolution of multidimensional software, an environment must ensure that the different dimensions evolve concurrently. This can be accomplished through an integration framework that maintains consistency of the different dimensions as they evolve. We have build a prototype of such a mechanism by setting up and maintaining constraints among artifacts representing the different software dimensions. This paper describes that prototype and our experiences with it to date.*

## 1 Motivation

Most developers think of a software system as the code and components that are the end result of the software development process. As code is written, developers gradually ignore the initial stages of development, the specifications and the design of the system, the documentation, the component specifications, and the test cases. This narrow view of software is one of the primary causes of the many problems associated with software and its development [9].

We visualize an environment where all the aspects of software development evolve consistently as the system is developed and maintained. As any one of these aspects changes, the environment would ensure that all other aspects are updated accordingly. This would simplify both software development and maintenance as it would guarantee that all aspects were relevant, accurate, and up-to-date.

Such an environment can be built by defining constraints between the artifacts representing the different software aspects. When an artifact is updated, all the constraints on that artifact are triggered and any related artifacts are checked for consistency. Consistency can then be maintained either automatically or by informing the developer what needs to be done before the change is complete. In this paper we demonstrate that such an environment based on constraints can be built and used effectively for a wide range of software artifacts.

### 1.1 Software Evolution

Software is not just the source code; instead it is multidimensional [19]. The specifications, design, architecture, test cases, user interfaces, coding conventions, components, constraints, design patterns, system architecture, development history, and documentation are all as much a part of a software system as is the physical code. Furthermore, these other artifacts are sometimes much more valuable than the code itself. Software development and programmer productivity therefore depend on being able to develop, relate, and consistently maintain all these different aspects of the software.

Today's software environments provide a wide variety of tools to handle the many of dimensions of software. Environments provide tools for managing, editing, and debugging the source code; tools for developing and experimenting with user interfaces; tools for specifying the design using UML or similar notations; and tools for creating and managing test cases for a system. Research environments have demonstrated tool prototypes for managing design patterns, components, and constraints [3,6,8,16,17,22,24,25,27,29,40]. There are also tools specifically designed to handle software evolution. Most of these tools only deal with the source code and use semantic analysis to either identify code affected by potential changes [2,15,42,47] or that assist in refactoring [5,24,41].

These tools, especially those dealing with different aspects of software, are rarely integrated with one another. As a result, software routinely evolves inconsistently along the different dimensions. Typically, developers create an initial design, but, as the code gets written and modified, they do not update this design to reflect the changes. The overall behavior of the system might have been specified initially, but this approach offers no guarantee to the future programmer that this specification is actually followed by the code after several change iterations. Testing may be the single dimension that receives proper attention, but the test suite usually bears little relationship to other dimensions. Likewise, the developers might emphasize coding conventions and constraints initially, but will not necessarily check these during later evolution so they may or may not be present in the evolved system.

This differential evolution of the various dimensions results in programmers getting inconsistent views of the system. Programmers quickly learn to not trust design documents or the original specifications when they are

faced with an evolving code base. Similarly, test cases lose relevance when developers fail to add new test cases as the software evolves. Component interaction, originally thought to be simple, becomes much more complex so that the developers might not ever fully describe or understand it in its final form. These and related problems are bad in a moderate-sized system; they are often fatal as systems get larger and more complex.

In short, programmers need a software development framework that supports the *consistent evolution of all the dimensions of software*. This framework should let the programmer specify the software along the different dimensions, using existing tools where possible. It should provide tools for design, code maintenance, test case generation and support, user interface design, documentation, component specification, behavioral descriptions, constraints, etc. More importantly, it should make sure that these different dimensions of the software remain consistent with one another as the software evolves and should provide tools for viewing and controlling this evolution.

## 1.2 Prior Work

Conceptually, the simplest approach to ensuring the consistency of different aspects of software is to combine all the aspects within a single programming language. Several environments such as Xerox Cedar Mesa environment [46] and Common Lisp [43] have combined documentation with code. These efforts led to literate programming [7,23] and, more recently, the use of *javadoc* and its corresponding conventions. User interface design is combined with code in programming environments such as Visual Studio where the user can design the interface and the system generates the code which the user never actually sees. Proponents of UML propose writing complete systems within its framework, thus making it a programming language that combines design with code. Batory [4] 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.

A number of current and past approaches to software development have attempted to provide a comprehensive, language-based environment. The Garden [31-33] and Escalante [26] systems tried to do this for visual languages; the proposed DARPA prototyping language tried to do this at the specification level; and, more recently, the intentional programming efforts at Microsoft [1] try to combine multiple approaches in a single textual framework. These approaches tend to concentrate on different programming dimensions, and generally ignore specifications and design as well as documentation and historical dimensions. Moreover, none of these approaches has shown itself to be practical.

An alternative that is being used today is to support a multidimensional programming technique such as aspect-oriented programming [20,21] or multi-dimensional separation of concerns [45]. Here a base language such as Java is augmented with cross-cutting encapsulations of code. This allows, for example, design patterns that affect multiple classes and methods to be defined in a single location. While these techniques address some of the issues of multidimensionality, they still focus on programming and neglect non-program dimensions.

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 is not fixed. Different types of software require different specifications and designs. As developers invent new types of software systems, they also devise new design and specifications techniques and languages. It is difficult to conceive of a language where new specification techniques can be easily added, integrated, and actually used. Finally, this approach does not address issues of legacy systems or the legacy components used in developing new software.

Rather than develop a single language incorporating the different dimensions of software, one could develop an intermediate representation supporting a variety of tools in an integrated fashion within a software development environment. Environments like Visual Studio already provide facilities for editing source, designing user interfaces, and creating UML diagrams. Environments such as Visual Age maintain the full semantics of the system in memory for fast compilation and analysis [30]. One could imagine extending an existing environment, first by providing tools for specifying other software dimensions and then by using the internal representation to ensure consistency. For example, the environment could track the structure of the source and of the UML class diagrams and could, using its internal data structures, ensure that the two are consistent.

Like a common language, this approach cannot succeed in general. It requires that the representation be designed to handle a much broader range of software aspects than current environments support. This involves reimplementing a broad range of tools within the environment in such a way that the environment can understand the semantics of the different aspects. Given the broad range of tools, each with different notations, features, and facilities, this quickly becomes impractical. This approach also will make it difficult to develop and use the new dimensions that will be needed for new types of software; it will, again, be difficult to use with legacy systems. Most importantly, before we can develop such an intermediate representation, we need to tackle the central problem of how to relate the different dimensions.

### 1.3 Consistent Software Evolution

Our goal is to provide a workable framework for maintaining the consistency of the many dimensions of software as the software evolves. To be effective, practical and complete, such a framework must meet a broad set of requirements. In particular, it should:

- *Work with existing tools*. Practicality implies that we should not have to reimplement or even significantly modify the broad range of existing tools. This is necessary to make use of the large effort represented by these tools and to handle legacy code and external components. A good mechanism should be able to extract the necessary information from these tools and interact with the tools.
- *Handle a wide range of software dimensions*. A good mechanism should not be geared to a specific problem such as ensuring the consistency of a UML class diagram with the source code. Instead it should be flexible enough to handle the broad range of dimensions that are actually involved in software development. This includes those dimensions that are not currently covered by existing tools.
- *Be extensible*. While some of the new dimensions can be foreseen now and should be dealt with, a practical mechanism will have to be adaptable to new design techniques and approaches, such as those implicit in Extreme Programming. To this end, the mechanism must be open and extensible.
- *Be bidirectional*. It is important that the mechanism handle changes in any aspect at any time. For example, a developer who changes the design will want to know what code is affected by the change and whether it is still consistent. At the same time, changing the code instead should tell the developer what aspects of the design are affected and whether they are still consistent.
- *Support partial checking*. A particular dimension generally does not provide a complete representation of the software. It is important to be able to support such partial representations. For example, the programmer might provide a UML diagram for all the externally viewable classes, but might omit diagrams for some internal support classes; it might be appropriate to provide external documentation for the public and protected methods of a class and to omit it for private methods. It should be possible for the mechanism to handle these and related cases and not force the programmer to provide unnecessary information.
- *Be able to locate points of inconsistency.* Not only does the mechanism need to determine when the dimensions are inconsistent, it needs to provide the programmer or tools with information on exactly what is inconsistent and why. At a minimum, this means identifying where in the different software artifacts the inconsistencies arise. Ideally, the mechanism should also provide simple and useful feedback.
- *Have low overhead and be unintrusive.* The mechanism should not interfere with existing tools or with the programmer. It should not take an excessive amount of time to find the inconsistencies. It should be as automatic as possible.
- *Handle both static and dynamic properties*. Many design and specification notations state something about the behavior of the software. While some of this can be checked statically, in general such checks are impossible (by equivalence to the halting problem). The mechanism must be able to extend to and deal with the dynamic properties of software.

The existence of such a framework would greatly enhance programming environments and the programming process itself.

## 2 Overview of our Solution

As we have discussed in Section 1.2, approaches to consistent software evolution that depend on developing a single comprehensive language or a single common semantic representation will not succeed for several reasons. First, they cannot adequately capture the wealth of dimensions inherent to software. Second, they will fail to handle new dimensions implicit to new types of software and software development. Third, they make it difficult to incorporate existing components into a system. To be successful, we need a more flexible approach to evolution management.

Our approach is to address this problem independent of the tools, languages, and notations needed for defining the various software dimensions. Here designers would describe software as they do now: using a combination of tools to create a set of artifacts or documents, each of which reflects one or more dimensions of the software. We then add a separate integration mechanism that provides for the consistent evolution of these artifacts. This integration mechanism analyzes the different artifacts and finds and flags any inconsistencies throughout the software development process. The mechanism is open both in that new dimensions can be handled through new types of artifacts and a variety of support tools can be used to show and manage the inconsistencies.

Integration mechanisms that work with existing tools have been quite successful. We pioneered the use of control integration in programming environments in the FIELD system [34-37] and the concept was then used in commercial systems such as HP's Softbench, Sun's ToolTalk, and DEC's Fuse. Here tools were either wrapped or slightly modified to send and receive messages using a relatively simple central message server. Integrating a new tool into the environment was a fairly straightforward task. Moreover, to the user, the overall environment behaved like a totally integrated suite of tools. In more recent work in this area, we showed in the Desert environment [38,39] and others demonstrated in the Sheets environment [44] that it is possible to provide an inexpensive form of data integration on top of existing tools. All these efforts concentrated on the programming aspects of software. In the proposed work we intend to extend this to as many other dimensions of software as possible.

Using an integration mechanism here has the promise of solving the problem of inconsistent software evolution in a practical way. It would allow the use of existing tools, languages, and notations. It would work with legacy systems, new code, as well as combinations of the two. It is simple enough to be adaptable to new dimensions, new tools, and new notations.

The integration mechanism that we have developed for maintaining the consistency of the different software artifacts is based on the notion of constraints. The key insight here is that **the design and specifications are simply constraints on the source code**. The whole process of specification and design can be thought of in general terms as specifying constraints on the final solution. Any implementation of the system the satisfies the full range of such constraints should be an acceptable solution. We can generalize this to take into account situations where different design and specification dimensions impose constraints on each other and to situations where the source imposes constraints on what should be included in the design. More importantly, this approach can be extended to interrelate the multiplicity of dimensions of a software system.

A constraint-based approach to consistency maintenance is both flexible and feasible. Most of the existing design notations can be viewed directly as a set of constraints on the source. For example a UML class diagram imposes constraints requiring the existence in the source of any class, method, or field specified by the diagram along with constraints about the class hierarchy and use relationships between the classes. It is possible to automatically take such a diagram and generate the corresponding list of constraints. Constraints can also be used for checking programming style, design patterns, coding conventions, as well as detailed and system-specific design rules [10-12,18,28,48]. Similarly, completeness of the design can be viewed as constraints imposed by the source. For example, a system-wide constraint can specify that every public class in the source be reflected in some UML class diagram.

In order to make this work in general, however, we must be cautious both in specifying the basis for the constraints and in specifying the constraints themselves. The constraints must provide for *accountability*; it must be possible to determine what portions of the source or other software artifact are in conflict when a constraint is not met. The constraints must also be easy to specify and relatively easy to check. The former is required to accommodate system-specific constraints that programmers will want to impose. The latter is needed to ensure that consistency maintenance is tractable even in the face of thousands of constraints. Finally, the basis for specifying constraints must be flexible enough to accommodate the wide variety of software dimensions.

The first step in this approach is to develop a common framework (as opposed to a common representation) for specifying all the software artifacts. This can be done by abstracting information from the different artifacts. Using an abstraction here provides independence from the actual tools being used, allows analysis to be done in order to provide a more practical basis for specifying constraints, and ensures that the information needed by the constraints is available. Using this approach, it is possible to abstract information in multiple ways from a single representation. This is most useful for the source code, where one can have separate abstractions based on structural information (the symbol table), semantic information (program dependency graphs), and dynamic information (trace data and performance summaries). Wherever possible, each item abstracted from an artifact identifies a location in that artifact as its provenance.

We then specify constraints as predicate equations over the corresponding abstracted data. We restrict constraints to be equations of the form: $\forall(x \in S)\varphi(x)\Theta(x)$ where S indicates the set containing data representing the source of the constraint, $\varphi(x)$ indicates the conditions under which the constraint is applicable, and $\Theta(x)$ is a qualified predicate the specifies the conditions the constraint must meet. Constraints of this form let the consistency manager handle accountability. The manager can determine for each constraint and each applicable object from S, what objects specified by $\Theta$ are used to either verify or disprove the constraint. The system then presents the user with the locations of the inconsistent elements in the appropriate artifacts.

We note that these constraints are one-way. They state that for a particular source object (say a class in a UML diagram or an instance of a design pattern) there must be some set of corresponding target objects (a class or a set of classes, methods and fields respectively in the source) that satisfy the constraint. Moreover, there is no inherent limit to what types of conditions can be checked by such constraints as both the predicates and the set of objects can be enhanced to accommodate the wide variety of software dimensions.

To demonstrate the feasibility of this idea, we have built a prototype system that can manage the evolution of a small set of software artifacts. The artifacts we have included cover a representative set of dimensions: UML class diagrams, source files, style conventions, design patterns, and design constraints.

An overview of the architecture of this system is shown in Figure 1. The details are presented in the next three sections.

## 3 Managing Abstractions

The first part of the system involves simple tools for abstracting information from artifacts and storing the result in a relational database. This includes the project manager, update monitor, artifact abstractor, and database manager of Figure 1.

The project manager is needed so that the overall system can determine the set of artifacts that define the particular software project of interest. Each project is

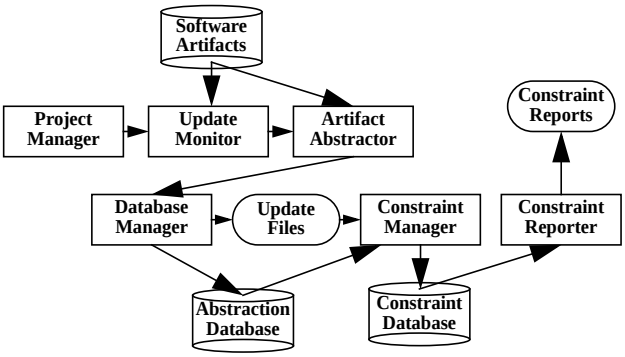


**Figure 1. Architecture for detecting and maintaining software constraints.**

defined by an XML file that specifies where in the file system to find the artifacts and the set of properties associated with the project. Properties include variables such as the Java class path, project-wide environment variables, and compiler options. The file system specification lets the programmer specify directories to include or exclude as well as patterns describing files to be explicitly included or excluded.

The update manager is a tool that runs periodically checking if any of the artifacts in the project defined by the project manager have changed since the last check. It keeps track of the previous set of artifacts and notes any artifacts that have been deleted, added, or modified. When such changes are detected, it runs an appropriate abstraction tool that scans the artifact and generate new abstraction information for the abstraction database. An XML file describes the different abstraction tools that are available and how they should be run (e.g. once for each file of the given type that has changed or once for all files of the type that changed).

The artifact abstraction tools are designed to be specific to a particular artifact type. For UML, we start with an XMI representation of the UML specification. Several UML tools generate XMI directly and translators from a tool-specific representation into XMI are available for the more common UML tools such as Rational Rose. Our front end for XMI reads the XMI file and sets up eight relations that reflect the information contained in the class diagram.

For source files, we use a modified IBM Jikes parser for Java that outputs symbol table and related information in XML format. We then have a separate tool that reads this data and generates eight relations to reflect the information.

The other dimensions of software that we currently cover are not reflected directly by artifacts. For example, there is not particular artifact that specifies style conventions. In these cases we define the data in the resultant relations directly and neither the update manager nor the artifact abstractor are used.

The result of running any of the artifact abstractors is an XML file that describes the changes needed in the database. These changes are done on a file-basis. The description file first indicates what elements should be removed from the database by specifying key-value pairs that generally identify the source artifacts that are deleted or changed. It then provides a full set of data from the new or modified artifacts.

The database manager takes this information and keeps the database up-to-date. It starts with a set of XML files that describe the database. These allow the definition of enumerated constants as well as all the relations. For each relation, they specify the keys as well any indices that should be maintained. The database manager can be used to force a recreation of the database if these files are changed. It does this both by setting up a new database and by informing the update manager that everything needs to be updated from scratch.

The database manager itself is designed to find and make incremental changes to the database even when the artifact abstractor provides it with completely new information for a modified artifact. It does this by loading the old information from any changed artifact, comparing that to the new information, matching as much as possible, and then making only the required updates. The database manager then produces a set of update files that describe the actual changes that were made in the database. This is the basis for constraint management in the next phase.

In addition to handling database updates, the database manager performs two other tasks. The first is to maintain unique identifiers (UIDs) for most of the items stored in the database. Each primary relation has a UID field. The corresponding UIDs are used later on to identify the source or affected items in constraints. UIDs also serve as keys and can be used to allow virtual pointers from one relation to another. The XML files that define the database relations specify which field is a UID and which fields the UID is dependent upon. The database manager uses this information wherever possible to preserve UIDs over updates. UID maintenance also provides facilities for mapping UIDs in the incoming change data (where they are defined locally rather than globally) into UIDs in the global database.

The second task performed by the database manager is to create and automatically maintain relations based on transitive closure. Much of the analysis that needs to be done in understanding programs requires predicates based on transitive closure. This includes analysis of scopes, class hierarchies, control and data flow information, and type inference [13]. The system provides a simple facility for defining transitive-closure based relations using an XML file. It then automatically maintains these relations whenever changes occur, again sending out update information as if these relations were updated directly.

```
<CLIME:CONSTRAINT TYPE="QUERY" NAME="uml_source_class_correspondence" >
    <CLIME:EXPR OP="FORALL" VAR="x" TABLE="UmlClass">
        <CLIME:EXPR OP="AND">
            <CLIME:EXPR OP="NOTNULL">
                <CLIME:EXPR OP="FIELD" FIELD="Name" VAR="x" />
            </CLIME:EXPR>
            <CLIME:EXPR OP="EQL">
                <CLIME:EXPR OP="FIELD" FIELD="ClassType" VAR="x" />
                <CLIME:EXPR OP="INT" VALUE="0" />
            </CLIME:EXPR>
        </CLIME:EXPR>
        <CLIME:EXPR OP="EXISTS" VAR="y" TABLE="SrcDefinition">
            <CLIME:EXPR OP="AND">
                <CLIME:EXPR OP="EQL">
                    <CLIME:EXPR OP="FIELD" FIELD="SymbolType" VAR="y" />
                    <CLIME:EXPR OP="INT" VALUE="3" />
                </CLIME:EXPR>
                <CLIME:EXPR OP="EQL">
                    <CLIME:EXPR OP="FIELD" FIELD="Name" VAR="x" />
                    <CLIME:EXPR OP="FIELD" FIELD="Name" VAR="y" />
                </CLIME:EXPR>
            </CLIME:EXPR>
        </CLIME:EXPR>
    </CLIME:EXPR>
</CLIME:CONSTRAINT>
```

**Figure 2. Example constraint definition.**

## 4 Defining Constraints

The heart of our prototype system is the constraint manager. This tool takes a set of metaconstraints defined against the abstraction database and maintains the implied set of actual constraints. The set of actual constraints and their current status is then maintained in a constraint database.

As previously noted, metaconstraints have the form $\forall(x \in S)\varphi(x)\Theta(x)$. In terms of the abstraction database, S is a relation whose elements has associated UIDs, φ controls which elements of S are appropriate to the metaconstraint, and Θ is the conditions that must be met for an instance of the metaconstraint to be satisfied. Metaconstraints are defined as set expressions using XML as shown in Figure 2. The constraint shown here checks to see that each UML class has a corresponding class in the source. The top-level operator is a FORALL that specifies the set S as the relation *UmlClass*. The AND clause immediately in the FORALL specifies the restriction φ on which elements of *UmlClass* are to be considered. Here is indicates that only those that have a non-null name and that are classes (rather than interfaces; this is indicated by *ClassType* = 0) should be considered. The EXISTS operator underneath that specifies the rest of the constraint, Θ. Here it indicates there has to be a definition in the source that is a class (*SymbolType* = 3) and that has a name that matches the name of the UML class.

In general the metaconstraint definitions can be quite complex, allowing the use of arbitrary predicates (including regular expression matching) and multiple nested EXISTS and FORALL clauses. The use of XML as an interface here is chosen so that it will later be possible to implement a user-friendly front end for specifying constraints.

The constraint manager takes each of the metaconstraint definitions and maps it into two SQL queries over the abstraction database. The first tests whether the constraint is viable for a given UID in the base set. This query can be used in one of two ways. It can be used first to find a complete list of UIDs to which the constraint applies. Second, it can be used to determine the status of the constraint for a given set of UIDs. Both return a set of UID-Boolean pairs where the UID indicates the element of S that generates the constraint and the Boolean indicates whether the constraint is currently satisfied. The first method is used initially to the initial set of constraints. The second is used in conjunction with the set of items that were updated that was provided by the database manager to update all constraints based on this metaconstraint.

The second query the constraint manager generates from a constraint definition takes a specific UID from the set S and returns a list of UIDs that specify elements of other sets that affect the validity of this constraint. If the database manager updates any of these elements, the constraint manager knows that it will have to recheck the constraint. This provides the facility for incrementally updating constraints in an intelligent way and allows the database manager to handle large numbers of constraints efficiently. Specific UIDs, however, are not always sufficient. Where the nonexistence of an element is essential to the validity of a constraint, no particular UID can be selected. In these cases, the constraint manager associates relations with each constraint and will recheck the constraint if any tuples in that relation have changed as a result of the update. While this is less efficient, it allows more powerful constraint specifications.

The constraint manager also keeps track of the constraint definition files. It notices when existing files have been modified or removed and when new files are added.

In each of these cases it will automatically recompute the set of affected constraints. This lets the developer specify new constraints and modify the definitions of old constraints dynamically, offering more flexibility to the system and making it easier to update and maintain the set of constraints.

The constraint manager stores the result of its analysis in a constraint database. (This is currently kept as part of the abstraction database, but can be logically viewed as a separate database.) This database keeps track of each of the constraints that are generated from the metaconstraints. For each it keeps the UID of the base object and well as the UIDs of all internal objects. These internal UIDs represent the portions of artifacts in the rest of the system that either validate or invalidate the particular constraint. This database is then accessible to other tools that want to track constraints and system evolution.

To demonstrate the feasibility of our approach we developed a range of different constraints. For relating UML class diagrams to the source code we defined metaconstraints that:

- Ensure each UML class has a corresponding source class.
- Ensure each UML interface has a corresponding source interface.
- Ensure each class generalization in a UML diagram corresponds to a superclass in the source.
- Ensure each interface generalization in a UML diagram corresponds to an extension in the source.
- Ensure each operation listed in a UML class has a corresponding source method with a compatible signature.
- Ensure each attribute listed in a UML class has a corresponding field with a compatible type.
- Ensure each association in a UML diagram has a correspondence in the source class is appropriate.
- Ensure each association in a UML diagram has a correspondence in the target class is appropriate.

For relating the source back to UML class diagrams we defined metaconstraints that:

- Ensure each unnested source class has a corresponding UML class.
- Ensure each unnested source interface has a corresponding UML interface.
- Ensure all superclass relationships between unnested classes have corresponding UML generalizations
- Ensure all interface relationships involving unnested user classes and interfaces are reflected in the UML diagram.
- Ensure each public method in an unnested class has a corresponding method in the UML diagram.
- Ensure each reference to another class based on field types from one unnested class to another has a corresponding UML association.

For checking programming style we added constraints that check naming conventions:

- Ensure that classes and interfaces have names of mixed case that start with an upper case letter.
- Ensure that methods have names that start with a lowercase letter.
- Ensure that fields have names that are all lowercase and that contain an underscore.
- Ensure that local variables have names that are all lowercase.
- Ensure that constants have names that are all uppercase and that may contain underscores.

To show that we could handle coding rules, we added a number of usage check metaconstraints. These include:

- Each parameter whose name does not include an underscore must be used in its routine.
- Each field defined in a class must be read at some point.
- Each field defined in a class must be written at some point.
- Each method defined in a class must be called either directly or potentially virtually.
- All fields must be either private or protected.

Finally, we defined metaconstraints that have to be specialized to a particular system. These assume new relations whose entries are specific to each system and thus are entered manually rather than being abstracted from an artifact. The metaconstraints defined in this way include:

- Let the programmer define classes that are supposed to be effectively pass-by-value. This implies that any routine returning an instance of this class has to return a new instance. This is a semantic constraint that arises in some applications. (In our case it arose for a bit-set type.) We provide a separate relation where programmers can define the appropriate types for their system.
- Let the programmer define instances of design patterns. We provide a relation that lets the user specify the pattern type and properties of that pattern. The only pattern we currently support is the facade pattern [14].

## 5 Maintaining Constraints

The remainder of our prototype system takes information from the constraint database and issues reports about the set of constraints that were generated. The current implementation is a simple tool that provides a textual view either of all constraints or of those constraints that are not satisfied. The interface to the database is designed so that more sophisticated visual tools can be constructed and so that violated constraints can be shown in the context of the artifact they originated in.

Two examples of constraint reports are shown in Figure 3. The first indicates that there is a class in the source for which there is no UML class. Here the report accesses the abstraction database to provide all the information available about the source class that is the culprit. This shows that enough information is present to locate that class in a tool if necessary. The second example indicates that a method was defined by never called. In this case, the constraint is explicitly violated by an element in the database. Here the report that is generate not only indicates which constraint is not satisfied, but lists the counterexamples that caused the constraint to fail. Again, the report demonstrates that enough information is available so that other tools could be used to show the problem.

```
source_uml_method_correspondence        FAIL (_19482) for
        Id :    _15267
        Name :  getName
        File :  /pro/clime/javasrc/edu/brown/clime/clide/ClideModelManager.java
        StartLine :     835
        EndLine :       835
        Scope : _589
        ScopeName :     <topscope>.edu.brown.clime.clide.ClideModelManager.FunctionBase
        SymbolType :    5
        StorageType :   3
        Access :        1
        Final : f
        Abstract :      f
        System :        f
        Synchronized :  f
        Volatile :      f
        Native :        f
        Static :        f
        DefTypeName :   java.lang.String()
        DefType :       _12685
        NewScope :      _593

    Update table: UmlOperation
    Update table: SrcType
    Update table: SrcTypeParameter
    Update table: UmlOperationArgument

check_methods_called    FAIL (_19632) for
        Id :    _18951
        CheckType :     5
        CheckValue :

CounterExample from SrcDefinition:
        Id :    _14524
        Name :  testFile
        File :  /pro/clime/javasrc/edu/brown/clime/clip/ClipFileType.java
        StartLine :     60
        EndLine :       60
        Scope : _106
        ScopeName :     <topscope>.edu.brown.clime.clip.ClipFileType
        SymbolType :    5
        StorageType :   3
        Access :        1
        Final : f
        Abstract :      t
        System :        f
        Synchronized :  f
        Volatile :      f
        Native :        f
        Static :        f
        DefTypeName :   boolean(java.io.File)
        DefType :       _12687
        NewScope :      _111
```

**Figure 3. Sample Constraint Report.**

## 6 Experience and Future Plans

We have used the constraint-based evolution framework for two different Java systems. The first involved the development and evolution of a 2500 line game-playing program. Here we did a UML design of the initial system before we worked on the corresponding implementation. Next we evolved the system by adding new capabilities and features sometimes by doing the design modifications first and at other times by doing the code modifications first. In all cases the constraint manager was able to keep track of what was inconsistent and was able to report the differences. We found it very useful to have a tool that automatically checked and told us what needed to be done. We also found that the existence of the constraints encouraged us to maintain the design as an accurate reflection of the code.

We also used the system on itself as it was evolving. Here we demonstrated that the system does scale (the system currently includes about 17,000 lines of code and generates about 1,340 constraints) and that it can be used practically and unobtrusively during software development. Rather than do a UML design for this system, we used Rational Rose's reverse engineering capability to construct the UML model. The fact that the various constraints on the UML were then satisfied acted as a check on both this reverse engineering process and on our constraint definitions. We also found several bugs in the system through the various style constraints.

Our prototype work has only demonstrated that the approach we plan to take is feasible and has potential. Extending the concepts to handle the full range of software dimensions and demonstrating that programmers can use the approach effectively requires significant additional research. The particular research directions we are working on include:

- Extending this approach to handle other artifacts.
- Extending this approach to handle dynamic information.
- Developing appropriate front ends for displaying inconsistencies.
- Extending the concept from detecting inconsistencies to fixing them.
- Validating this approach through more extensive use and eventually controlled experiments.
- Integrating our mechanisms into an existing development environment.

## 7 Conclusion

The approach we have taken involves the use of a constraint-based integration mechanism to maintain and manage the consistency of software artifacts during evolution. Our work to date has demonstrated that this approach is practical and worthwhile. In particular, the approach meets most of the requirements set out initially for such a framework:

- The mechanism works with existing tools using whatever artifacts the tools provide.
- The mechanism extends to a variety of software dimensions. While we have limited the prototype to a small set of dimensions, the methods and techniques are general and should extend easily. Our ongoing work includes providing such extensions to the more commonly used dimensions such as documentations.
- The mechanism is extensible. New artifacts and dimensions can be added by creating abstraction tools and updating a small set of XML description files. New constraints can be added dynamically for these or existing dimensions.
- The mechanism supports associating inconsistencies with artifacts. Each constraint is based on a simple tuple in a database relation. The abstractions that generate the relations are designed so that each such tuple has associated source information. Moreover, the system also is able to determine other tuples in other relations that validate or invalidate the constraint, thus providing further source information.
- The mechanism has relatively low overhead and runs unobtrusively. The current system, once started, will automatically maintain all the constraints. Most of the constraints are checked quickly. A few currently generate queries that the database optimizer has some problems dealing with and that run a bit slowly. However, we are hoping to fix these problems in the near future.

The only requirement that the current mechanism does not address at all is that of dynamic properties. The mechanism can only really handle static properties defined by the artifacts and has trouble expressing properties that are only reflected during system execution. We are beginning to explore what is needed to specify and check dynamic dependencies. Here it appears that simple static queries are insufficient; we are looking at alternatives such as extended finite state machines.

Finally, we note that while maintaining the consistency of the various dimensions of software is not a panacea for making software easier to write or enabling the construction of better systems, it is a good first step. We believe one needs an approach such as ours in order to move beyond the narrow view of the source code being the system. Being able to view and consistently evolve software along multiple dimensions should give developers more confidence in their systems and should ensure that the resultant systems are more understandable and do what they are meant to.

## 8 Acknowledgements

This work was done with support from the National Science Foundation through grants ACI9982266, CCR9988141 and CCR9702188 and with the generous support of Sun Microsystems. Manos Renieris and Shriram Krishnamurthy provided significant advise and feedback.

## 9 References

1. William Aitken, Brian Dickens, Paul Kwiatkowski, Oege de Moor, David Richter, and Charles Simonyi, *Transformation in Intentional Programming*. sep 1997.
2. Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter, "The Pan language-based editing system for integrated development environments," *ACM Software Engineering Notes* Vol. **15**(6) pp. 77-93 (December 1990).
3. Jagdish Bansiya, "Automatic design-pattern identification," *Dr. Dobbs' Journal*, pp. 20-28 (June 1998).
4. 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).
5. Robert W. Bowdidge and William G. Griswold, "Supporting the restructuring of data abstractions through manipulation of a program visualization," *ACM Trans. on Software Engineering and Methodology* Vol. **7**(2) pp. 109-157 (April 1998).
6. 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).
7. 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).
8. John Clements, Paul T. Graunke, Shriram Krishnamurthi, and Matthias Felleisen, "Little Languages and their Programming Environments," in *Monterey Workshop on Engineering Automation for Software Intensive System Integration*, (jun 2001).
9. President's Information Technology Advisory Committee, *Information Technology Research*: *Investing in Our Future*. feb 1999.

10. Carolyn K. Duby, Scott Meyers, and Steven P. Reiss, "CCEL: a metalanguage for C++," *Proc. Second Usenix C++ Conference*, (August 1992).
11. 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).
12. David Evans, "Using specifications to check source code," MIT LCS Technical Report (June 1994).
13. C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen., "Catching bugs in the web of program invariants," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 23-32 (May 1996).
14. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley (1995).
15. Susan L. Graham, Michael A. Harrison, and Ethan V. Munson, "The Proteus presentation system," *Software Engineering Notes* Vol. **17**(5) pp. 130-138 (December 1992).
16. 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).
17. C. Frederick Hart and John J. Shilling, "An environment for documenting software features," *Sofware Engineering Notes* Vol. **15**(6) pp. 120-132 (December 1990).
18. Daniel Jackson, "Aspect: a formal specification language for detecting bugs," MIT/LCS Technical Report 543 (June 1992).
19. Frederick P. Brooks Jr., "No silver bullet -- essence and accidents of software engineering," *IEEE Computer* Vol. **20**(4) pp. 10-19 (April 1987).
20. 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).
21. 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).
22. Jung J. Kim and Kevin M. Benner, "An experience using design patterns: lessons learned and tool support," *Theory and Practice of Object Systems* Vol. **2**(1) pp. 61-74 (1996).
23. Donald E. Knuth, "Literate programming," *The Computer Journal* Vol. **27**(2) pp. 97-111 (1984).
24. Walter Korman and William G. Griswold, "Elbereth: tool support for refactoring java programs," UCSD Dept. of Computer Science and Engineering Technical Report CS98-590 (June 1998).
25. S. Krishnamurthi, Y.-D. Erlich, and M. Felleisen, "Expressing structural properties as language constructs," *European Symposium on Programming*, pp. 258-272 Springer-Verlag, (March 1999).
26. Jeffrey D. McWhirter and Gary J. Nutt, "Escalante: an environment for the rapid construction of visual language applications," U. Colorado at Boulder report CU-CS-692-93 (December 1993).
27. Marco Meijers, "Tool support for object-oriented design patterns," Dept. of Computer Science INF-SCR-96-28, Utrecht University (August 1996).
28. Scott Meyers, Carolyn K. Duby, and Steven P. Reiss, "Constraining the structure and style of object-oriented programs," *Proc. First Workshop on Principles and Practice of Constraint Programming*, (April 1993).
29. Gail C. Murphy and David Notkin, "Lightweight source model extraction," *Software Engineering Notes* Vol. **20**(4) pp. 116-127 (October 1995).
30. Lee R. Nackman, "An overview of Montana," *IBM Research*, (1996).
31. Steven P. Reiss, "Working in the Garden environment for conceptual programming," *IEEE Software* Vol. **4**(6) pp. 16-27 (November 1987).
32. Steven P. Reiss, "A Conceptual Programming Environment," *9th Intl. Conf. on Software Engineering*, (March 1987).
33. 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).
34. Steven P. Reiss and Scott Meyers, "FIELD support for C++," *Proc. USENIX C++ Conference*, pp. 293-300 (April 1990).
35. Steven P. Reiss, "Connecting tools using message passing in the FIELD environment," *IEEE Software* Vol. **7**(4) pp. 57-67 (July 1990).
36. Steven P. Reiss, "Interacting with the FIELD environment," *Software Practice and Experience* Vol. **20**(S1) pp. 89-115 (June 1990).
37. Steven P. Reiss, *FIELD*: *A Friendly Integrated Environment for Learning and Development*, Kluwer (1994).
38. Steven P. Reiss, "Simplifying data integration: the design of the Desert software development environment," *Proc. 18th Intl Conf on Software Engineering*, pp. 398-407 (March, 1996).
39. Steven P. Reiss, "The Desert environment," *ACM TOSEM* Vol. **8**(4) pp. 297-342 (October 1999).
40. Steven P. Reiss, "Working with patterns and code," *Proc. HICSS-33*, (January 2000).
41. Don Roberts, John Brant, and Ralph Johnson, "A refactoring tool for Smalltalk," Dept. of Computer Science, U. of Illinois at Urbana-Champaign (1997 .ds [K rbjrefac).
42. Barbara G. Ryder and Frank Tip, "Change impact analysis for object-oriented programs," *ACM PASTE '01*, pp. 46-53 (June 2001).
43. Guy Lewis Steele, Jr., *Common Lisp*: *the Language*, Digital Press, Bedford, MA (1990).
44. Robert Stockton and Nick Kramer, "The Sheets hypercode editor," Carnegie Mellon University (1998).
45. 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).
46. Warren Teitelman, "A tour through Cedar," *IEEE Software* Vol. **1**(2) pp. 44-73 (April 1984).
47. Wuu Yang, Susan Horwitz, and Thomas Reps, "A program integration algorithm that accommodates semantics-preserving transformations," *ACM Software Engineering Notes* Vol. **15**(6) pp. 133-143 (December 1990).
48. Amy Moormann Zaremski and Jeannette M. Wing, "Signature matching: a key to reuse," *Software Engineering Notes* Vol. **18**(5) pp. 182-190 (December 1993).