An Approach to Generate Tools for i* Languages

Josias Paes, Jaelson Castro
Centro de Informática
Universidade Federal de
Pernambuco (UFPE)
Recife, Brazil
{jpsj2, jbc}@cin.ufpe.br

Carla Silva
Depto. de Ciências Exatas/ CCAE
Universidade Federal da
Paraíba (UFPB)
Rio Tinto, Brazil
carla@dce.ufpb.br

Emanuel Santos, Carlos Lima
Centro de Informática
Universidade Federal de
Pernambuco (UFPE)
Recife, Brazil
{ebs, cdql}@cin.ufpe.br

Abstract—i* is goal-oriented requirement modeling framework with an increasing use in industry and academy. One of the main challenges in adopting this framework, is the diversity of variants/dialects of the i* modeling language. These variants were created by different research groups to address their particular purposes and are supported by specific CASE tools. Considering them, it is possible to identify a set of common modeling elements, as well as a set of different modeling elements. We understand that these variants are part of the same family of i* based languages. Hence, a specific i* based language can be obtained from an i* language family, similarly to a product obtained from a software product line. To define the core assets of such i* language family, we identify their common and variable characteristics by comparing various i* based languages. From this comparison, we propose a core metamodel for the i* language family and a process to configure it to generate graphical editors for any i* based language. As a running example, we show how to derive the metamodel for the Aspectual i* modeling language and generate a graphic editor for this language.

Keywords—Goal Oriented Requirements Engineering, Software Product Lines, Variability, Metamodeling

I. INTRODUCTION

The i* framework [1] is a goal oriented requirements engineering approach with rich semantics to describe different social and intentional dependencies between actors in their organizational environment, as well as functional and non-functional requirements of a system-to-be.

Due to its large use in industry and academy [2, 3, 4], different research groups have developed their own variant/dialect of the i* modeling language, including their own supporting tools. Existing variants include the i* Wiki [5], GRL [6], Tropos [7], i*-c [8], Aspectual i* [9], among others. In this scenario, obtaining a suitable tool support for each variant has high development cost. Moreover, according to [10], this diversity of versions of the i* language can cause:

- Division of effort, since each research group will focus on the development of tools to support its own dialect;
- Semantic mismatches among the writers and readers of a particular i* model;
- Inhibition of i* adoption by new users.

These problems can also occur with any modeling language that does not have a standard version.

Different i* variants have common and variable modeling elements. Variability is a concept widely explored for Software Product Lines (SPL) [11]. A SPL is a set of software-intensive systems sharing a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets [12]. The core assets are artifacts or resources specifically designed to be reused in the products of a SPL [12]. This paradigm supports [11]: (i) development time reduction, once core assets are produced and components are created and reused, with flexibility; (ii) development cost reduction, due to the systematic reuse, with focus on products development productivity; and (iii) high quality products, once reused artifacts could be reviewed and tested more carefully since they are present in more than one product, providing higher chances to detect errors.

Based on SPL principles, we have identified the common and variable modeling elements of several i* based languages. As a result, in this work, we present a core metamodel for the i* modeling language. Using this core metamodel, it is possible to derive a specific product (i.e. a specific i* based language). Therefore, we can benefit from the high productivity provided by the SPL paradigm to obtain a set of metamodels for specific i* based languages and generate, automatically, a graphical editor for these languages. To configure these new products, we developed an approach called AGILE, briefly presented in [11]. Our approach aims to increase the abstraction level in the metamodeling activities related with the development of graphical editors for i* based languages. The metamodels are based on the GMF [13] technology. To illustrate the use of the AGILE Approach in detail, in this paper we use the Aspectual i* [9] as running example.

This paper is organized as follows. Section II presents an overview of the i* framework. Section III defines the core assets for a family of i* based languages. Section IV presents the AGILE approach to guide the definition of metamodels and the automatic configuration of graphics editors. Section V compares our work to related works and section VI discusses our contributions and points out to some future works.

II. THE FRAMEWORK i*

The i* framework [1] is a goal oriented requirements engineering approach used to support organizational modeling and analysis.
It is centered on defining relationships between strategic actors, also called actor dependencies. Actors represent stakeholders involved with the system and the system itself. Actors depend on each other to achieve their goals, to accomplish their tasks, and to obtain or to share needed resources. These dependencies between actors represent social networking that models the system and the environment where it is inserted. Two models are used to represent the different levels of abstraction in the i* framework: the Strategic Dependency (SD) model and the Strategic Rational (SR) model.

The SD model represents through dependencies the network of intentional strategic relationships among actors. A set of nodes and links are instantiated, where the dependent actor is called depender, while the actor in charge to achieve the dependency is called dependee (see Figure 1(a)).

The SR promotes a structural representation of nodes (goal, task, resource and softgoal) and links (means-end, task-decomposition, and contributions) that work together to detail the “rationale” of an actor. The intentionality of each actor is identified and represented inside its boundary (see Figure 1(b)).

III. DEFINITION OF THE CORE ASSETS FOR A FAMILY OF i* BASED LANGUAGES

In order to identify the common and variable modeling elements present in i* based languages, we performed a comparison between the Original i*, proposed by Eric Yu [1], and some other i* based languages consolidated in academia, such as: i* Wiki [5], GRL [6], Tropos [7], i*-c [9] and Aspectual i* [7]. The details of the comparison can be found in [14]. It was based on the following classification for the modeling elements: actors, intentional elements and relationships.

After the comparison among the languages, we identified differences in the specialization of actors. For instance, the Original i* has four possible actors (Actor, Agent, Role, and Position) while the GRL has just one (Actor). Moreover, several differences in the use of intentional elements can be pointed out. Comparing the Original i* with Tropos, we found changes of concepts from Task to Plan, and from Goal to Hardgoal. Similarly, the i*-c modifies the set of intentional elements by adding new ones (Cardinality Task and Cardinality Resource). Finally, there are relationships that were included in the languages to satisfy specific needs. For example, Tropos included new types of elements decomposition (And- Decomposition and Or-Decomposition); likewise, GRL included new links, such as links of correlation, and links of quantitative and qualitative softgoal analysis.

We also performed a constraint-level comparison (see [14]), because there are some languages that contains the same modeling elements but differ in their use (e.g., Original i* and i* Wiki). Thus, it is important to define a strategy to deal with both metamodeling and constraints definition for i* based languages.

Considering the principles of Software Product Lines, the set of common modeling elements identified should be treated as domain variation points. In this case, variation points are places where some modeling elements vary. For instance, in the Original i* there are four actor variants while the GRL [6] has only one. Hence, the modeling element actor is a point of common variation between these two languages. Variant is the representation of some modeling element that will cover a variation point. In [14], there is a list of the variation points and variants identified from the comparative analysis performed previously.

Finally, identifying the variation points and variants among the compared languages, we use a feature model to represent, in a structured way, the variability present in i* based languages. Besides, this modeling technique is of easy comprehension to represent variability. Figure 2(a) presents a feature model with cardinality [15] where a mandatory feature is represented by a filled circle; an optional feature by an empty circle; and grouped features by an arch. The grouped features can be modified by an Or, XOr, or by using cardinality. The cardinality (e.g., [0..*], [1..3], etc.) of a feature is an interval with the number of times that a feature and it sub-features can be present in a product [15].

When a language is built based on the i*, it is necessary to choose which elements will be inherited from the i* language. Our approach just allows inheritance from
the Original i* or Wiki i*, since they have the same modeling elements and differ just in the constraint level (see [14] for further details). Our approach allows the selection of the elements to be inherited from the i* language as if we were selecting them from the feature model presented in Figure 2(a), but it increases the abstraction level of this activity by using a CASE tool (to be detailed in section IV.A).

A. Core Metamodel

The comparison performed among some i* based languages and the identification of the variation points of the i* based languages family (considering the multiple variants) have lead us to build a core metamodel for that family. The features marked with X in Figure 2(a) will be a metaclass in the core metamodel. It was described using the language Ecore [16], as presented in Figure 2(b).

Differently from other metamodels proposed for the i* language [2, 8, 18], which are intended to represent all the modeling elements present in the language, the core metamodel was built to support the flexible addition and removal of modeling elements, enabling an easier creation of i* based languages.

To do so, this metamodel unifies many i* based languages, once any of these languages could be based in a minimal set of modeling elements (i.e. the core of the i* language). From these main metaclasses, it is possible to classify all modeling elements of the existing i* based languages, as well as to create new elements for a new i* based language, by choosing the most general metaclass that matches with the purpose of this new element and establishing an inheritance relationship between them.

The information contained in the core metamodel is explained, as follows:

- **Model** - It represents the region where the elements will be inserted, i.e. the repository where the NodeObjects and the Relationships are created and constrained. In the Original i* language, two models are allowed: SD and SR models.
- **NodeObject** - It is an abstract metaclass that represents all modeling elements that may be interpreted as a node. A NodeObject can be a Compartment or an IntentionalElement. This strategy reduces the metamodel complexity because it helps in the definition of the sources and targets of a link.
- **Compartment** - This class is capable to represent actors. Several Compartments could be inserted into a Model and several Intentional Elements could be inserted into a Compartment or into a Model. The CompartmentType enumerator

Figure 2. (a) Model showing the main modeling constructors for a new i* base; (b) i* core metamodel to support variability.
represents all variants of i* actors supported by the language.

- Intentional Elements: It is an abstract metaclass used to represent the existing intentional elements in the i*. This metaclass represents the elements that will be placed into the Model and/or the Compartment.

- ElementMC: This metaclass is a specialization of the IntentionalElement metaclass to represent elements that can be created both inside the Model and the Compartment.

- Relationship: Existing relationships in the i* have several features distinct from each other, in other words, do not share their constraints (source and target elements), their attributes and composition in the model (location can be inserted in the model). From this we believe that relationships should not be expressed directly in the core metamodel but must be configured/created by the users. Thus for each new relationship created by a user in the core metamodel, a new metaclass must be created to represent it. To provide the relationship semantics of these metaclasses created, e created as a solution in the core metamodel an abstract metaclass called Relationship to represent any new relationship. For this, the creation of each new relationship a new metaclass must be created to represent him and this new metaclass must extend the Relationship metaclass.

IV. THE AGILE APPROACH

The AGILE approach (Automatic Generation of i* Languages) offers an automation of the process to create graphical editors for i* based languages. To support the approach, a CASE tool is integrated to the GMF (Graphical Modeling Framework) [11], which is a framework of Eclipse platform that simplifies the development of graphical editors. The GMF is based on the Model-Driven Development (MDD) [17] paradigm that prioritizes the development of models instead of the development of algorithms.

Metamodels (also called domain models in the GMF) are essential for the development of graphical editors (see the process described using the Business Process Modeling Notation [18] (Fig 3(a)). They define the syntax of a language. The definition of the metamodel is performed manually by developers and the GMF models (i.e., Tool model, Graphical Model and Mapping Model) can be automatically generated according to the metamodel.

These models complement the language metamodel in the development of a graphical editor. They need to be consistent with the language metamodel, since the decisions done during the metamodel definition can affect their configuration. The use of an enumeration in the metamodel, for example, requires several manual interventions in the creation of these models to maintain consistency. These manual interventions could slow down the development process as well as to introduce errors and inconsistency among the artifacts.

Considering this limitation, we propose the automation of some activities of the GMF process (see Figure 3(a)), by defining a new process (Figure 3(b)) that substitutes the GMF process.

As can be seen in Figure 3(b), the manual development activities were replaced by new automated ones: (i) Configuration the i* base; (ii) Creation and configuration of new modeling elements; and (iii) Automatic generation of GMF models (Tool model, Graphical Model and Mapping Model). The process automation is supported by a tool called AGILE Tool, that can be found in [14].

The AGILE approach aims at improving productivity of the activities related to the creation of metamodels. The creation of a new metamodel is done by specializing the core metamodel proposed in Section III.

---

Figure 3. (a) Process for creating a graphical editor using GMF; (b) Process for creating a graphical editor using the AGILE approach.
Our approach deals with the definition of the language metamodel and the configuration of GMF models in a higher abstraction level. As a result, the effort required to develop graphical editors for i*-based languages is reduced.

The new activities will be detailed in the next sections using as running example the Aspectual i* language [9]. This language is rich enough to illustrate most of the AGILE approach capabilities. Based on the Original i*, the Aspectual i* requires more than just configuring the original i* base, but also modeling elements of all types (i.e., Compartments, Elements and Relationships). In doing so, we can demonstrate the reuse of the i* metaclasses, through the configuration of the i* base, the creation, configuration and insertion of new metaclasses, and the treatment of constraints for the metamodel.

Figure 4 shows an example of an Aspectual i* model, illustrating its modeling elements such as the AspectualActor (e.g., Payment Processor, Security Manager and Adaptability Manager), the link CrosscuttingME (between the goal Confirm Payment and the goal Get Bought Items), and the link CrosscuttingTD (between the task Update Encryption Strategy and aspectual actor Adaptability Manager).

A. Configuring the i* base

The core metamodel proposed in Section III.A (Figure 2(b)) is not suitable to develop a graphical editor as is, because it does not represent all characteristics of a language. Actors, intentional elements or relationships are still not concrete in this metamodel, for example, the enumerations of compartments and elements are empty.

To define a specific i* based language, first it is necessary to choose the elements inherited from the Original i*. This step is the configuration of the i* base and can be done in the main screen of the AGILE Tool (Figure 5).

Using the AGILE Tool, it is possible to choose the elements that are to be inherited from the i* language by selecting the correspondent items in the configuration screen. It implies in the automatic inclusion of the chosen elements in the core metamodel, resulting in a metamodel with elements inherited from the i* language (Figure 6).

The configuration of the core metamodel, for the Aspectual i* language [9], was performed as follow:

- Actors – The Compartment metaclass (Figure 6(a)) represents actors present in the language. Thus, all types of actors selected are to be included in the CompartmentType Enumeration;
- Intentional Elements – Intentional elements (Figure 6(b)) that created both in a Compartment and a Model, are to be included in the ElementMCType Enumeration;

Figure 4. Medi@ example in Aspectual i* adapted from [7].
• **Dependency Link** – To represent the dependency link, we create a metaclass called `DependencyLink` (Figure 6(c)) with an attribute called “type” of the type `DependencyLinkType Enumeration`. Since we have several types of dependency, we deal with this variation using an Enumeration. A dependency link is a specialization of the `Relationship` metaclass that can be used just in a model, thus, the `DependencyLink` metaclass is composed in the `Model` metaclass. Finally, there is also the definition of how the link can be used. Every link contains a source (the element from where it begins) and a target (the element to where it ends). In the dependency link, both the source and target elements can be any type of actors or any type of intentional elements. Thus, the source and target of the `DependencyLink` metaclass are associated with the `NodeObject` metaclass. The attribute name was created to represent the label existing in some relationships. For example, the `contribution` link have labels that have distinct semantic from each, such as: +, -, + - and other (Figure 1(b)).

• **Actor Link** – The `ActorLink` metaclass (Figure 6(d)) was created and inherits from the `Relationship` metaclass. `ActorLink` is composed in the `Model` metaclass because it is used only in the model. It has an attribute “type”, of the type `ActorLinkType Enumeration`, to represent the sub-types of the actor link. The source and target, of the `ActorLink` metaclass, are associated with the `Compartment` metaclass since this relationship occurs only between actors;

• **Contribution Link** – The same rationale applied to dependency and actor links is applied to contribution links. A metaclass called `ContributionLink` (Figure 6(e)) is created and inherits from the `Relationship` metaclass. It has an attribute “type” of the type `ContributionType Enumeration` to represent the possible types of contribution. The `ContributionLink` metaclass is composed in the `Compartment` metaclass because contribution links can be used only inside the boundaries of an actor. The source and target, of the `ContributionLink` metaclass, are associated with the `IntentionalElement` metaclass, since it occurs only between intentional elements;

• **Means-End and Task Decomposition Links** – A metaclass for each link is created: `MeansEndLink` and `TaskDecompositionLink` (Figure 6(f)). These metaclasses are composed in the `Compartment` metaclass because they relate only intentional elements contained in the actor’s boundary. These metaclasses inherit from `Relationship`, and their sources and targets are associated with the `IntentionalElement` metaclass.

This metamodel is still not complete. It is necessary to define a set of constraints not included in the metamodel, to do so we use OCL (Object Constraint Language) [19]. OCL is a formal language to specify restrictions over models.
The AGILE Tool provides support to the automatic generation of OCL code according to the configurations made by the developer for the metamodel. Still in the main screen of AGILE Tool, it is possible to define constraints to the links present in the language. Through a graphical interface, the developer can select the possible associations of a link and the corresponding OCL code is automatically generated (Figure 7). These codes are composed automatically into metamodel of GMF.

B. Creation and configuration of new modeling elements

Next activity allows the creation and configuration of new modeling elements. It makes possible to define the characteristics of a new modeling element, such as: where the element is composed, its classification, its attributes and constraints.

To create a new modeling element we use the setup screen for new modeling element (Figure 7). In the screen 1 it is possible to define where the new element can exist, in other words, where it can be composed (in the model, compartment or both) (Figure 7(a)) as described above in section III. If this element is of type Link, still on the same screen, is possible to define its cardinality (Figure 7(b)) as well as its possible source and target elements (Figure 7(c)). Furthermore, the user can insert any kind of attributes needed to configure your new element, these attributes can be of type enumeration (Figure 7(d)).

For an element of type Link the user can define the constraints involved (only if the element is of type Link) through the button Constraints New Relationship (Figure 7). With this action the user has access to the screen 2. In this screen you can select the elements that may be sources and targets of the link in question. Through this configuration will be automatically generated OCL code regarding the restrictions that will be attached automatically in the configuration models of the GMF.

Now, we are going to include modeling elements specific for the Aspectual i* language [9], using the AGILE Tool. The particularities of this language can be resumed as: (i) the Aspectual Actor; (ii) the crosscutting concerns and (iii) crosscutting relationships. The composition of these new modeling elements in the metamodel is described as follow:

- **Aspectual Actor** – An Aspectual Actor acts as a container of modeling elements, similarly to the actors of the Original i* (Figure 8 (a)). The modeling element called Aspectual Actor is defined as a Compartment. Therefore, the Aspectual Actor is represented, in the metamodel, as a literal of the CompartmentType Enumeration. As a result, the Aspectual Actor will inherit all configurations already defined previously for a Compartment, but specific constraints shall be defined afterwards, such as the constraints of actors’ links that can not involve Aspectual Actors;

- **Crosscutting Concerns** – Elements called crosscutting concerns are modeling elements that shall be created inside an Aspectual Actor. The possible types of crosscutting concerns are: Goal, Task, Resource and Softgoal. Although having the same denomination of the Intentional Elements present in the Original i*, the crosscutting concerns have a set of constraints when compared to the Intentional Elements: (i) they cannot be related to dependency links; (ii) they can only be created inside an Aspectual Actor, and; (iii) they are related to other crosscutting concerns and to Intentional Elements through crosscutting relationships. As these elements can just appear inside Aspectual Actors, they do not inherit from any element of the configured metamodel (Figure 8(b)) and a new metaclass must be specified to represent elements that can be created only inside a Compartment. This metaclass, called ElementCompartment, has an attribute “type” of...
Figure 7. Creating crosscutting relationships of the Aspectual i* and setting constraints

type ElementCompartmentType Enumeration, representing all elements that can be created only inside Compartments. All types of crosscutting concerns (i.e., Goal, Softgoal, Task, and Resource) were included in this enumeration. All constraints will be included afterwards;

- CrosscuttingME and CrosscuttingTD – The CrosscuttingME (or Crosscutting Meand-Ends) and the CrosscuttingTD (or Crosscutting Task Decomposition) are relationships that have, as source element, a ElementCompartment or an AspectualElement, but can have, as target element, ElementCompartment, AspectualElements, IntentionalElements and Actors (Figure 8(c)). A metaclass is created to represent each link (CrosscuttingME and CrosscuttingTD). They are specializations of the Relationship metaclass and are composed into Compartment and Model. The definition of how the elements can be connected is done through constraints, as can be seen in Figure 7-1. The screen presented in Figure 7 shows the definition and configuration of a new relationship (i.e., CrosscuttingME). It allows defining where the relationship is composed (Figure 7-1(a)), its cardinality (Figure 7-1(b)), its sources and targets (Figure 7-1(c)), and, if required its attributes. Figure 7-2 shows the screen where are defined the constraints over the sources and targets (Figure 7-2(a)), and the OCL code automatically generated (Figure 7-2(b));

- Crosscutting Contribution – To include this element, we follow the same rationale used for including the CrosscuttingME and the CrosscuttingTD relationships. Thus, the CrosscuttingContribution metaclass is created (Figure 8(d)). The CrosscuttingContribution can be of several types (e.g., Help, Hurt, etc.), similarly to the contribution of the Original i*. Therefore, this metaclass has an attribute called “type” of type ContributionType Enumeration.

As a result, we obtain a metamodel for the Aspectual i* (Figure 8). More details about how the AGILE Tool was implemented to generate a metamodel for a new i* language, based on the core metamodel, can be found in [14].

C. Automatic generation of configuration models of the GMF

This is the last configuration step of AGILE approach. In summary, in this step occurs the generation of the GMF models (Tool model, Graphical Model and Mapping Model). All project of GMF requires a metamodel that describes the structure of the language to allow the generation of the other models required by the GMF. This metamodel was created and configured in the previous steps.

In the GMF process (Figure 3(a)), even with a well-defined metamodel and all the structure provided by the GMF, such as the use of enumerations, many project configurations need to be done manually by the developers (e.g., creation of graphical representation of modeling elements, definition of constraints, etc.), and the definition of the configuration models are not a trivial task.

In the AGILE approach, these configurations are performed automatically. As the modeling elements are defined in the metamodel, their configurations are automatically reflected in the GMF models. When the metamodel is finally created, the GMF models can be generated with one click in the button Create Files of the AGILE Tool. After that, the final configurations are performed and the code of the graphical editor can be generated using the GMF Dashboard, finishing the graphical editor generation.
It is worth mentioning that the automatic creation of graphical representation of basic modeling elements of i* is supported by the AGILE Tool. However, if a new modeling element is included in the new i* based language, its configuration in the Graphical Model is automatically generated but the creation of its graphical representation needs to be done manually.

V. RELATED WORKS

In studies realized by Ayala et al. [20], a structural comparison between three variants of i* (Original i*, Tropos, GRL [1, 22, 23]) was performed with the aim to build a generic conceptual model used as reference framework of these three variants. We did a structural comparison including three more languages (i.e., Wiki i*, i*-C, and Aspectual i*) in order to identify the existing variability among them.

The work presented in [2] proposed other metamodel for i* language to be used as a reference between research groups. The structure of this metamodel makes it difficult to insert new elements due to the metamodeling strategies used by the authors. Although both metamodels proposed by Ayala et al. [20] and Franch [2] support the inclusion of new elements, their metamodels were not planned to support the inclusion of new generic elements as our metamodel does. These elements represent the basic constructors that any i* based language should contain and are required to properly support extensibility and variability.

In relation to the configuration approach, Amyot [6] proposed a profile to insert, in the GRL, concepts of the original i*, such as roles, and agent positions, through a tool called jUCMNav. Our work goes one step further because we benefit from a shared base of artifacts in the pre-defined tool. It is possible to create a new language based on i* (may be even GRL) by reusing this shared base of common features, by incorporating new modeling element and by defining constraints.

VI. CONCLUSION AND FUTURE WORKS

The existence of several i* based languages is a problem encountered for those who want to understand the expressivity provided by a goal-oriented language. However, from the analysis conducted in this work, it was possible to notice that the i* based languages have many common modeling elements and they could be
reused among the different versions by means of SPL principles.

This paper presents the definition of a core metamodel for *i* based languages, in consonance with the Software Product Line Engineering area, in order to explicit the common modeling elements of many *i* based languages and to make it easier the design of new *i* based languages, as well as their corresponding modeling tool. The AGILE approach is supported by the AGILE Tool to aid the configuration of metamodels and the automatic generation of graphical editors for a family of *i* based languages. As a running example, we used the Aspectual *i* through which we have shown how the core metamodel could be configured to generate a variant of *i* and its tool support. Using our approach on this example, we could observe a reduction of effort for developing graphical editors for *i* based languages.

As future work, we intend to: (i) perform new case studies to identify any possible limitations in the approach that couldn’t be identified through the presented case study; (ii) improve the AGILE Tool to support the creation of the graphical representation of new modeling elements using a graphical user interface; (iii) generate SR models from SD models and vice-versa, through model transformations; (iv) promote integration between graphical editors supporting different *i* based languages; and (v) extend the approach to support other domain specific languages.

ACKNOWLEDGMENT

This work was supported by CNPq, CAPES and FACEPE Brazilian agencies for research support.

REFERENCES