UNICORN: A Tool for Modeling and Reasoning on the Uncertainty of Requirements Evolution *

Le Minh Sang Tran and Fabio Massacci

University of Trento, Italy
{tran, fabio.massacci}@disi.unitn.it

Abstract. Long-living software systems keep evolving to satisfy changes in their working environment. New requirements may arise, while current requirements may become obsoleted. Such requirements evolution fortunately could be foreseen at some level of (un)certainty. The paper presents UNICORN, a CASE tool for modeling and reasoning on the uncertainty of requirements evolution. The tool provides graphical constructs as well as different views of requirements evolution to assist users to model requirements evolution. The tool also supports the evolution analysis in which facilitate the selection of design alternative.

Keywords. requirements evolution, known-unknown evolution, max belief and deferral belief, CASE tool.

1 Introduction

Long-living software systems keep evolving as they need to continue to satisfy changing business needs, new regulations and standards, and the introduction of new technologies. Some of future requirements may be known to be possible, but it is unknown whether they would actually be needed: the *known unknown*. The final standardization of two competing alternatives is a simple example of such phenomenon. Unfortunately, a company who ship or buy software cannot wait until all unknowns become known. The process of tendering and organizational restructuring requires a significant amount of time and planning. Therefore, when having a number of possible design alternatives for the system-to-be, decision makers at high-level need to choose a viable design alternative that is evolution-resilient (*i.e.* minimize the risks that it cannot satisfy evolving requirements and thus needs replacement). Obviously, implementing a new design to replace for an obsoleted one may be more expensive than having a design that still be applicable when new requirements arise.

In this paper we present a CASE tool which aims to support an RE approach to model and reason on requirements evolution (previously proposed in [4, 5]. The objective of the approach is to capture what Loucopoulous and Kavakli identified as the knowledge shared by multiple stakeholders about *"where the enterprise is currently"*, *"where the enterprise wishes to be in the future"*, and

* This work is supported by the European Commission under the project EU-FP7-NoE-NESSOS

"alternative designs" for the future state [3]. The approach assists choosing an appropriate design alternative, in consideration of uncertainty of requirements evolution, based on quantitative metrics.

This paper is organized as follows. In §2 we give an overview about the RE approach [4, 5] as a baseline for the tool. We then present the tool architecture in §3. We run a demo scenario in §4 and conclude the paper in §5.

2 The Approach on Requirements Evolution

Our approach [5] aims at dealing the uncertainty of the requirements evolution to leverage the selection of an 'optimal' design alternative. Requirements evolution is captured in terms of evolution rules. There are two kinds of rules: *observable rule* and *controllable rule*.

The observable rule captures the way requirements evolve. Requirements evolution might be either intentional (*e.g.,* planned changes) or unintentional (*e.g.,* changes dues to new business needs). Both may be uncertain because when mentioning about future, "the only certainty is that nothing is certain"(*Pliny the Elder*[1]). Concretely, the unintentional evolution is uncertain because we do not know whether it happens. The intentional evolution is planned, but it is still not 100% for sure because some expected reasons might impact the future plan, *e.g.,* financial issues, introduction of new standards. Therefore, intentional evolution is also uncertain. The uncertainty of evolution is captured by the *evolution probability* which is the belief that evolution might materialize in future. The semantic of such belief could be accounted by using the game-theoretic approach described in [5].

Let RM be the original requirements model, and RM_i be one of the evolution possibilities that RM may evolve to. For sake of simplicity, all RM_i are complete and mutual exclusive, *i.e.* one and only one RM_i materializes. The observable rule (r_o) is as follows.

$$r_o(RM) = \left\{ RM \xrightarrow{p_i} RM_i \,\middle|\, \sum_{i=1}^{n} p_i = 1 \right\} \tag{1}$$

where p_i is the evolution probability for which RM evolves to RM_i; n is the number of all evolution possibilities of RM. The sum of all p_i equals to one.

The controllable rule captures the way how requirements are satisfied. A requirements model usually have different design alternatives whose implementation will satisfy the requirements. This is described in a controllable rule. Let RM be a requirements model, and DA_j be a design alternative (*i.e.* set of elements satisfying all mandatory requirements) of RM. The controllable rule (r_c) is as follows.

$$r_c(RM) = \{RM \rightarrow DA_j | j = 1..m\} \tag{2}$$

where m is the number of design alternatives of RM. Here we abuse the arrow notation ($\rightarrow$) to express both observable and controllable rules.

[1] Gaius Plinius Secundus (23 – 79), a Roman naturalist, and natural philosopher.

Our analysis on design alternatives relies on two quantitative metrics namely *Max Belief* and *Deferral Belief*[2]. The former indicates the maximum belief that a design alternative will still be applicable when evolution happens. The latter is the belief that a design alternative turns out to be not applicable when evolution happens. The criterion to justify among design alternatives in terms of evolution resilience is that: *"Higher max belief, lower deferral belief"*. Interested readers are referred to [5] for more detailed discussion on these two metrics.

3 The Main Features and Architecture

3.1 Features Overview

UNICORN is an Eclipse-based tool which aims to support the approach described in [5] (and briefly reviewed in §2). The tool is provided as a set of EMF-based Eclipse plug-ins written in Java, relying on standard EMF technologies such as GMF, Xtext. The features of the tool can be categorized into two major categories: *Modeling support* and *Reasoning support*.

The *modeling support* includes features necessary to model requirements evolution. Important features in this category are as follows:

- *Support requirements evolution modeling.* UNICORN provides several constructs to capture evolution rules in a requirements model. Different evolution possibilities and different design alternatives are supported. Evolution is supported in different levels: from high level requirements to low level ones.
- *Support different views.* Several views are supported to assist designers. In particular, *Normal View* shows the complete requirements with evolution rules; *Evolution View* presents only evolving parts of the model; and *Original View* displays the requirements model without any evolution.
- *Support large model.* A large requirements model can be partitioned into several sub models. Sub models are edited in separated windows. Each model can reference to other models. Changes in a model will be automatically reflected to other models.
- *Support customization and extension.* The graphical constructs of UNICORN are highly customizable. Adding a new constructs with custom figures and attributes, or modifying existing constructs can be done without changing the UNICORN source code.

Fig. 1 presents the basic constructs by which we draw requirements models in UNICORN. A requirement entity represents a requirement. A refines relation connects a requirement to other requirement. It means that the parent requirement can be fulfilled if its child is fulfilled. If more than one children are required, these children connect to an extra compound node which in turn connects to the parent requirement. By allowing several refines relations to connect to a requirement, we can model the different design alternatives of a controllable rule. An observable entity represents an observable rule where the original requirement is

[2] We rename the *Residual Risk* metric in [5] to *Deferral Belief* to distinguish it with the concept of *Residual Risk* in the field of risk management.

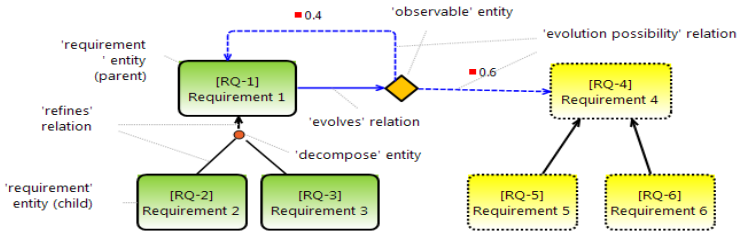

Fig. 1. The constructs to modeling requirements evolution in UNICORN.

connected by an evolves relation. Evolution possibilities are connected by evolution possibility relations. Elements in other diagram could be reference by special construct off-diagram reference.

In Fig. 1, the original requirements model RM has three requirements: RQ-1, RQ-2, and RQ-3. RQ-1 is refined to RQ-2 and RQ-3. Therefore, RM has one design alternative which is $\{\text{RQ-2}, \text{RQ-3}\}$. RM might evolve to a possibility RM_1 in which RQ-4 is refined to either RQ-5 or RQ-6. The evolution probability for this evolution is 0.6. Besides, RM might remain unchanged with the probability of 0.4. The observable and controllable rules captured by this figure are as follows.

$$r_o(RM) = \left\{ RM \xrightarrow{0.6} RM_1, RM \xrightarrow{0.4} RM \right\}$$
$$r_c(RM) = \{RM \rightarrow \{\text{RQ-2}, \text{RQ-3}\}\}$$
$$r_c(RM_1) = \{RM_1 \rightarrow \{\text{RQ-5}\}, RM_1 \rightarrow \{\text{RQ-6}\}\}$$

The *reasoning support* provides an environment for developing automated analyses on requirements models. For example, the graphical models could be transformed into a data structure that facilitates the analysis. The traceability between the modeling constructs and transformed data structure is also maintained. Currently, we have implemented following analysis:

– *Evolution analysis*: This analysis walks through the entire requirements models and calculate quantitative metrics (*Max Belief, Deferral Belief*) for each design alternative. The analysis can incrementally update the metric values with respect to changes in the model as soon as the user changes the models.

3.2 Architectural Overview

The tool architecture is specially designed to support a high level of customization and extension. Fig. 2 presents the overall architecture of the UNICORN tool. In this figure, components are depicted by rectangles. The headed-arrow connections denote the interaction between components where the source components invoke (or use) the target ones. These components are briefly described as follows:

– The *Universal Data Model* is a common storage for the constructs of all models. Since graphical constructs could be defined by users (*e.g.,* add new

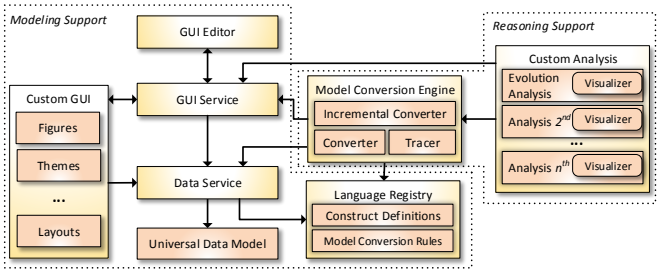

Fig. 2. The overall architecture of the UNICORN tool.

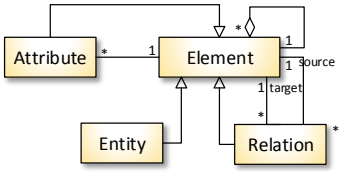

Fig. 3. The class diagram of the *Universal Data Model*

.

construct with custom attributes, or add new attributes to existing constructs), the Universal Data Model is a meta-meta model (see Fig. 3).

- The *Language Registry* maintains definitions of graphical constructs in requirements models, as well as conversion rules to transform the data model to other data structures used by analysis. The construct definitions and conversion rules are defined in configuration files which are fully customizable.
- The *Data Service* uses the construct definitions in the Language Registry to allow other components to manipulate the data stored in the data model.
- The *GUI Service* is in charge of manipulating graphical objects and data model. It employs the *Custom GUI* components to create several GUI objects (*e.g.,* construct figures, themes, and so on) consumed by the *GUI Editor*, which is a front-end graphical editor.
- The *Model Conversion Engine* uses the conversion rules stored in the Language Registry to convert the requirements model to the underlying data structures used by the custom analysis.
- The *Custom Analysis* is a set of analyses run on the editing requirements model. Each custom analysis has a *Visualizer* to show the analysis result.

Fig. 3 describes the class diagram of the *Universal Data Model*. The *Element* is an abstract class representing any element in the model, which could be either an entity, or a relation. An *Entity* could be a requirement, or an observable node. A relation captures the relationship between entities. An *Attribute* is a special kind of element, holding the attribute value of an element.

Fig. 4 shows the syntax of the construct definition file in the Extended Backus-Naur format. To keep the syntax tidy and clear we do not provide complete definition of some non-terminal production rule such as expression, as well as common terminal symbols such as identifier (ID). The definition file begins

```
1 language ::= "language" ID (element)* ";"
2 element ::= ("entity"|"relation") ID "{" attribute * "}"
3 attribute ::= ("field")? ID (":" type)? "=" expression ";"
4 tag ::= ID ":" expression
5 type :: = "string"|"float"|"date"|"boolean"|ID
```

Fig. 4. The compact syntax of the construct definition file.

```
1 entity requirement {
2     figure = new RoundRectFigure() { Size=(50,60),
3                 addChildFigure(new CenterLabelFigure("req_name"), DOCK_FILL)};
4     req_name_parser = {pattern="{0}:{1}", fields=(Name, Description)};
5     field Actor: string = "";}
```

Fig. 5. A fragment of a construct definition file.

with a keyword language followed by an ID which is the language name and a set of elements. An element is either an entity or a relation. Each element has a set of attributes which has name, data type (optional), and initial value.

Fig. 5 exhibits an example where the requirement construct is defined with respect to the grammar denoted in Fig. 4. The requirement construct is an entity whose graphical representation is a round rectangle with a label inside. The label is to show and edit the name and the description of this requirement. There is one text field Actor in requirement. The initial value of this field is a blank.

4 Demo Scenario

We demonstrate the features supported by our tool in a scenario taken from an industrial project: the System Wide Information Management (SWIM) [1,2]. The scenario concerns the evolution in the requirements models of the Enterprise Information System Security and the External Boundary Protection Security [1, section 5.6]. In this scenario we focus on the authentication and the implementation of boundary protection (BP) services.

Table 1 reports the list of requirements, and their design alternatives. The table is divided into two parts: design alternatives on top, and requirements on bottom. The check mark (✓) in the cross join of a design alternative and a requirement indicates that this design alternative satisfies the corresponding requirement. Both design alternatives and requirements have unique identifiers, and short descriptions.

Modeling requirements evolution. Fig. 6 illustrates the requirements model with evolution rules of the scenario. The model says that the requirement RQ-0 is refined to both RQ-1 and RQ-2. RQ-1 is later refined to RQ-3, and so on. RQ-1 has an evolution rule where RQ-1 might remain unchanged with probability 0.4, or might evolve such that RQ-1 will be refined, in a new way, into RQ-3, RQ-4, and RQ-5. The rest of the diagram can be read in the similar manner. Due to space limit, some screen shots (*e.g.,* different views) are not provided. Interested readers are referred to the web site of the tool [3].

[3] http://disi.unitn.it/~tran/pmwiki/pmwiki.php/Main/Unicorn

Table 1. The requirements and design alternatives.

ID	Design Alternative						
RQ-10	Simple IKMI						
RQ-11	Ad-hoc SSO solution						
RQ-12	OpenLDAP						
RQ-13	Active Directory						
RQ-14	Oracle Identity Directory						
RQ-15	Ad-hoc solution for BP services						
RQ-16	Common gateway for BP services						
RQ-17	Centralized Policy Decision Point (PDP)						
ID	**Requirements**	**Alternative**					
RQ-3	Manage keys and identities of system entities			✓	✓	✓	✓
RQ-4	Support Single sign-on			✓	✓	✓	✓
RQ-5	Support large number of entities			✓	✓		
RQ-6	Less program dependencies BP services			✓			
RQ-7	Robust and scalable BP services		✓				
RQ-8	Simpler operation of BP services	✓					
RQ-9	Overall security assessment supported		✓				

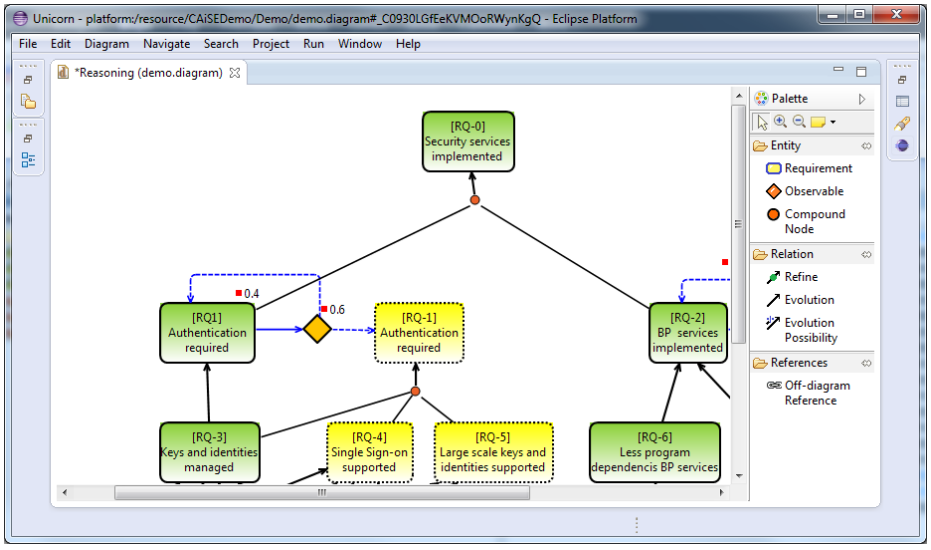

Fig. 6. The requirements model of the scenario with evolution rules.

Reasoning on requirements evolution. Fig. 7 shows the evolution analysis on the requirements model of the scenario, in which the evolution metrics for each design alternative are calculated. The analysis result is shown in two tabs. The first tab reports possible alternatives derived from the model and their corresponding evolution metrics. The second tab displays the DAT which is an internal structure stored at every node in the model to calculate the evolution metrics. Additionally, users can specify their own alternative, and have its evolution metrics calculated.

Any changes in the diagram will be automatically reflected in the analysis result. Since the analysis on requirements evolution is incremental, only changed nodes in the model are recalculated. This improves the overall performance of the tool.

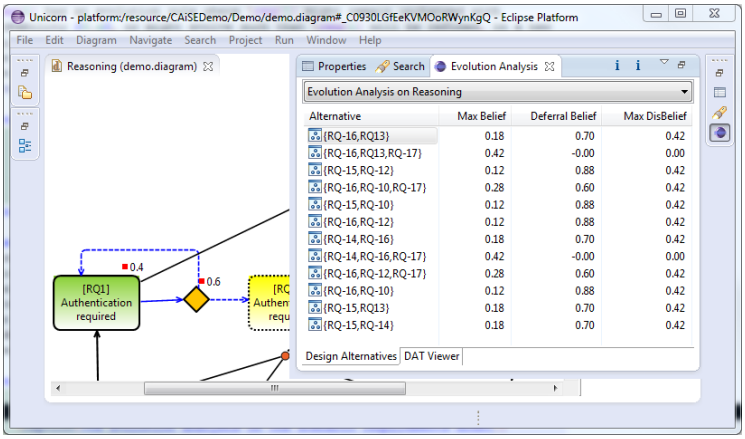

Fig. 7. The evolution analysis on the requirements model of the scenario.

5 Conclusion

We have presented UNICORN, a tool for modeling and reasoning on requirements evolution. By modeling support, UNICORN provides several customizable graphical constructs to model the requirements evolution. By reasoning support, UNICORN provides an environment where the graphical notation could be transformed to a data structure facilitating the analysis. UNICORN demonstrates this by implementing an analysis for requirements evolution.

As a part of future work, we will develop some plug-ins that allow our tool to read requirements models drawn by other tools (for example Si*[4] models). These models could then be referenced in the evolution rules.

References

1. F. A. Administration. System Wide Information Management (SWIM). Segment 2 Technical Overview. Technical report, October 2009.
2. Federal Aviation Administration. System Wide Information Management (SWIM) segment 2 technical review. Technical report, FAA, 2009.
3. P. Loucopoulos and E. V. Kavakli. Enterprise knowledge management and conceptual modelling. In *ER'99*, 1999.
4. F. Massacci, D. Nagaraj, F. Paci, L. M. S. Tran, and A. Tedeschi. Assessing a requirements evolution approach: Empirical studies in the air traffic management domain. In *EmpiRE'12*, 2012.
5. L. M. S. Tran and F. Massacci. Dealing with known unknowns: Towards a game-theoretic foundation for software requirement evolution. In *CAiSE'11*, 2011.

[4] http://www.sistar.disi.unitn.it