Creating Modelling Tools for i* Language Extensions

João Pimentel[1], Jaelson Castro[2], Moniky Ribeiro[2], Alberto Souza[2], Breno Ramos[2]

[1] Universidade Federal Rural de Pernambuco, UFRPE, Brazil
[2] Universidade Federal de Pernambuco, UFPE, Brazil
joao.hcpimentel@ufrpe.br, jbc@cin.ufpe.br, smsr@cin.ufpe.br, ass6@cin.ufpe.br, brr@cin.ufpe.br

Abstract. In spite of the expressiveness of the i* modelling language, in excess of 90 extensions have been proposed in order to address specific concerns of some domains or development approaches, among other reasons. When new extensions are proposed, it is common practice to develop accompanying modelling tools. Despite many advances in meta-modelling technologies, the creation of such tools is still cumbersome and time-consuming. This paper presents an approach and tool support that facilitates the creation of modelling tools for extensions of the i* language. The approach is illustrated with a safety-related extension that features four new kinds of elements and a single new kind of link.

Keywords: Modelling Language Extension, Modelling Tools, Requirements Engineering.

1 Introduction

The *i** research community has a long history of proposing extensions to the language, with more than 90 extensions catalogued [3]. It is common practice to develop supporting tools to proposed extensions, in order to allow the proper creation of models in the extended language. In fact, the *i** wiki[1] lists a total of 29 *i** tools.

Despite strongly facilitating the development of visual modelling tools, generic meta-modelling frameworks are naturally not able to provide out-of-the-box support for the specific characteristics of every possible language. Thus, considerable effort is required in order to extend them to provide support for the specificities of *i** and its extensions, such as collapsible actor boundaries, automatically resizable containers, non-trivial shapes, and dependency links.

Paes et al. [4] presented an approach to generate modelling tools for *i** variants, borrowing concepts from Software Product Lines. Like our proposal, the focus there is on the pragmatics of tool developments rather than on conceptual mappings. Cares et al. [1] defined an XML-based file format for the interoperability of *i** models. The idea is to be able to use a single format to represent the dialects and variants implemented by different modelling tools.

[1] Available at http://istarwiki.org

This paper presents mechanisms built into the piStar tool [5] to facilitate the creation of tool support for *i** extensions. Section 2 presents an overview of the mechanisms, which are exemplified in Section 3 with an extension for critical systems. Section 4 presents related work and Section 5 concludes the paper describing ongoing and future work.

2 Meta-meta-model for Language specification

In order to support the automatic generation of modelling tools for *i** extensions, the piStar tool [5] executes on top of language specification files. The tool reads the language specification at runtime and automatically generates a web-based modelling tool, complete with user interface (UI), model loading and model saving capabilities, image saving capabilities, and properties editing. Additionally, helper methods are also generated (e.g., addGoal, isGoal, goal.delete).

The language specification is split into three files, one being mandatory (the meta-model of the language), and two others optional (language constraints and concrete syntax). While constraints are defined as JavaScript functions, the remaining content (meta-model, concrete syntax and UI details) is specified declaratively in a textual notation (JavaScript Object Notation – JSON). Additional UI information can also be specified if desired. Examples are given in the next section.

The meta-meta-model supported by the tool is presented in Fig. 1 in the form of a Unified Modelling Language (UML) class diagram. The base concept is a Cell, which can be either an Element or a Link. Besides a textual id for unique identification as well as custom properties, every Cell also contains an isValid() method that is used to determine whether a given Cell is valid or not – i.e., the instantiation of this method defines the constraints of the language. The custom properties may be developer-defined (included in the meta-model of a language) or user-defined (created at runtime by a user). They can be used to define meta-information such as rationale, author, date, and status. Moreover, every Element contains a name, which is displayed on the model's diagram as a label.

Containers are Elements that can contain Nodes, whereas a Node is an Element that is not a Container. This definition implies that Containers cannot contain other Containers. The instantiations of Containers in iStar 2.0 are the Actor, Agent and Role concepts. They can be collapsed as to hide their contained nodes, expanded back, and also toggled (collapse if expanded, expand if collapsed).

Nodes represent atomic elements, such as goals, tasks, resources and qualities. Three boolean attributes are used to indicate where a Node can be added: inside a container (*canBeInnerElement*), as part of a Dependency (*canBeDependum*), or directly onto the diagram (*canBeOnPaper*).

Links can have a single textual label. In some cases, the user may be able to change the label (*changeableLabel)*, choosing a value from the *possibleLabels* enumeration. This is the case, for instance, of contribution links in iStar 2.0, which can assume labels such as make, help, hurt and break. Additionally, the *tryReversedWhenAdding* attribute of Link can be set to true when there is only one way to link an Element with another

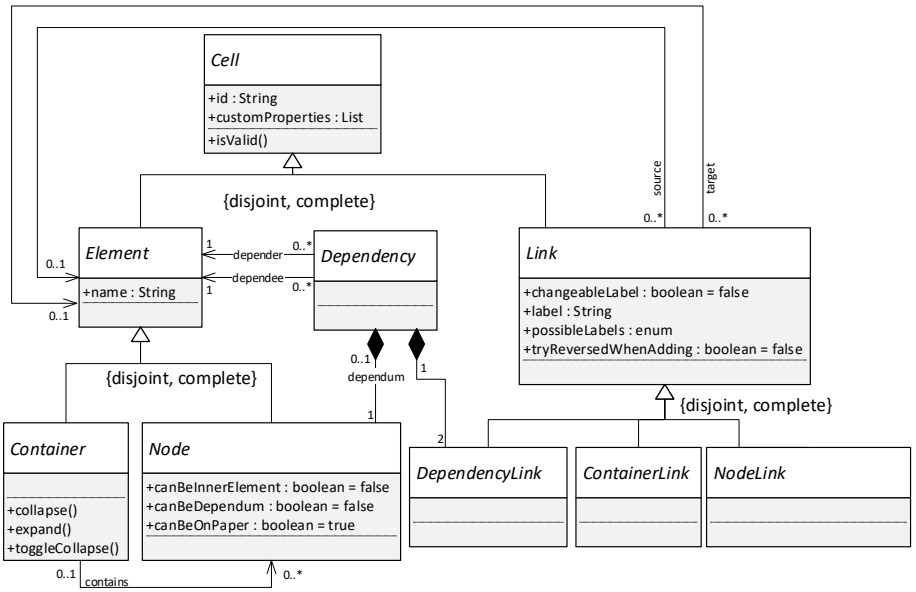

Fig. 1. Meta-meta-model supported by the piStar tool

Element correctly; if the user tries to create the link in a wrong direction, the tool will then try to create the link with reversed source and target (e.g., qualification links and needed-by links).

There are three kinds of Link: ContainerLink, NodeLink, and DependencyLink. A ContainerLink links a Container to another Container: is-a and participates-in links. A NodeLink links a Node to another Node: and-refinement, or-refinement, needed-by, qualification, and contribution links. A DependencyLink links a depender or a dependee element to a dependum.

The meta-meta-model already provides the basis for some constraints. For instance, in this framework, a ContainerLink cannot be used to link Nodes. Similarly, the Node attributes define where Nodes can be inserted. Additional constraints can be defined in a constraints' specification. Even though the Object Constraint Language (OCL) is the standard for defining meta-model constraints, we have opted to define the constraints with JavaScript functions, which not only is more developer-friendly but is also executable on the target platform (web browser). The downside is that previously defined constraints in OCL cannot be directly transferred to the tool – instead, the constraints must be translated to JavaScript code. Along with the constraints, it is possible to define explanatory messages that are shown to the user when an invalid construction is attempted.

Developers can define the concrete syntax of Elements and Links with custom shapes. Particularly, whether links can be curved or not is also defined in their shapes. The shapes are defined with Scalable Vector Graphics (SVG) elements and attributes. Unlike rasterized images such as PNG and JPG, SVG images can be scaled up and down without distortions. If no shapes are defined, default stereotyped shapes are used

for the diagram: dashed rectangles for Nodes, dashed circles for Containers, and dashed arrows for links.

In summary, the meta-model and constraints files define the abstract syntax of the language, whereas the shapes file defines the concrete syntax. Additionally, the user can customize the textual content of the UI describing, for instance, written instructions on how to create specific kinds of links as well as images for the interactive buttons. The next section describes the instantiation of these files for the iStar4Safety extension.

3 The iStar4Safety extension

iStar4safety [6] is an extension of iStar 2.0 [2] aiming to enable early requirements engineering for critical systems by supporting the main concepts of Preliminary Safety Analysis [7]. It extends iStar 2.0 with four new kinds of elements (Hazard, SafetyGoal, SafetyTask, and SafetyResource) and one new kind of link (obstruct).

Metamodel. The new kinds of elements are *nodes*. All these elements must go inside actor boundaries; they cannot be dependums in dependency links, nor added to an empty portion of the diagram (the paper). Thus, their *canBeInnerElement*, *canBeDependum* and *canBeOnPaper* attributes are set to true, false and false, respectively. Moreover, all Safety Goals must have an Accident Impact Level attribute. This is achieved by adding it as a custom property of the Safety Goal object. The new kind of link is defined as a *nodeLink*. Fig. 2 presents an excerpt of the metamodel file for iStar4Safety, where the added elements are written in bold.

Shapes. In iStar4Safety, the new elements are colour variations on the shapes of the original iStar 2.0 elements: Hazards and Safety Goals have the shape of goals; Safety Tasks have the shape of tasks; and Safety Resources have the shape of resources. Hence, the definition of their concrete syntax is trivial – copy the definitions of the original elements and change their colour. Moreover, the *obstruct* link shape is a copy of the shape of contribution links. These variations are illustrated in Fig. 4.

Constraints. Constraints are defined as isValid() functions for each kind of element or link. The function must return a Boolean validity value and, optionally, it can also return an error message to be displayed to the user. Fig. 3 shows two constraints of Obstruct links: their source must be a Hazard and their target must be a Safety Goal.

User Interface. Besides the language itself, developers can declaratively customize some elements of the user interface, such as the name and images of buttons in the tool's palette. Their images are defined as SVG files named after the actual name of the element or link.

Fig. 2 presents a screenshot of the piStar tool with the iStar4Safety extension. A live version of the tool is available at https://www.cin.ufpe.br/~jhcp/pistar/4safety

```
{
  ...
  "nodes": {
    "Goal": {
      "canBeInnerElement": true, "canBeDependum": true,
      "canBeOnPaper": false },
    "Quality": {
       "canBeInnerElement": true, "canBeDependum": true,
       "canBeOnPaper": false },
    "Resource": {
      "canBeInnerElement": true, "canBeDependum": true,
      "canBeOnPaper": false },
    "Task": {
      "canBeInnerElement": true, "canBeDependum": true,
      "canBeOnPaper": false },
    "Hazard": {
      "canBeInnerElement": true, "canBeDependum": false,
      "canBeOnPaper": false },
    "SafetyGoal": {
      "canBeInnerElement": true, "canBeDependum": false,
      "canBeOnPaper": false,
      "customProperties": {
        "accidentImpactLevel": "" } },
    "SafetyTask": {
      "canBeInnerElement": true, "canBeDependum": false,
      "canBeOnPaper": false },
    "SafetyResource": {
      "canBeInnerElement": true, "canBeDependum": false,
      "canBeOnPaper": false }
  },
  "nodeLinks": {
    "AndRefinementLink": { },
    "OrRefinementLink": { },
    "NeededByLink": { "tryReversedWhenAdding": true },
    "QualificationLink": { "tryReversedWhenAdding": true },
    "ContributionLink": {
      "changeableLabel": true,
      "possibleLabels": ["make", "help", "hurt", "break"] },
    "ObstructLink": { "label": "obstruct" }
  }
}
```

Fig. 2. Excerpt of the metamodel file for iStar4Safety

```
if ( !source.isHazard() ) {
  isValid = false;
  result.message = 'the source of an Obstruct link must be a Hazard';
}
if ( isValid && !target.isSafetyGoal() ) {
  isValid = false;
  result.message='the target of Obstruct links must be a Safety Goal';
}
```

Fig. 3. Excerpt of the constraints for Obstruct links in iStar4Safety

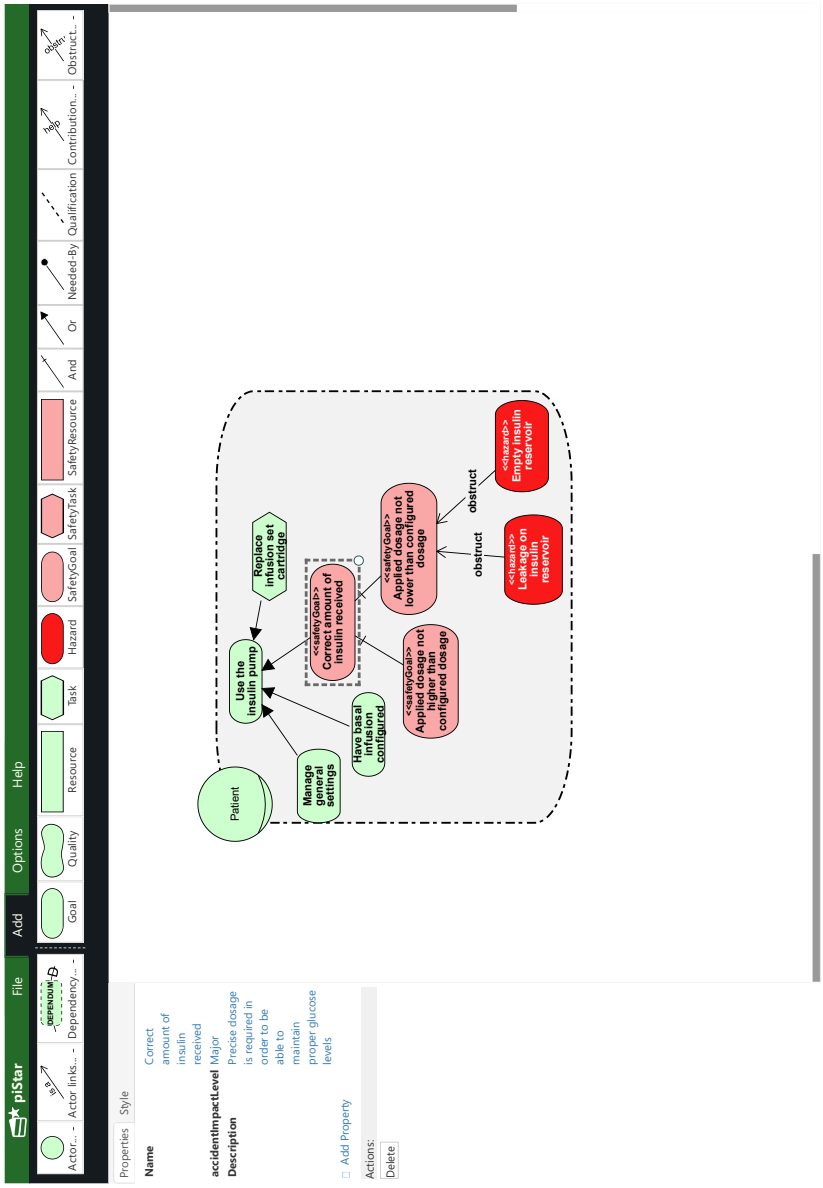

Fig. 4. Screenshot of the piStar tool with the iStar4Safety extension

4 Ongoing and future work

Currently we are developing a centralized repository for piStar-based implementations of *i** extensions. The idea is that, instead of extension developers deploying their tools in different servers, they will submit them to this centralized repository. Submitted extensions could then be selected by users from the original piStar tool website. This is similar to what has already been accomplished by the OME and RE-Tools, which allow users to select from different modelling languages when creating a new model.

As future work, we expect to extend this repository to also support the submission of functionality plugins. These are plugins that, instead (or besides) extending the *i** language, provide additional functionalities to the base tool, such as metrics calculation, automated reasoning, and visualization options.

Acknowledgments: The authors thank CNPq & FACEPE for their financial support.

References

1. Cares, C., Franch, X., Perini, A., Susi, A. (2011). Towards interoperability of *i** models using iStarML. Computer Standards & Interfaces, 33(1), 69-79.
2. Dalpiaz, F., Franch, X., & Horkoff, J. (2016). iStar 2.0 language guide. In: arXiv preprint arXiv:1605.07767
3. Gonçalves, E., Heineck, T., Araújo, J., Castro, J.: CATIE: A catalogue of istar extensions. Cadernos do Ime. Série Informática, v. 48, p. 23-37, 2018.
4. Paes, J., Castro, J., Silva, C., Santos, E., Lima, C. (2011). An Approach to Generate Tools for i* Languages. In 25th Brazilian Symposium on Software Engineering (pp. 243-252). IEEE.
5. Pimentel, J., & Castro, J. (2018). piStar Tool–A Pluggable Online Tool for Goal Modeling. In IEEE 26th International Requirements Engineering Conference (RE) (pp. 498-499). IEEE.
6. Ribeiro, M., Castro, J., Pimentel, J. iStar for Safety-Critical Systems. In 12th iStar Workshop (iStar), 7 pages, 2019.
7. Vilela, J., Castro, J., Martins, L.E.G., Gorschek, T., & Silva, C.: Specifying Safety Requirements with GORE Languages. Proceedings of the 31st Brazilian Symposium on Software Engineering. pp. 154–163, 2017.