Incorporating Modules into the *i** Framework*

Xavier Franch

Universitat Politècnica de Catalunya (UPC)
c/Jordi Girona, 1-3, E-08034 Barcelona, Spain
franch@lsi.upc.edu

Abstract. When building large-scale goal-oriented models using the *i** framework, the problem of scalability arises. One of the most important causes for this problem is the lack of modularity constructs in the language: just the concept of actor boundary allows grouping related model elements. In this paper, we present an approach that incorporates modules into the *i** framework with the purpose of ameliorating the scalability problem. We explore the different types of modules that may be conceived in the framework, define them in terms of an *i** metamodel, and introduce different model operators that support their application.

Keywords: goal-oriented models, *i**, modularization, modules, scalability.

1 Introduction

The *i** framework [1] is currently one of the most widespread goal- and agent-oriented modelling and reasoning frameworks. It has been applied for modelling organizations, business processes, system requirements, software architectures, among others.

Several challenges have been identified with the goal of overcoming different reported problems (see e.g. [2]). Among them, one of the most important issues is to make *i** models more manageable and scalable by defining modularity constructs. This paper presents a proposal for converting *i** into a modular language. This is a basic notion for any language expected to create big models as *i** is, but it has been not yet proposed for *i** except for some proposals of new constructs in the language. Instead, we do not propose to extend the language, but to add modularity facilities to the metamodel of *i** in a loosely coupled way.

The rest of the paper is structured as follows. Section 2 provides the background on case studies and empirical evaluation as well as related work. Section 3 presents an *i** metamodel to be used for reference the rest of the paper. Section 4 proposes the different types of modules for the *i** language. Section 5 defines two module operations, combination and application. Section 6 includes some discussion about the presented work. Finally, Section 7 states the conclusions and future work.

Basic knowledge of *i** is assumed in the paper, see [1] and the *i** wiki (http://istar.rwth-aachen.de) for a thorough presentation.

* This work has been partially supported by the Spanish project TIN2007-64753.

B. Pernici (Ed.): CAiSE 2010, LNCS 6051, pp. 439–454, 2010.

2 Background and Related Work

2.1 Use of *i** in Industrial Projects

Some industrial experiences on the use of *i** have been documented. In [3], a report is presented about three air traffic management projects in which *i** was applied to model requirements for new socio-technical systems. Among other remarks, the problem of managing large SR models is highlighted. This large size cannot be lowered due to the absence of structuring mechanisms.

In [4] an experience is reported about how to support the continuous alignment of corrective software maintenance processes with the strategic goals of a Software Design Maintenance Organization at Ericsson Marconi Spa. The authors used a model slicing technique to break the model into pieces. However, this partition was done by hand since the authors did not bring the notion of module into the *i** framework, with the inherent drawbacks of proceeding this way.

In [5], the authors report the use of *i** for architecting hybrid systems in two industrial experiences at the Etapatelecom Ecuadorian company. One of the cornerstones of the proposed method is the reconciliation of the individual models that the different stakeholders build. The models have not been encapsulated into modules due to the absence of this capability in the *i** framework, making this process more difficult to implement. Also reusability is highlighted as a key concept, but at the time being it has been supported simply by manual management of the models.

As a summary of these cases, the existence of modularity constructs in the *i** framework's language could help to ameliorate some of the problems mentioned.

2.2 Empirical Evaluation of *i**

At the time being, the only in-depth empirical evaluation of the *i** framework reported in the community we are aware of is [2]. The authors propose a feature-based evaluation scenario and they assess *i** with respect to these features in the light of three industrial projects. The results of the analysis is that *i** supports well the expressiveness and domain applicability features, provides some insufficient support to the refinement, repeatability, complexity management and traceability features, and does not support at all modularity, reusability and scalability features. It is also stated that reusability and scalability have a causal relationship with modularity, which means that providing some solution to the latter feature impacts on the former. We may conclude then that providing a solution to the modularity problem can be a topmost productive effort in terms of improving the evaluation of the *i** framework.

2.3 Existing Approaches to Model Modularity

Although we have stated that the *i** framework does not include modularity constructs, there are some lines of research addressing this issue. The two most remarkable contributions at this respect are the incorporation of aspects and services into *i**.

Concerning the first point, a line of research [6][7] proposes the use of aspects for modelling cross-cutting concerns. Although it is true that separation of concerns may help to structure the *i** models, the proposal still does not include modules to support

the basic concept of stepwise refinement. Also, the addition of aspects into *i** results in a framework that is more complex and may eventually require a steeper learning curve. Therefore we do not consider this proposal as a general solution for the modularization problem.

In [8], the concept of service is incorporated into the *i** framework. This type of modularity unit is closer to the concepts managed in the domain (i.e., business services) and from this point of view fits better than aspects to the natural stepwise refinement process. However it is true that this particular proposal introduces a lot of complexity to the framework, with the fundamental concepts of "service" and "process", and also with the configuration of services inside SR boundaries using a variability-like model with mandatory and optional features combined in several ways.

In this work, we have preferred to search for other solutions that do not require the addition of new constructs into the *i** framework and that are basic enough to be bound to different concepts in different methods.

3 The *i** Metamodel

The *i** community has defined several dialects of *i** that add new constructs for particular purposes (e.g., trust constructs, temporal constructs, …), remove some that are not of primary interest for their purposes (e.g., types of actors) or modify some conditions of use (e.g., which type of intentional element is a valid end for a means-end relationship). In [9][10] we provide a thorough analysis of these variations. Also, in [11] we may find a survey of variations used by the community in several proposals. Variations occur both in Strategic Dependency (SD) and Strategic Rationale (SR) diagrams. This diversity makes advisable to identify which constructs we do consider.

Following our previous work [9][12], we propose in this section an *i** metamodel that is used as reference in the rest of the paper. The metamodel is built under the principles of generality (i.e., trying to host as most as possible the existing variations), extensibility (for incorporating future extensions) and suitability for modularity (being this the goal of the paper). As a result, we remark here the most important innovations with respect to the metamodel we have proposed so far:

- We have added an abstract *Link* class that holds binary relationships among *Nodes* (being *Node* the general concept of *i** model element). This *Link* class generalises the concepts of link among actors (e.g., is-A relationships), among SR elements (e.g., means-end link) and among dependums (and thus dependencies). These three particularizations are considered abstract links themselves specialized into the available types of links in a uniform way. As a modelling decision and for clarity purposes, we provide the conditions that each particular type of link may impose (e.g., *Covers* is a many-to-many association from position *Actors* to role *Actors*) as OCL constraints instead of graphically (although we show the resulting dependencies to make evident this relationship).
- For illustration of dependum links, and considering the goal of this paper, we have added a class for the *Support* dependum link as introduced in [13]. A dependum *d1* supports another dependum *d2* when *d1* has been introduced in a later development stage than *d2* in a way that *d1* provides details about the form that *d2* has.

This proposed construct keeps track of refinement of dependums in a similar way than means-end and task-decomposition links do for SR elements.

– Due to the objectives of the paper, it will become necessary to work with dependencies in which either the depender or the dependee are temporarily unknown. We have modified the core of the dependency concept to support this need, by defining *Dependency* as a class with two specializations. Again for clarity, we have preferred to model the restrictions on each subclass with OCL constraints instead of graphically (i.e., instead of redefining the associations).
– We have included a class *Model* that records the *Nodes* that compose an *i** model (and transitively also the links, i.e. those that connect two nodes from the model).

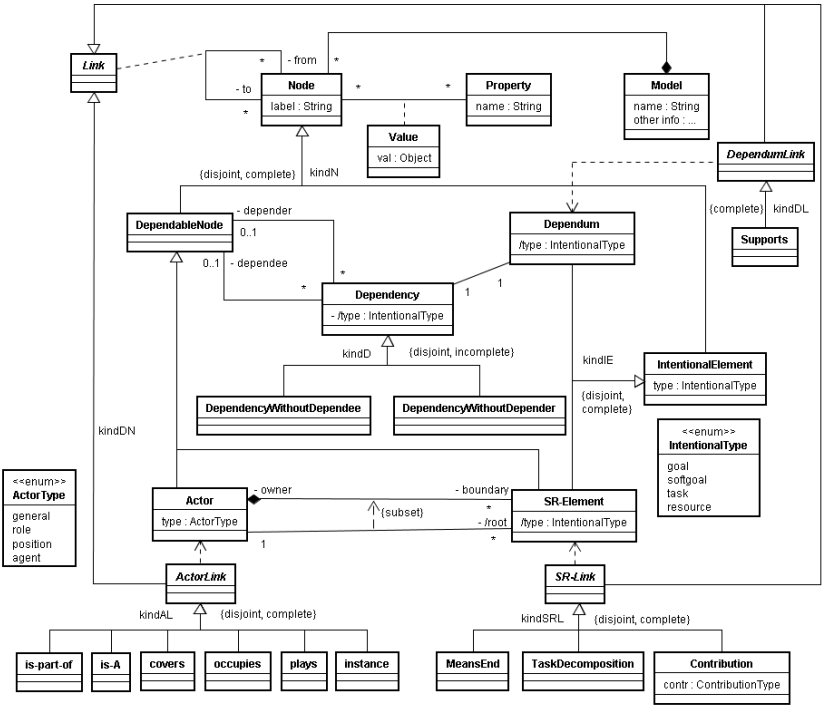


Fig. 1. The *i** metamodel

Table 1. Constraints on the *i** metamodel: a sample

Constraints on the *i** metamodel
context Link **inv** Nodes_Are_From_The_Same_Model: self.from.model = self.to.model -- both ends of link belong to the same model

context SR-Link **inv** Nodes_Are_SR_Elements:
-- both ends of an SR-Link are SR elements
  self.from.oclAsType(IntentionalElement)->oclIsTypeOf(SR-Element) and
  self.to.oclAsType(IntentionalElement)->oclIsTypeOf(SR-Element)

context TaskDecompositon **inv** To_Node_Is_Task: -- the node object of a Task Decomposition ("to" role) is a Task   self.to.oclAsType(IntentionalElement).type = task

The resulting metamodel is shown in Fig. 1. We also show some representative OCL constraints, especially to illustrate how the different levels of abstraction in the class diagram have also their counterpart in the OCL constraints (see Table 1). In the following sections, we will define new classes and associations corresponding to the modularity constructs that will be linked with the required elements of this metamodel.

4 Types of Modules

In this section we present the two types of modules we envisage for encapsulating meaningful *i** model pieces: SR modules and SD modules. Both types are subclasses of a more general class that declares the common attributes of interest, at least the name of the module and other required information not relevant for this paper (e.g., metadata as author, date, etc.). In addition, also the whole model can be encapsulated in a module, in which case the metamodel of Fig. 1 describes the allowed contents.

4.1 SR Modules

SR modules are the most obvious type of module because the elaboration of SR models relies upon the application of several kinds of refinement operators.

According to the metamodel presented in Section 3, the two usual kinds of refinement operators are decomposition of tasks and identification of means for an end. Also, softgoal contributions need to be considered. However, aligning with the general guidelines of our approach, we define first the general concept of SR module and then show how to customize it to the cases above, leaving open the door for incorporating further types of modules if new decomposition operators are proposed.

Fig. 2 shows the connexion of SR modules to the *i** metamodel and Table 2 lists some additional integrity constraints expressed in OCL, which need to be considered as additions to the ones already defined in the metamodel. In its more basic form, an SR module is composed of SR elements and links among them. Upon this basic structural form, we have added as few additional constraints as possible to allow defining in the future different types of SR modules:

- Multiplicities show that the module shall contain at least two SR elements. Also, an OCL constraint (not shown) requires at least one link among them.
- At least one of the SR elements shall be a root (see the definition of *root* at Table 2). We have considered that constraining to one single root could be unnecessarily restrictive (as illustrated below).
- From the root elements, all other intentional elements shall be reachable (see *All_SR_Elements_Reachable_From_Roots* in Table 2). That is, no unconnected partitions are allowed since we consider that they would represent different conceptual units that would require encapsulation in different modules.
- We remark that we do not impose any restriction on the decomposition depth. This means that the decomposition complexity is left up to the modeller's decision.

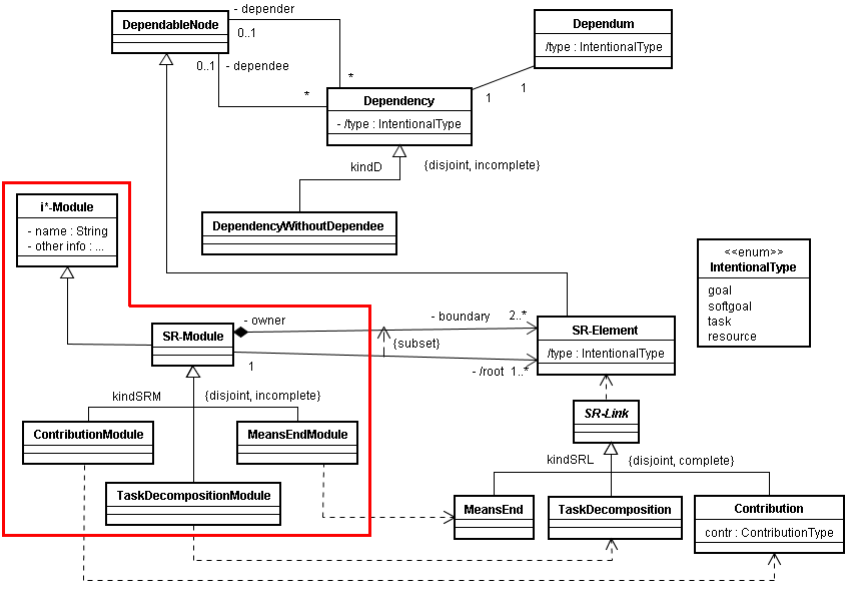


Fig. 2. Integrating SR modules with the *i** metamodel

Table 2. Constraints over SR modules

General constraints on SR Modules

`context SR-Module inv All_SR_Elements_Reachable_From_Roots:`
`  let descendants(x: Set(SR-Element)): Set(SR-Element) =`
`  x->union(descendants(x.from)) in`
`  descendants(self.root)->includesAll(self->boundary)`

`context SR-Module inv All_Dependencies_Are_Without_Dependees:`
`  self.boundary->forAll(x | x.dependency[depender]`
`  ->forAll(d | d.oclIsTypeOf(DependencyWithoutDependee))`

Particular constraints on particular types of SR Modules

`context TaskDecomposition-Module inv Valid_Task-Decomposition_Module:`
`  self.root->select(x | x.type = task)->size() = 1 and`
`  self.root->reject(x | x.type = task)->forAll(x | x.type = softgoal) and`
`  self.root->select(x | x.type = task).link[to]`
`  ->forAll(l | l.oclIsTypeOf(TaskDecomposition)) and`
`  self.root->select(x | x.type = softgoal).link[to]`
`  ->forAll(l | l.oclIsTypeOf(Contribution))`

`context MeansEnd-Module inv Valid_Means-End_Module:` `  ...similar to the one above`

`context Contribution-Module inv Valid_Contribution_Module:`
`  self.root->size()= 1 and any(self.root).type = softgoal and`
`  self.root.link[to]->forAll(l | l.oclIsTypeOf(Contribution))`

This basic form could be enough in those cases where the intentional elements may fulfil the required goals by themselves. But most often, they will require the collaboration of other actors and this will be represented, as usual, by dependencies. The fundamental point here is just to show the dependers and dependums of those dependencies, not the dependees (see *All_Dependencies_Are_Without_Dependees*). As a

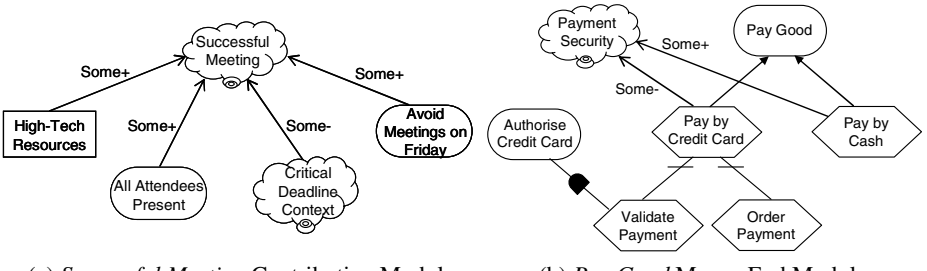
result, the SR module does not include any assumption about what intentional element will collaborate with these dependers. The connection of the dependencies defined in the SR module and the surrounding actors will be established as part of the module operations (see Section 5). The structure itself of the enlarged metamodel ensures that dependers are always SR elements (i.e., dependencies at the level of actor are not allowed when decomposing at the SR level).

SR modules in general may contain any arbitrary decomposition of elements. From this general form, we define three different types of SR modules that appear as specializations of the SR module class in the class hierarchy (we remark that the partition is incomplete). Their particular constraints are shown in Table 2.

- Task-decomposition SR modules. The intentional element of interest is a task decomposed into subelements. These subelements may be further decomposed. The intentional elements that appear in this multi-level decomposition may contribute to softgoals (that are also roots in the diagram), and these contributions may also be included in the module.
- Means-end SR modules. The intentional element of interest is a goal whose means are tasks. As happened above, tasks may be further decomposed and all the intentional elements may contribute to softgoals.
- Contribution SR modules. They identify intentional elements that contribute to one softgoal. In this case, we consider methodologically convenient to allow just one root, namely the softgoal of interest. Also just intentional elements that directly contribute to the root softgoal are included.

New types of modules could be eventually defined by adding specializations with the corresponding integrity constraints.

Fig. 3 shows examples of these modules. At the left we have a contribution module with no stemming dependencies, whilst on the right a means-end module states the need of collaboration with some undefined actor and a contribution to softgoal.



(a) *Successful Meeting* Contribution Module (b) *Pay Good* Means-End Module

Fig. 3. Examples of SR modules

4.2 SD Modules

This type of module is conceived to contain subsets of actors and dependencies among them. In their general form, SD modules encapsulate actors and dependencies without any restriction. From a methodological point of view, it is interesting to

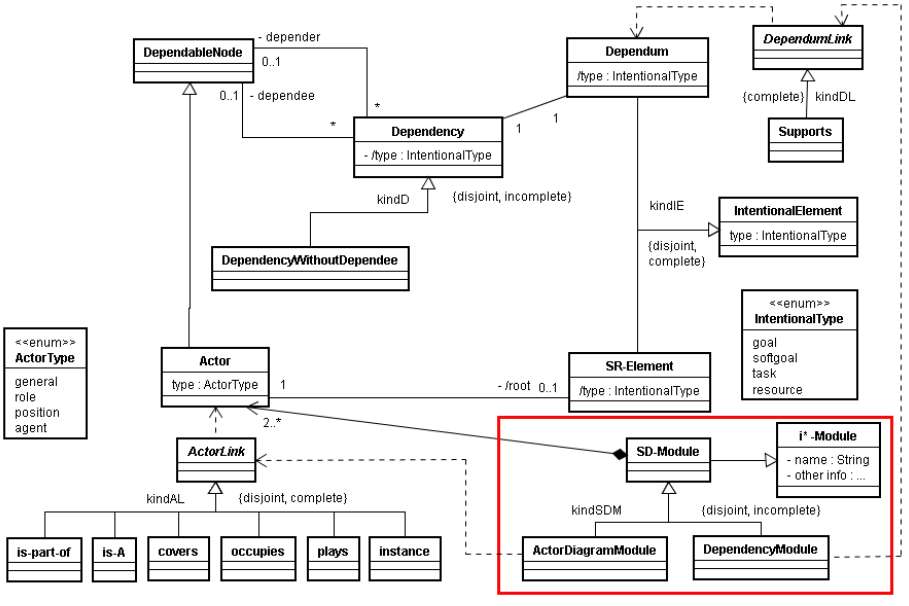


Fig. 4. Integrating SD modules with the *i** metamodel

designate some particular types of SD modules that seem convenient to define, namely actor diagram modules and dependency modules.

Fig. 4 shows the integration of SD modules into the *i** metamodel and Table 3 shows some representative constraints. Some similarities may be established with SR modules: (1) a minimum of two actors are required (since recursive dependencies are not allowed by the metamodel), as well as at least one link or dependency among them (by an OCL constraint not shown); (2) all the actors should be interconnected somehow, otherwise they would represent different abstractions and should be encapsulated in different modules; (3) there is no restriction about the number of actors, links or dependencies, it's up to the modeller to decide the appropriate size of the module; (4) dependencies are established among actors, not intentional elements; (5) there may be some dependencies from actors that have not dependees inside the module, mixed with dependencies whose both ends are actors belonging to the module (see Fig. 5, (a)). Furthermore, for methodological reasons (see [13]) we allow actors to include a primary objective in the form of an intentional element inside their boundaries. This way, the SD diagram may declare the overall intention of its enclosed actors. Even in this case, the dependencies are between actors. To reinforce that these goals are roots, SR links are not allowed in SD diagrams. OCL constraints take care of these conditions.

The two specializations of the general concept of SD module are defined as:

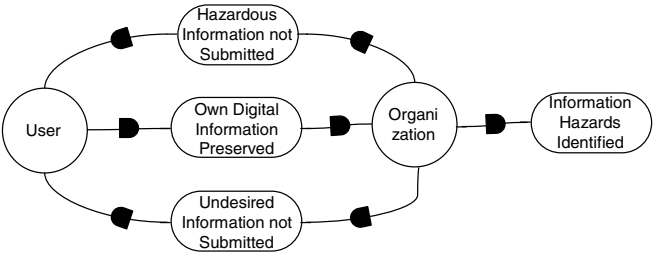
– Actor diagrams SD modules. The module just contains actors and links among them, i.e. no dependencies are included, see [14]. This module recognizes the rich variability of actor types and their relationships by creating networks of roles,

positions and agents with specialization and aggregation information. See Fig. 5 (b) for an example.

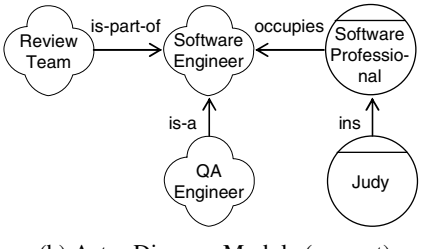
– Dependency SD modules. The module just contains a dependency of interest between two actors and then some supporting dependencies (making use of the *support* dependum link), that may be decomposed at their turn. Therefore, we can refine dependencies in a similar way than SR elements. In Fig. 5 (c) we show an example that reflects the refinement process as mutual needs of both actors.

Table 3. Constraints over SD modules

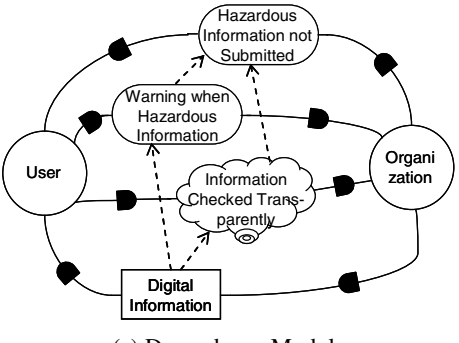
General constraints on SD Modules
context SD-Module **inv** All_Actors_Are_Connected: self.actor->forAll(x,y | existsPath(x, y)) -- auxiliary function not included
context SD-Module **inv** There_Are_Not_Dependencies_Without_Depender: not self.actor.dependency[dependee].oclIsTypeOf(DependencyWithoutDepender)
context SD-Module **inv** No_SR-Links_Allowed: self.actor.root->forAll(SR-Link[from]->isEmpty() and SR-Link[to]->isEmpty)
Particular constraints on particular types of SD Modules
context ActorDiagramModule **inv** No_Dependencies_Allowed: self.actor.dependency[depender]->isEmpty() and self.actor.dependency[dependee]->isEmpty()
context DependencyModule **inv** Just_Two_Actors: self.actor->size() = 2



(a) General SD Module



(b) Actor Diagram Module (excerpt)



(c) Dependency Module

Fig. 5. Examples of SD modules

5 Module Composition

Once module types and their valid contents have been defined, it is necessary to formulate the operations needed to manage them. Basically we need to cover two particular needs: merging two modules into one, and including a module into a model.

5.1 Model Combination

Fig. 6 defines an abstract module combination operation at the level of the *i*-Module* superclass. This operation fixes common preconditions and postconditions with some protected auxiliary functions (refinement not included), to be enriched in the subclasses. It is worth noting along the section that, since modules and models are defined as composition of elements, node comparison is done not by oid but by identifier.

```
combine(a1: i*-Module, a2: i*-Module, name: String, other info...)
  /* there is not a module with the name of the new one */
pre not i*-Module.allInstances().label->includes(name)
  /* common nodes are of the same type */
pre compatibleNodes(allNodes(a1), allNodes(a2))
  /* the new module has been created */
post oclIsNew(s) and
     s.label = name and s.otherInfo = other info... and oclIsTypeOf(i*-Module)
  /* nodes in the new module are the union of those from starting modules */
     and sameNodes(allNodes(s), allNodes(modelUnion(a1, a2)))
  /* nodes in the new module are compatible to those from the starting mods. */
     and compatibleNodes(allNodes(s), allNodes(modelUnion(a1, a2)))
```

Fig. 6. The combination operation in the superclass

A question arises to know which type of module results from the combination of two modules. In some cases the question is straightforward, e.g. the case of combination of Actor Diagram modules, which results in another module of the same type if preconditions hold. In other cases the answer depends on the contents of the module, e.g. when combining two Task Decompositions modules A and B, if A's task root appears as a leaf inside B, then the operation yields another Task Decomposition module, otherwise the result does not comply with the constraints on this type and needs to be considered as an instance of the more general concept of SR module.

```
combine(a1: ActorDiagramModule, a2: ActorDiagramModule,
        name: String, other info...)
   /* there is at least one actor in common */
pre a1.actor.label->intersection(a2.actor.label)->isNotEmpty()
   /* the result is also an Actor Diagram Module */
post oclIsNew(s) and s.label = name and s.oclIsTypeOf(ActorDiagramModule)
```

Fig. 7. Module combination operation: the Actor Diagram Module case

Fig. 7 illustrates the refinement for the case of Actor Diagram modules combination. A new precondition demanding at least one common actor is requested to ensure the invariant of this type of module. Also, the type of this kind of combination is detailed. The rest of conditions are fulfilled by inheriting the superclass definition

(which is more general than needed, e.g. Actor Diagram modules do not have SR links, but this is not a problem). This particular example of combination refinement is quite straightforward since by definition this type of module does not have dependencies, see 5.2 for this case.

5.2 Module Application

Application of a module over an *i** submodel or element relies on the same principles and in fact, several auxiliary functions appearing in the OCL definition will be shared.

```
apply(m: Model, a: i*-Module, depMtch: Set(dpdm: Dependum, x: DependableNode))
   /* common nodes are of the same type */
pre compatibleNodes(allNodes(m), allNodes(a))  -- nodes in m not in a are not
   /* the dependency matching is correct */    -- considered
pre depMtch->forAll(
       allNodes(a)->includes(dpdm) and
                    dpdm.dependency.isOclTypeOf(DependencyWithoutDependee) and
       allNodes(m)->includes(x) and not allNodes(m).label->includes(dpdm.label)
       and compatibleLinkEndPoints(dpdm.dependency.depender, x))
   /* the nodes in the module are included in the model */
post hasNodes(m, allNodes(a))
   /* the nodes keep being compatible after the application */
post compatibleNodes(allNodes(m), allNodes(a))
   /* the matching has been applied in the model */
post depMtch->forAll(
       allNodes(m).label->includes(dpdm.label)) and
       allNodes(m)->select(label = dpdm.label).
              dependency.depender.label = dpdm.dependency.depender.label and
       allNodes(m)->select(label = dpdm.label).dependency.dependee = x))
```

Fig. 8. Applying an *i** module to an *i** model

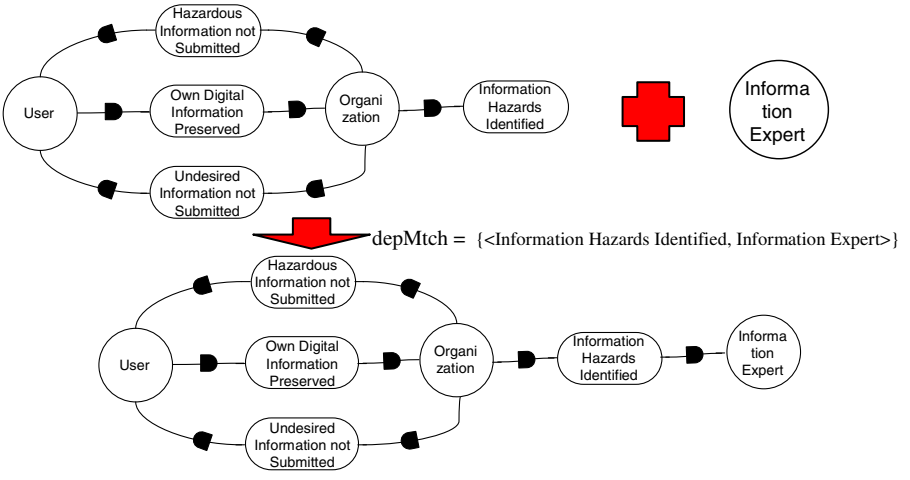


Fig. 9. Example of application of an SD module (left, from Fig. 5) to a model excerpt (an actor)

Fig. 8 shows the general application function. We remark here the connection of dependencies stemming out of the module. The header of the function includes a matching from dependums of the module to dependable nodes of the model. Correctness of this matching requires the dependum to correspond to a dependency without dependee, and compatibility of the already defined depender and the proposed dependable node that acts as dependee in the resulting node. Note that the matching may be partial, meaning that after its application some dependencies may remain without dependee (of course, this means that the model is still incomplete).

Fig. 9 provides a very basic example applying the module in Fig. 5(a) to a model that includes an actor, being in this case the matching: *Information Hazards Identified* → *Information Expert*.

In Fig. 10 we refine the general application function to the particular case of applying a Task Decomposition module to an *i** model. In this case it is necessary to add a precondition to check that the task decomposed in the module is not yet decomposed in the model.

```
apply(m: Model, a: TaskDecompositionModule)
let theTask: SR-Element = a.root->select(type = task) in  -- the root task of a
   /* the task is in the model and it is not decomposed (it is a leaf) */
pre m.boundary.label->includes(theTask.label) and
    m.boundary->select(label = theTask.label).to->isEmpty()
```

Fig. 10. Module application operation: the Task Decomposition Module case

6 Discussion and Further Issues

In Sections 4 and 5 we have presented the types of modules and two basic module management operations. We have presented in detail the structure of the modules and the specification of operations. But there are some additional issues that we have not tackled due to lack of space that we enumerate below.

Relationships among modules. In the proposal, relationships between modules are implicit: a module is related with another if some component of the former is bound somehow with some component of the latter. This kind of relationship is quite basic and could be improved by defining semantic ones. We envisage two types of such relationships. On the one hand, adding rationale to the implicit relationship mentioned above. On the other hand, refinement-related relationships, in which it may be established, for instance, that a module refines another, or that it is a different version, etc.

In addition to this, structural relationships among modules can be convenient, e.g. composition or nesting. The UML metamodel may be a source of inspiration with the objective of having a uniform treatment with respect to this widespread modelling notation. Nevertheless, trade-offs need to be assessed (see future work at Section 7).

Concept-driven modularization. The proposal presented here is not much linked to the problem domain, since the criteria used to identify the modules is not established explicitly. In the specializations of SR modules, it may be argued that the criterion is the intentional element to be decomposed. This also happens in Dependency modules.

But in the general case, the criterion is missing. A simple solution to this problem is to add the *Property* class from the metamodel to the modules metamodel. For instance, going back to Section 2, one property could be Business Service, and then each different business service in [8] could be represented by a module. The same applies to the Etapatelecom case, where the property could simply be Stakeholder and then each stakeholder may have her own model encapsulated in a module.

Model matching and model integration. In this work, operations defined in Section 5 are binding elements by name. But more sophisticated forms may be defined. The most immediate extension is to provide a mapping of names. In fact, this extension is almost mandatory if we think about reusability (see below). But also we may think of more sophisticated integrations. For instance, one possibility is to consider *i** model merging as proposed in [15]. We remark that the consideration of these proposals impacts on the definition of the module combination and application operations but not in the proposal of modules presented in this paper.

Reusability. A natural consequence on having a modularization structure available is thinking about reusability. Currently, reusability is just a copy-and-paste process due to this lack of modularity. Having modules available makes it easier to organize repositories of modules. Several consequences can be listed.

First, new types of modules could be considered, for instance, a specialization of SR module called Actor module that contains all the rationale of an actor, ready to be reused. In this reusability context, the most general type of module, corresponding to a whole model (as mentioned at the beginning of Section 4), would surely play a prominent role.

Second, it may be convenient to add incoming dependencies to modules. These incoming dependencies would synthesise the intentionality provided by the (sub)model encapsulated in the module. Therefore, it would not be necessary to analyse any single intentional element inside that (sub)model to reuse the actors.

Last, as mentioned above, in the reusability context, binding by name as proposed here is clearly insufficient and at least a name mapping is required.

Views. The proposal of this paper is oriented to provide support to the modeller whilst developing the models, and facilitate their latter understandability and maintainability. Another possibility could have been to define views over *i** models that could eventually be stored in modules if required. Views are a powerful mechanism to extract information from models and in fact it is natural to think in this to make modular existing models. This is the idea followed in [6][7] to create aspects from existing *i** models. But still the problem to build the model remains, and scalability is still an issue. Therefore, we see views not as a different alternative but complementary to our proposal.

Tool-support. At the bottom line, modules are a model management mechanism and not a fundamental ontological construct in *i**. Because of their operational nature, tool support is fundamental to make them usable. In fact, a good tool support for this proposal should hide the metamodel details presented here and provide functionalities that represent practical needs for the modeller whilst constructing her model.

7 Conclusions and Future Work

In this paper we have proposed some modularity constructs for structuring *i** models. These constructs have taken the form of building blocks, i.e. modules, together with a model combination operator. We have determined the *i** metamodel, and then defined those modules as new classes that refer those metamodel elements. Finally, we have defined some module operations and outlined some further issues.

In the rest of the section, we assess the modularity proposal using the features defined in [2].

Features considerably improved

- *Modularity*. According to [2], this feature is not currently supported by *i**. The main rationale was "[…] *i** doesn't have mechanisms for using building modules […]". We have tackled this issue in the paper. In [2] the emphasis was on defining building modules for business processes. In our approach, we have adopted a more neutral view, presenting the building modules more related to the structure of the models than to the ontology of the domain. As a consequence, modules may be eventually bound to whatever concept may be considered of primary interest.
- *Refinement*. [2] states the need of "[…] incrementally add more detail […] until we reach concrete models of business processes and their actor dependencies". Having building modules impacts positively in this stepwise refinement of *i** models, since modules can be used to encapsulate elements that are at the same level of abstraction.
- *Complexity Management*. This feature is defined in [2] as "the capability of the modelling method to provide a hierarchical structure for its models, constructs and concepts". In this proposal, since an element that appears in a module may, at its turn, be decomposed itself, there is an implicit hierarchy of models (i.e., those that are enclosed in the modules) and thus of constructs and concepts (as part of the models).
- *Reusability*. As stated in [2], "this feature is causally related to modularity", thus it can be said that the existing modules are providing the basic foundations for reusability of models and model elements.
- *Scalability*. Also there is a causal relationship to modularity, therefore we may argue that the existence of building modules is expected to improve the scalability of the *i** framework.

Features slightly improved

- *Expressiveness*. Although not a fundamental issue in this work, it must be remarked that a couple of characteristics of the metamodel enhance expressiveness. First, the capability of linking model nodes to any external concept represented in the *Property* class. *Property* instances may represent ontological concepts (e.g., the concept of business process) or instances of these concepts (e.g., a particular business process). Second, the high-level abstraction classes *Node*, *Intentional Element* and especially *Link* support extension of the language from the syntactic point of view (the most fundamental perspective in this modularity-related work).
- *Traceability*. The *Support* dependum link increments the degree of traceability in *i** models, although of course this is a quite limited contribution. More fundamental traceability mechanisms are still missing.

Features not affected. *Repeatability* and *Domain Applicability* are not related to this work.

Our future work moves along three main directions. First, to enrich the modularity constructs especially by supporting module nesting and a language of module relationships. This second feature may help to record semantic relationships among models (a way to support traceability), e.g. a model for a socio-technical system derived from a pure social model. Second, to settle an experimentation program oriented to gain insights about the advantages and possible obstacles of the proposal, whilst obtaining quantitative feedback about production time, learning curve, etc. Assessment of some decisions taken (e.g., to force all elements in a module to form a connected graph) will be stem from this program. Third, to implement the framework both in *i** edition tools (our H*i*ME, http://www.essi.upc.edu /~llopez/hime/, but also try to incorporate it in OME, jUCMNav, REDEPEND, etc.) and in the iStarML interchange format [16]. This is a critical point, since most of the concepts presented here at the modeling level need to be naturally generated by adequate tool support, as transparently as possible for the final user.

References

1. Yu, E.: Modelling Strategic Relationships for Process Reengineering. PhD Dissertation, University of Toronto (1995)
2. Estrada, H., Martínez, A., Pastor, O., Mylopoulos, J.: An Empirical Evaluation of the *i** Framework in a Model-Based Software Generation Environment. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 513–527. Springer, Heidelberg (2006)
3. Maiden, N.A.M., Jones, S., Ncube, C., Lockerbie, J.: Using *i** in Requirements Projects: Some Experiences and Lessons Learned. In: Yu, E., Giorgini, P., Maiden, N.A.M., Mylopoulos, J. (eds.) Social Modeling for Requirements Engineering. The MIT Press, Cambridge (2010)
4. Annosi, M.C., De Pascale, A., Gross, D., Yu, E.: Analyzing Software Process Alignment with Organizational Business Strategies using an Agent- and Goal-oriented Analysis Technique - an Experience Report. In: Procs. 3rd *i** International Workshop, CEUR Workshop Proceedings, vol. 322 (2008)
5. Carvallo, J.P., Franch, X.: On the use of *i** for Architecting Hybrid Systems: A Method and an Evaluation Report. In: Procs. 2nd PoEM International Conference. LNBIP, vol. 39 (2009)
6. Alencar, F., Castro, J., Moreira, A., Araújo, J., Silva, C., Ramos, R., Mylopoulos, J.: Integration of Aspects with *i** Models. In: Kolp, M., Henderson-Sellers, B., Mouratidis, H., Garcia, A., Ghose, A.K., Bresciani, P. (eds.) AOIS 2006. LNCS (LNAI), vol. 4898, pp. 183–201. Springer, Heidelberg (2008)
7. Alencar, F., Castro, J., Lucena, M., Santos, E., Silva, C., Araújo, J., Moreira, A.: Towards Modular *i** Models. In: Procs. 25th SAC International Conference – RE Track. ACM, New York (2010)
8. Estrada, H.: A Service-oriented Approach for the i* Framework. PhD Dissertation, Universidad Politécnica de Valencia (2008)
9. Ayala, C.P., Cares, C., Carvallo, J.P., Grau, G., Haya, M., Salazar, G., Franch, X., Mayol, E., Quer, C.: A Comparative Analysis of *i**-Based Goal-Oriented Modeling Languages. In: Procs. 17th SEKE International Conference, KSI (2005)

10. Cares, C., Franch, X., Mayol, E., Quer, C.: A Reference Model for *i**. In: Yu, E., Giorgini, P., Maiden, N.A.M., Mylopoulos, J. (eds.) Social Modeling for Requirements Engineering. The MIT Press, Cambridge (2010)
11. Horkoff, J., Golnaz, E., Abdulhadi, S., Yu, E.: Reflective Analysis of the Syntax and Semantics of the *i** Framework. In: Song, I.-Y., Piattini, M., Chen, Y.-P.P., Hartmann, S., Grandi, F., Trujillo, J., Opdahl, A.L., Ferri, F., Grifoni, P., Caschera, M.C., Rolland, C., Woo, C., Salinesi, C., Zimányi, E., Claramunt, C., Frasincar, F., Houben, G.-J., Thiran, P. (eds.) ER Workshops 2008. LNCS, vol. 5232, pp. 249–260. Springer, Heidelberg (2008)
12. Franch, X., Grau, G.: Towards a Catalogue of Patterns for Defining Metrics over *i** Models. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 197–212. Springer, Heidelberg (2008)
13. Franch, X., Grau, G., Mayol, E., Quer, C., Ayala, P., Cares, C., Haya, M., Navarrete, F., Botella, P.: Systematic Construction of *i** Strategic Dependency Models for Socio-technical Systems. IJSEKE 17(1) (2007)
14. Leite, J., Werneck, V., de Pádua Albuquerque Oliveira, A., Cappelli, C., Cerqueira, A.L., de Souza Cunha, H., González-Baixauli, B.: Understanding the Strategic Actor Diagram: an Exercise of Meta Modeling. In: Procs. 10th WER International Workshop (2007)
15. Sabetzadeh, M., Easterbrook, S.: View Merging in the Presence of Incompleteness and Inconsistency. Requirements Engineering Journal 11(3) (2006)
16. Cares, C., Franch, X., Perini, A., Susi, A.: Towards interoperability of i* models using iStarML. Computer Standards & Interfaces (2010), http://dx.doi.org/10.1016/j.csi.2010.03.005