*i** Modules: a jUCMNav Implementation[†]

Daniel Colomer, Xavier Franch

Software Engineering for Information Systems Research Group (GESSI)
Universitat Politècnica de Catalunya (UPC)
c/ Jordi Girona 1-3, 08034, Barcelona, Spain
{dcolomer, franch}@essi.upc.edu
`http://www.essi.upc.edu/~gessi`

Abstract. When building large-scale goal-oriented models using the *i** framework, the problem of scalability arises. Modules have been proposed to structure *i** models into reusable and combinable fragments. In this work we present an implementation of the module concept over the jUCMNav tool.

Keywords: *i**, iStar, modules, jUCMNav.

1 Introduction

One research challenge for the *i** community is to make *i** models more manageable and scalable. In [1] we defined a theoretical approach for adding modularity facilities to the *i** metamodel in a loosely coupled way, also tailored to a particular domain, namely the modularization of goal models for data warehouse schemata [2]. In this work, we present an implementation of the general concept of module as an extension of the jUCMNav 4.2.1 plug-in. The tool may be downloaded from http://www.essi.upc.edu/~gessi/mod_extension/resources.html where a basic tutorial in the form of user's manual may be found, as well as details on the metamodel used.

jUCMNav is a graphical editor and an analysis and transformation tool for the *User Requirements Notation* (URN). URN is intended for the elicitation, analysis, specification, and validation of requirements. It combines modeling concepts and notations for *goals and intentions* (with GRL) and *scenarios* (with UCM). We will focus on the GRL notation because of its *i**-based nature. It is a graphical language for supporting goal-oriented modelling and reasoning about requirements, especially non-functional requirements and quality attributes. It provides constructs for expressing various types of concepts that appear during the requirement process. GRL has its roots in two widespread goal-oriented modeling languages: *i** and the NFR Framework. Major benefits of GRL over other popular notations include its integration with a scenario notation and a clear separation of model elements from their graphical representation, enabling a scalable and consistent representation of multiple views/diagrams of the same goal model.

[†] This work has been partially supported by the Spanish project TIN2010-19130-C02-01.

2 Module Implementation

We extended the last jUCMNav metamodel available (`URN_23.mdl`), see Fig. 1. In order to guarantee later graphical and usability efficiency we made some decisions that differ from the model presented in [1]. A `State` pattern was implemented in order to allow dynamic state (i.e., type) changes during module definition. Then a new attribute was added to the existing `IntentionalElement` definition representing the notion of root (for graphical purposes) so the relationship `root` introduced in [1] was no longer needed. Constraints such as multiplicities were assigned to integrity constraints due to modeling software limitations. The implemented structure also facilitates later extensions such as new module definitions.

Fig. 2 shows a snapshot of module in jUCMNav. In the left-hand side we may find module references. They have two different functionalities: to inform the user about the nature of the module that is currently being edited and about the different sources from which the current module was obtained (they are only shown if the module was obtained as a result of one or more module operations) for traceability purposes. This second type of references is shown in green background.

In [1], constraints are proposed for ensuring the structural correctness of the different types of modules. Both general and particular constrains over SR and SD Modules have been implemented as Static Semantics checking rules (see Fig. 3).

A crucial point of the approach in [1] is that of module operations. *Combination* and *Application* are somehow similar, so we decided to implement both of them as a single abstract operation. When this abstract operation is applied to an undefined module, *Module Application* will be executed and then a list of dependency matches is needed. When applied to any type of module (different from a undefined module) *Module Combination* will be executed. In this case a simple merge is carried out and the resulting module is created. Both operations were implemented as part of the set of Eclipse navigator view functionalities (see Fig. 4). A simple merge algorithm is used and so some limitations appear (see Section 3).

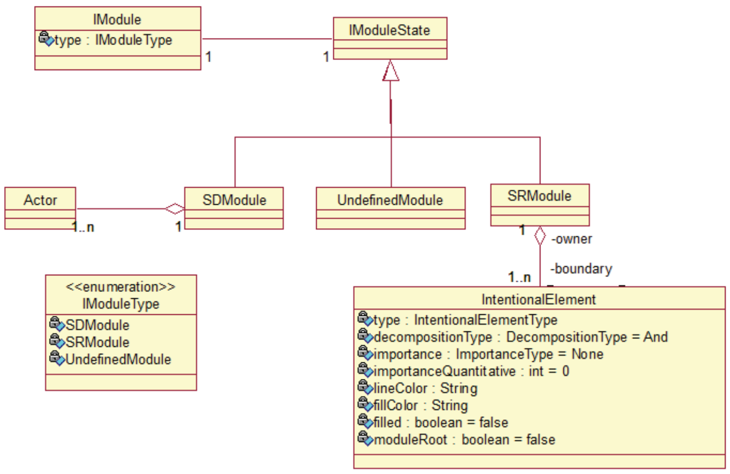


Figure 1. The metamodel part related to modules as implemented in the jUCMNav extension.

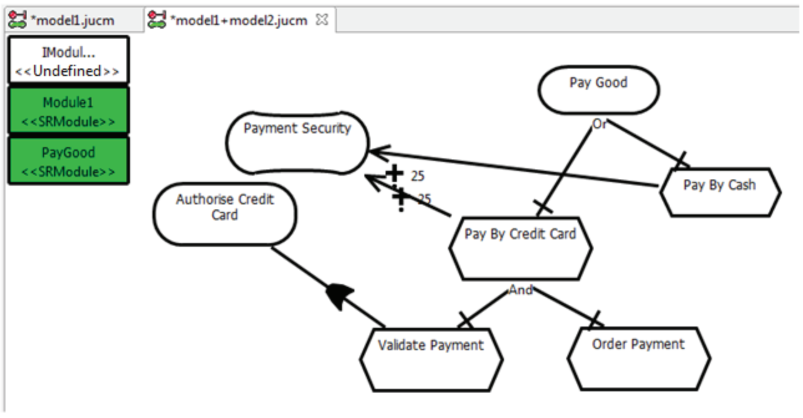
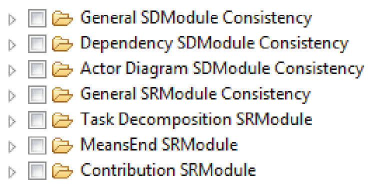


Figure 2. Module definition in jUCMNav extension.


General SDModule Consistency
Dependency SDModule Consistency
Actor Diagram SDModule Consistency
General SRModule Consistency
Task Decomposition SRModule
MeansEnd SRModule
Contribution SRModule

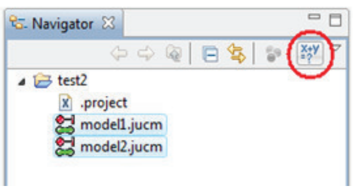


Figure 3. Static Semantics checking rules. Figure 4. Module operations in Eclipse.

3 Limitations and Future Work

jUCMNav makes a clear separation of model elements from their graphical representation, enabling a consistent representation of multiple diagrams of the same goal model. This multiple-diagram representation is not covered in [1] and although the metamodel extension was made taking this into account, the current solution only supports files with a single diagram. Future work aims at solving this limitation.

Extensibility has been a goal. New module specializations can be easily added by extending the current implemented hierarchy. Functionalities for collapsing and expanding are yet to be implemented. Module operation constraints can also be easily added through the `ModuleCombinationAction` class. Last, there are two different ways of extending module restrictions: 1) jUCMNav offers the possibility to add, remove and edit current integrity constraints through Eclipse's preferences view; 2) new OCL constraint packages could be easily added to the plug-in by incorporating their XML description and extending the default integrity constraint loader.

References

1. X. Franch: "Incorporating Modules into the *i** Framework". CAiSE 2010.
2. A. Maté, J. Trujillo, X. Franch. "A Modularization Proposal for Goal-Oriented Analysis of Data Warehouses using *i**". ER 2011.