Adding a Textual Syntax to an Existing Graphical Modeling Language: Experience Report with GRL

Vahdat Abdelzad, Daniel Amyot(✉), and Timothy C. Lethbridge

School of Electrical Engineering and Computer Science,
University of Ottawa, Ottawa, Canada
{v.abdelzad,damyot,Timothy.Lethbridge}@uottawa.ca

Abstract. A modelling language usually has an abstract syntax (e.g., expressed with a metamodel) separate from its concrete syntax. The question explored in this paper is: how easy is it to add a textual concrete syntax to an existing language that offers only a concrete graphical syntax? To answer this question, this paper reports on lessons learned during the creation of a textual syntax (supported by an editor and transformation tool) for the Goal-oriented Requirement Language (GRL), which is part of the User Requirements Notation standard. Our experiment shows that although current technologies help create textual modelling languages efficiently with feature-rich editors, there are important conflicts between the reuse of existing metamodels and the usability of the resulting textual syntax that require attention.

Keywords: Goal-oriented Requirement Language · Graphical modeling language · jUCMNav · Metamodel · Textual syntax · Xtext

1 Introduction

A model is an abstraction of the reality that helps engineers and other users focus on specific aspects of a problem or a system in order to support communication, understanding, analysis, and decision making. Modeling languages often have a graphical and/or a textual representation, called *concrete syntax*. The concepts of a modeling language are often captured with an *abstract syntax*, for example in the form of a grammar or a metamodel [14]. Concrete syntaxes bring understandability, usability, and often visualization to the concepts defined at the abstract level.

Graphical and textual syntaxes both have strengths and limitations [25]. With diagrams, it is often easier to understand non-linear relationships (such as graphs) and appreciate analysis results than with text. On the other hand, textual models are often easier to create and manipulate (e.g., through intelligent editors or simpler copy-pasting). It is also challenging to find appropriate symbols and metaphors in a graphical language in order to assure a suitable cognitive fit for all users. The cognitive effectiveness of notations has been explored

J. Fischer et al. (Eds.): SDL 2015, LNCS 9369, pp. 159–174, 2015.
DOI: 10.1007/978-3-319-24912-4_12

substantially in the past few years, based on frameworks such as Physics of Notations [21], and illustrated on different graphical languages such as for goal modeling [21] and scenario/process modeling [11].

Ideally, modelers should be given the choice of using among the concrete syntaxes that best suit the tasks they have to perform, for example, a textual syntax to create a model and a graphical syntax to communicate the model and visualize analysis results. Several standardized languages already support textual and graphical syntaxes (e.g., [13,15,17]), but often they have been designed to support both from the beginning. In this paper, we are more interested in exploring the challenges related to the *addition* of a textual concrete syntax to an existing language for which only a concrete graphical syntax already exists. This exploration is done through the design of an actual editor-supported textual syntax for the Goal-oriented Requirement Language (GRL), a requirements-level goal modeling language standardized as part of the User Requirements Notation (URN) [3,16]. One challenge here is that the abstract syntax of URN is based on a metamodel oriented towards the graphical representation of its concepts, without consideration for a potential concrete textual syntax.

Section 2 presents work related to modeling language design, together with background on GRL and existing tool support (jUCMNav [4,28]). Section 3 introduces some of the main challenges we have observed when adding a textual syntax to an existing metamodel-based graphical language, together with elements of solutions. Section 4 presents our case study, where we created a grammar for a *Textual GRL* (TGRL), together with an Eclipse-based rich editor and a transformation mechanism that converts TGRL models to URN models readable by jUCMNav. Not all modeling languages are based on metamodels and not all language editors are using Eclipse, but our experience report does involve metamodels and Eclipse technologies. A short discussion of lessons learned is presented in Sect. 5, followed by conclusions and future work in Sect. 6.

2 Background

This section reviews closely-related work on modeling language design and highlights the background concepts on GRL and jUCMNav required to understand the examples of challenges and solutions discussed later in the paper.

2.1 Related Work on Textual and Graphical Languages

Several languages already support textual and graphical syntaxes. Among the languages standardized by the *International Telecommunication Union - Telecommunication Standardization Sector* (ITU-T), common examples include the Specification and Description Language (SDL) [13], Message Sequence Charts (MSC) [15], and the Testing and Test Control Notation (TTCN-3) [17]. All are supported by many tools, some of which allowing modelers to use both syntaxes interchangeably and transparently. TTCN-3 even offers an additional *tabular* concrete syntax [18]. While SDL uses an abstract grammar as abstract

syntax, TTCN-3 is based on a metamodel. However, these languages have developed their concrete textual syntax, concrete graphical syntax, and abstract syntax more or less at the same time. With URN and GRL, there are already a metamodel and a graphical syntax that have been in place for many years, so adding a textual syntax is more problematic in that context.

In the Unified Modeling Language (UML) world, several textual syntaxes have been proposed for subsets of UML, often as a means to create models and then visualize them. Cabot as collected a list of such languages and tools [6]. We are not really interested in these technologies as they do not allow modelers to create instances of the UML metamodel, which would have enabled analysis and transformations based on standard UML. These tools have their own internal representations.

In a different and more recent approach, we find *Umple*, a textual language that integrates concepts from UML class/state diagrams and patterns with programming languages such as Java [8]. Umple models are written using human-readable text seamlessly integrated with code. Umple models can also be visualized with the UML notation. This *model-is-the-code* approach helps developers maintain and evolve code as the system matures simply by the fact that both model and code are integrated as aspects of the same system [10]. Still, Umple uses its own metamodel, not UML's.

2.2 Related Work on Enabling Technologies

The *Object Management Group* (OMG) has proposed the *UML Human-Usable Textual Notation* (HUTN), a technology for automatically supporting user-readable concrete syntaxes of models and model instances based on the *MetaObject Facility* (MOF) [23]. One interesting feature of HUTN is that the textual syntax does not need to reflect exactly the structure of the metamodel. Parameters can be used to create *short-hands* and make the syntax more readable and usable. For example [23], one can set:

- The use of a class attribute as the class unique identifier for a given scope;
- The representation of a Boolean or enumerated attribute as a keyword;
- The use of default values for mandatory attributes (making them optional);
- The selection of an alternate name for any model element;
- Alternative representations for associations.

Unfortunately, HUTN is supported only a few tools, including the one proposed by Rose et al. [27], which is part of the Eclipse Epsilon project. HUTN was shown to be complicated to use, and resulting editors have limited capabilities.

Eclipse's *Xtext* [32] is one of the enabling technologies used to produce feature-rich editors for a textual language. Xtext usually takes a language grammar as input and the corresponding metamodel is automatically built in the background. It also allows reusing already-existing metamodels, but then there is no flexibility in the design of the language syntax. In other words, importing metamodels constrains the design of language. Changing the grammar changes

the underlying metamodel, which might create some issues with transformations that use such metamodel as a source. In that context, Schmidt et al. [30] proposed a category of refactorings for Xtext that use asymmetric bidirectional model transformations to synchronize the various artifacts of language descriptions, including transformations (based on Xtend [31]).

Other technologies for textual syntax development include EMFText from Heidenreich et al. [12], which generates automatically default syntax from *Eclipse Modeling Framework* (EMF) models, with some possibilities for syntax tailoring before the generation of text editors. Jouault et al. [19] also proposed *Textual Concrete Syntax* (TCS), a generative solution that transforms grammars into editors and tools for model-to-text and text-to-model transformations. Both EMFText and TCS are however far less popular and mature than Xtext, and their development seems to have stopped several years ago.

Finally, as graphical syntaxes often include textual syntaxes for various kinds of expressions, Scheidgen presented techniques to *embed* textual editors into graphical model editors and provided a proof of concept involving Eclipse-based technologies [29]. However, we are more interested here in generating a new textual syntax than in embedding one in a graphical syntax.

2.3 Goal-Oriented Requirement Language (GRL)

The URN standard is composed of two complementary sub-languages: (i) GRL for modeling the intentions of actors and systems, together with their various relationships, and (ii) Use Case Maps (UCM) for modeling causal scenarios and processes superimposed on a structure of components [16]. GRL core concepts include *actors*, *intentional elements* (e.g., goals, softgoals, tasks, resources and beliefs), *links* (decompositions, dependencies, weighted contributions) and *indicators* (Fig. 1). GRL model elements are URN model elements. As such, they can have *metadata* (name-value pairs) and typed *URN links* connecting pair of elements; these concepts are useful to extend and tailor URN to specific domains, in a standard way [3].

Many of the concepts of GRL have a visual representation. In URN, the graphical language metamodel is a pure superset of the abstract syntax metamodel. For example in Fig. 2, an actor reference (ActorRef) is the visual representation of an actor in a GRL graph and hence possesses attributes such as a size, a label, and a position. The actor itself has color-related attributes, which are shared by all its references.

GRL model analysis, whether qualitative (using contribution, satisfaction, and importance values from their respective enumerated types in Fig. 1) or quantitative (using integer values in specific ranges), is done through *strategies*. A strategy provides initial satisfaction values to some of the intentional elements in the GRL model, and an evaluation algorithm propagates this information (through the GRL links) to the other intentional elements and to the actors in order to compute their resulting satisfaction values [3,16]. As it is often difficult to agree on the weights of contribution links in GRL models, the standard also includes *contribution changes* as a mechanism to specify and group (in collection

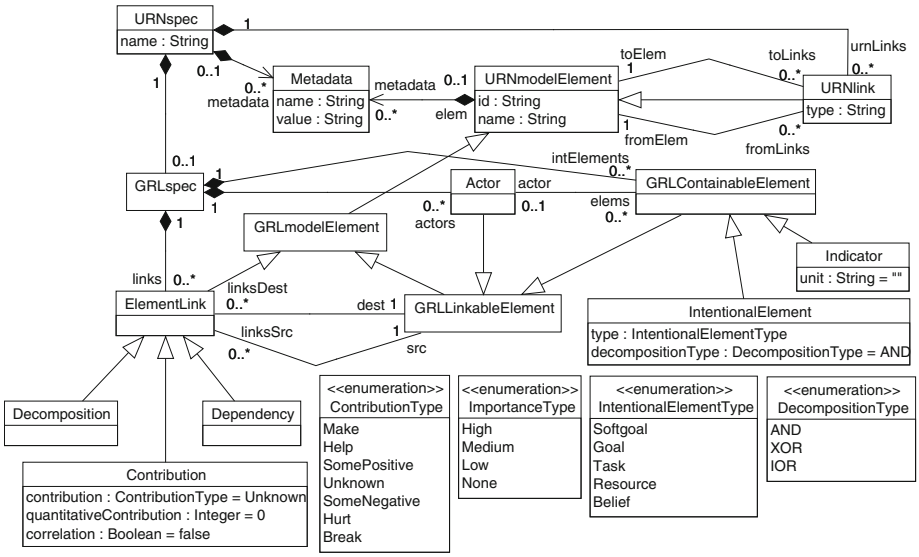

Fig. 1. GRL metamodel: core GRL concepts (adapted from [16])

contexts) a local modification to the weight of a contribution link, which can be applied to a base model before evaluating its strategies. What is important to observe here is that strategies (not shown here) and contribution changes (Fig. 3) do not currently have any concrete syntax, and hence these parts of a model need to be specified through a tool's user interface and tree-structured views, as is currently done in jUCMNav [4].

The absence of a complete graphical syntax and of a textual syntax has already been observed as a "sin" in the design of the GRL language [22]. In addition, the graphical syntax has similar cognitive efficiency weaknesses as those observed for the *i** goal modeling language [21], as GRL's syntax is based in part on the one from *i**.

2.4 jUCMNav Tool

jUCMNav is an open-source Eclipse plugin for URN modeling, analysis, reporting and transformation, developed since 2004. The GRL modeling and analysis part was first provided by Roy et al. [28], and has substantially evolved since then to support new features and newer concepts now found in the standard [16].

Given that jUCMNav was initiated before the first version of URN was standardized in 2008, and given that jUCMNav is also used as a platform for exploring new language concepts that could be integrated into URN in the future (like contribution changes were integrated in the 2012 edition of URN [16]), there are many differences between jUCMNav's metamodel and URN's (see http://bit.ly/1GCbhNa for details). For example, jUCMNav's metamodel uses interface classes (for reusability across its UCM and GRL editors) and packages, there are

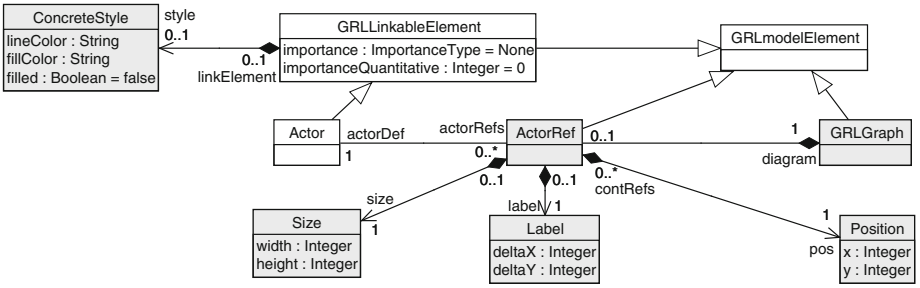

Fig. 2. GRL metamodel: graphical classes for actor references (adapted from [16])

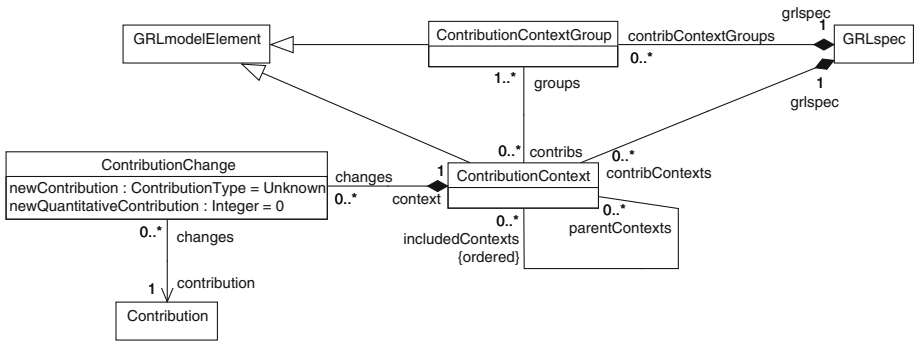

Fig. 3. GRL metamodel: contribution changes, with no graphical syntax [16]

minor mismatches in how indicators and strategies are supported, and there are additional classes to support aspect-oriented modeling. jUCMNav however can import and export models in the Z.151 XML-based interchange format.

3 Challenges Faced When Adding a Textual Syntax

Adding a concrete textual syntax to a metamodel-based language with an existing concrete graphical syntax involves steps in which there are technical and non-technical challenges. The designer of the textual syntax has to define related keywords, build a consistent structure associated with the definitions and assignments, keep the metamodel of the textual language (if any) compatible with the abstract syntax (language metamodel), select the proper technology to implement the textual language, implement a mechanism to apply specific restrictions and rules, and finally develop a mechanism to synchronize the textual and graphical syntax representations.

3.1 Choice of Keywords

Keywords of a textual language have an important role in the usability and adoption of the language. They must be chosen from the domain vocabulary

and be close to the language abstract syntax (assuming that the abstract syntax constructs have meaningful names). If the textual keywords are also aligned with the graphical syntax symbols and keywords, then this will also help the adoption of the textual language by already-existing users while also decreasing the learning curve. Balancing this closeness is not an easy task and usually results in tradeoffs. The users of graphical languages work with graphical notations and often names are hidden implicitly in the shape of notations. These hidden names sometimes cannot be expressed by a single word, supposed to be considered as a keyword in the textual syntax. Using the exact graphical or abstract names may potentially result in a bulky language.

During the design of TGRL, we have faced that challenge and we decided to consider three important criteria while defining keywords: (1) be consistent with the abstract syntax in terms of the semantics; (2) favor usability over rigid following of the metamodel; and (3) avoid defining keywords when possible. For example, we defined the keyword decomposedBy for the element link decomposition and did not use a specific keyword for defining evaluation elements for strategies. Note that when discussing usability in a textual modeling language, we do not try to compare it with that of a graphical modeling language (these are separate problems).

In terms of process, we first defined keywords similar to concept names from the abstract syntax. This helped us have a blueprint of the textual language and revealed some challenges, e.g., conflicting keywords or a high number of keywords. We then changed some keywords in order to solve conflicts, modified them to be more human readable, and finally simplified the language by removing unnecessary keywords.

3.2 Structure Consistency

Each defined keyword would have some structure and properties that need to be set during the development of models. The values assigned to properties and their structure should be kept similar, because this promotes language learnability. For this purpose, they are several patterns that can be adapted from either programming or modeling languages. For our language, we adopted a structure inspired from Umple [8,10]. For example, if there are several properties needed to be set, we use the name of the property along with its value. However, if there is only one property, we just assign the value without requiring the property name.

3.3 Alignment of Metamodels

When keywords and structures are defined, the grammar must be implemented. There are two general ways to do this. The first one is to use the already-existing language metamodel and cover it with the definition of the textual language. This approach makes the implementation process straightforward, but there might be situation where the grammar and the metamodel, which becomes a constraint, cannot be aligned properly without greatly affecting usability. The second approach is to let the textual language build its own metamodel, à la Xtext. This

approach allows getting the maximum benefits of having a simple and human-readable textual syntax, but it might result in an underlying metamodel that will require the creation of major internal model transformations from instances of the textual metamodel to instance of the abstract syntax metamodel.

We have experienced both approaches and recognized that the first approach results in a textual syntax that is too synthetic, especially if the language abstract syntax was never designed with a potential concrete textual syntax in mind (which is the case for GRL's). Furthermore, the second approach allows having several alternatives for a definition while it is not the case in the first approach. For example, we could design two alternatives ways of defining element links. The first alternative has an independent structure and needs both the link source and the link destination to be specified while the second alternative depends on the location in the source and just needs the destination to be specified. In our case study (next section), we have chosen to adopt the second approach.

3.4 Technology Selection

In order to implement one of the approaches discussed in the previous section, a suitable technology must be selected. As discussed in Sect. 2.2, there are several technologies such as Xtext, EMFText, and TCS that can be used for these purposes. The choice will be largely influenced by how potentially usable a textual syntax (automatically) generated from the abstract syntax can be, by the intended usage of the textual language, and by the required quality of resulting editor tooling.

In our study of TGRL, we have selected Xtext because it is an active project and provides a rich editor for the language. Working with Xtext is simple and fast, but everything has to be based on the Xtext grammar. This prevents developers from improving or "tweaking" the structure of the final metamodel.

Hence, this choice came at the cost of having to transforms Xtext-based models (from TGRL) to the target abstract syntax (in our case, URN's metamodel). In such a context, such transformation can be done with model-to-model transformations (e.g., with Java or specialized languages such as Eclipse's ATL Transformation Language [5]) or with model-to-text transformations targeting serialized models (e.g., again with Java or with enhanced technologies such as Xtend [31] or Acceleo [2]). As we had good experiences using Acceleo in the past, we opted for this path.

3.5 Handling Restrictions and Rules

The implemented syntax comes with restrictions and rules that need to be checked and applied continuously. These rules and restrictions come from two main sources: the abstract syntax (and its static semantics constraints) and the concrete syntax itself. For example, the identifier (ID) could have to follow a specific pattern, or cyclical definitions may need to be prevented. The rules from the abstract syntax are usually clear and already defined, but the ones from the concrete syntax must be specified. Such rules may be used to improve the

readability of the concrete syntax (especially if alternative representations are supported) or keep the syntaxes compatible. Part of this validation can be supported automatically by the technology used to implement the language (e.g., Xtext). However, the rest must be implemented manually. For example, checking the validity of a reference is supported by the editor provided by Xtext while checking for duplicates of a link must be implemented manually.

3.6 Synchronizing Textual and Graphical Models

Keeping connections between the textual syntax and the graphical syntax is important in order to fully benefit from the iterative use of both syntaxes by modelers. Generally, there are two ways to do this: *synchronously* and *asynchronously*. In the synchronous case, the transformation is done automatically and both syntaxes are refreshed continuously so as to show a consistent representation (in a way to the model-view-controller pattern). This is the most desirable case but its feasibility depends on the technology employed to develop the concrete syntaxes. If the technology used in either the textual or the graphical syntaxes does not support external synchronization, then this option might be impossible. In the asynchronous case, users work on a concrete syntax and when one needs to have the other representation, the transformation is done explicitly, on demand. This approach is a solution to the cases where the synchronous approach in unfeasible or when synchronization is too costly in terms of speed or memory usage.

In our case study, we used asynchronous transformations because of issues regarding the synchronization with the technology used for graphical syntax (e.g., jUCMNav). So far, we investigated only one transformation (from TGRL to URN), the reverse one being left for future work. As explained in Sect. 3.4, the current transformation is performed through a model-to-text approach implemented with Acceleo, which is a pragmatic implementation of OMG's *MOF Model to Text Language* (MTL) standard. jUCMNav can read the files generated in that way, and its auto-layout mechanism can be used to visualize the models.

4 Case Study: TGRL

4.1 TGRL Concrete Syntax

In this section, we describe a case study involving the design of a concrete textual syntax for GRL (called TGRL) with tool support (editor and automated transformation). Any concrete syntax has general rules that are applied for all keywords and their related structures. In our concrete textual syntax, the general rules are as follows:

- GRL elements are usually defined through keywords using camelCase boundaries (e.g., a softgoal intentional element is represented by a softGoal).
- Model element properties and sub-elements are set inside curly brackets.

- Every definition ends with a semicolon except when a pair of curly brackets is utilized to include properties or sub-elements.
- String values are surrounded by quotation marks.
- Comments are delimited by //.

TGRL model 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. 4 shows the TGRL representation of a simple GRL model with three actors, their intentional elements, and various links. This is a common GRL pattern where alternative ways of achieving some system functionality have different impacts on the concerns of stakeholders (such as users and developers). IDs are used as names unless specified otherwise. Qualitative values 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 [8,10], links can be specified inside one element or outside the relevant elements, depending on the modeler's preference. In Fig. 4, one contribution is defined inside the System actor, one is defined in the User actor and targets an element of another actor, and two other contributions are defined outside all actors.

Note that scoping is also used to resolve potential naming issues. For example, in the contribution link inside the System actor, task FirstOption is local but softgoal ReuseComponents is defined elsewhere, and hence must be prefixed by its containing actor (Developer). GRL dependency links are handled in a similar way.

TGRL also supports contribution changes (for which there is no graphical syntax in standard URN, see Fig. 3) and handles advanced constructs such as contribution inclusion and value ranges. For example, Fig. 4 contains a group (SomeOverrides) of two sets of contribution changes that make reference to two contribution links named C1 and C2. The first set (FirstOverride) changes C1 and C2 with new quantitative and qualitative values, respectively. A tool such as jUCMNav will substitute the targeted contributions weights with the new values specified in this contribution set before analyzing any strategy. The second set (SecondOverride) extends the first one (and hence inherits the make value for C2), but now C1 is defined as a range of values that go from −40 to 0 by steps of 10 (i.e., $\{-40, -30, -20, -10, 0\}$). In jUCMNav, when such a range is specified, the selected strategy is evaluated iteratively for all the contribution values in that range, leading to sets of resulting evaluations for all intentional elements and actors in the model (which is useful for sensitivity analysis). In TGRL's grammar, it was decided to keep the start, end, and step keywords in order to make the meaning of the values explicit and more easily understandable.

Similarly, TGRL supports groups of evaluation strategies, with strategy inclusion (for reuse), indicator initialization, and value ranges. Again here, TGRL provides a concrete textual syntax for elements that do not have a graphical syntax in URN.

```
grl SDL2015 {
    comment "This is a simple TGRL 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
       // Link definition inside the actor, qualitative weight
       LowLearningTime contributesTo EasyToUse {name=C2;help};
    }
    actor Developer {
       softGoal ReuseComponents {importance=100;}
    }
    actor System {
       // Goal with qualitative importance, and OR decomposition type
       goal SomeFunctionality {importance=high; decompositionType=or;}
       task FirstOption {metadata stereotype="SomeValue";}
       task SecondOption {description = "Better alternative";}
       SomeFunctionality decomposedBy FirstOption, SecondOption;
       // Link across actors, quantitative weight
       FirstOption contributesTo Developer.ReuseComponents {75};
    }

   // Links defined outside their actors
   System.FirstOption contributesTo User.EasyToUse {hurt};
   System.SecondOption contributesTo User.EasyToUse {name=C1;60};

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

   // Strategy examples
   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;

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

Fig. 4. Simple illustrative TGRL model

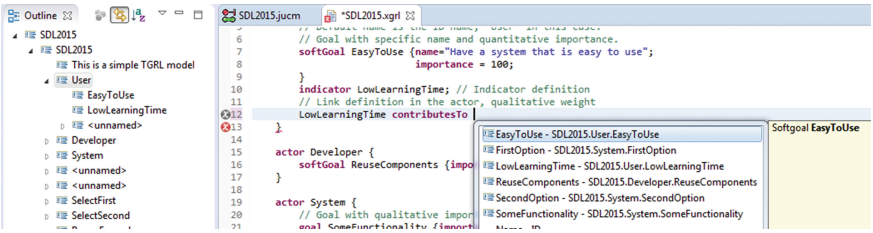

Fig. 5. Overview of the TGRL editor, with content assistance

Note also at the end of Fig. 4 that URN links are also supported. In this example, a link of a user-defined type mustUse connects the User to the System. Again, standard URN does not have a concrete graphical for this element, and jUCMNav relies on dialog boxes for creating such model elements (which are not displayed on the diagrams).

4.2 TGRL Editor and Transformation to jUCMNav

As Xtext was used to implement the TGRL syntax, we were able to get a feature-rich editor for very little effort. The Eclipse-based TGRL editor comes with configurable syntax highlight (as shown in the code in Fig. 4), an outline view, annotation of syntactic errors, content assistance, and code formatting. Figure 5 gives an overview of the editor.

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. 5, several suggestions are provided as potential targets of an incomplete contribution link. This greatly accelerates the coding, and also the learning of the language as one can get suggestions at any step.

The transformation between TGRL models and URN/jUCMNav models serialized in XML was implemented with Acceleo. While designing the TGRL syntax, we were able to make quick iterations from changing the Xtext-based grammar to adapting the Acceleo code and regenerating the editor and executable transformation, often within two minutes. Our transformation does not handle the layout of the generated GRL diagrams, but jUCMNav has several features for creating views of a model and for automatically laying out elements. For example, Fig. 6 shows the GRL model corresponding to the ongoing example, as imported by jUCMNav. The evaluation of the strategy SelectFirst is also shown, using quantitative values, and without any contribution change applied.

5 Discussion

In their original draft proposal for GRL in 2001, Liu and Yu defined a GRL ontology with a graphical syntax, a textual syntax, and an XML interchange format (but without a fully defined abstract syntax) [20]. The textual notation they proposed was cognitively hard to understand and did not cover the advanced

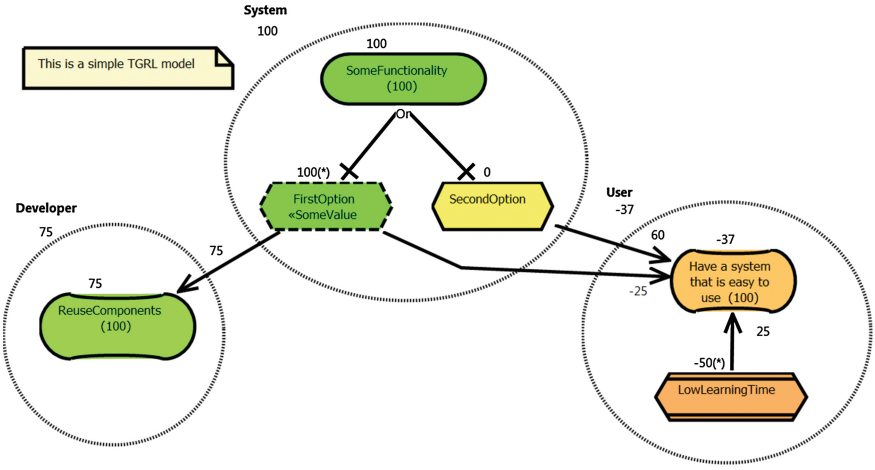

Fig. 6. Sample GRL model imported in jUCMNav, with a strategy evaluated quantitatively

GRL features found in standard URN (e.g., indicators, strategies, quantitative values, metadata, URN links, and contribution changes). We believe that the syntax for GRL should be intuitive, without requiring keywords when the context is clear (e.g., TGRL contribution weights do not require a keyword as they are expected to be provided 99 % of the time). TGRL is the first concrete syntax to cover all of GRL's constructs, and in that sense it goes beyond URN's standard graphical syntax [16].

Rashidi-Tabrizi et al. also proposed and implemented (in jUCMNav) an import mechanism for GRL models and strategies in a *tabular concrete syntax*, as comma-separated value files [26]. This allows people knowledgeable in tools like Excel to create GRL models without having to use jUCMNav, and then use jUCMNav for visualizing and analyzing models (as we do). However, their mechanism is limited to a subset of GRL (e.g., without contribution changes), and targets a very specific type of GRL models for the laws and regulations domain. Hence, their solution is not as generic and exhaustive as TGRL's.

Engelen and Van Den Brand have used two techniques, named *grammarware* and *modelware*, for the integration of textual and graphical modeling languages by implementing a textual surface language as an alternative for activity diagrams in UML [7]. In the grammarware technique, a text-to-text transformation was used while model-to-text, text-to-model, and model-to-model transformations were used in modelware. Their approach enabled them study the benefits and drawbacks of both techniques. Other similar comparisons were done by Gargantini et al. [9]. In our implementation, we utilized a modelware-like approach in which we have a model-to-text transformation used to generate jUCMNav files from TGRL models. However, more importantly, the availability of a GRL graphical editor (jUCMNav) and of a textual editor (TGRL) now enables us to compare both approaches quantitatively and answer usability questions in requirements engineering and system development contexts.

The integration of textual and graphical multi-view domain-specific languages was explored by Pérez Andrés et al. [24], in which they utilized the AToML model transformation tool. In their approach, a metamodel of the whole language must be defined first and then subsets have to be selected for different viewpoints. Then, a viewpoint metamodel is transformed into a textual model, from which a parser is automatically derived and integrated with the generated multi-view environment. This approach can be seen as a bridge between the modelware and the grammarware approaches. This viewpoint approach might be revisited in our context as in fact GRL is a view of URN. If a textual syntax is eventually produced for UCM (the other sub-language of URN), it might be interesting to evaluate whether it is beneficial to support URN models, UCM models, and GRL models (the three views) with standalone tools. It would also be interesting to consider providing different concrete textual syntaxes for GRL, e.g., for goal modeling in general, or with different keywords for specific domains such as law and regulation modeling, as needed in [26].

6 Conclusions and Future Work

In this paper, we have reported on the challenges that exist when trying to add a usable concrete textual syntax to a rich metamodel-based language predominantly oriented towards a concrete graphical syntax. We have discussed several alternatives for addressing challenges related to the choice of keywords, to structure consistency, to the alignment of metamodels, to the selection of suitable language design technologies, to rule handling, and to the synchronization between textual and graphical representations.

Many of these challenges were illustrated based on our case study, where we created a textual syntax for the GRL modeling language called TGRL. In addition to the observations we have made based on our experience, this paper led to the creation of the first textual syntax for an i^*-based goal-modeling language (as far as we know). TGRL also covers GRL fully, even concepts for which there is no standard graphical syntax (e.g., strategies, contribution changes, and URN links). A feature-rich, Xtext-based editor is now available for TGRL, together with a transformation to a URN model serialized in XMI and readable by the jUCMNav tool [1]. The availability of TGRL now enables researchers to compare the efficiency and usability of textual and graphical syntaxes in a goal modeling context, for different tasks and types of users.

In terms of future work, as this paper reports on only one language, it would be important to gather additional experience on other modeling languages, including some that are not based on metamodels. This would help identify common problems and trends across languages of different natures. It would also be interesting to better separate concerns related to language definition and tool integration.

On the TGRL side, further validation of the correctness and usability of this language is needed. 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 support for additional well-formedness and semantic rules in the TGRL editor would also represent a good improvement. Another obvious step is the extension of this language to support the whole URN standard, including UCM (where, again, several concepts do not have a graphical syntax [11]). This could even lead to improvements to the URN standard at ITU-T. We also envision opportunities to combine TGRL (for goals) with Umple (for design and implementation) as they provide complementary concepts. Finally, the study of various concrete syntaxes for GRL (or URN), targeting different domains, is also something to explore, especially in terms of cognitive fitness.

Acknowledgement. This work was sponsored in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) through its Discovery grant program.

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. Amyot, D., Mussbacher, G.: User requirements notation: the first ten years, the next ten years. J. Softw. (JSW) **6**(5), 747–768 (2011)
4. Amyot, D., Shamsaei, A., Kealey, J., Tremblay, E., Miga, A., Mussbacher, G., Alhaj, M., Tawhid, R., Braun, E., Cartwright, N.: Towards advanced goal model analysis with jUCMNav. In: Castano, S., Vassiliadis, P., Lakshmanan, L.V.S., Lee, M.L. (eds.) ER 2012 Workshops 2012. LNCS, vol. 7518, pp. 201–210. Springer, Heidelberg (2012). http://softwareengineering.ca/jucmnav
5. ATL Transformation Language (2015). https://eclipse.org/atl/
6. Cabot, J.: UML tools - textual notations to define UML models (2009). http://sumo.ly/5Mb. Accessed 6 June 2015
7. Engelen, L., Van Den Brand, M.: Integrating textual and graphical modelling languages. Electron. Notes Theor. Comput. Sci. **253**(7), 105–120 (2010)
8. Forward, A., et al.: Model-driven rapid prototyping with Umple. Softw. Pract. Exper. **42**(7), 781–797 (2012)
9. Gargantini, A., Riccobene, E., Scandurra, P.: Deriving a textual notation from a metamodel: an experience on bridging modelware and grammarware. In: 3M4MDA. CTIT Workshop Proceedings Series WP06-02, pp. 33–48 (2006)
10. Garzón, M., Aljamaan, H.I., Lethbridge, T.C.: Umple: A Framework for Model Driven Development of Object-Oriented Systems. In: SANER 2015, pp. 494–498. IEEE CS (2015)
11. Genon, N., Amyot, D., Heymans, P.: Analysing the cognitive effectiveness of the UCM visual notation. In: Kraemer, F.A., Herrmann, P. (eds.) SAM 2010. LNCS, vol. 6598, pp. 221–240. Springer, Heidelberg (2011)
12. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and refinement of textual syntax for models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 114–129. Springer, Heidelberg (2009)
13. International Telecommunication Union: ITU-T Recommendation Z.100 (12/11) - Specification and Description Language - Overview of SDL-2010 (2011). http://www.itu.int/rec/T-REC-Z.100-201112-I

14. International Telecommunication Union: ITU-T Recommendation Z.111 (11/08) - Notations and guidelines for the definition of ITU-T languages (2008). http://www.itu.int/rec/T-REC-Z.111-200811-I
15. International Telecommunication Union: ITU-T Recommendation Z.120 (02/11) - Message Sequence Chart (MSC) (2011). http://www.itu.int/rec/T-REC-Z.120-201102-I
16. International Telecommunication Union: ITU-T Recommendation Z.151 (10/12) - User Requirements Notation (URN) - Language Definition (2012). http://www.itu.int/rec/T-REC-Z.151-201210-I
17. International Telecommunication Union: ITU-T Recommendation Z.161 (11/14) - Testing and Test Control Notation Version 3: TTCN-3 Core Language (2012). http://www.itu.int/rec/T-REC-Z.161-201411-I
18. International Telecommunication Union: ITU-T Recommendation Z.162 (11/07) - Testing and Test Control Notation Version 3: TTCN-3 Tabular Presentation Format (TFT) (2012). http://www.itu.int/rec/T-REC-Z.162-200711-I
19. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: GPCE 2006, pp. 249–254. ACM Press (2006)
20. Liu, L., Yu, E.: GRL - goal-oriented requirement language. University of Toronto, Canada (2001). http://www.cs.toronto.edu/km/GRL
21. 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)
22. Mussbacher, G., Amyot, D., Heymans, P.: Eight deadly sins of GRL. In: 5th International i^* Workshop (iStar 2011), CEUR-WS, vol. 766, pp. 2–7 (2011)
23. OMG: UML Human-Usable Textual Notation (HUTN). Version 1.0, formal/2004-08-01 (2004). http://www.omg.org/spec/HUTN/1.0/
24. Pérez Andrés, F., de Lara, J., Guerra, E.: Domain specific languages with graphical and textual views. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 82–97. Springer, Heidelberg (2008)
25. Petre, M.: Why looking isn't always seeing: readership skills and graphical programming. Commun. ACM **38**(6), 33–44 (1995)
26. Rashidi-Tabrizi, R., Mussbacher, G., Amyot, D.: Transforming legulations into performance models in the context of reasoning for outcome-based compliance. In: RELAW 2013, pp. 34–43. IEEE CS (2013)
27. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: Constructing models with the human-usable textual notation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 249–263. Springer, Heidelberg (2008)
28. Roy, J.-F., Kealey, J., Amyot, D.: Towards integrated tool support for the user requirements notation. In: Gotzhein, R., Reed, R. (eds.) SAM 2006. LNCS, vol. 4320, pp. 198–215. Springer, Heidelberg (2006)
29. Scheidgen, M.: Textual modelling embedded into graphical modelling. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 153–168. Springer, Heidelberg (2008)
30. Schmidt, M., Wider, A., Scheidgen, M., Fischer, J., von Klinski, S.: Refactorings in language development with asymmetric bidirectional model transformations. In: Khendek, F., Toeroe, M., Gherbi, A., Reed, R. (eds.) SDL 2013. LNCS, vol. 7916, pp. 222–238. Springer, Heidelberg (2013)
31. Xtend (2015). http://www.eclipse.org/xtend/
32. Xtext (2015). http://www.eclipse.org/Xtext/