Learning from Inconsistency

Steve Easterbrook

NASA/WVU Software Research Lab
NASA IV&V Facility, 100 University Drive, Fairmont, WV 26554
steve@atlantis.ivv.nasa.gov

Abstract

This position paper argues that inconsistencies that occur during the development of a software specification offer an excellent way of learning more about the development process. We base this argument on our work on inconsistency management. Much attention has been devoted recently to the need to allow inconsistencies to occur during software development, to facilitate flexible development strategies, especially for collaborative work. Recent work has concentrated on reasoning in the presence of inconsistency, tracing inconsistencies with 'pollution markers', and supporting resolution. We argue here that one of the most important aspects of inconsistency is the learning opportunity it provides. We are therefore concerned with how to capture this learning outcome so that its significance is not lost. We present a small example of how apprentice software engineers learn from their mistakes, and outline how an inconsistency management tool could support this learning. We then argue that the approach can be used more generally as part of continuous process improvement.

1. Introduction

During the development of a specification, software developers are directed by methods (for technical guidance) and process models (for co-ordinating development activities). Although both are prescriptive, neither are perfect. We argue in this paper firstly that a flexible approach to the application of methods and process models is needed, and secondly that much can be learnt from studying instances of deviation, especially in terms of process improvement.

In the case of methods, any particular method is developed from experience on a set of cases in a particular domain or domains; it is rare that subsequent projects will map on to the original cases perfectly. In fact, most of the common methods available now have evolved considerably from their original design. Poor method fit has hampered the uptake of CASE tools: the tools often force developers to apply a method too rigidly for practical use. There is no reason to assume that because a method is mature enough to be used widely, it should not also continue to evolve.

In the case of process models, the long term rationale for process modelling is that it facilitates process improvement. This implies an acknowledgement that process models are never perfect, that there is always room for improvement. In general, there are two ways in which process improvements are identified: retrospectively or dynamically [7]. Retrospective improvement identifies areas of stress in the process enactment, and incorporates improvements based on hindsight. Dynamic process improvement allows developers to change the process model as problems are encountered. In both cases, adherence to the process model is maintained. In practice, a combination of the two approaches is desirable, incorporating the local contextual knowledge available in dynamic process improvement, with the benefit of hindsight offered by retrospective improvement. As Cugola *et. al.* [2] argue, this can be achieved by allowing deviations from the prescribed process, and providing support for dealing with the resulting inconsistencies.

In this paper, we are concerned primarily with (deviation from) specification methods. However, the arguments apply equally to the type of fine-grained process modelling described by Nuseibeh *et. al.* [9], and perhaps to process modelling in general.

2. Inconsistency Management

In Easterbrook *et. al.* [3] we introduced a broad definition of inconsistency, as any situation in which a relationship between two parts of a specification should hold but does not. This allows us to consider inconsistencies in any notation. Of course, this also makes inconsistency entirely method-dependent, as the method (or possibly the process model) defines which relationships should hold.

The need for a tolerant approach to inconsistency has been recognised by a number of authors [1; 5; 8; 10]. While these approaches offer ways of proceeding with development in the presence of inconsistency, and of analysing and resolving inconsistencies, none have yet addressed the question of what can be learnt from the occurrence of inconsistency. We regard the occurrence of inconsistency as a good indicator of problems in the prescribed development process. For example, in [4] we show how analysis of inconsistencies can reveal conceptual disagreements between developers.

Existing work on managing inconsistency concentrates on identifying the deviations, and reasoning about the correctness of the resulting process. For example, Cugola

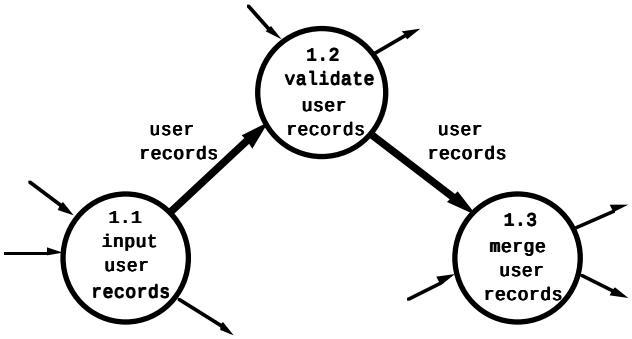

Figure 1: A portion of a dataflow diagram in which the author has deviated from the method, as process 1.2 passes its input to its output without transformation.

et. al. [2] introduce 'pollution markers', to track deviations from a process model, but do not offer any way of identifying lessons from such deviations. In the next section we present a small example, to show how learning can result from such deviations.

3. Example

When students first learn to use dataflow diagrams, they make a number of conceptual errors. For example, they may fail to distinguish between physical dataflow and logical dataflow, because they confuse the abstract notion of *process* with the concrete notions of *place* or *person*. Partly, this is because software engineering students take a while to become comfortable with the use of abstraction. This leads to a number of typical mistakes, of which we will consider just one: a failure to appreciate that a dataflow diagram is concerned with *transformation* of data.

Figure 1 shows a portion of a dataflow diagram illustrating a typical mistake: the data item 'user records' is shown as both an input to and an output from the process 'validate user records', without any apparent transformation. This is a typical naive attempt to model an observed system in which (say) paper files are passed around an office. This diagram is inconsistent according the semantic rules for dataflow diagrams.

We can recognise this as an inconsistency, using any one of a number of techniques for detecting inconsistency. Indeed we would expect a specification tool to detect such problems. The detection of the inconsistency is not what we are interested in here, but rather, what the student then does. Imagine the student is using a CASE tool, and the tool reports the inconsistency in the above diagram. The tool may even provide some analysis of the problem, perhaps identifying the edit action that led to the inconsistency. However, the student still does not understand the problem, because he has not grasped the notion of data transformation. The student needs some guidance on what the options are from this position.

A brief analysis of the mistake leads us to suggest four likely options (see figure 2):

(a) fork: the same item of data should be passed to both processes 1.2 and 1.3;
(b) rename: process 1.2 does in fact transform the data before passing it on to 1.3;
(c) bypass: process 1.2 doesn't need this data item, and it

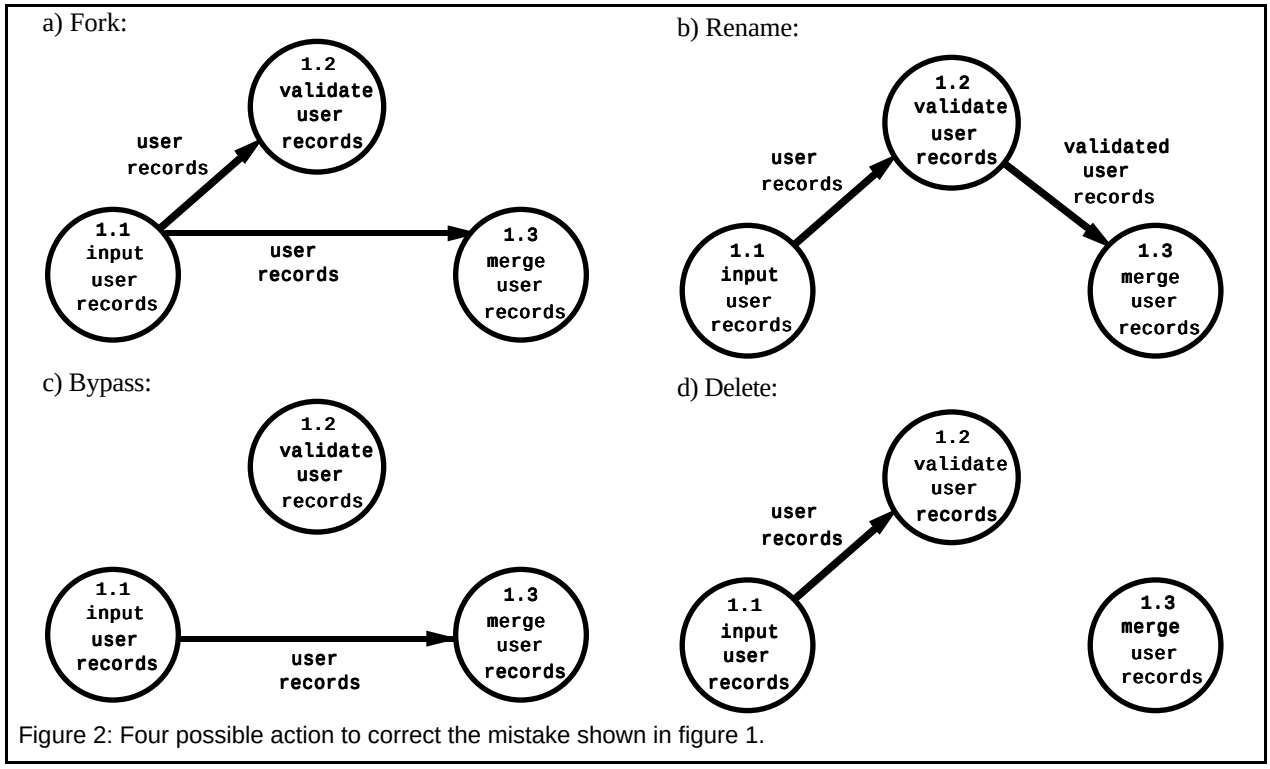

Figure 2: Four possible action to correct the mistake shown in figure 1.

should be passed straight to 1.3

(d) delete: process 1.3 doesn't need this data item, and the flow to 1.3 should be deleted.

We have given each of the actions a name for convenience: these could be offered as a menu of actions in a support tool. There are of course other actions that could also be taken that may resolve or ameliorate the inconsistency; we have merely selected the most likely.

Note at this point that the names of the processes suggest that the most likely choice is action (b). However, this is only an informal observation, available because the student has chosen particularly redundant names for the processes. In the general case, it is not possible to extract such semantic information to determine which action should be taken.

3.1. Prescribing Repair Actions:

Although the example and the four actions are relatively simple, we can imagine a large number of possible inconsistencies that can arise during software specification, and it is worth asking how feasible it is to generate a list of suggested actions for each possible inconsistency. There are three main sources of information from which the list of possible actions is constructed:

- Initial observations by the method designer - as each inconsistency rule must be explicitly defined, the designer may also be able to suggest some basic repair actions
- Analysis of the inconsistency rule - some actions can derived directed from the formal representation of the inconsistency rule. For example, any action that negates a precondition of the rule is a candidate for suggestion: actions (b) and (d) above have this effect, as they remove one of the items in the specification that caused the rule to fail.
- Past experience - each time the inconsistency occurs and is resolved, it provides data about possible actions. This is one of the learning outcomes we described below.

In addition to identifying possible actions, it is possible to reason about which to recommend. Such reasoning can take into account the context under which the inconsistency occurred. If we have available the development history of the specification, including the actions that led to the inconsistency, then some choices can be eliminated as they will be retrograde steps. Similarly, if the inconsistency occurs between parts of the specification developed by different people, and one has been updated more recently than another, then a transfer of information between the developers might be needed. Again, we expect to build up a set of heuristics to guide the selection as we gain more experience with the approach.

To explore these issues we have developed a framework for representing repair actions, in which each action has a name, for menu selection; pre- and post-conditions, to facilitate reasoning about its effects; and a rationale, to offer informal guidance about its applicability [3].

3.2. Learning Outcomes

There are five ways in which learning may occur in this example:

1) The developer learns what actions the method prescribes (or more often, proscribes). In this case the developers are students, making a rather basic mistake, and a tool that merely enforced the method *might* have induced this learning outcome. However, the lesson is reinforced by allowing the developer to explore the consequences of not obeying the rules.
2) The developer learns why it is that the method prescribes a particular way of working. In our example, this is a more important learning outcome that the first one, because it helps address the student's underlying conceptual confusion about data transformation in dataflow diagrams. It is unlikely that students would gain this learning outcome if they were prevented from making the mistake, unless a tutor observed their difficulty.
3) The method designer learns about whether the method needs updating. In our rather simple example, the method does not need changing, but one could equally well envisage an example in which the inconsistency turned out to be a new exceptional case, in which an alternative approach is needed. This is especially likely with experienced developers, who would normally have good reason to deviate from the method. One of the repair actions available in our framework is to disable the consistency rule. If the developer chooses this action, this is a strong indicator that there is a problem with method fit.
4) The method designer learns more information about how to guide subsequent developers. The resolution action chosen by the developers, and the context in which the choice was made, can be taken into account when reasoning about recommendations next time that inconsistency occurs. In this way a case library can be built up for various classes of inconsistency.
5) The developer learns more about the system being specified, because correction of the problem focuses attention on areas that are poorly understood. In our example, the student may need to go back to the domain and study further how data is passed around.

The first two of these outcomes are limited to the developers involved, and can play a useful role in training; however, as with most forms training it is impossible to observe the learning taking place directly. The next two outcomes form part of institutional learning, or process improvement. Both these types can be captured directly by the toolset, either as data for a process review activity, or as data for case-based approach to guiding development activities. The fifth outcome improves the quality of the specification, and can be seen as a validation activity.

There is one further learning outcome, not demonstrated in the example, but which we can expect to apply in team projects:

6) The developer learns more about how colleagues understand the system being specified. In cases where inconsistencies arise between portions of the specification developed by different people, they each may gain some insight into one another's view. In other cases the inconsistency may cause the developer to consult other members of the team.

It is unlikely that all six learning outcomes will apply at once, but we would expect at least one of them to apply each time an inconsistency occurs.

4. Empirical Experience

Having observed that exploration of inconsistency can lead to learning, we are currently investigating how to facilitate this learning process. We suspect that in many projects, the opportunity for this kind of learning is wasted. Our investigations are based on empirical work in conjunction with analysts currently working with NASA to assess the software requirements specifications for the International Space Station, as part of an Independent Verification and Validation (IV&V) contract.

We have been working on improvements to the methods used for performing traceability and completeness analysis on the Fault Detection, Isolation and Recovery (FDIR) requirements. At present, the development contractor produces failure models using the multigraph modelling method and associated tools. These models are then used to generate the FDIR requirements, which are currently represented in natural language, as part of a SRS conforming to DOD-STD-2167A. The IV&V team receives the SRS and validates it using their domain knowledge, and their own modelling tools. Our work concerns the introduction of a formal method that will allow the IV&V team to perform completeness analyses on these requirements. The method is based on SCR [6], and makes use of logic tables to represent state changes and the conditions under which they occur.

During the initial exploration of this new method, we have witnessed a number of inconsistencies occurring, and have used these as a way of learning more about how to fit the method to the project. We have also observed how investigation of inconsistency has helped us to develop a better understanding of the system being specified.

For example, one commonly occurring inconsistency in our initial trials with the method was that phrases expressing conditions in the SRS would be re-worded when they were placed in the tables. We had expected that a consistency check on the wording of these phrases could help with traceability and validation of the tables. In practice, the wording in the original phrases often did not make sense when removed from their original paragraphs. Forcing the two to be consistent is possible, but would reduce the readability of the tables. Hence, our preference is to allow the inconsistencies to stand for now, and to develop method guidance for certain kinds of re-wordings.

A second example arises when the requirements are analysed by different people. In some cases, the tabular representations produced by different people to represent the same requirement have differed in the number of conditions identified, and the ways in which conditions are combined. Literally, the tables had different numbers of both rows and columns. Investigation has demonstrated that the style used in the natural language specifications is inherently ambiguous. As a result we are exploring improvements in the way in which these original requirements are expressed, including introducing the tabular form earlier in the process.

Both of these examples show the method designer learning about how the method can be improved to fit the practice: learning outcomes (3) and (4) above. Whether we solve the problems by altering the method, or by evolving guidance on how to cope depends to some extent on the cost of changing the existing process. Most importantly, we have used the occurrence of inconsistencies to strengthen the argument for process change. Previous arguments about the ambiguity of natural language have foundered on the fact that replacing them completely with formal notations is prohibitively expensive. Now, however, we can show that certain kinds of ambiguity lead to specific inconsistencies. This allows us to identify smaller, incremental, changes to the process.

We have also observed how the occurrence of inconsistency leads to a better understanding of the system being specified. For example, one of the consistency checks we applied revealed that a particular mode was unreachable (Heitmeyer *et. al.* [6] describe this kind of consistency checking in more detail). Investigation revealed that the phrases "the current channel has not been reset within the last major [processing] frame" and "the current channel has not been reset within the major [processing] frame" had been interpreted as semantically equivalent. In fact the former refers to the *previous* processing frame, while the latter refers to the *current* processing frame. This distinction reveals an important aspect of the underlying model which had not been appreciated by our team: fault recovery actions can depend on events in both the previous and the current processing frames. We had assumed that only events in the previous processing frame were available for monitoring. Hence, the inconsistency allowed us to improve our understanding of the requirements.

The learning outcomes we have described in this section occurred only through careful (manual) analysis of particular inconsistencies. For learning associated with method improvement, this may be reasonable: this learning takes place outside the critical path of a project, and the time and effort needed may be reasonable. For other types of learning, scheduling deadlines may not permit the kind of reflection required. Hence our next step is to investigate how better to support the link between the actions taken to handle an inconsistency and the potential learning outcomes, so that the learning outcome is not lost. This will require further empirical work with the tools in place, to observe the conditions that help or hinder the learning effect.

5. Conclusions

A flexible approach to the application of methods allows designers the freedom to adopt development strategies that are appropriate to their particular needs. Rather than rigidly enforcing adherence to a method or process model, developers should be allowed to deviate, and to analyse the consequences. It is more important to be able to recognise that a deviation has occurred, than to prevent it. The ability to deviate from the prescribed method is important because:

- the method is never perfect
- every project is different
- developers have local expertise
- people learn best from trying things out
- deviations provide data on how to improve the method

This last point is particularly important. If changes to the method are inspired merely by stress points when the method is rigidly enforced, we will know where changes are needed, but not what those changes might be. If, on the other hand, changes are inspired by observed deviations from the method, the form and context of the deviation provides a great deal more data on how to improve the model.

This is not to say that every inconsistency would lead to an improvement in the method (or process model), but rather that every deviation has a potential learning outcome. Some inconsistencies may indicate that a method improvement is needed. Others may just provide a lesson about why the method is the way it is, and that it should be applied more rigidly in that respect in the future. Most importantly, it is not necessarily the case that the latter type of lesson is already known. Our example focused on apprentices, who are more in need of such lessons than experienced developers. However, even experienced developers and mature methods still need to learn. A commitment to continuous improvement implies a commitment to checking whether methods and process models are prescribing the right behaviour, if and when experienced developers have cause to doubt it.

5. Further Work

We are following up the ideas presented in this paper in a number of ways:

- further empirical observations of the types of inconsistency that occur in specification development, and the ways in which they induce learning.
- development of heuristics to improve the guidance offered for particular types of inconsistency, including reasoning about the development history (Eg. so that a recommended action does not take the developer back to a form that has already been considered and abandoned)
- case-based support for reasoning about which action is most likely by comparing the current situation with previous occurrences of the mistake.
- exploration of ways to alert developers to possible learning outcomes, using on our framework for inconsistency management.

6. Acknowledgements

Thanks to Bashar Nuseibeh, Jack Callahan, Chuck Neppach, Todd Montgomery, Jeff Morrison and Amer Al-Rawas for contributions to the ideas described here. This work is supported by NASA through cooperative agreement NCCW-0040.

7. References

[1] Balzer, R. (1991). Tolerating Inconsistency. In *Proceedings, 13th International Conference on Software Engineering (ICSE-13),* Austin, Texas, USA, pp158-165.

[2] Cugola, G., Di Nitto, E., Ghezzi, C., & Mantione, M. (1995). How to Deal with Deviations During Process Model Enactment. In *Proceedings,17th International Conference on Software Engineering,* Seattle, Washington, pp265-273.

[3] Easterbrook, S. M., Finkelstein, A. C. W., Kramer, J., & Nuseibeh, B. A. (1994). Co-ordinating Distributed ViewPoints: the anatomy of a consistency check. *Concurrent Engineering: Research and Applications*, *2*(3), 209-222.

[4] Easterbrook, S. M., & Nuseibeh, B. A. (1995). Managing Inconsistencies in an Evolving Specification. In *Second IEEE Symposium on Requirements Engineering,* York, UK, pp48-55.

[5] Gabbay, D., & Hunter, A. (1991). Making Inconsistency Respectable: A Logical Framework for Inconsistency in Reasoning, Part 1 - A Position Paper. In *Proceedings, Fundamentals of Artificial Intelligence Research '91,*, pp19-32.

[6] Heitmeyer, C. L., Labaw, B., & Kiskis, D. (1995). Consistency Checking of SCR-Style Requirements Specifications. In *Second IEEE Symposium on Requirements Engineering,* York, UK, pp56-63.

[7] Jamart, P., & van Lamsweerde, A. (1994). A Reflective Approach to Process Model Customisation, Enactment and Evolution. In *Third International Conference on the Software Process,* Reston, Virginia.

[8] Narayanaswamy, K., & Goldman, N. (1992). "Lazy" Consistency: A Basis for Cooperative Software Development. In *Proceedings, International Conference on Computer-Supported Cooperative Work (CSCW'92),* Toronto, Canada, pp257-264.

[9] Nuseibeh, B., Finkelstein, A. C. W., & Kramer, J. (1993). Fine-Grain Process Modelling. In *Proceedings, Seventh International Workshop on Software Specification and Design (IWSSD-7),* Redondo Beach, CA, USA, pp42-46.

[10] Schwanke, R. W., & Kaiser, G. E. (1988). Living With Inconsistency in Large Systems. In J. F. H. Winkler (Ed.), *Proceedings of the International Workshop on Software Version and Configuration Control,* Grassau, Germany, pp98-118.