Consistent Software Evolution

White Paper

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 result is that different software artifacts representing different dimensions tend to evolve at different rates and in different ways as the software grows and ages. In order to ensure that software can evolve in a way that maintains its inherent multidimensionality, one must ensure that the different dimensions evolve together in a consistent manner. While one could imagine this being done through a common language or a common internal representation, the most practical approach is to develop an integration framework that maintains consistency as the software evolves. Such a mechanism can be built by viewing the design and other software artifacts as a set of constraints on the source code and then providing a means for checking and maintaining these constraints. This paper describes what is necessary to make this work and what research needs to be done to make it practical.

1. Software Evolution

To most people software is the code that is the end result of the software development process. Here the initial stages of development, the specifications and the design of the system in question, the documentation, and the test cases, are ignored once the code has been developed. This narrow view of software is one of the primary causes of the many problems associated with software and its development.

Software is not just the source code; instead it is multidimensional. The specifications, design, architecture, test cases, user interfaces, coding conventions, components, constraints, design patterns, and documentation are all as much a part of a software system as is the physical code. Software development and programmer productivity depend on being able to develop and relate all these different aspects of the software.

Today's software environments provide a wide variety of tools to handle the various dimensions of software. There are tools for managing, editing, and debugging the source code. There are tools for developing and experimenting with user interfaces. There are tools for specifying the design using UML or similar notations. There are tools for creating and managing test cases for a system. There are tool prototypes for managing design patterns, components, and constraints.

These tools, however, are not coordinated or integrated with one another and the result is that the software tends to evolve inconsistently along the different dimensions. Typically, a design is created initially, but as the code gets written and modified, the design is not updated to reflect the changes. The overall behavior of the system might have been specified initially, but there is no guarantee to the programmer that this specification is actually followed by the code several years down the line. Test cases tend to be written and then become irrelevant to the evolving code. Coding conventions and constraints are emphasized initially, but are not necessarily checked or present in the developing system.

This differential evolution of the various dimensions ensures that programmers get inconsistent views of the system down the road. Programmers quickly learn to not trust design documents or the original specifications when they are faced with an evolving code base. Similarly, test cases become irrelevant and new cases are not added appropriately as the code evolves. Component interaction, originally thought to be simple, becomes much more complex and is never fully described or understood. These and related problems are bad in a moderate-sized system; they are often fatal as systems get larger and more complex.

What is needed is a software development framework that supports the consistent evolution of the various dimensions of software. This framework should let the programmer specify the software along the different dimensions. 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.

2. Approaches to the Problem

There are several approaches that can be taken to achieve consistent software evolution. The first is to develop a comprehensive language that covers all the dimensions of software. A second involves developing an semantic representation for software development that handles all the dimensions and maintains their consistency. A third approach, and the one we feel is the most practical and viable is to develop a mechanism that integrates tools for the different dimensions.

Conceptually the simplest approach to ensuring the consistency of different aspects of software is to combine all the aspects within a single programming language. There are already some attempts at combining aspects with a language framework. For example, documentation is combined with code in Java using *javadoc* and its corresponding conventions. User interface design is combined with code in programming environments such as Visual Studio or Forte for Java 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.

This approach, however, is doomed to failure in the long run. Software has too many dimensions to combine within a single framework. Not only do the different dimensions require different notations, but they also do not interact hierarchically. Even something as simple as a design pattern can have consequences in multiple methods in multiple classes throughout a system. Moreover, the set of software dimensions is not fixed. Different types of software require different specifications and designs. As new types of software systems are development, new design and specifications techniques and languages are developed. It is difficult to conceive of a language where new specifications can be easily added, integrated, and actually used. Finally, this approach does not address issues of legacy systems or the legacy components that are used in developing new software.

Rather than develop a single language that incorporates the different dimensions of software, one could develop an intermediate representation that supports a variety of tools in an integrated fashion within a software development environment. Already environments like Visual Studio 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. 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 keep track of 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 is doomed to failure. It requires that the representation be designed to handle a much broader range of software aspects than is currently done. 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 new dimensions as they are needed for new types of software and will be difficult to use with legacy systems. Most importantly, before such an intermediate representation can be developed, the central question of how the different dimensions can be related must be answered.

The third alternative is to address this central question independent of the tools, languages, and notations that are needed for defining the various software dimensions. Here software would be described as it is now as a set of artifacts or documents each of which reflects a particular dimension. There would be a new tool that acts as an integration mechanism to ensure that these artifacts remain consistent as the software evolves.

This approach 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 on legacy systems as well as 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 question that remains is: can it be achieved in a reasonable manner?

3. Requirements

Any mechanism that insures the consistency of a variety of artifacts representing the different dimensions of software must meet a broad set of requirements. In particular, to be practical and complete 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. A good mechanism should be able to extract the necessary information from the external representations used by 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.
- *Be bidirectional*. It is important that the mechanism handle changes in any aspect at any time. For example, if one changes the design one wants to know what code is affected by the change and whether it is still consistent. At the same time, one might change the code and want to know what aspects of the design are affected by the change and whether they are still consistent.
- *Be extensible*. New types of software are going to require new design techniques and approaches. The mechanism must be able to encompass these approaches as they are developed.
- *Support partial checking*. The different dimensions do not always 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 the diagram the diagram for some internal support classes. It should be possible for the mechanism to handle this and related cases and not force the programmer to provide design diagrams for low level details. Similarly, it might be appropriate to provide external documentation for public and protected methods of a class and to omit it for private methods.
- *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.
- *Have low overhead.* 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 checking*. 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 (equivalent to the halting problem). One way around this is to check constraints statically where possible, but then to make additional checks dynamically by looking at actual runs of the system in which appropriate data is collected.

If such a mechanism could be developed, it could be easily incorporated into a software development environment by providing an additional tool that told the programmer what items were inconsistent and identified the locations of the inconsistencies.

4. A Possible Approach

A key insight in developing a mechanism for maintaining consistency is that **the design and other software artifacts 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. This can be generalized to take into account situations where different design and specification dimensions impose constraints on each other and even situations where the source imposes constraints on what should be included in the design.

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 that require 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. Similarly, completeness of the design can be viewed as constraints imposed by the source. For example, a constraint can specify that every public class in the source be reflected in some UML class diagram.

In order to make this work overall, however, care must be taken 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 should 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 a wide variety of different software dimensions.

The first step in this approach is to develop a common framework (but not a common representation) for specifying all the software artifacts. This should be done by abstracting information from the different artifacts. Using an abstraction here provides independence from the actual tools being used and allows analysis to be done in order to provide a more practical basis for specifying constraints. Our initial approach here uses a relational database as the common form and defines a set of relations for each type of artifact to represent the corresponding information. This approach is flexible in that the type of information abstracted can be designed to reflect the underlying needs of the constraints. Moreover, it is possible to abstract information in multiple ways from a single representation. This is most useful for the source code where one could have separate abstractions (and hence sets of relations) 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 identities its source within that artifact

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

We have used this approach to build a simple prototype tool that can manage a small set of software artifacts. We have developed tools for abstracting information from UML class diagrams and source files. We have developed a set of thirteen constraints that relate the source and UML class diagrams. In addition, we have a set of five constraints to handle simple naming conventions, four sample constraints to handle programming style and correctness, and a specific constraint to specify a system-specific design constraint. The corresponding system has been tested through the development of a sample program and its corresponding design as well as on itself. Our experiences to date show that the approach is quite practical and workable and meets most of the requirements previously specified. Moreover, we can see easy extensions of the concept to handle a much wider variety of software dimensions.

5. Research Issues

Our work to date has only demonstrated that the approach is feasible and has potential. Significant additional research is needed both to extend the concepts to handle the full range of software dimensions and to demonstrate that the approach can be used effectively by programmers. The particular research directions we foresee include:

- *Extending this approach to handle other artifacts*. Here one needs to determine what are the relevant artifacts, create appropriate abstractions, and then attempt to define constraints based on these abstractions relating the artifacts to the source and to other artifacts. The dimensions we are particularly interested in include design patterns, other aspects of UML beyond class diagrams, behavior specification, test cases, and documentation.
- *Extending this approach to handle dynamic information*. Many of the design and specification artifacts imply constraints not on the source code itself but on the behavior of the system. One needs to determine ways of checking such constraints. This could be done, for example, by analyzing the constraints to determine what information needs to be collected at run time, minimally instrumenting the system to obtain this information, and then checking that the constraints are satisfied when the program is run.
- *Developing new approaches to software development based on this approach*. The use of a constraint framework should enable and encourage new ways of specifying the design and behavior of software. Of particular interest are specification techniques for component and web-based software.
- *Determining the appropriate representations and constraint languages*. Our initial approach uses a relational database and a set-oriented one-way constraint language. It is unclear if this is the best approach. For example, the use of a relational database makes constraints that involve following links much more complex and requires the creation of intermediate relations outside the database in order to handle constraints involving transitive closure (such as X is a subclass of Y). This question should be addressed once constraints for a broader range of artifacts have been specified.
- *Developing an appropriate front end and determining the utility of the approach*. This approach to software development is useful only if programmers will actually use it to maintain consistency. This requires that the approach be integrated into a programming environment in such a way that programmers will understand what is going on and will want to use it.
- *Extending the concept from detecting inconsistencies to fixing them*. There are many cases in software development where consistency among the artifacts can be maintained mechanically. It should be possible to augment the constraints with the additional information needed to either automatically make things consistent or to at least suggest alternatives to the programmer.

6. Conclusion

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 that an approach such as we have outline is needed if one wants to move beyond the narrow view of the source code being the system. Being able to view and maintain 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.