A Textual Syntax with Tool Support for the Goal-oriented Requirement Language

Vahdat Abdelzad, Daniel Amyot, Sanaa A. Alwidian, Timothy C. Lethbridge

EECS, University of Ottawa, Ottawa, Canada
{v.abdelzad, damyot, salwidia, Timothy.Lethbridge}@uottawa.ca

Abstract. Most goal-oriented modeling languages, including *i**, Tropos, KAOS and the Goal-oriented Requirement Language (GRL), offer a graphical syntax, sometimes accompanied by a textual interchange format (e.g., in XML). Graphical representations of goal models excel at supporting discussions and at visualizing analysis results. However, creating/modifying goal models is often a tedious task with current graphical environments. Textual languages are often more efficient for creating/ modifying models, in particular large ones. This paper proposes a programming-like textual syntax for GRL supported by an advanced editor for the Eclipse platform. Such syntax and editor enable modelers to create GRL models with complex features (e.g., strategies and contribution overrides) in a way that is simpler than with the most popular GRL editor, namely jUCMNav. The paper also introduces a converter from the GRL textual syntax to jUCMNav, so that models can be visualized and analyzed.

Keywords: Editor · Goal-oriented Requirement Language · Textual Language.

1 Introduction

Graphical modeling languages bring benefits over textual languages in that they can represent information in two dimensions (rather than linearly) using intuitive pictograms and other visual clues. Unsurprisingly, goal modeling languages such as *i**, Tropos, Techne, KAOS and the Goal-oriented Requirement Language (GRL) have chosen graphical syntaxes for supporting visual modeling and analysis activities [5]. However, there are still two important issues in that context: 1) it is difficult to design a graphical modeling language that offers good cognitive fitness for different types of users and purposes, and most goal modeling languages have much room for improvement in that regard [8]; and 2) graphical editors are often cumbersome to use and inefficient for *creating* goal models [9]. Textual syntaxes, although not very useful for visualizing analysis results, are often less cognitively challenging than graphical syntaxes, and are often faster to use for creating/modifying models via intelligent editors and simpler copy/pasting semantics. This is something we have observed with Umple, a textual language that integrates concepts from UML class/state diagrams and patterns with programming languages such as Java [3].

In this context, we have decided to explore the design of a textual syntax for GRL, which is part of the User Requirements Notation (URN) standard [6]. GRL modeling, analysis and transformations are currently supported by jUCMNav [10]. The designed language, called TGRL, supports full GRL, including basic concepts such as intentional elements, links and actors, but also advanced features such as indicators, metadata, strategies, and contribution overrides.

Section 2 provides an overview of our textual syntax, based on standard GRL and jUCMNav's metamodel. The Eclipse-based editor and the converter that transforms TGRL models into jUCMNav ones are introduced in Section 3. Section 4 discusses early experience with the language and its editor, and provides pointers to future work items. Please note that the full grammar and the editor are available online [1].

2 TGRL: A Textual Syntax for GRL

To define the TGRL textual syntax, we used guiding principles inspired from the design of Umple, including simplicity, consistency, and a programming language-like look and feel. In addition, we aligned the syntax and especially keywords with jUCMNav's metamodel (which served as inspiration for GRL in the standard URN metamodel [6], except for several exploratory features) in order to simplify the conversion from/to GRL models in jUCMNav. In addition:

- GRL elements are usually defined through keywords using CamelCase boundaries (e.g., a softgoal intentional element is represented by a `softGoal`).
- String values are surrounded by quotation marks.
- Model element properties and sub-elements (if any) are set inside curly brackets.

```
grl SimpleExample {
    comment "This is a simple TGRL illustrative model"; // Model comment

    actor User {
      // Default name is the ID name, "User" in this case.
      // Goal with specific name and quantitative importance.
      softGoal EasyToUse {name = "Have a system that is easy to use";
                          importance = 100;}
      indicator LowLearningTime; // Indicator definition
    }
    actor System {
      // Goal with qualitative importance, and OR decomposition type
      goal ProvideMainFunctionality {importance = high;
                                     decompositionType = or;}
      task FirstOption {metadata stereotype="SomeValue";}
      task SecondOption {description = "Better alternative";}

      ProvideMainFunctionality decomposedBy FirstOption, SecondOption;
      FirstOption contributesTo User.EasyToUse {hurt}; //Inside element
    }

   // Links defined outside its elements, with quantitative value
   System.SecondOption contributesTo User.EasyToUse {name=C1;50};
   User.LowLearningTime contributesTo User.EasyToUse {name=C2;40};
}
```

Fig. 1. Simple illustrative TGRL model

- Every definition ends with a semicolon except when a pair of curly brackets is utilized to include either sub-elements or properties.

Most elements have a textual identifier (ID) as well as optional *metadata* (name-value pairs). Intentional elements (goals, softgoals, tasks and resources) also have qualitative/quantitative importance values (to their containing actor). For example, Fig. 1 shows the TGRL representation of a simple GRL model with two actors, their intentional elements, and various links. IDs are used as names unless specified otherwise (e.g., a `name` attribute can be quite long, with special symbols). Qualitative values (e.g., `high` for importance, `make`/`hurt` for contributions) and quantitative values (between –100 and 100) can be used interchangeably. Lists can be used for definitions and usages (e.g., see the `decomposedBy` relationship in the example).

As in Umple [3], links can be specified *inside* one element or *outside* the relevant elements, depending on the modeler's preference. In Fig. 1, one contribution is defined inside the `System` actor, whereas two other contributions are defined outside both actors. Note that scoping is also used to resolve potential naming issues. For example, in the contribution inside the `System` actor, task `FirstOption` is local but softgoal `EasyToUse` is defined elsewhere, and hence must be prefixed by its containing actor (leading to `User.EasyToUse`). Dependency links are handled similarly.

TGRL also supports evaluation *strategies* (initial values given to some intentional elements before invoking a propagation algorithm for model analysis) and handles advanced constructs such as strategy inclusion (for reuse), indicator initialization, and value ranges. For example, Fig. 2 adds a group of three strategies to the model in Fig. 1. The first one selects the first task option in the system, and sets the parameters of the `User.LowLearningTime` indicator. During analysis, an indicator converts a strategy's `eval` value to a satisfaction value by comparing it to the `target`, `threshold`, and `worst` values [6]. The second strategy extends the first one (and hence includes its initializations) but overrides existing initializations or adds new ones. The third strategy refines the first one through a *range* of values (in this example: 10, 15, 20, 25, 30, 35, 40), leading to ranges of results for all intentional elements and actors after evaluating the strategy against the model. In jUCMNav, the creation and management of GRL strategies is rather complicated and not user-friendly, especially if indicators and ranges are involved. This is handled in a much simpler and explicit way in TGRL.

```
strategy SelectFirst {
   System.FirstOption = satisfied;
   User.LowLearningTime = {unit="minutes"; target=30.0; threshold=60.0;
                           worst=120.0; eval=90.0;}
}
strategy SelectSecond extends SelectFirst { // Strategy inclusion
   System.FirstOption = none; // Overridden
   System.SecondOption = 100; // Added, quantitatively this time
}
strategy RangeExample extends SelectFirst {
   System.FirstOption = {start = 10; end = 40; step = 5;}
}
strategyGroup MyGroup includes SelectFirst, SelectSecond, RangeExample;
```

Fig. 2. Sample GRL strategy definitions in TGRL

TGRL also supports advanced model modifications such as *contribution overrides*, which can be applied to a GRL model to change the weights of existing contribution links, prior to the evaluation of strategies [6]. Overrides are useful when stakeholders disagree on the weights of contributions; different options can be evaluated while having only one model to manage [9]. As shown in Fig. 3, contribution overrides can be grouped and their new values can be specified via a name given to the modified contribution (in Fig. 3, one contribution was named `C1` and another one `C2`). The syntax is quite similar to those of TGRL strategies; as illustrated by `SecondOverride`, contribution overrides can `extend` others, possibly with ranges of values.

The last example is shown at the bottom of Fig. 3, and is concerned with URN links, which are user-specified typed links that can connect any pair of elements in a URN model [6]. In this example, a new type of link (`independentFrom`) is defined and then used to connect two actors in the model. URN links and metadata are constructs that are useful in extending or *profiling* URN to specific domains.

```
// Contribution overrides
contributionGroup SomeOverrides includes FirstOverride, SecondOverride;
contribution FirstOverride {
    C1 = 30;
    C2 = make;
}
contribution SecondOverride extends FirstOverride {
    C1 = {start = -40; end = 0; step = 10;}
}

// URN links
link independentFrom; // Link type definition
User independentFrom System; // Link instance between two actors
```

Fig. 3. Sample GRL contribution overrides and URN links

3 TGRL Editor

We developed a TGRL editor for the Eclipse environment. We specified our grammar with *Xtext* [11], often used for the development of textual domain-specific languages. The TGRL editor supports syntax highlight (as shown in the code snippets from the previous figures), an outline view, annotation of syntactic errors, content assistance, and code formatting. Fig. 4 gives an overview of the editor.

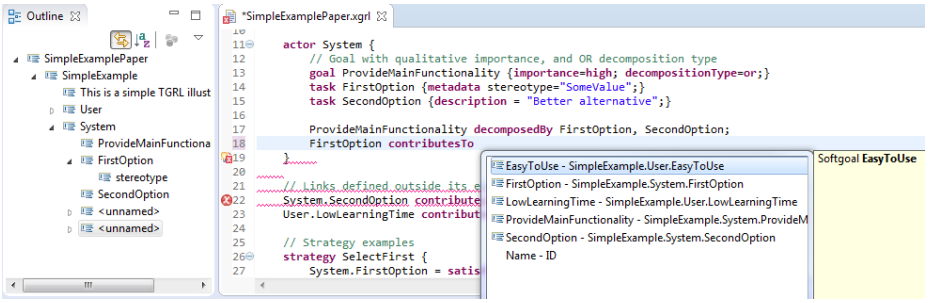

Fig. 4. Overview of the TGRL editor, with content assistance

The modeler, using Control-Space, can invoke code completion at any moment. Not only is this available for the keywords found in the grammar, this is also available for references to existing elements. For example, in Fig. 4, several suggestions are provided as potential targets of an incomplete contribution link.

Together with the editor, we provided a mechanism (*converter*) that enables the transformation of a textual model to a graphical model in jUCMNav, ready to be analyzed and visualized. The transformation was developed using *Acceleo* [2], which is a pragmatic model-to-text transformation language. We have chosen to use such transformation rather than a model-to-model one because jUCMNav stores its models in textual (XML) files. Further validation of the models, based on GRL's semantics, is performed through rules deployed in the body of the converter. This transformation does not handle the layout of diagrams, but jUCMNav has several features for creating views of a model and for automatically laying out elements. For example, Fig. 5 shows the GRL model corresponding to the ongoing example, as imported by jUCMNav (with automatic and some manual layout). The evaluation of the strategy `SelectFirst` is also shown, using quantitative values.

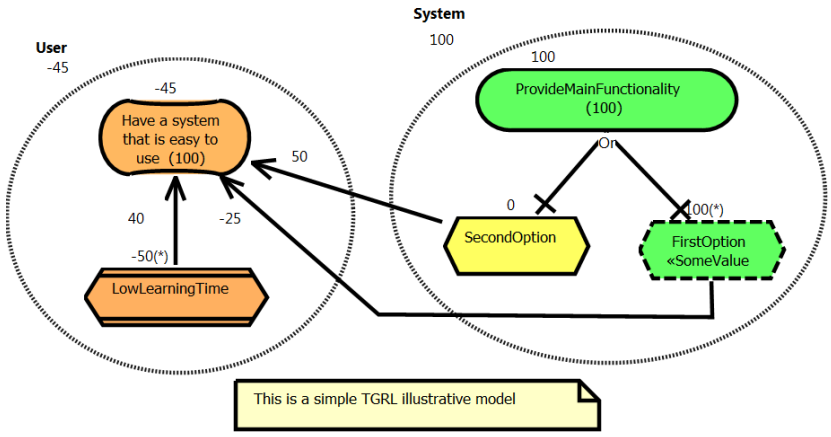

Fig. 5. Sample GRL model imported in jUCMNav, with a strategy evaluated quantitatively

4 Discussion and Future Work

In this paper, we illustrated a new textual syntax for GRL, called TGRL, with a full coverage of the language. TGRL is supported by a feature-rich Eclipse-based editor, supplemented by an automated conversion to GRL models readable by jUCMNav. This work contributes a simple and practical way of creating/modifying GRL models.

The idea of having a textual syntax for GRL is not new. In the first draft of the GRL language (from 2001), Liu and Yu provided a textual grammar and an XML-based interchange format [7]. TGRL is however not based on this earlier attempt. Rather, it focuses on adding a textual syntax to an already existing metamodel definition. TGRL also covers many concepts that did not exist in [7], such as indicators, strategies, contribution overrides, metadata and URN links. Formal Tropos also has a textual syntax [4] but its goal modeling syntax (outer layer) is more declarative, ver-

bose, and limited in scope. Formal Tropos however supports an inner layer for declaring constraints on attributes and supports temporal logic properties. Tool support for Formal Tropos (T-Tool) does not include a feature-rich editor. To our knowledge, TGRL is the first tool-supported textual syntax for an *i**-like modeling language.

In terms of coverage and usability, one of us (S.A. Alwidian, not involved in the design and implementation of the grammar and tool, to avoid bias) validated the language and the tool through two simple examples from the GRL literature. Her feedback was overall positive, and solutions to some issues raised with earlier versions were incorporated in the grammar and the tool to improve their usability.

This new technology opens the door to many future opportunities. The language and the tool obviously require further and more rigorous validation, for example based on how well existing models are supported. However, they also enable comparisons with graphical tools (e.g., jUCMNav) in terms of efficiency and usability for model creation and manipulation tasks. One important feature currently missing is the availability of a transformation from jUCMNav to TGRL, which would enable modelers to go back and forth between the two representations. The editor could also be improved by the inclusion of additional static semantic rules to ensure the correctness of the GRL models created (e.g., to prevent cyclical contribution links or the mixed use of quantitative/qualitative values, or to detect bad smells and anti-patterns). We also envision opportunities to combine TGRL (for goals) with Umple (for design and implementation) as they provide complementary concepts.

References

1. Abdelzad, V.: Textual Modeling Language for GRL (2015) https://github.com/vahdat-ab/TGRL
2. Acceleo (2015) http://www.eclipse.org/acceleo/
3. Forward, A., Badreddin, O., Lethbridge, T.C., Solano, J.: Model-driven rapid prototyping with Umple. Softw., Pract. Exper. 42(7), 781–797 (2012) http://umple.org/
4. Fuxman, A. et al.: Specifying and analyzing early requirements in Tropos. Requir. Eng. 9, 2, 132–150 (2004).
5. Horkoff, J., Yu, E.S.K.: Comparison and evaluation of goal-oriented satisfaction analysis techniques. Requir. Eng. 18(3), 199–222 (2013)
6. ITU-T: Recommendation Z.151 (10/12): User Requirements Notation (URN) – Language definition. Geneva, Switzerland (2012)
7. Liu, L., Yu, E.: GRL – Goal-oriented Requirement Language. University of Toronto, Canada (2001) http://www.cs.toronto.edu/km/GRL
8. Moody, D.L., Heymans, P., Matulevičius, R.: Visual syntax does matter: Improving the cognitive effectiveness of the *i** visual notation. Requir. Eng. 15(2), 141–175 (2010)
9. Mussbacher, G., Amyot, D., Heymans, P.: Eight Deadly Sins of GRL. 5th International *i** Workshop (iStar 2011). CEUR-WS, Vol-766, 2–7 (2011)
10. Roy, J.-F. Kealey, Amyot, D.: Towards Integrated Tool Support for the User Requirements Notation. SAM 2006. LNCS 4320, 198–215. Springer (2006) http://softwareengineering.ca/jucmnav
11. Xtext (2015) http://www.eclipse.org/Xtext/