University of Trento
Italy

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

Le Minh Sang Tran and Fabio Massacci, University of Trento, Italy. *Email: {leminhsang.tran, fabio.massacci}@unitn.it*

SEVENTH FRAMEWORK PROGRAMME

NESSoS

Requirements Evolution in Software Systems

Long-life software systems need to evolve to satisfy continuous changes in their working environment. Intuitively, dealing with such changes in the early phases of the development is much cheaper and simpler than modifying deployed systems. These changes could be forecasted at some levels of (un)certainty.

We propose an approach to support designers to incorporate potential evolution of requirements and its uncertainty into requirements models. We provide analysis on the evolution uncertainty to facilitate the selection of an optimal design alternative which behaves when evolution happens.

Rule-based Approach for Requirements Evolution

Requirements changes are captured in terms of observable rules. An observable rule is a set of triplets: a model 'as-is' (RM), a model 'to-be' (RM_i) where changes occur, and a likelihood (p_i) that RM evolves to RM_i.

$$r_o(RM) = \left\{ RM \xrightarrow{p_i} RM_i \middle| \sum_i^n p_i = 1 \right\}$$

Different design alternatives of models ('as-is'/'to-be') are captured in terms of controllable rules. A controllable rule is a set of tuples: a model (RM) and a design alternative (DA).

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

Reasoning on Requirements Evolution

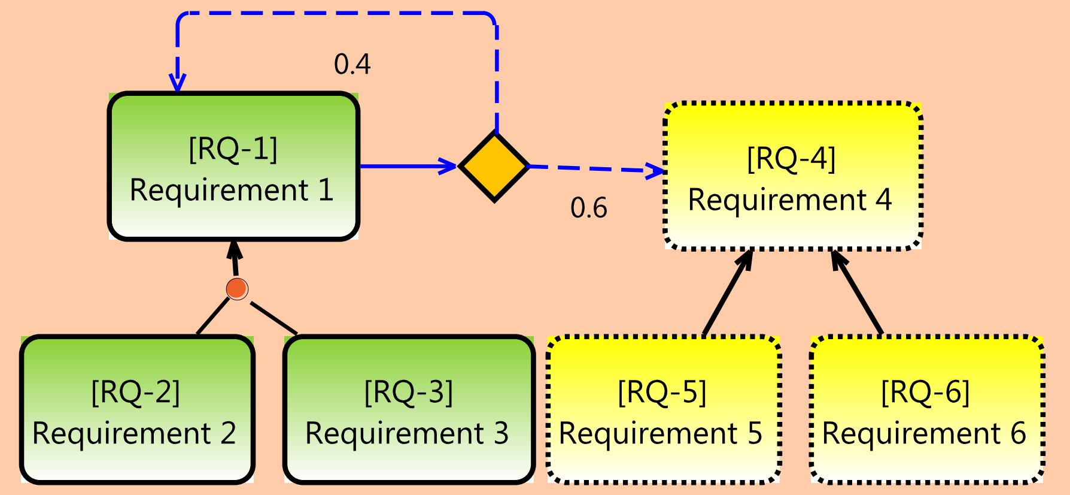
We introduce different evolution metrics to support the selection of an optimal design alternative.

- Max Belief is the maximum belief that an alternative is still usable after evolution.
- Deferral Belief is the complement of total belief that an alternative is still usable after evolution.
- Max Disbelief is the maximum belief that an alternative is useless after evolution.

The Major Features of UNICORN

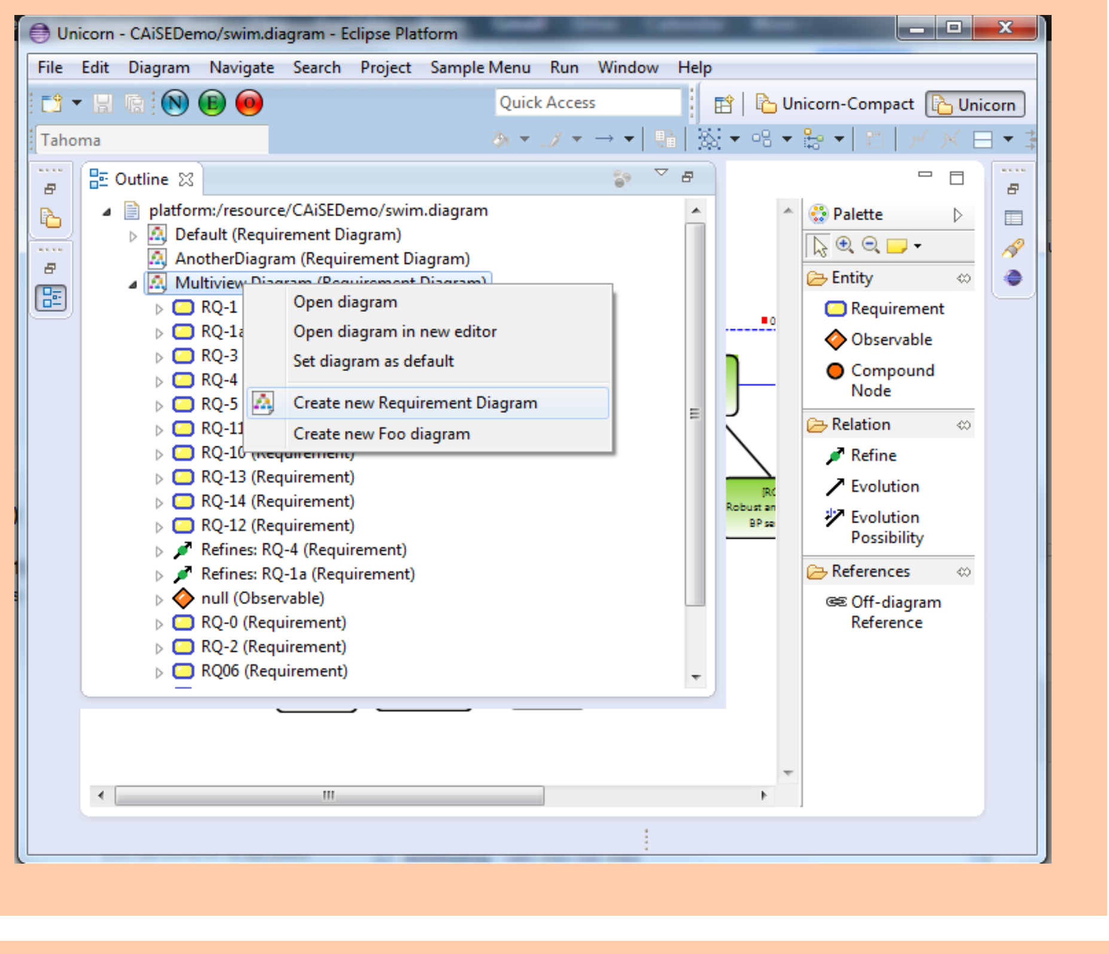
Evolution Modeling Support

- Each requirement is represented by a **requirement node** (round rectangle). A requirement model consists of a number of requirement nodes
- An **observable node** (diamond) corresponds to an observable rule.
- The model 'as-is' connects to the observable node by an **evolution relation** (solid line).
- The model 'to-be' connects to the observable node by an **evolution possibility** relation (dash line).

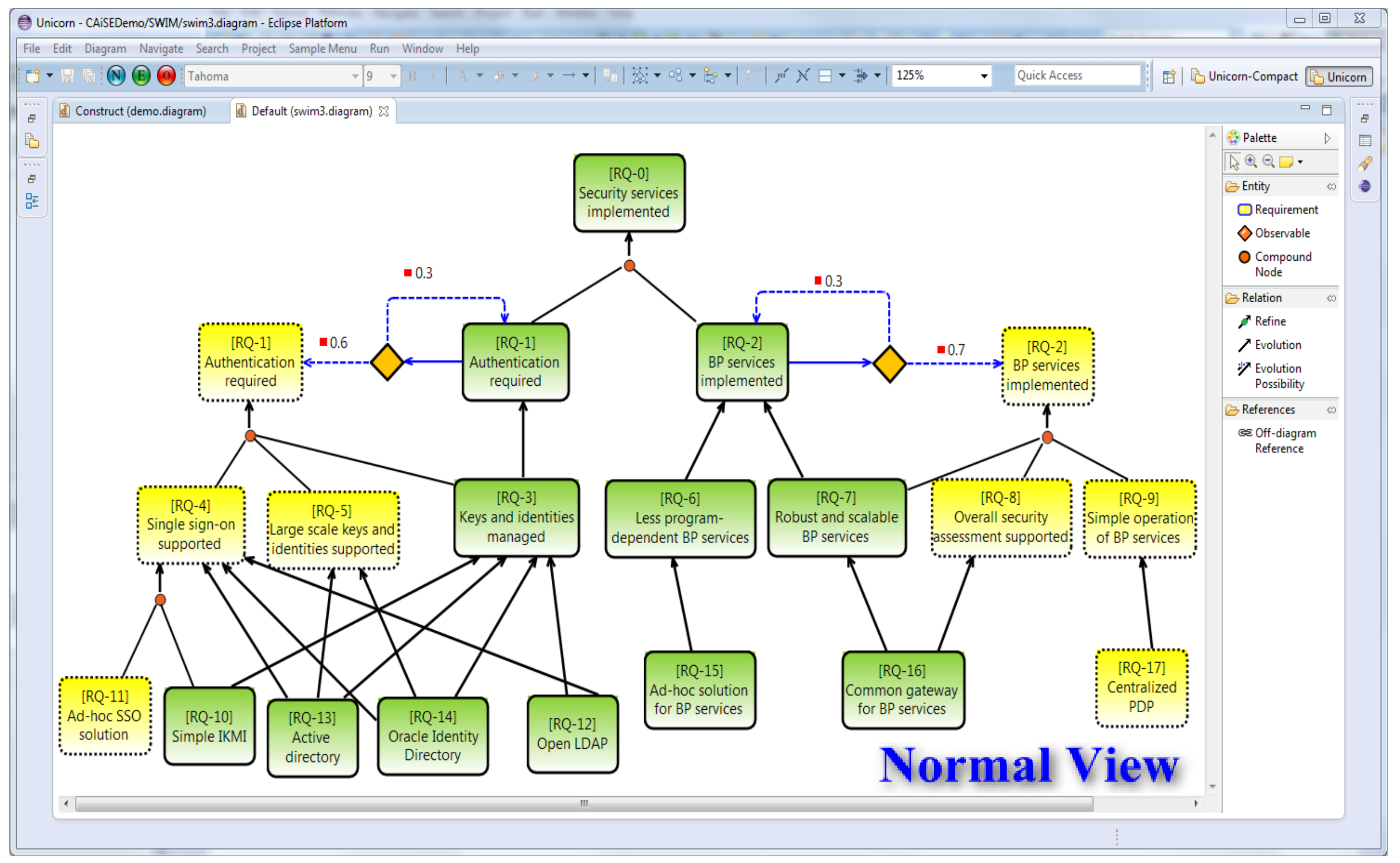


Large-Model Support

- Multiple diagrams are supported in a single file.
- Each diagram can refer to elements in other diagrams by using **Off-Diagram Reference** construct.



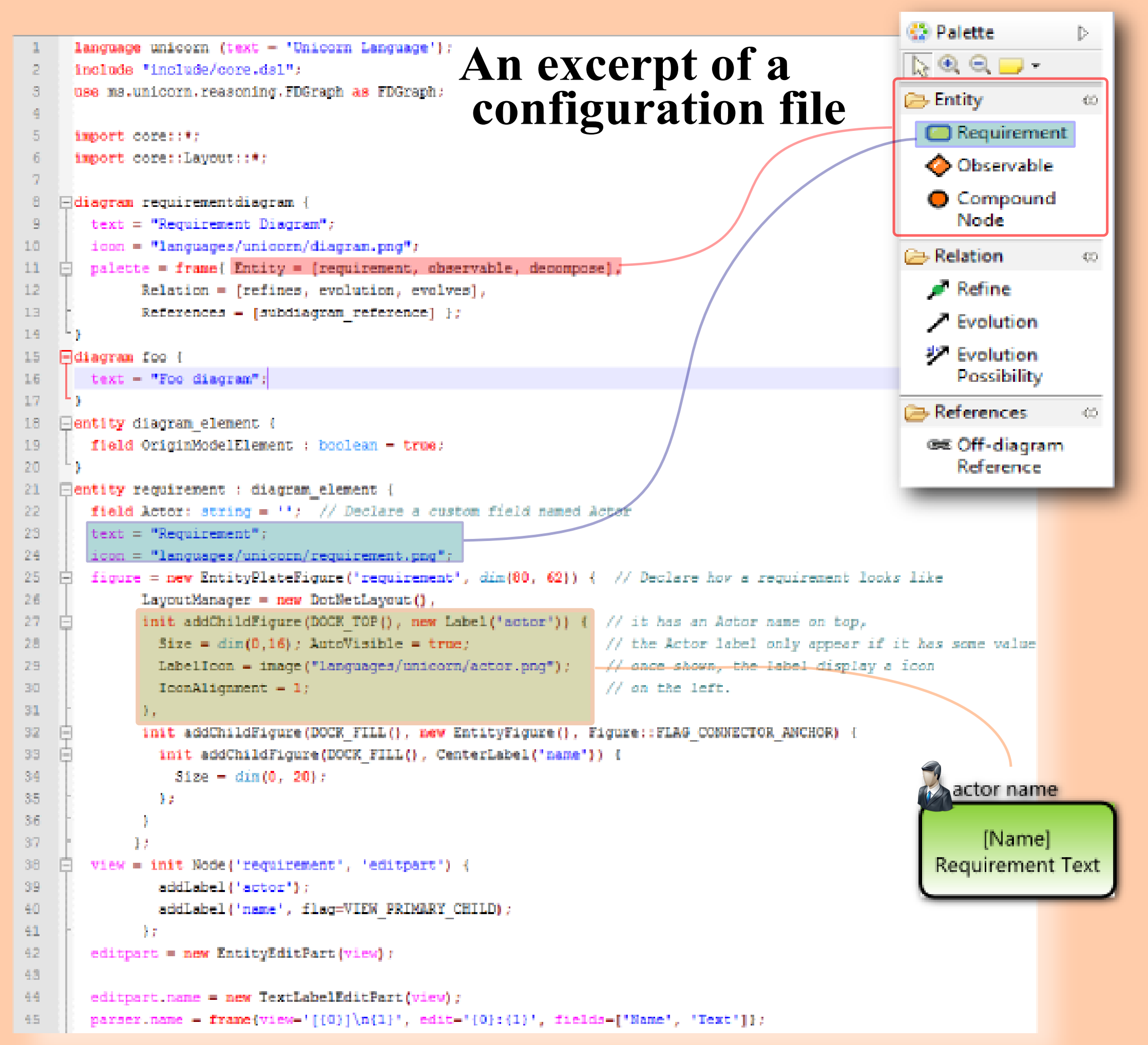
Unicorn - CAiSEDemo/swim.diagram - Eclipse Platform
Outline
platform/resource/CAiSEDemo/swim.diagram
Default (Requirement Diagram)
AnotherDiagram (Requirement Diagram)
Open diagram
Open diagram in new editor
Set diagram as default
Create new Requirement Diagram
Create new Foo diagram
Palette
Entity
Requirement
Observable
Compound Node
Relation
Refine
Evolution
Evolution Possibility
References
Off-diagram Reference



Unicorn - CAiSEDemo/SWIM/swim3.diagram - Eclipse Platform
[RQ-0] Security services implemented
[RQ-1] Authentication required
[RQ-1] Authentication required
[RQ-2] BP services implemented
[RQ-2] BP services implemented
[RQ-4] Single sign-on supported
[RQ-5] Large scale keys and identities supported
[RQ-3] Keys and identities managed
[RQ-6] Less program-dependent BP services
[RQ-7] Robust and scalable BP services
[RQ-8] Overall security assessment supported
[RQ-9] Simple operation of BP services
[RQ-11] Ad-hoc SSO solution
[RQ-10] Simple IKMI
[RQ-13] Active directory
[RQ-14] Oracle Identity Directory
[RQ-12] Open LDAP
[RQ-15] Ad-hoc solution for BP services
[RQ-16] Common gateway for BP services
[RQ-17] Centralized PDP
Normal View

Highly Customizable Constructs

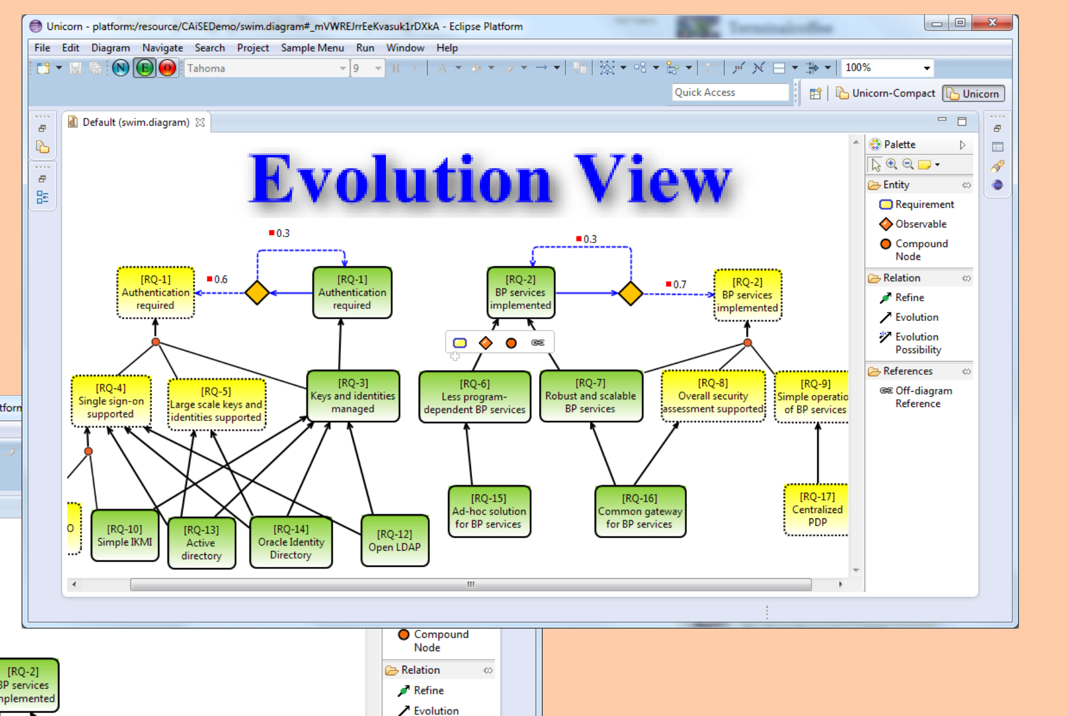
- Every graphical construct is customizable via a configuration file which is separated from the source code.
- A graphical construct can be inherited from another construct.

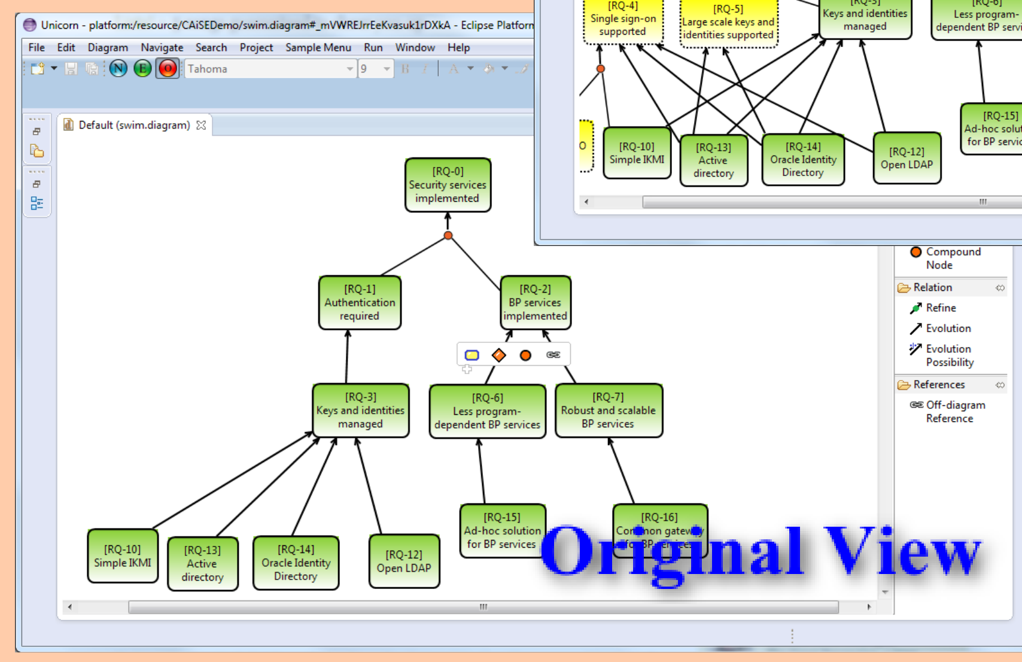

An excerpt of a configuration file

```
language unicorn (text = 'Unicorn Language');
include "include/core.dsl";
use ms.unicorn.reasoning.FDGraph as FDGraph;

import core::*;
import core::Layout::*;

diagram requirementdiagram {
  text = "Requirement Diagram";
  icon = "languages/unicorn/diagram.png";
  palette = frame( Entity = [requirement, observable, decompose],
        Relation = [refines, evolution, evolves],
        References = [subdiagram_reference] );
}
diagram foo {
  text = "Foo diagram";
}
entity diagram_element {
  field OriginModelElement : boolean = true;
}
entity requirement : diagram_element {
  field Actor: string = ''; // Declare a custom field named Actor
  text = "Requirement";
  icon = "languages/unicorn/requirement.png";
  figure = new EntityPlateFigure('requirement', dim(80, 62)) { // Declare how a requirement looks like
        LayoutManager = new DotNetLayout(),
        init addChildFigure(DOCK_TOP(), new Label('actor')) { // it has an Actor name on top,
          Size = dim(0,16); AutoVisible = true;               // the Actor label only appear if it has some value
          LabelIcon = image("languages/unicorn/actor.png");  // once shown, the label display a icon
          IconAlignment = 1;                                  // on the left.
        },
        init addChildFigure(DOCK_FILL(), new EntityFigure(), Figure::FLAG_CONNECTOR_ANCHOR) {
          init addChildFigure(DOCK_FILL(), CenterLabel('name')) {
           Size = dim(0, 20);
          };
        }
      };
  view = init Node('requirement', 'editpart') {
          addLabel('actor');
          addLabel('name', flag=VIEW_PRIMARY_CHILD);
        };
  editpart = new EntityEditPart(view);

  editpart.name = new TextLabelEditPart(view);
  parser.name = frame(view='[{0}]\n{1}', edit='{0}:{1}', fields=['Name', 'Text']);
```

Different Views Support

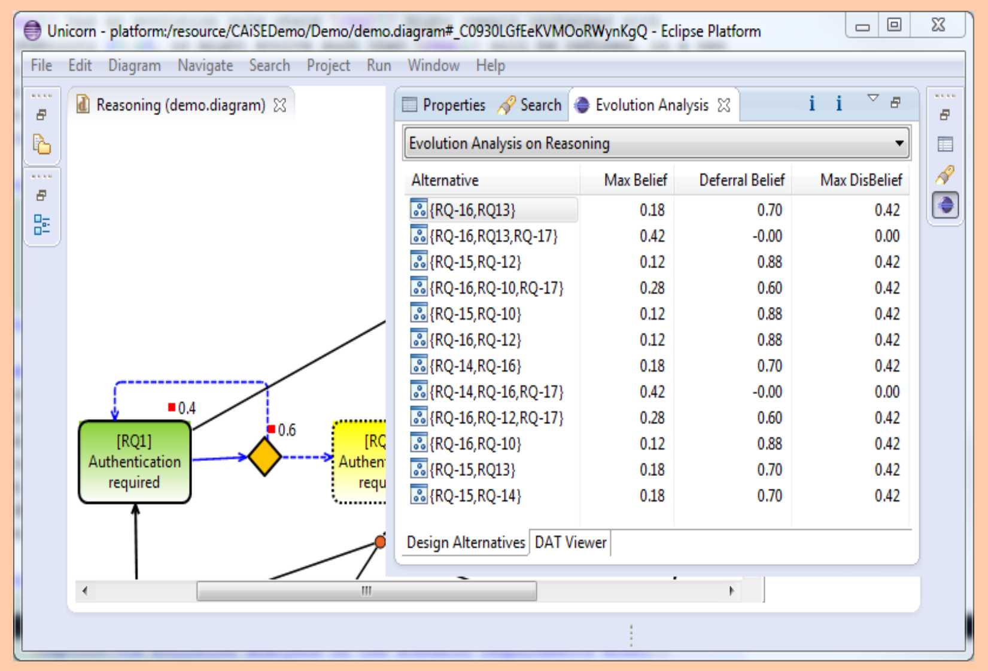
- **Normal view**: shows the complete requirements with evolution rules.
- **Evolution view**: shows only elements involved to evolution.
- **Original view**: shows only elements that appear in the model 'as-is'.





Reasoning Support

- Enumerate all possible design alternatives.
- Calculate evolution metrics for each alternative.


Demo/Demo/demo.diagram#_C0930LGfEeKVMOoRWynKgQ - Eclipse Platform

Properties | Search | Evolution Analysis

Evolution Analysis on Reasoning

Alternative	Max Belief	Deferral Belief	Max DisBelief
{RQ-16,RQ13}	0.18	0.70	0.42
{RQ-16,RQ13,RQ-17}	0.42	-0.00	0.00
{RQ-15,RQ-12}	0.12	0.88	0.42
{RQ-16,RQ-10,RQ-17}	0.28	0.60	0.42
{RQ-15,RQ-10}	0.12	0.88	0.42
{RQ-16,RQ-12}	0.12	0.88	0.42
{RQ-14,RQ-16}	0.18	0.70	0.42
{RQ-14,RQ-16,RQ-17}	0.42	-0.00	0.00
{RQ-16,RQ-12,RQ-17}	0.28	0.60	0.42
{RQ-16,RQ-10}	0.12	0.88	0.42
{RQ-15,RQ13}	0.18	0.70	0.42
{RQ-15,RQ-14}	0.18	0.70	0.42

Design Alternatives | DAT Viewer

Acknowledgement

This work is supported by the European Commission under the project EU-FP7-NoE-NESSoS.

References

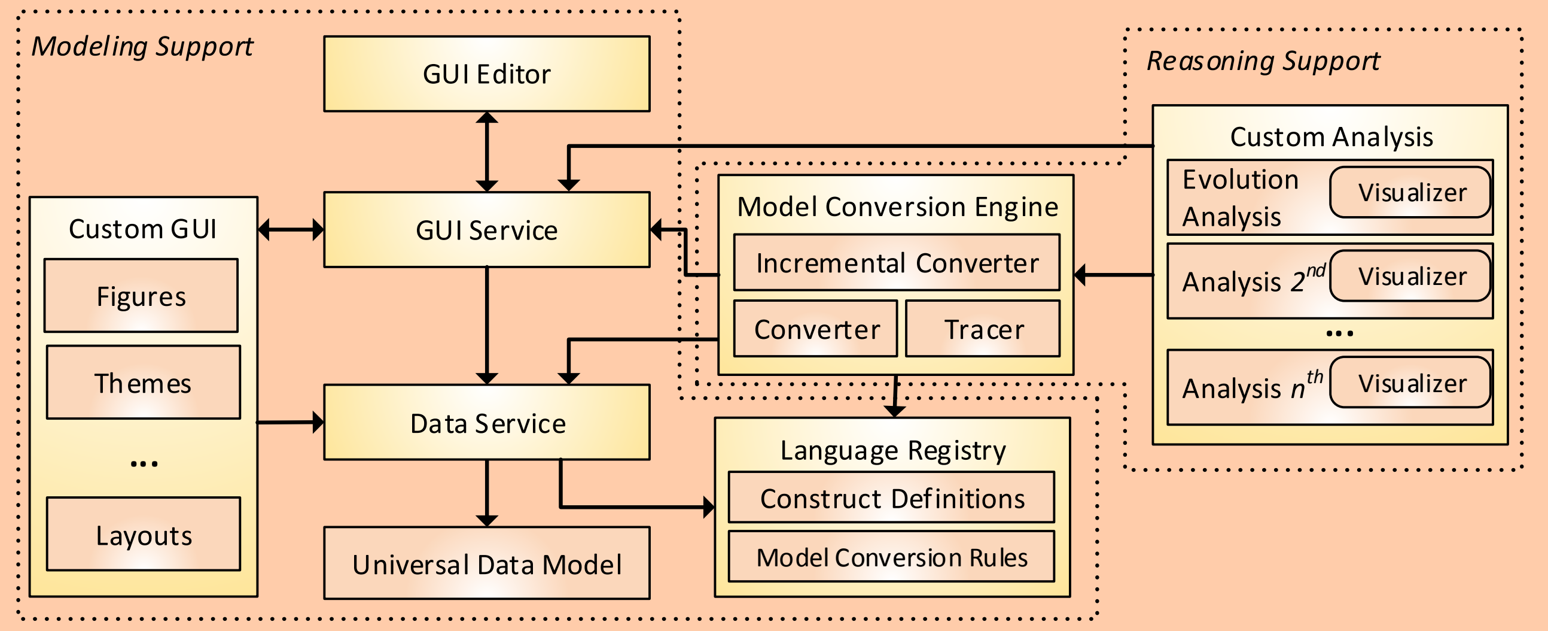
1. F. A. Administration. *System Wide Information Management (SWIM)*. Segment 2 Technical Overview. Technical report, October 2009.

2. 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.

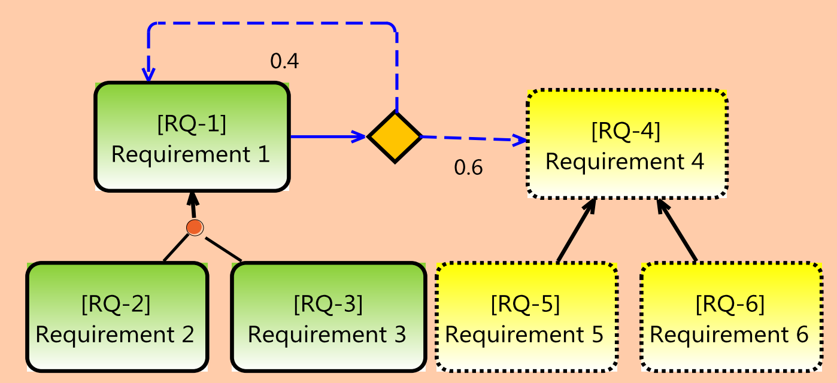
3. L.M.S. Tran and F. Massacci. *Dealing with Known Unknowns: Towards a Game-theoretic Foundation for Software Requirement Evolution*. In CAiSE'11, 2011.

Under the Hood

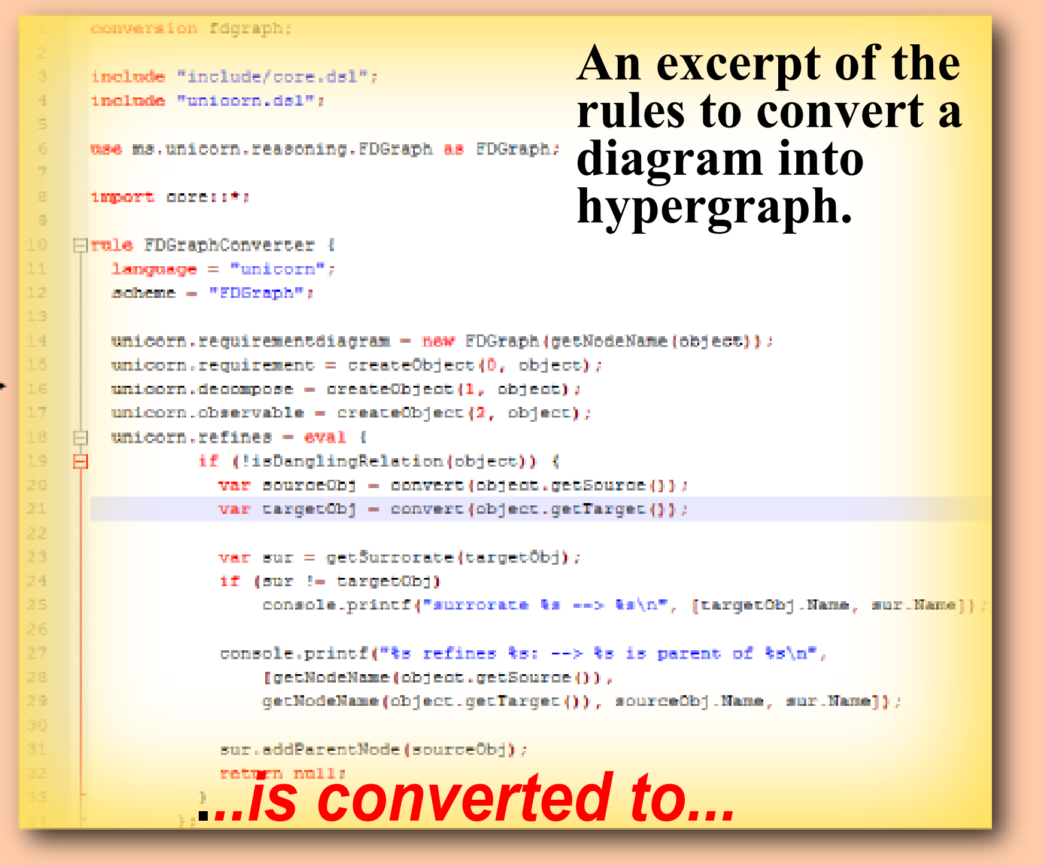
Tool Architecture



The Diagram Conversion Mechanism for Reasoning Support

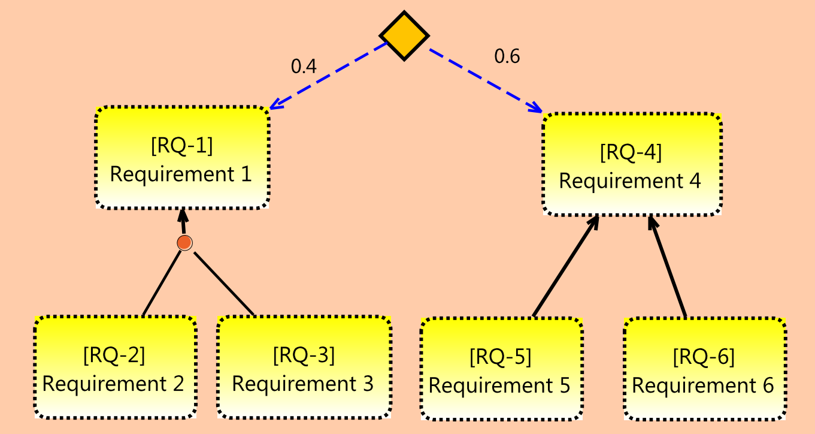


A user-friendly diagram...


An excerpt of the rules to convert a diagram into hypergraph.

```
conversion fdgraph;

include "include/core.dsl";
include "unicorn.dsl";

use ms.unicorn.reasoning.FDGraph as FDGraph;

import core::*;

rule FDGraphConverter {
  language = "unicorn";
  scheme = "FDGraph";

  unicorn.requirementdiagram = new FDGraph(getNodeName(object));
  unicorn.requirement = createObject(0, object);
  unicorn.decompose = createObject(1, object);
  unicorn.observable = createObject(2, object);
  unicorn.refines = eval {
          if (!isDanglingRelation(object)) {
            var sourceObj = convert(object.getSource());
            var targetObj = convert(object.getTarget());

            var sur = getSurrorate(targetObj);
            if (sur != targetObj)
                console.printf("surrorate %s --> %s\n", [targetObj.Name, sur.Name]);

            console.printf("%s refines %s: --> %s is parent of %s\n",
                [getNodeName(object.getSource()),
                getNodeName(object.getTarget()), sourceObj.Name, sur.Name]);

            sur.addParentNode(sourceObj);
            return null;
          }
  };
```

...is converted to...



....a hypergraph structure.