

# Supporting requirements model evolution throughout the system life-cycle

Neil Ernst, John Mylopoulos  
Dept. of Comp. Science  
Univ. Toronto, Canada  
{nernst,jm}@cs.toronto.edu

Yijun Yu  
Dept. of Computing  
Open University, UK  
y.yu@open.ac.uk

Tien Nguyen  
Elec. Comp. Engineering  
Iowa State Univ., USA  
tien@iastate.edu

## Abstract

*Requirements models are essential not just during system implementation, but also to manage system changes post-implementation. Such models should be supported by a requirements model management framework that allows users to create, manage and evolve models of domains, requirements, code and other design-time artifacts along with traceability links between their elements. We propose a comprehensive framework which delineates the operations and elements necessary, and then describe a tool implementation which supports versioning goal models.*

## 1. Introduction

Software-based systems are subject to change pressures. Many of these change pressures affect a system's requirements [1, 2], which in turn affect system architecture, design and code. To support the evolution of requirements for systems subject to adaptive pressures, we need requirements models throughout the software lifecycle, and particularly post-implementation. These models provide the maintainer guidance and motivation for code-level changes such as refactorings or the addition of new functionality. To manage models of these requirements, a requirements model management system that can cope with change is necessary.

## 2. Requirements model management

To formalize our discussion, we make use of the notion of an abstract data type (ADT). A requirements model management system deals, broadly, with three domain models, namely requirements goal model  $G$ , implementation model  $I$ , and (reified) traceability model  $T$ . An element in  $G$  is either a goal or a relation between two goals; an element in  $I$  is a component or a relation between them, categorized as calls, implements, uses; an element in  $T$  is an object reifying a relation between an element in  $G$  and an element in  $I$ .

**Operations in the ADT** – Operations on  $I$ , the implementation model, include ones used during design and implementation. For the traceability model  $T$ , the operations reflect the creation and maintenance of the traceability elements. In addition, it is desirable to be able to **merge** models. For example, changes to the initial model may need to be merged into a derivative of that model. The merge operator takes two models of the same type, a matching relation between them (e.g., simple text matching), and produces a new model. There is also an inverse operation, **split**. Match takes two models and generates a relationship between their elements. **Trace** (between a goal model and an implementation model) is a related operator that matches between models  $G$  and  $I$ . We define a specialized **diff operator** which takes two versions of  $G$ , and describes a set of operations,  $D$ , required to reproduce the original from the changed model. The operations include metaproperties such as the date of the diff, size of diff, and user who called the function. There are several possible techniques to generate such operations. The most common is to use a text-based diff which can generate line by line comparisons. Other approaches include those demonstrated in model diff tools. The next operator we expand on is the **slice operator**. A slice in our framework produces a subset of a requirements model satisfying the criterion. We also define three operators specific to the problem of changing requirements models.

The **version operator** takes a goal model at some point in time, appends a logical timestamp  $v$ , and places it in persistent storage. We call the right-hand side  $M$ , a mirror of the goal model at time  $v$ . The **Temporal commit operator** updates the persistent layer with the changesets as defined by **diff**, where  $M1$  is the "HEAD" version of the repository. This **temporal query operator** uses the diff result. It produces a set of objects matching the criterion at either the meta-level (properties of the diff) or object level (versions of object  $x$  in

time).

Finally, an important quality of goal models is **evaluation**. There are several different means to assess how well a goal model is satisfied, which our model management system needs to keep track of. In other words, we also want to store metadata for the individual models as to evaluation results, for some evaluation algorithm  $F(G)$ .

### 3. Implementation

Our implementation, which we call OpenOME, is a goal modeling tool supporting several languages. At its most elemental, the tool implements fundamental goal modeling operations (such as create, edit, delete, evaluate). OpenOME has three flavours of editor to work with goal models: a) a text editor, leveraging XText DSL tools, supporting syntax highlighting and checking; b) an object-oriented editor of goal models with tree-based outline of objects and table-based field property views (generated from the Eclipse Modeling Framework (EMF)); c) a graph editor of the goal models. This editor is generated from the Graphical Modeling Framework (GMF).

On top of this, we integrated a customizable, object-oriented, configuration management (CM) infrastructure, Molhado [4]. Our implementation minimizes the cognitive dissonance of text-based systems, such as CVS, as developers no longer need to translate between the problem domain and the development environment. The implementation of the ADT concentrates principally on supporting the **version** and **temporal commit** operators, but we have also implemented **diff**, **match** and **temporal queries**. We are currently working on generating model **slices**. There is existing work which addresses model **merging and matching**. We evaluated the implementation using three versions of a goal model of several thousand entities, spanning two years of development.

Molhado has a novel slot-based property mechanism. We represent the properties associated with a goal model entity, such as a goal, using these slots to store metadata about the element over time. To manage our models, we created five algorithms that implement the **version** model management operation we mentioned previously. These algorithms are responsible for mirroring, mapping, and recording the operations performed by the user into the version control system. We assessed the space and time complexity of the approach to evaluate **scalability**. We found that the algorithms were either linear or  $\log N$  in time complexity. A reporting framework displays changes between different versions of an artifact. Since our CM is model-driven, we describe a reporting and differ-

encing scheme that addresses domain elements, rather than typical line-oriented differencing algorithms. Our evaluation showed our tool minimized the size of the changesets by a factor of three over XML and text diff tools. Finally, to show that our implementation maintains change history conformance, we ‘replayed’ a series of changes in a diff  $D$ , to confirm that our algorithms do not alter the models beyond the operations the user specified.

### 4. Conclusions and future work

This paper has highlighted the need for support for requirements evolution throughout the software lifecycle, that is, post-implementation. To support this, we called for a requirements model management framework which could support requirements modeling, versioning, merging, reporting, and traceability to code. We noted that merging and traceability are supported in existing research, but that the areas of versioning and reporting are under-studied. We described a tool which supports these two areas of requirements model management, and evaluated the ability of that tool to manage large goal models. An extended version of this paper is available at <https://se.cs.toronto.edu/ome/wiki/REPaperDetails>.

We intend to use our framework to guide work in tracing from source to goal models and vice versa, possibly using incremental LSI as in [3]. The issues associated with changing requirements – tracing, impact assessment, etc. – will be difficult to manage with existing tools. We are interested in exploring how our model management techniques will fit with less formal requirements practices.

### References

- [1] V. R. Basili and D. M. Weiss, “Evaluation of a software requirements document by analysis of change data,” in *Intl Conf. on Software Engineering*, San Diego, USA, 1981, pp. 314–323.
- [2] S. D. P. Harker, K. D. Eason, and J. E. Dobson, “The change and evolution of requirements as a challenge to the practice of software engineering,” in *Intl Symp on Requirements Engineering*, 1993, pp. 266–272.
- [3] T. Nguyen, H. Jaygarl, and I.-X. Chen, “Incremental latent semantic indexing for traceability link evolution management,” in *Intl Conf on Software Engineering*, Leipzig, Germany, September 2008, to appear.
- [4] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao, “An infrastructure for development of object-oriented, multi-level configuration management services,” in *Intl Conf on Software Engineering*, St. Louis, MI, May 2005, pp. 215–224.