

Goal Model Reference Guide

1 An Overview of Goal Modelling

In requirements engineering, a requirement is a statement or specification that describes a feature, functionality, or constraint that a system, product, or service must possess in order to satisfy the needs of its stakeholders (such as users, customers, or other systems).

Requirements are essentially the building blocks for the development of any system, serving as a clear agreement between stakeholders and developers about what the system is expected to do, how it should perform, and what limitations it must operate within.

Goal Oriented Requirements Engineering refers to the use of goals for requirement elicitation, evaluation, negotiation, structuring, documentation, analysis and evolution. A goal is thus an objective related to a set of requirements that the system should satisfy through the action of its agents.

Goal Modelling refers to a hierarchical model for a system's objectives detailing the relationship between agents and between the system and its environment, clarifying what must be done to accomplish a requirement. Notably, it allows for large goals to be analysed into small realisable goals through decomposition allowing designers to more accurately perceive goal completeness.

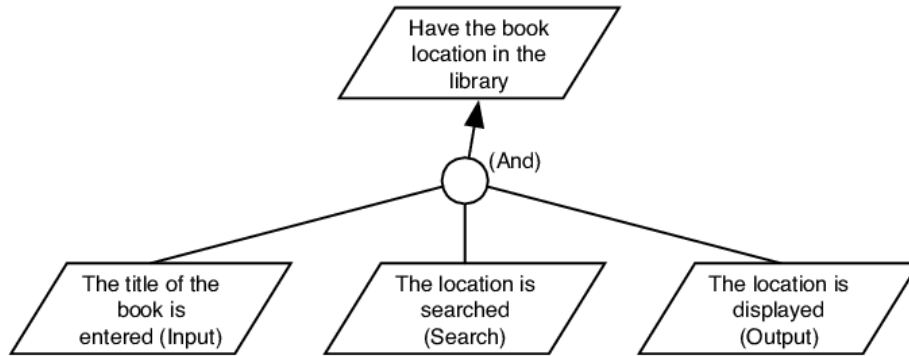


Figure 1: Goal model example of a library system

2 I* Goal Modelling Language

I* is a modelling language for goal modelling, originally developed for modelling and reasoning about organizational environments and their information systems composed of heterogeneous actors with different, often competing, goals that depend on each other to undertake their tasks and achieve these objectives.

Similarly, robotic environments often have many different heterogeneous actors (the robots) with different goals, which often compete with one another, such as in systems with limited resources (eg. battery). Below we showcase the commonly used notations for the language:

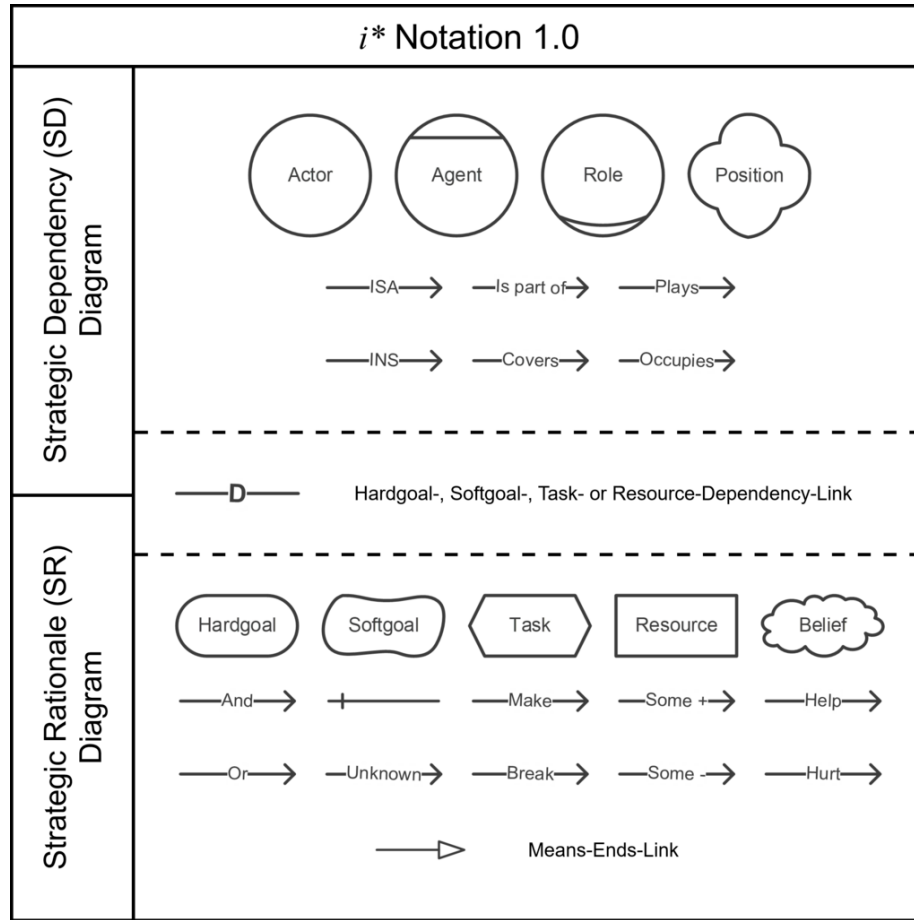
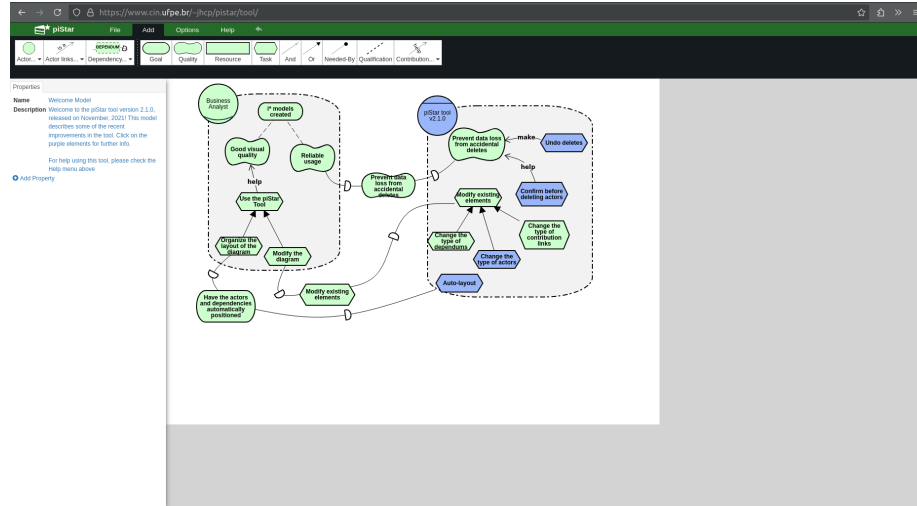


Figure 2: Element notation of the I* language

3 piStar Modelling Tool

We will be using the piStar Tool to model goal models in the i* language. When accessing the link above, you should see the following screen with an example model:



In the *Add* button in the upper part of the screen we can add different elements to our model. You can click and drop them to place them on the model. Namely, those elements are:



- **Actor:** An Actor is an active autonomous entity that uses its capabilities to achieve their goals in collaboration with other agents. For each agent in our system, we describe a root goal that must be achieved and its decomposition, which we shall explained later;
- **Actor Links:** We can establish links between different actors: (i) *Actor is a*: We can say two actors have the same capabilities and are of the same type, ie. student A is a student; (ii) *Participates In*: We can say an actor is part of another larger actor, ie. student A participates in Department of Computer Science. Note that the Department of Computer Science is also an actor with its own respective goals and capabilities;
- **Dependencies:** We can establish dependencies between goals from one actor to another, such as a student requires a pen to do the test, but he depends on another student lending him his pen.

- Goal: Goals are the main structure of a goal modelling notation and represent what needs to be done to achieve a set of system requirements;
- Qualities: Qualities are additional requirements that can be added to a system, but are not necessary to the system, such as maintaining a good service to the user versus maintaining the bare minimal service.
- Tasks and Resources: Tasks usually represent the bottom of a goal model, representing concrete actions that must be performed to achieve a goal. Resources can be added onto tasks by using the *needed-by* element, such as the task of writing on a paper requires the resource of a pen or pencil.
- And/Or: We can decompose goals into additional sub-goals using an AND or OR decomposition. An AND decomposition implies the goal is achieved if and only if all of its AND decomposed goals are also achieved. An OR decomposition implies the goal is achieved if and only if atleast one of its GOAL decomposed subgoals are achieved.

4 MutRoSe - Multi-Robot mission Specification and Decomposition

For the specifying and decomposing the mission, we will be using the MutRoSe extension for Visual Studio Code. The extension not only ships the MutRoSe binary compiled for Ubuntu, but also ships the piStar Modelling Tool as a custom text editor for *.gm files* which integrates with a custom tree view that represents all the goal models in the workspace, so it's possible to create and decompose the goal models without the need of multiple workspaces.

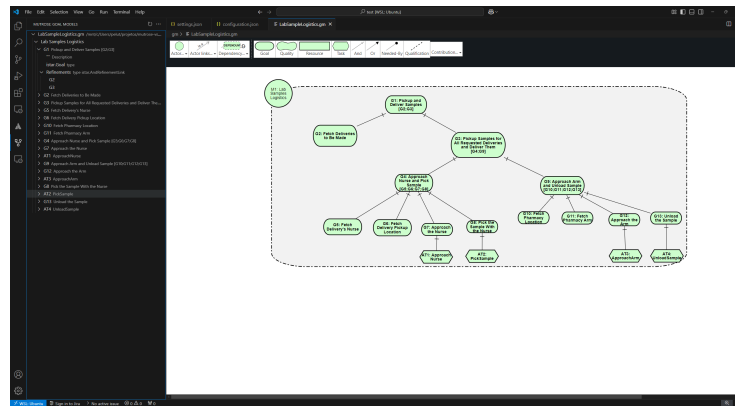
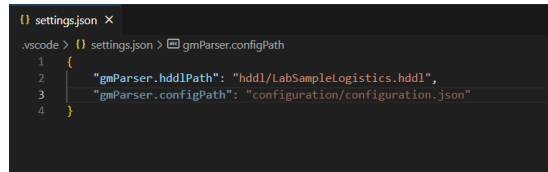


Figure 3: Visual Studio Code Extension with Tree View and piStar.

4.1 The project structure inside a VSCode workspace

For the extension to work properly, the created workspace shouldn't contain any spaces in the name and must have the following folders:

- `.vscode`: should contain the `settings.json` file, where the configuration e hddl files path is defined for the MutRoSe decomposition. See example in the figure 4.
- `gm`: where all `.gm` files should be. They must have the same content as the ones generated by the piStar hosted by the UFPE.
- `hddl`: that is where you should place your HDDL file, keeping in mind that it still needs to be declared in the `settings.json` file to be used in the decomposition.
- `knowledge`: the folder to store the `world.db.xml` file.
- `configuration`: Where the `configuration.json` needs to be stored.
- `output`: should be an empty directory since it's the output path for the decomposed mission.
- `ihtn`: does not need to be manually created, it is where the iHTN will be generated at.

A screenshot of a Visual Studio Code editor window showing the `settings.json` file. The file is open in the editor, and the content is as follows:

```
{
  "gmParser.hddlPath": "hddl/LabSampleLogistics.hddl",
  "gmParser.configPath": "configuration/configuration.json"
}
```

The file is named `settings.json` and is located in the `.vscode` directory. The editor shows line numbers 1 through 4.

Figure 4: Example of the `settings.json` file.

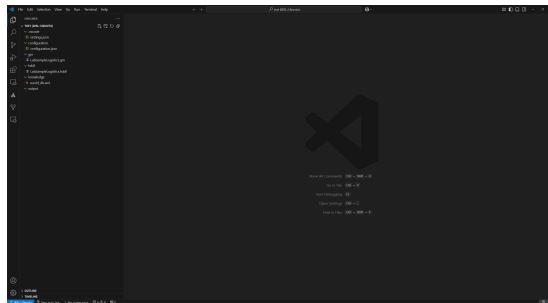


Figure 5: Example of a workspace inside Visual Studio Code.

4.2 Using the Tree View for viewing and editing a mission

The Tree View have multiple levels of depth, each representing one type of element. The hierarchy is given by:

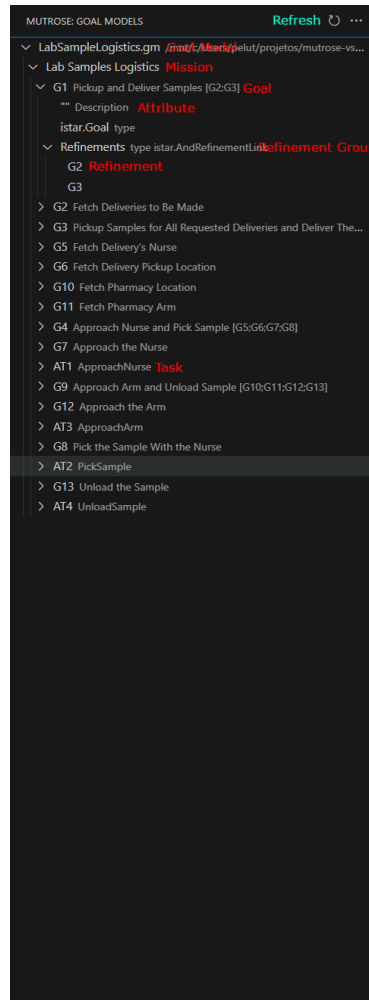


Figure 6: Example of a Goal Model being visualized in the Tree View.

- Goal Model
 - Mission
 - * Goal
 - Attribute
 - Refinement Group

- - Refinement
- * Task
- Attribute

Keep in mind that even though the Task and the Goal elements can have the same attributes and refinements because of the current implementations of the Tree View, that is not necessarily correct semantically or syntactically.

Each type of element can be edited by using the context menu that is accessed by right click the element.

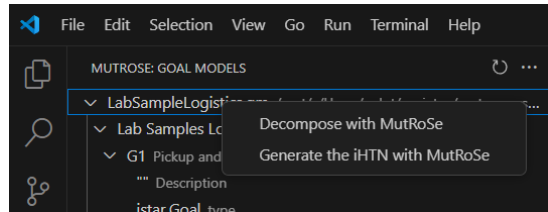


Figure 7: Goal Model File's context menu.

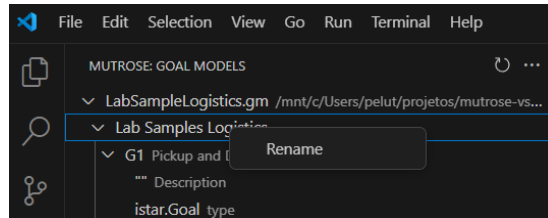


Figure 8: Mission's context menu.

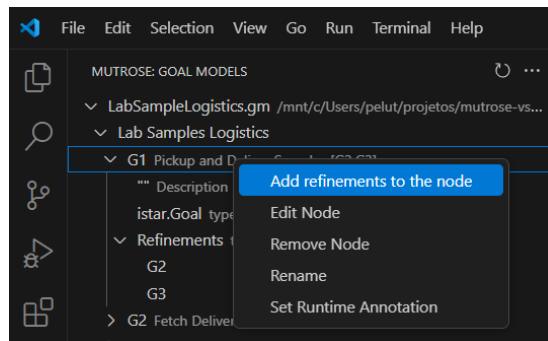


Figure 9: Task and Goal's context menu.

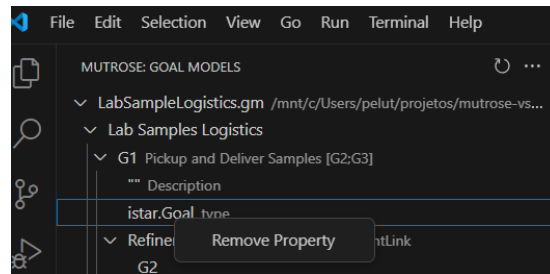


Figure 10: Property's context menu.

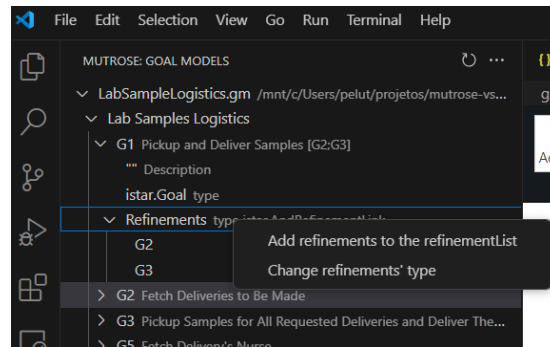


Figure 11: Refinements' context menu.

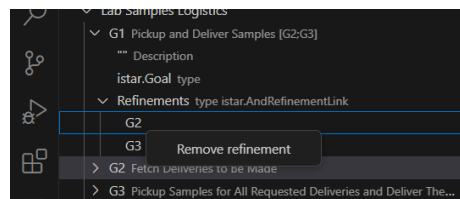


Figure 12: Refinement's context menu.

When clicking on one of the menu's options, there are 5 types of behavior that can happen:

- Open a Custom Edit Flow.
- Open a Quick Pick.
- Open an Input Box.
- Remove the element from the text file directly.

- Execute the binary corresponding to the selected option.

Specifically, in the Custom Edit Flow, a list of attributes with name and value will be shown. after choosing one of these, either a quick pick or an input box should appear so you can define the chosen attribute's value. If the needed attribute doesn't exist in the list, there's a custom property options that allows to type both the name and the value of the attribute, thus creating a new entry in the list. Lastly, for all the alterations to be written in the Goal Model, is necessary to select the last option of the list, named as "Confirm Edition".

After any alteration in the files it's possible that the tree view did not show the new changes. When that happens, there's a refresh button, which it's highlighted in the figure 6, that update the Tree View.

4.3 Integration between piStar and the Tree View

Mostly of the integration between the piStar and the Tree View is done by using the Goal Model file as medium. That means that each alteration made by piStar will be reflected in the Tree View only after the file is saved in the custom text editor. Also, there are some redundancies between the Tree View and the piStar's actions e.g. adding a refinement for a node, so the user can choose the preferred way without any drawbacks.

Besides that, there are two interactions that are specific between the Tree View and the Custom Text Editor:

- Creating an element with an auto-generated tag.
- Revealing in the Tree View the selected element (that isn't a refinement) in piStar.

When creating an element using the piStar menu represented in the figure 3, a tag will be generated automatically. The only caveat related to that is the need to save the edited file after each element creation, otherwise the tag won't appear in the name of the element, even though it'll be in the text file.

Related to revealing selected element in the Tree View, the only caveat is that it won't collapse the Tree View element after losing focus in the piStar.

4.4 Decomposing a Goal Model with MutRoSe

The MutRoSe generates two types of decompositions:

- The default.
- The iHTN.

For generating both of these, it's necessary to have all the file paths inside the configuration.json and the settings.json relative to the workspace root. After that, if the all the definitions are correct, it should be possible to decompose the Goal Model by using the Goal Model's context menu, represented in the figure 7.

4.5 Known bugs & workarounds

Since the extension is currently under the early stages development, there're a few known bugs. These are mapped and have the respective workaround listed in the extension README.md, which can be accessed in the Visual Studio Code's extensions tab.

5 Final Considerations