

Environment Support for Software Consistency*

Jun Han

Software Verification Research Centre
Department of Computer Science
The University of Queensland, 4072, Australia

*A paper appeared in *Proceedings of 8th IEEE Region 10 International Conference on Computers, Communications, Control and Power Engineering (TENCON '93)*, Beijing, China, October 1993, pp. 399-402. International Academic Publishers.

ENVIRONMENT SUPPORT FOR SOFTWARE CONSISTENCY

Jun Han

Software Verification Research Centre
University of Queensland, Qld 4072, Australia

ABSTRACT

We present three strategies which can be used to enforce, in a flexible but controlled manner, the consistency of software artifacts in software development environments. Examples of applying these strategies are given. A prototype experiment implementing these strategies is briefly described.

INTRODUCTION

Software development environments (SDEs) are software systems that assist the user (i.e., the software engineer) with software development activities. One of their functionalities is to ensure the consistency of the software artifacts developed with their assistance. A dominant approach adopted by the current CASE tools and prototype SDEs to enforce software consistency is to preserve, in one way or another, the consistency of software artifacts at all time. Experience has shown that such an approach is too restrictive and the user may become frustrated in many situations. In general, allowing temporary inconsistency is necessary to achieve flexible manipulation of software artifacts. But occurrences of inconsistency must be controllable and detectable so that they can be finally rectified. For example, the software engineer may wish to reuse a program segment by modifying it in its new context with environment assistance regarding consistency.

Based on a systematic analysis, in this paper we present three general strategies aimed at environment support for software consistency. First, we introduce the notions of software objects and object consistency. Then, we discuss the three consistency strategies and their implementation issues, and present examples of consistency problems and the enforcement of their consistency strategies. Finally, we briefly describe a prototype experiment

implementing these strategies.

SOFTWARE OBJECTS

Systematic software development requires that the software engineer follow well-established methodologies. A software development methodology concerns software products and the software processes that produce these products. To provide SDEs for such methodologies, we must identify in detail the support requirements of the software products and processes pertaining to them. In doing so, we adopt an object-oriented approach to methodology modelling: the software process or sequence of development operations, that a given methodology permits on a software product, is inherent in the product itself. We call such a software product *a software object*. An SDE assists the manipulation of all software objects allowed by the supported methodology.

A software object is defined by a structure and a number of applicable operations. The structure reflects its development process by recording the initial assumptions, the intermediate and final results, and the development relations among these assumptions and results. The operations are used to construct and modify the object (structure), and can be classified into *construction*, *editing* and *query* operations. Construction operations introduce new features into the object by creating them. Editing operations make changes to the object based on existing features of itself and other objects. Query operations access existing features of the object, but do not change them.

OBJECT CONSISTENCY

Developing a software object usually involves many steps, i.e., applications of operations. At certain stages, the object may be invalid with respect to its

may occur in terms of the various features of the object. Both construction and editing operations may introduce inconsistency into the object.

In a software development methodology which requires that a code segment must be related to a specification segment, for example, introducing a code segment without relating to a specification segment results in inconsistency in the software objects involved, and so does deleting a specification segment which has a related code segment.

CONSISTENCY STRATEGIES

As indicated above, allowing temporary inconsistency in software objects is desirable for certain consistency problems. For other consistency problems, however, preserving object consistency at all time is still necessary. For example, a methodology may insist on that no design steps be attempted for an inconsistent system specification. Therefore, an SDE should support a variety of consistency strategies. Whether a particular kind of inconsistency should be allowed and how it is treated if allowed are a matter of choosing a consistency strategy according to the nature of the consistency problem. In this section, we present three general strategies which can be used to deal with consistency problems of software objects.

1. Consistency Preserving

Under the *consistency preserving* strategy, any operation which raises the concerned inconsistency is precluded, rejected, or has additional effects to enforce consistency. As far as this consistency problem is concerned, the software objects are guaranteed to be consistent at all time.

Construction operations are very often subject to this consistency strategy to ensure that the newly introduced features of software objects do not contain inconsistency of any kind. In many CASE tools, for instance, only those operations that are applicable to the currently highlighted document segments are offered to the user for selection. This precludes the cases where improper operations are applied. In some language-based editors, input of an expression in the place of a statement will be immediately rejected. This is to avoid syntactic inconsistency in the program under development.

Editing operations may also be under the control of the consistency preserving strategy. For example, the use occurrences of a procedure name may be

The consistency preserving strategy can be realised by suppressing operations, by adding “failing-conditions” to operations, or by enforcing additional operation effects. The operations being suppressed are those that are likely to introduce the concerned inconsistency. The “failing-conditions” work in such a way that if they are satisfied, the operations will fail and no changes are made to software objects. The additional operation effects are to preserve consistency by making further changes to software objects.

2. Automatic Consistency Checking

Under the *automatic consistency checking* strategy, the concerned inconsistency is allowed, and checked by the system for every relevant operation. If inconsistency occurs, it is brought to the user’s attention immediately. Therefore, the concerned software objects may be inconsistent at certain stages but the user is always aware of the inconsistency instances.

In an example given above, deletion of a specification segment may also result in the related code segment being identified as “un-specified”. Certain Construction operations may also adopt the automatic consistency checking strategy. For example, an isolated code segment may be introduced, but identified as “un-specified”.

The automatic consistency checking strategy can be realised by associating additional consistency checking functionalities with the relevant operations, to bring the raised inconsistency to the user’s attention.

3. When-Required Consistency Checking

Under the *when-required consistency checking* strategy, the concerned inconsistency is allowed, but is not checked until the system is required to do so. The requirement comes from the user directly or indirectly. In the latter case, the requirement is due to a different task required by the user. If inconsistency is identified by a required check, it is brought to the user’s attention. Thus, the concerned software objects may be inconsistent at certain stages and the user might not be aware of all the inconsistency instances at all time.

In a language-based editor, for example, the static semantics of a program is usually checked only upon the user’s request. If any inconsistency occurs, such as an un-declared variable, it is brought to the user’s

The when-required consistency checking strategy can be realised by new operations for consistency checking or by associating additional consistency checking functionalities with existing operations, to bring the identified inconsistency to the user's attention.

In general, software development involves many kinds of software objects. They demand flexible application of consistency strategies: different software development methodologies require different consistency strategies for their objects; even in a same methodology, different consistency problems require different consistency strategies. Selection of consistency strategies for individual consistency problems is methodology-dependent.

EXAMPLES

The examples given above all come from traditional software development. In recent years, software development by formal methods has increasingly gained popularity in developing reliable software systems. It uses mathematically based techniques to ensure that the implementation of a software system meets the requirements for the system. Because of the requirement for high reliability, the support for software consistency in SDEs for formal methods plays a more significant role than in those for traditional software development methods. In this regard, we have systematically analysed formal methods' requirements for environment support, in particular for consistency support¹.

Software development by formal methods involves two major sub-tasks: program refinement and theorem proving. In this section we demonstrate, via an example proof, the effects of applying consistency strategies in software development by formal methods.

Fig. 1 shows a proof of the **and-associative-left** rule

$$\{(e_1 \wedge e_2) \wedge e_3\} \vdash e_1 \wedge (e_2 \wedge e_3)$$

constructed with the Mural proof assistant². First, hypothesis h_1 and conclusion $main$ are introduced. Then, conclusion $main$ is decomposed into assertions 2 and 5 by applying the **and-introduction** rule in a backward style. Similarly, assertion 5 is decomposed into assertions 3 and 4. By applying the **and-elimination-right** rule to hypothesis h_1 in a forward style, we obtain and prove assertion 1. Assertion 4 is proved using hypothe-

1	$e_1 \wedge e_2$	by and-E-right on $[h_1]$
2	e_1	by and-E-right on $[1]$
3	e_2	by and-E-left on $[1]$
4	e_3	by and-E-left on $[h_1]$
5	$e_2 \wedge e_3$	by and-I on $[3,4]$
<i>main</i>	$e_1 \wedge (e_2 \wedge e_3)$	by and-I on $[2,5]$

Fig. 1 A Mural proof.

sis h_1 under the **and-elimination-left** rule. Finally, assertions 2 and 3 are proved using assertion 1 under the **and-elimination-right** and **and-elimination-left** rules respectively.

Fig. 2 shows the dependence relations among the assertions involved in the above Mural proof. An important property of a proof is that an assertion should not depend on itself directly or indirectly. In other words, the dependence relation graph of a proof should be acyclic.

The above acyclicity property can be enforced in a variety of ways as the user desires.

A. When-required consistency checking. Suppose that after decomposing assertion 5 into assertions 3 and 4, we prove assertion 4 based on assertion 5 under the **and-elimination-left** rule. Then we have a dependence relation circle: $\langle 5, 4, 5 \rangle$, and the proof's acyclicity property is violated. Under the when-required consistency, this proof step is carried out as usual and the violation is not reported until the user explicitly requires an acyclicity check.

B. Automatic consistency checking. Under this strategy, the step in which assertion 4 is proved using assertion 5 is carried out but the violation of the acyclicity property is reported immediately.

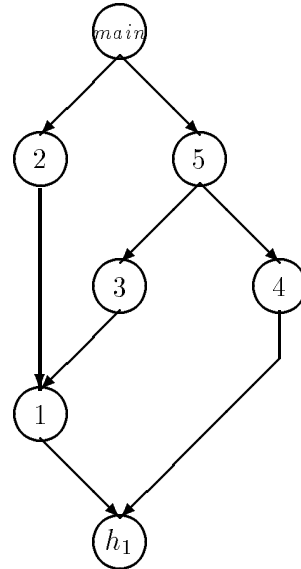


Fig. 2 Dependence relations in a Mural proof.

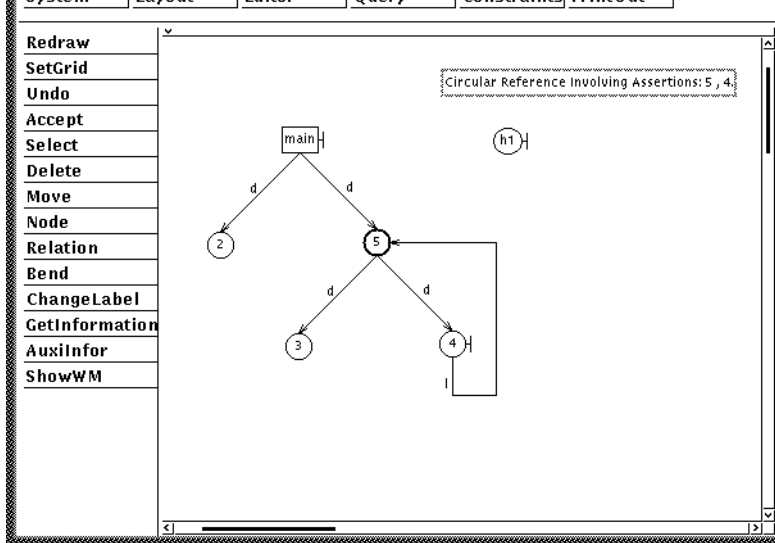


Fig. 3 Circular dependence relations in a Mural proof.

C. Consistency preserving by rejecting operations. Under this strategy, the attempt to prove assertion 4 using assertion 5 will be rejected immediately and the proof is not affected.

D. Consistency preserving by precluding operations. Under this strategy, the only existing assertions which are available for use in proving assertion 4 are assertions 2, 3 and h_1 because using any of the assertions *main*, 4 and 5 would result in a dependence circle.

In the proof construction process, the user may delete an existing assertion from a proof. A relevant issue is how to deal with the proof steps in which the assertion is involved. We may choose the consistency preserving strategy by enforcing additional effects, i.e., also deleting the directly related proof steps. In the dependence relation graph, this means that removing a node also removes the edges connected to the node.

EXPERIMENT

The consistency strategies introduced in this paper have been prototyped in an experiment for providing environment support for formal methods¹, using the language-based editor UQ1³ and the graphical visualisation system UQ-Snake⁴. Fig. 3 shows the on-screen effect of simulating the automatic consistency checking strategy in relation to the acyclicity property discussed above. (The node shapes and edge labels are for capturing the construction process, and are explained elsewhere¹.)

CONCLUSIONS

It has been recognised that different consistency

problems in software development require different consistency strategies. In this paper, we have introduced three general strategies aimed at environment support for software consistency. The effects of applying these strategies have been demonstrated via a proof example. The implementation issues of these strategies have been discussed and prototyped in an experiment. Based on our experience, we are now incorporating the support for the proposed consistency strategies in a generic software development environment.

The author would like to thank Professor Jim Welsh for his advice, and Mr. Tao Lin for his cooperation.

REFERENCES

1. J. Han, A Structural Model for Methodology-Based Interactive Rigorous Software Development, PhD thesis, The University of Queensland, St. Lucia, Australia (1992), 284.
2. C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore, *mural*: A Formal Development Support System, Springer-Verlag (1991), 421.
3. J. Welsh, G.A. Rose, and M. Lloyd, Australian Computer Journal, 18(2)(1986), 67.
4. T. Lin and P. Bernus, In: Proceedings of the 1st International Conference on Computational Graphics and Visualisation Techniques, Portugal (1991), 361.