Edit, Compile, Debug – From Hacking to Distributed Engineering

Christian Nentwich, Wolfgang Emmerich and Anthony Finkelstein
Department of Computer Science
University College London, London WC1E 6BT
{c.nentwich,w.emmerich,a.finkelstein}@cs.ucl.ac.uk

1 Background

Specifying a system from different viewpoints, in heterogeneous design notations, and using a distributed team, introduces several challenges that test the state of the art in software engineering. One of these challenges is to check the consistency of such heterogeneous notations, and to deal with the problem of inconsistency throughout the lifecycle — including the management of inconsistency between notations at the same stages of the lifecycle, such as multiple UML models, and inconsistency between stages, such as the consistency of design and code.

We envisage a scenario where developers can execute consistency checks at arbitrary stages of the lifecycle in order to get feedback. Developers are presented with reports on the current consistency status, and can decide whether or not to take action. Because of the desynchronised nature of distributed software engineering, we take a lightweight approach that does not require total consistency, but instead fetches specifications when developers are prepared to evaluate consistency, and points out inconsistencies without forcing changes.

Our most recent work is aimed at providing developers with a range of choices for repairing documents, and at addressing problems of scalability in checking distributed documents.

2 Consistency Checking

To support the kind of activity outlined above we have developed xlinkit [2], a generic technology for checking the consistency of distributed, heterogeneous documents. xlinkit is a fully implemented, working system that has been used in several industrial case studies and can be downloaded at `http://www.xlinkit.com`.

While xlinkit can be used in many application domains, it is very suited to software engineering documents. In [3] we describe the application of xlinkit to checking the consistency of the design, implementation, and deployment descriptors of Enterprise JavaBeans (EJB)-based systems.

Consistency is defined in xlinkit through a set of constraints, called *consistency rules*. A consistency rule defines how elements in different documents are interrelated. Figure 1 gives an example of a consistency rule (using xlinkit's XML rule syntax) that prescribes that interfaces that inherit from "`EJBObject`" must be implemented by a class that inherits from "`EntityBean`" or "`SessionBean`".

xlinkit returns as its output a set of links that connect inconsistent elements in document. This is an improvement on the boolean evaluation of first order formulae because the links pinpoint precisely the combination of elements that causes the inconsistency. Figure 2 shows a link generated by evaluating the sample rule. In this case, the link points to only one location, the interface for which no corresponding class was found.

3 Consistency Management

Consistency management, the problem of detecting, assessing and potentially resolving inconsistency, is a delicate topic that is subject to a variety of influences, from domain specific ways of accomplishing goals, down to individual working preferences. Having provided a sophisticated system for detecting inconsistency, our goal is to specify a very basic, rudimentary mechanism for reporting inconsistencies and for making simple changes in order to remove inconsistencies. Once such a basic mechanism is in place, it can form the infrastructure for a more elaborate consistency management or conflict resolution process that can take into account information such as workflow, or policies that set priorities for different kinds of specifications to enable overriding.

On a very abstract level, we expect a consistency man-

```
<forall var="i" in="/java/interface[extends/@name='EJBObject']">
  <exists var="c" in="/java/class[../package/@name=$i/../package/@name
                      and (implements/@name='EntityBean' or
                           implements/@name='SessionBean')]"/>
</forall>
```

Figure 1: Sample xlinkit consistency rule

```
<xlinkit:ConsistencyLink ruleid="javaejb_inter.xml#id('r2')">
    <xlinkit:State>inconsistent</xlinkit:State>
    <xlinkit:Locator
        xlink:href="Job.java#/java/interface"/>
</xlinkit:LinkBase>
```

Figure 2: Sample consistency link

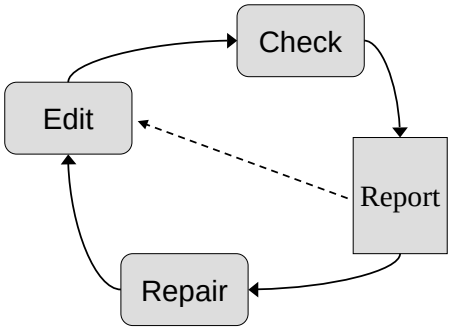

Figure 3: Consistency management process

agement process to look like Figure 3. Developers will decide to assess the consistency of their documents, make a check, get reports back, and either decide on an action for "repairing" the documents, or defer the decision and go back to editing. We have already addressed the checking process, and will concentrate on report generation and repair in the next two sections.

4 Report Generation

xlinkit's linking diagnostics, which connect inconsistent elements in distributed specifications and deliver them in the form of an XLink [1] linkbase, are powerful but not particularly friendly for human consumption. We have developed a report generation tool, *Pulitzer*, that can read such linkbases and display marked up information about the elements that the locators point to.

Using Pulitzer it is possible to provide reports similar to traditional compiler error output — but at the interviewpoint level! — in a variety of formats. Figure 4 shows an HTML report generated after checking an EJB system.

5 Repair

When elements in distributed specifications have been linked and identified as inconsistent, it should be possible to offer some simple repair choices to developers. As an example, in our sample rule given above we have an inconsistency because for some Java interface we did not find a correct implementation. In this case, we could offer the developer to automatically introduce a new Java class that fulfills the requirements.

In general, it is straightforward to enumerate the repair options available to developers. One case add elements to specifications, delete elements, and change properties of elements. When adding or changing elements, one can let developers enter values for the properties of elements, search the specifications for existing values to use, or fall back to defaults.

The challenging task is then not to enumerate repair options but to prune them in order to leave only "sensible" choices. In our sample rule, we could remove the inconsistency by deleting the offending Java interface — although in most cases this would be attacking the symptom instead of the problem, and would not be a sensible choice. We believe that the developers responsible for establishing the constraints should be able to determine which repair actions make no sense. We are planning to analyse the formulae so as to come up with an exhaustive list of actions and presenting a simple interface for pruning them down.

Should this kind of pruning process still result in an overwhelming number of repair choices, domain specific heuristics can be used to cut them down further. For example, it may be possible in a software engineering setting to use a "differential" approach that compares the consistency status of specifications before and after the last consistency check, and uses this information to determine more precisely which elements may have to

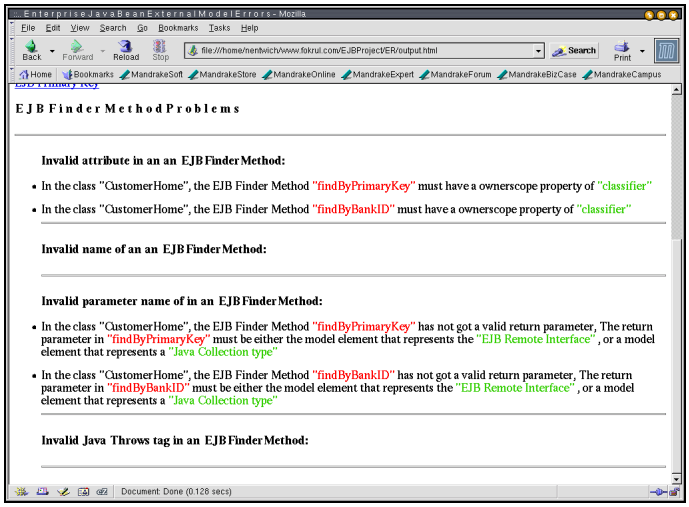

Figure 4: Sample report

be changed. Whether this approach works or not, by providing an enumeration of the repair choices based on formula analysis, we will establish a sound infrastructure on top of which such heuristics can be built.

6 Scalability

Scalability problems occur in various guises in consistency management. A good system must be able to scale as the number of documents and constraints increases, and as the size of documents increases. It must also be scalable in terms of user-friendliness, by providing mechanisms that simplify the expression of a large number of constraints.

We are currently working to address the problems of checking large documents and large numbers of documents through a variety of means: by providing incremental checks, which analyse changes to documents and minimize the number of constraints that have to be rechecked; by implementing a distributed *supervisor – worker* architecture that can deal with large amounts of data; and by using the caching facilities of native XML databases to lower the main memory requirements of our service. Substantial progress has been made in these areas.

We have also implemented a macro mechanism and a predicate plugin mechanism for our formula language that greatly facilitate the expression of a large number of constraints. We are currently evaluating these features in case studies.

7 Conclusion

We have outlined our current position with regard to achieving a system for managing the consistency of distributed specifications. xlinkit takes a big step in this direction by providing a solid base for checking the consistency of distributed, heterogeneous documents.

We are now working to establish a simple method for suggesting repair options to developers, without compromising our view of inconsistency as something that cannot necessarily be eliminated. We are also evaluating our tools for dealing with scalability problems that arise as documents sizes and the number of documents grow.

References

[1] S. DeRose, E. Maler, and D. Orchard. XML Linking Language (XLink) Version 1.0. W3C Recommendation http://www.w3.org/TR/xlink/, World Wide Web Consortium, June 2001.

[2] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2001. To appear.

[3] C. Nentwich, W. Emmerich, and A. Finkelstein. Flexible Consistency Checking. Research note, University College London, Dept. of Computer Science, 2001. Submitted for Publication.