Context Transformations for Goal Models

Paola Spoletini†, Alessio Ferrari*, and Stefania Gnesi*
† Università dell'Insubria, Varese, Italy
paola.spoletini@uninsubria.it
* ISTI-CNR, Pisa, Italy
{name.surname}@isti.cnr.it

***Abstract*—This paper proposes a technique to support the requirements engineer in transforming existing models into new models to address the customer's needs. In particular, we identify a set of possible categories of context change that indicate in which direction the original model needs to evolve. Furthermore, we associate a transformation to each category, and we formalise it in terms of graph grammars. Our results are a generalisation of an experimental evaluation based on 10 models retrieved from the literature and 25 scenarios of context change. This work represents a step forward in the formalisation of requirements models since it provides the foundations of a tool to support the automatic transformation of models, and employs graph grammars to provide a formal layer to the approach.**

I. Introduction

Requirements elicitation and modelling are parts of a process influenced by several human and relational factors. Among these factors, *creativity* plays a fundamental role. Indeed, as pointed out in Neil Maiden's keynote at IEEE RE'13 [1], writing a requirement is the result of an act of creativity to propose some change in the world. In general, creativity is considered a process to open up the space of ideas. However, as pointed out by Boden [2], creativity can be also a *combinatorial* and *transformational* process. Combinatorial creativity occurs when different ideas are combined to generate new ones, while transformational creativity arises with changes of the problem space. In this paper, we propose a technique that supports transformational creativity by reusing and transforming models through a set of algorithms that allow the requirements engineer to build new models from existing ones. Since the reuse is based on a set of models collected from different sources, the proposed approach allows the requirements engineer to take advantage of other engineers' modelling experience. Moreover, transforming existing models instead of starting for scratch has the potential to shorten the development time considerably, and to cut the development costs.

The idea is to identify a set of transformation patterns that are triggered by suitable changes of context that should accommodate the already existing models to be compliant to customer's requests. In particular, we identify a set of possible categories of context change that indicate in which direction the original model needs to be modified, and we associate to each category a transformation, formalised in terms of graph grammars [3]. While the idea of reusing models is not related to any particular kind of model, the defined transformations depend on the syntax of the used model. In this work, we focus on goal models, considering popular formalisms such as i* [4] and KAOS [5]. Notice that with context change for a system, we do not refer specifically to a change in the environment in which the system is defined, which may require autonomic adaptation of the system. Instead, we focus on context changes that involve the requests of a customer, and which can be addressed by modifying an existing goal model.

Though there are several works in the literature that propose to *reuse* requirements (e.g., [6], [7]), and *transform* software models (e.g., [8], [9]), none of the previous works provide a systematic review of context changes that might occur in practice, and a formal categorization of such changes. The present work aims to address these aspects. In order to identify the context change categories and the possible transformations, we have performed an analysis of literature models and we have hypothesized which kinds of mechanic transformations could be performed on them to obtain new models suitable for different contexts or problems. The obtained results are generalised into 8 categories of context change and for each of them a set of required transformations to be performed on the original model is defined.

This work represents a step forward in the formalisation of requirements models since it offers a tool to support the automatic transformation of models in the required direction, and uses graph grammars to provide a formal layer to the approach. Then, the introduced categorisation can be used to recall the origin and the history of a model. Indeed, a context change that motivates a transformation can be seen as a trace between the old model and the existing one. Such a trace can be employed for requirements management, model versioning, as well as to support possible analyses on the evolution of the models.

The reminder of the paper is structured as follows. Section II exemplifies the proposed approach through a case study that informally shows some context change and the associated transformation. Section III gives a general description of our experimental phase and describes the obtained context change categories. In Section IV, we formally describe the transformation associated to each context change and, in Section V, we discuss the limit and the potentiality of our approach. Then, Section VI analyzes related contributions in the literature and Section VII concludes the paper.

978-1-4799-6342-3/14/$31.00 © 2014 IEEE

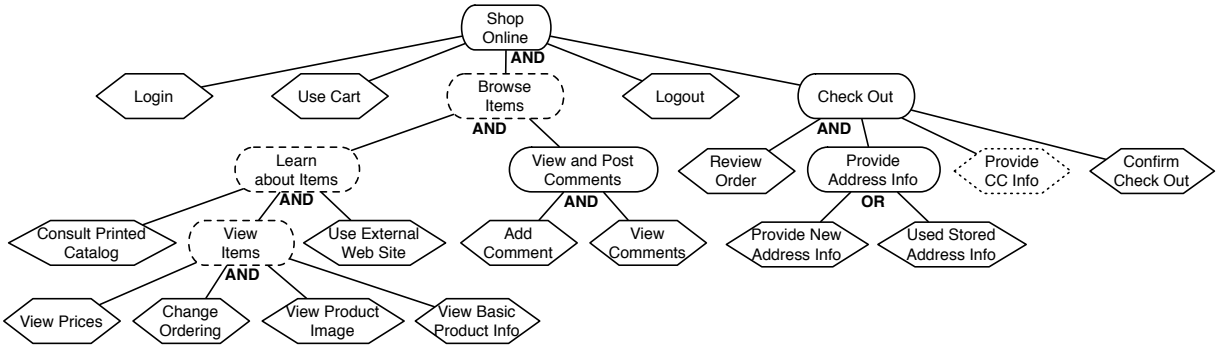

Fig. 1: Online Shop Goal Model.

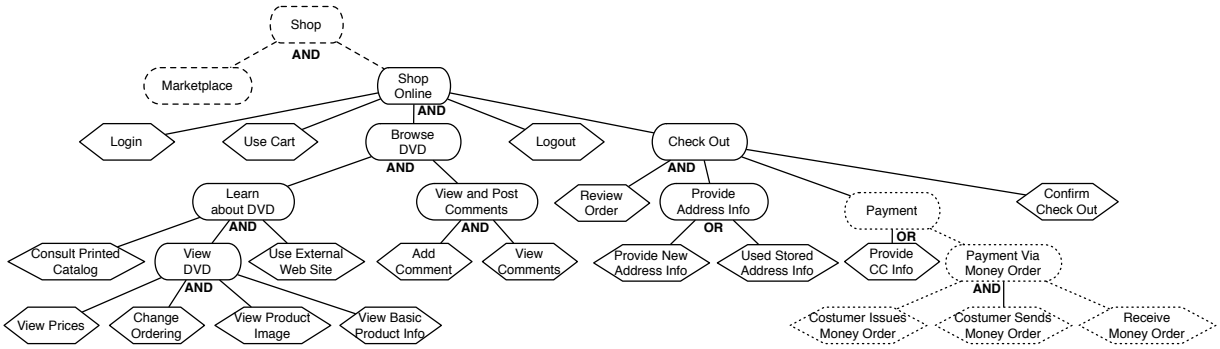

Fig. 2: Online DVD Shop Goal Model with multiple payment methods.

II. Example

To better understand how models can be transformed to obtain new models that operate in different contexts, in this section we define an example that describes some relevant use-cases of the idea. The proposed transformations are an example of the experimental work we have done to analyse how models can be transformed according to new customer's requests.

Consider the case in which a requirements engineer has already developed a model for an `online shop`, as the one presented in [10]. The model is reported in Fig. 1 (ignore, at this point, that some entities are dashed or dotted). In the following, we outline some practical change scenarios applied to the model.

A. From"items" to "DVDs"

The requirements engineer is now serving a customer who wishes to build a system for an online DVD store. If the requirements engineer thinks that the level of detail of the previously developed model is good enough to represent the new system, s/he can obtain the new model by changing the *object* on which the already developed model operates. Indeed, an online shop aims to "sell items" and, analogously, the online DVD shop still aims to "sell items", but is specialised on a particular kind of items, namely DVDs. This means that the new model has the same goal of the existing one, but the goal is reached operating on a specialisation of the original object.

Hence, the `online shop` model can be transformed by replacing the word "item" with the word "DVD", i.e., in Fig. 1 the word "DVD" substitutes the word "item" in the dashed nodes.

B. A New Payment Method

Suppose that, after discussing with the customer, the requirements engineer understands that the customer wants to add a new payment method, for example "money order", to the current model. To build the new model, s/he can develop or search in other models for the requested payment type and, once retrieved, s/he adds the subtree associated to the new payment method to the model. Imagine that, in our case, the requirements engineer selects the subtree "money order" from the `book seller` model in [11]. This additional subtree represents an alternative payment method, but, since in the original model there was only a payment method, the engineer needs to introduce an additional node that represents the parent of the subtree that is going to be added. This happens because the new subtree is *alternative* to the unique subtree that appears in the current model. To accomplish the extension, the new parent goal "Payment" is created and the new payment method and the old one become an OR-decomposition of "Payment": the dotted node in Fig. 1 is substituted with the dotted subtree in Fig. 2.

C. Adding a "Marketplace"

Suppose now that a new customer requires a model for an online shop that can manage both a traditional online DVD

shop and a marketplace, in which users can sell and buy DVDs among each others. To fulfil the request, the requirements engineer can reuse the already developed model, and transform it to represent the extension in the system. To perform this transformation, the requirements engineer creates a new node that represents the new root goal. Then, it decomposes it as an AND-decomposition where one of the sub-models is the already existing model and the other is searched among others existing models or is created from scratch. The obtained model is represented in Fig. 2, where a placeholder is added to represent the "Marketplace" goal. The requirements engineer can now provide a refinement of the "Marketplace" goal with additional goals. The title of the new model is `Online DVD shop with marketplace`.

D. Only a "book marketplace"

As a last example, suppose that, after refining the marketplace subtree in the obtained model, the requirements engineer now needs a "book marketplace" model. In this case, s/he can obtain it by performing a group of transformations on the existing model. Indeed, s/he first need to simplify the model considering only the marketplace subtree of the original model and, then, needs to modify the object on which the model operates by substituting "DVD" with "book".

III. Context Change Categories

In the previous section, we have outlined a practical example of how a goal model could evolve according to new customer's requests. The question is now: *how* can we generalise and categorise the possible context changes of the goal models? To answer this question, we have first collected a set of goal models from scientific papers and, for each one of them, we have imagined a set of practical transformation scenarios, trying to figure out how the models could be reused or readapted to satisfy new customer's requests.

Each transformation scenario is made of a goal model and a *transformation trigger*. The transformation trigger is a natural language sentence that states the semantic change that is expected from the goal model. For example, considering the `online shop` model in [10], a transformation trigger could be *"the system sells DVDs"*.

The initial list of considered models (10 in total), together with the transformation triggers (25 in total), is reported in the first three columns of Table I. A complete summary of the transformations applied, with pseudo-code that describes the modifications to each model is available at `https://sites.google.com/site/tramframework/context-change-analysis`.

The output of each scenario was a re-designed goal model - sketched on paper - that addressed the transformation trigger. From the considered scenarios and the changes applied to the models, we have been able to define a set of categories of context changes that might occur in practice.

The context change category associated to each scenario is reported in the last column of Table I. Indeed, as one could expect, similar triggers were driving similar changes at the level of the goal models. Therefore, we argued that a category could identify both a *conceptual* change of the goal model (expressed by the transformation trigger), and a consequent *structural* change. The categories are classified in three different groups according to the "part" of the original model that they involve in the transformation. In particular, *object-related* categories include those triggers that aim to impact on the object on which the original model is defined. We call *system-related* categories those triggers that aim to modify the main goal of the system (independently from the object on which the transformation is applied). Finally, *subtree-related* categories are those categories that modify the refinements of a sub-goal of the model (a goal that is not the main goal of the model).

The object-related group contains the following 4 categories:

- *Object change*: The trigger aims to generate a new model that represents a system with the same goal of the current one but works on a different object (e.g., the system in [11] sells "DVDs" instead of "books");
- *Object extension*: The trigger aims to create a new model with the same goal of the current one that however works with at least another object (e.g., the system sells "DVDs" and "books");
- *Specialisation*: The desired model works only with a specific category of the objects on which the current system works (as in Sect. II-A, we specialise "items" into "DVDs");
- *Generalisation*: The new model works only on a generalisation of the objects on which the current system works (the inverse change of Specialisation).

The system-related triggers aim to modify the goal of the system to be modelled instead of the object on which it works. We have identified the following 2 categories in this group:

- *System extension*: The main goal of the new model is an extension of the main goal of the current model (i.e., it is a higher-level goal, as in Sect. II-C, we extend the system with a "Marketplace");
- *System simplification*: The main goal of the new model is obtained by simplifying the main goal of the current model (i.e., a lower-level goal is achieved). Imagine for example, that we remove the "Marketplace" goal from the model in Fig 2.

The subtree-related categories group those triggers that aim to add to or simplify parts/functionalities of the model. These categories are the following:

- *Subtree extension*: A new subtree is added to a goal of the current model (as in Sect. II-B, we add a new payment method with an associated goal subtree);
- *Subtree simplification*: The system is simplified by removing one or more of the subtrees that can fulfil a goal (imagine we now remove the "Payment via Money Order" subtree).

Subtree-related categories can be distinguished by the system-related categories because they do not aim to change the goal

TABLE I: The list of goal models and transformation triggers

Paper	Topic	Transformation Trigger	Category
[11]	book seller	the system sells DVD instead of books	Object Change
		the system sells DVD and books	Object Extension
		the system manages the orders of a library	System Extension
		the system does not allow to pay with money order	Subtree Simplification
		the systems has a new functionality to compare supplier costs	Subtree Extension
[12]	meeting scheduler	the system defines a lesson timetable	Object Change
		the system finds a proper place and period for a conference	Object Change
		the system arranges the catering	Subtree Extension
[10]	online shop	the system adds a "money transfer" among the types of payment	Subtree Extension
		the system includes a marketplace to sell products among private people	System Extension
[13]	public transport service	the system becomes free	System Simplification
		the system allows tickets to be sold online	Subtree Extension
		the system is used for taxis insead of public transportation	Object Change
[14]	objectives of a car manufacturer	motorbikes are sold instead of cars	Object Change
		the car company becomes a monopoly	System Simplification
[15]	test case generation	the system allows the generation of test cases also from formal models	System Extension
		the system allows the generation of test cases only for Simulink models	Specialisation
[16]	library management system	the system allows managing only media items	Specialisation
		the system allows checking-in the material	Subtree Extension
[17]	document sharing system	the system allows photo sharing only	Specialisation
		the system allows to remove documents	Subtree Extension
		the system allows sharing among sub-groups of people	System Extension
[18]	healthcare data access	the data of patients are checked by an additional authority	System Extension
[19]	media shop	the system allows selling only books	Specialisation
		the system sells only books and provides a recommender system for books	Specialisation + Subtree Extension

of the model, but they want to enrich or simplify some of its sub-parts. In general, they act on the refinement of any possible node but the root.

IV. Patterns to Transform Goal Models

For each one of the identified categories, we define an automatic procedure to change the model to meet the customer's requests. To be able to do it formally, we need to discuss how we represent and store the existing models that are used as a starting point for creating new models. In the following we first discuss how the models are formally represented (Section IV-A) and, then, we present the automatic transformations associated to each category, formalising them through graph grammars [3] (Section IV-B).

A. Model Representation

The models are represented as XML documents, based on an XML schema that includes all the elements of the most popular formalisms, namely i* [4] and KAOS [5]. The sketch of the XML schema is represented in Figure 3 (the full schema, together with the models in Table I in graphical and XML form, is publicly available at `https://sites.google.com/site/tramframework/models-database`).

A `MODEL` that is represented through the schema is characterised by its `title`, the `object` on which operates, its `type` (i.e., the formalism used to represent it), and its `level`, which indicates if the model describes an entire system or only a component. The model is composed of a sequence of three kinds of elements: `ENTITY`, `RELATIONSHIP` and `GROUP`. An `ENTITY` represents a type of node in the model (e.g., goal, softgoal, requirement, task, etc.). An `ENTITY` can be refined in AND- or OR-decompositions of other entities (`REFINEMENT` in the schema). A `GROUP` represents a sub-model associated to a certain actor, as typical in i*. Finally, a `RELATIONSHIP` represents a connection between elements, and its attributed meaning depends on the `type` attribute (dependency, contribution and any type of relationship - except refinement - defined in i*, and KAOS). Notice that we assume that any model contains an `ENTITY` that represents the main goal of the model. Furthermore, we assume that the refinement of this goal is the main tree of the model. Though goal models are, in general, graphs, the models from the literature that we

have considered in this work normally include a main tree structure. Therefore, we argue that the proposed assumption is acceptable in practice.

```
<element name=MODEL title="string" object="string"
   type="string" level="integer">
    <xs: element name=ENTITY
      id="string" name="string"  type="string" >
        <xs: element name=REFINEMENT type="string" >
            <xs: element name=ENTITY type=entity minOcc=2/>
        </xs: element>
    </xs: element>
    <xs: element name=RELATIONSHIP type="string"
      value="string" entityAid="string"
      entityBid="string" type="string"/>
    <xs: element name=GROUP
      id="string" name="string"  type="string" >
        <xs: element name=ENTITY type=entity/>
        <xs: element name=RELATIONSHIP/>
    </xs: element>
</xs: element>
```

Fig. 3: Sketch of the XML-schema of the stored models.

B. Transformation Rules

The proposed transformation rules are formally defined in terms of graph grammars [3]. Such grammars operate on *graphs* instead of operating on strings like classical approaches (e.g., Chomsky grammars). A graph grammar is composed by three elements: a *typed graph* — which represents the meta-model of the adopted graph language —, an *initial graph* — the graph to be transformed —, and a set of *rules* — which describe which parts of the graph need to be transformed and how. Requirements goal models are often based on graph or tree structures. Hence, graph grammars are a suitable tool to define how to transform them.

In our case, we associate a grammar to each defined category of transformation. In all the grammars, the initial graph is the model that needs to be transformed. Moreover, the meta-model defines the structure of the supported requirements models, which is our XML schema. We assume that the model contains a main tree structure that shows how the system main goal is decomposed and operazionalized. In the following, with "model" we refer to the main tree structure. However, we will also show how our approach takes care of the ENTITIES of the model that do not belong to the main tree.

Each grammar has a specific set of rules. The rules define how to modify the model to obtain the desired result. A rule is composed by three parts: LHS (left-hand side), NAC (negative application condition) and RHS (right-hand side). Each part is a graph. The LHS and the NAC enable a rule if the former is matched by at least one part of the current graph, and the latter is not matched in any part of the graph. If applied, the rule substitutes the matched part with the graph in RHS.

We consider 8 categories of transformations, one for each identified category, and, hence, 8 graph grammars, classifiable in three main groups according to the classification proposed in Section III. In the following, we describe the transformations through a pseudocode description of the algorithms that we use to apply them. The pseudocode uses primitive functions that are outlined in Table II.

TABLE II: Primitive functions used in the pseudocode.

Name	Description
ELEMENT(M)	Returns all the nodes in M
NAME(x)	Returns the name of the node x
RENAME(n_1,n_2,x)	Writes n_2 instead of n_1 in node x
DUPLICATE(g,g')	The goal g is duplicated in g'
REMOVE-SUBTREE(g)	Removes the subtree of g
REMOVE(g)	Removes g and all its subtrees
ADD(g_1, g_2, t)	Adds the goal g_2 and its subtree as a t-decomposition of goal g_1

Object-related Transformations. Object-related transformations include *object-change*, *specialisation*, *generalisation*, and *object-extension*. Let us first consider the operations associated to an *object change*. The operations associated with this transformation modify a given model M by replacing the object o on which M is defined with a given object o' without changing the model structure. In term of graph grammars, this corresponds to a single rule that is applied multiple times, in which the NAC is empty, the LHS contains an entity that contains o in the name and the RHS contains the same node in which the name has o' instead of o. Formally:

```
1  transObjChange(M, o, o'){
2      ∀ e ∈ ELEMENT(M) do {
3          if (o ∈ NAME(g))
4              RENAME(o, o', g);}}
```

The algorithm considers all the elements of the model (line 2) and it modifies those that contain o in their names (line 3), by substituting o with o' (line 4). In order to transform also the ENTITITES and the RELATIONSHIPS outside the main tree, the above described algorithm can be extended with additional instructions that replace o with o' also in the elements that do not belong to the main tree.

The transformations related to *specialisation* and *generalisation* operate exactly as the one for object change. Indeed, the only difference among these transformations is due to the relationship between o and o' and it does not affect the transformations.

The transformations associated with *object extension* modify a model M by duplicating it for each new object and replacing the object in the duplicate. The new objects are supposed to be stored in an input set $O = \{o_1, \ldots, o_n\}$.

```
1  transObjExtension(M, o, O){
2      DUPLICATE(M, r); REMOVE-SUBTREE(r);
3      ADD(r, M, AND);
4      ∀ o' ∈ O{
5          DUPLICATE(M, r');
6          transObjChange(r', o, o');
7          RENAME(o, o ∧ o', r);
8          ADD(r, r', AND);}}
```

The algorithm duplicates the root of the main tree (line 2). This new node becomes the new root of the model and its goal is AND-decomposed. The original model becomes a component of this decomposition (line 3). Then, for each object to be added (line 4), the original tree is duplicated (line 5) and, once

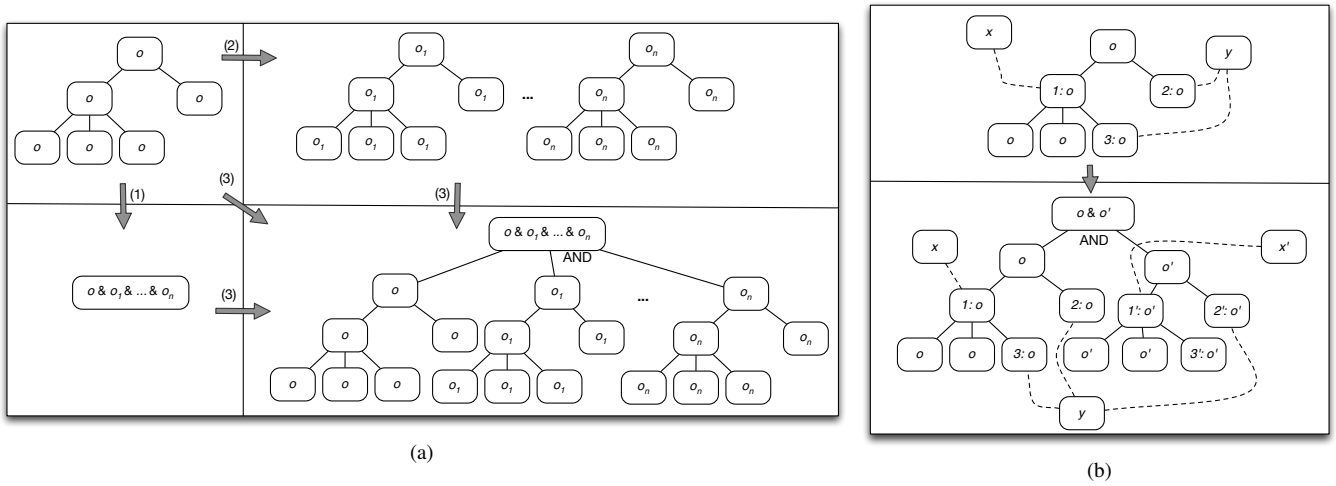

Fig. 4: (a) Sketch of the transformations required to *object extend* a model; (b) *Object extension* on a complete model.

its object is changed with the new one (line 6), the modified tree becomes part of the AND-decomposition (line 8). At each iteration of the loop, i.e., every time a new object is considered, the root name changes to consider also the new object (line 7). The idea behind this transformation is to use a copy of the original model as a model for all the new objects. This is obtained by duplicating the model and changing the object in each copy. All the obtained models, including the original one, concur together to guarantee the satisfaction of the new system main goal, that is obtained by generalising the main goal of the original model to all the objects. Figure 4a exemplifies the application of the transformations on a model defined on object o, extended to serve also the objects in O. Transformation (1) generates the new root and transformation (2) creates n duplicates of the original model in which o is changed into the new object. Finally, using the original model and the results of (1) and (2), the transformation (3) creates the final model. To this end, the transformation aggregates all the duplicates and the original model as an AND-decomposition.

To transform also the ENTITITES and RELATIONSHIPS outside the main tree, the algorithm is extended. It duplicates a selection of the external ENTITIES and, in each duplicate, it replaces o with a new object. The RELATIONSHIPS of a duplicated node are also duplicated, but, instead of referring to the original ENTITIES, they refer to the corresponding duplicates. Moreover, for each non-duplicated ENTITY e, a set of selected RELATIONSHIPS that connect e to an ENTITY related to o is duplicated and directed to the corresponding ENTITY for the new object. The selection of ENTITIES and RELATIONSHIPS to duplicate is up to the user. Figure 4b shows an example of this procedure. The original model contains two ENTITIES external to the main tree, x and y, connected to some node of the main tree by some RELATIONSHIP, represented as a dashed line. In the example, x is duplicated and its relationship with it (the duplicated x' is related to $1'$ as x is connected with 1), while y is not duplicated. The ENTITY is connected with 2 and 3. Between these two RELATIONSHIPS, the former is duplicated, while the latter is not (so y is also related with $2'$).

Notice that, since the object extension transformation duplicates the model as many times as many new objects are added, this context change is suitable only when the extension concerns few objects. When the number of new objects grows, the use of a *generalisation* is preferable (e.g., from "DVD" and "book" to "item"). In such cases, an ontology could be associated to the model, to keep trace of the specific objects included in the generalisation.

System and Component Transformations. The operations associated to system or component transformations are grouped in the following description, since their behaviour is quite similar.

The transformations associated to a *system extension* are analogous to the ones associated to an object extension, but, instead of using the modified original model to represent the new objects, they require to find the model that needs to be added to extend the system. The input parameters of the algorithm are the starting model M, the query q, and, if it exists, a model M' for the systems extension:

```
1  systemExtension(M, q, M'){
2        DUPLICATE(M, r); REMOVE-SUBTREE(r);
3        RENAME(NAME(r), NAME(r) and q);
4        ADD(r, M, AND);
5        if (M' = ∅)
6              M' = placeholder;
7        ADD(r, M', AND);}
```

The root on the main tree is duplicated (line 2). In the copy, which becomes the new root, the name is changed to include both the name of the original root and the name of the system that needs to be added (line 3). The root is AND-decomposed and the original model becomes one of its ENTITIES (line 4). If the requirements engineer does

not have developed the new extension (line 5), the system is added (line 7) through a placeholder (line 8), that can be later manually replaced by the requirements engineer. Notice that the function `systemExtension` only supports an extension of the system at a time. To extend the original model with multiple models, we need to apply `systemExtension` sequentially by adding an extension at each call. This procedure generates the expected final model in which the models of all the extensions, including the original model, concur in an AND-decomposition to the final system goal. However, the lack of a function that manages all the extensions at a time has a drawback. Indeed, at each call the algorithm adds a level at the tree of the model to create an AND-decomposition with two `ENTITIES`. The transformation is expected to generate a unique extra level in which all the extensions are at the same level. Instead, with the current definition, the transformation generates multiple levels. This problem can be easily solved by manually refactoring the obtained model. Besides the operations on the main tree, the transformations to apply a system extension requires only to add the other `ENTITIES` of the model together with their `RELATIONSHIPS`.

The algorithm that performs the *subtree extension* is analogous to the one needed to perform a system extension. The only difference is that, in the former case, the algorithm takes as additional parameters the node where to add the subtree (or a placeholder for it), and, if the subtree is added to a single alternative (as seen in the example in Section II), the kind of refinement to be used to compose the existing subtree with the new one.

The input parameters of the algorithm that performs the transformations associated to a system *simplification* are the model to modify and the node to identify the part to delete.

```
1  systemSimplification(M, n){
2       e=MATCH(M.children, n);
3       REMOVE(e));}
```

The sub-system to be deleted is identified by matching the node name (n) among the *direct* children of the root (line 2) and, then, it is pruned with all the remaining pending `RELATIONSHIPS` (line 3). Once the model is pruned, if some of the nodes in the model remains decomposed in only one option, the model is refactored, as represented in the rule in Figure 5. The rule matches all the nodes that are AND- or OR-decomposed by an element, but not more than one, and, in that case, substitutes the root of the decomposition (x) with its only child (y).

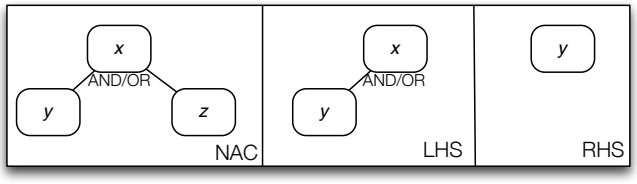

Fig. 5: Rule to refactor a model after a simplification.

Notice that the algorithm deletes also the `RELATIONSHIPS` between `ENTITIES` deleted in the main tree and `ENTITIES` that do not belong to the main tree. When removing such `RELATIONSHIPS`, if the external `ENTITY` remains isolated, the algorithm identifies it as a candidate to be deleted.

The *subtree simplification* acts exactly as the subsystem simplification. The only difference is that the match to find the part to be deleted is performed on all the `ENTITIES` of the main tree.

V. Discussion

In the previous sections, we presented a set of categories of goal model transformations, and their formalisation in terms of graph grammars. This is a first step in the direction of providing the theoretical foundations for a system that supports the requirements engineer in reusing goal models. Graph grammars can be employed as a formal underlying layer to support the transformations in an automated and sound manner. However, along with the practical scenarios that we have analysed while transforming goal models, we have seen that there are relevant issues to be taken into account when foreseeing a system for automated model transformations.

A. The Role of the Requirements Engineer

The *decision role* of the requirements engineer, and his/her rational in the model transformation appears to be crucial. Though transformations can be generalised, categorised and formalised in their core parts, the validation of the transformed model cannot be automatised and remains a task to be performed by the requirements engineer. Moreover, we found out that, in practice, requirements engineers do not only validate the model, but tend to apply some re-arrangements to the models that could not be standardised and, hence, automated.

For example, consider an excerpt of the `public transport service` goal model in [13] (Figure 6a). The trigger *"the system allows tickets to be sold online"* induces an additional functionality expressed by means of a goal subtree "Provide tickets online" (the dashed subtree in Figure 6b). Since the original system does not have different options to provide tickets, we need to provide a parent node to be refined by the added subtree and the original one (the node "Provide tickets", which is the root of the dashed subtree in Figure 6b). The node is attached to the "Avoid fraud" node. However, "Provide tickets" was the name of the already existing subtree (the dotted node in Figure 6a), which did not need any additional qualification in the goal name, since it was the only option for providing tickets in the original model. This node needs to be renamed to distinguish the goal of providing tickets "in place" from the general goal of providing tickets, as shown by the dotted node in Figure 6b. These operations evidently require a decision and an intervention of the thinking requirements engineer, and can be hardly automated.

Another case in which the contribution of the requirements engineer becomes crucial can be outlined by considering again the `public transport service` model

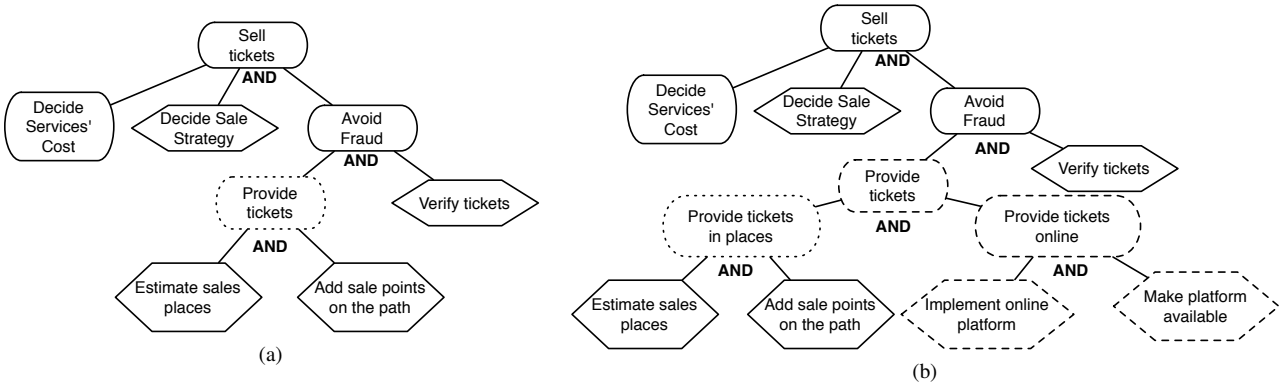

Fig. 6: (a) An excerpt of the model in [13] and (b) its transformed version.

in [13], and the trigger *"the system becomes free"*. The trigger leads to a system simplification change, by deleting the subtree associated to a node named "Sell tickets". Now, the `systemSimplification` algorithm does not take into account the other nodes of the goal model that have some implicit relation with "Sell tickets", such as, for example, the node associated to the requirement "Manage profits" (the full model is not reported here for sake of space, but can be seen at `http://goo.gl/fjr1PX`). Such a node shall be manually deleted by the requirements engineer, if profits are not foreseen in absence of tickets. Analogously, any other object-related transformation may require further manual modification to remove or modify parts of the model that are not adequate for the new object, or to include additional details that are peculiar to the new object.

Another example of the relevant role played by the requirements engineer is the choice that the s/he has to make between a *System* and a *Subtree extension*, which are in practice rather similar. The decision depends on several factors. (a) It depends on the emphasis that the requirements engineer wishes to give to the new subtree that will extend the system: if major emphasis is given to the novel tree, a system extension shall be probably chosen. (b) It depends on the foreseen development phases of the system. Indeed, separating the goals eases the separation of concerns at the development stage. Different high-level subtree can be apportioned to different system architecture components that can be developed by different development teams. In this sense, a system extension might clarify which are the goals of the system that shall be addressed by a specific development team. (c) The choice also depends on the type of model that is considered: is the model built for evolving through a system extension or through a subtree extension? If some already existing goal can be regarded as a parent or sibling of the additional subtree – as in the previous example for the "Provide tickets" node and the "Provide tickets online" node –, it is probably the case to perform a subtree extension. (d) Finally, the choice depends on how many goals could be shared between the current model and the future extension and how these node are distributed in the original model. In some cases, it is probably preferable to perform a *manual* refactoring of the model, without relying on the transformations that we have provided. Moreover, notice that, in general, an extension that represents an alternative refinement for a goal (OR-decomposition) is a subtree extension. Indeed, the subtrees in the OR-decompositions show different ways to satisfy the goal they decompose. Such alternatives are not even necessarily implemented in the final system. Hence, adding a new OR-subtree does not extend the goal of a system, even if it is added to the main goal, but suggests a new possible way to satisfy it.

B. The Role of the Model

To merely apply the transformations that we have outlined, the models have to show specific characteristics that favour the model *evolution*. In particular, each model shall explicitly specify the "object" of interest, to facilitate *object-related* transformations. Moreover, the model shall have a clearly identifiable main tree among the hard-goals to favour system extension and subtree extension. A tree-like hierarchical structure enables clear separation of concerns, and, in general, fosters reusability. We have seen that models with an interwoven graph-like structure are often hard to transform. Finally, another aspect is the number of types of components belonging the goal modelling language that are used in the model. Though employing different types of components helps in identifying different semantic aspects, it has some drawbacks from the reusability point of view: indeed, employing more "abstract" components (e.g., only goal, softgoal and requirements), ease the adaptation of such components to different contexts.

In practice, even if many existing models in the literature present these characteristics (i.e., tree-like structure and abstract components), models are not always formal and this might hamper their manipulation. To overcome this problem, we have proposed the XML schema presented in Figure 3 to represent, when possible, existing models in a common formal format.

C. Contribution

Besides some limitations caused by the need of inputs and refactoring steps from requirements engineers and by the need

of well-formed goal models, the proposed approach gives two main contributions. First, it is a step forward in classifying categories of change for existing models. Even if we do not claim that our classification is complete, it represents a good starting point. The categories allow the requirements engineer to understand which existing models s/he can reuse and for which request. Second, the transformations simplify the adaptation process, and guide the modification in the direction of having well-structured, component-based models.

Moreover, the context change categories that we have identified and formalised provide a formal basis not only to automate model transformations, but also to enforce *traceability* and *versioning* between models at different stages of evolution. Indeed, given a goal model and its transformed version, the transformations can be regarded as *operators* that have been sequentially applied to the model to change it. Therefore, the history of a model can be represented in terms of different operators applied to the original model. More formally, a transformed model M' is regarded as $t_1(t_2(\dots t_n(M)))$, where M is the original model and $t_1, t_2, \dots t_n$ are the transformation operators (i.e., the algorithms associated to the context changes). Since each transformation can be associated to a trigger, the history is also enriched with semantic traces of the model evolution.

D. Towards a General Framework

We can consider this work as a first step in the direction of developing a more general framework to support transformational creativity. The envisioned framework will be formed by two components: a recommender system and a transformation engine. The recommender system will rely on a set of existing requirements models, efficiently stored in a database, using the XML representation introduced in this work, and categories of context changes. Given a generic customer specification in natural language, the recommender system will suggest a set of relevant requirements models among the stored ones and will recommend suitable context changes that can be applied to such models to suit the customer's needs. The requirements engineer will identify the most suitable pair (model and associated context change) among the proposed ones, and will input it to the second component of the framework, namely the transformation engine. This component will apply the transformations associated to the chosen context change, using the algorithms presented in this work. The engine will produce a transformed model, which can be used as a seed to inspire the novel design, and boost structured creativity within the requirements process. Scenarios and requirements for the envisioned framework are reported at `http://goo.gl/GuVrsF`.

VI. Related Work

The work presented in this paper aims at *reusing* and *transforming* requirements models expressed in the form of goal models. The rationale is to reuse previously acquired knowledge to inspire requirements elicitation and formalisation of new systems. This idea is also the main focus of the works of Sutcliffe [20]. Among these works, the approach presented in [21] proposes to reuse generic models of applications as *templates* for modelling and evaluating requirements for new applications. Furthermore, approaches for the reuse of requirements *patterns* have been explored in the literature, e.g., [22], as well as techniques to model context variability in requirements, e.g., [23]. Moreover, in [6], an approach is proposed to reuse *natural language* requirements that have been already employed in previously defined systems. However, none of these methods provides means to identify the *categories* of domain changes of a product, and to support the requirements engineer in the management of the domain-dependent transformations of the requirements.

In the field of *product line engineering*, approaches have been defined to reuse the requirements of similar products to build reusable assets by means of feature models (i.e., a hierarchical representation of the features of a set of similar products) [7], [24], [25]. A relevant example in this field is the work of Dumitru *et al.* [7], who propose a method that mines product descriptions from online project repositories. The descriptions are clustered into features, and a feature model is built for a given product category. The feature model is employed to recommend features and combinations of features that might be relevant for the new product under development. The approach differs from ours, since it only reuses features as they are in the original model, and does not provide means for transforming and adapting such features to a novel system.

More related to *evolution* of goal models than to reuse, but still with several affinities with ours, is the work in [26]. The work distinguishes between autonomic – i.e., which does not require human intervention – and designer-supported evolution of goal models. The latter category partially fits with our view of context-dependent reuse of goal models. In particular, adding and removing sub-requirements from a goal model are operations that are conceptually related to our subtree extension and simplification transformations, respectively. Nevertheless, the cited work does not focus on adaptation to different domains, which is among the primary objectives of our work. Souza *et al.* [27] provide the new concept of *evolution requirements*, which defines the changes on other requirements in case some conditions apply. The approach is complementary to ours, since we do not plan model reuse in advance, but such planning – and therefore the presence of evolution requirements – could in principle facilitate model reuse. Again focused on goal model evolution is also the work of Ernst *et al.* [28], who provide a framework to support goal model versioning according to a set of basic evolution operators. Though these works on model evolution are a valuable contribution for understanding *how* a model evolves and to *control* such evolution, none of them provide categories of context-changes and domain adaptation contexts.

Approaches that propose to *transform* software models normally address the need of adapting models to novel domains, or to translate the models into other modelling languages. For example, [29] presents an approach for matching and firing of transformation rules on software models to adapt them to novel needs. The same goal is addressed in Agraval *et al.* [30].

Moreover, in [9], an environment for *refactoring* models is described, which employs user-defined transformation rules. However, all these approaches for model transformation – a survey on the topic is provided in [8] – are more focused on *design-level* models, and do not deal with requirements models. We argue that inspiration can be taken from these works. In particular, most of the approaches rely on meta-models that enable sound transformations. In our case, beside the XML schema, which is our meta-model for any goal model, additional meta-models for the different domains could be provided, to better support the transformations, and trigger sound domain adaptations of the goal models. Nevertheless, domain-level meta-models, for example in the form of ontologies, should be handled with care in the requirements elicitation phase. Indeed, we argue that a too strict degree of formalisation of the domains could hamper the creativity of the requirements engineer: the definition of such domain meta-models should also be flexible enough to enable freedom in transforming the goal models. In principle, a goal model has to *extend* the domain to which the system-to-be is going to operate, and should not be constrained with too strict, previously established, boundaries.

VII. Conclusion

In this paper we presented a support for the requirements engineers in creating new models by exploiting and transforming existing models. In particular, we introduced a set of categories that indicate in which direction a model can be modified to address current customer's needs and, for each category, we defined a set of transformations to automatically modify the original model in the required direction. As a future work, we plan to perform an empirical evaluation of the approach to verify the practical effectiveness of the transformations, and the degree of understandability of the produced models.

In this work, we focused on goal models, but in the near future, we plan to extend the propose approach to deal with different graph-based requirements languages, exploiting model translation approaches [8]. To this end, we also plan to consider other transformation-oriented technologies for model representation, such as the Eclipse Modeling Framework meta model (Ecore). Finally, we aim to extend our approach to obtain a recommender system, that not only selects the model to be modified, but also suggest the transformation needed to modify it.

Acknowlegment

This work was partially supported by the LearnPAd FP7-ICT-2013.8.2 European Project.

References

[1] N. Maiden, "Requirements engineering as information search and idea discovery (keynote)," in *RE'13*, 2013, p. 1.

[2] M. A. Boden, *The Creative Mind: Myths and Mechanisms*. Weidenfeld and Nicholson, 1990.

[3] L. Baresi and R. Heckel, "Tutorial introduction to graph transformation: A software engineering perspective," in *Graph Transformation*, 2002.

[4] E. Yu, P. Giorgini, N. Maiden, and J. Mylopoulos, *Social Modeling for Requirements Engineering*. The MIT Press, 2011.

[5] A. Van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *RE'01*, 2001.

[6] C. Castro-Herrera, C. Duan, J. Cleland-Huang, and B. Mobasher, "A recommender system for requirements elicitation in large-scale software projects," in *SAC'09*, 2009, pp. 1419–1426.

[7] H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, and M. Mirakhorli, "On-demand feature recommendations derived from mining public product descriptions," in *ICSE*, 2011.

[8] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Sys. J.*, vol. 45, no. 3, pp. 621–645, 2006.

[9] J. Zhang, Y. Lin, and J. Gray, "Generic and domain-specific model refactoring using a model transformation engine," in *Volume II of Research and Practice in Software Engineering*, 2005, pp. 199–218.

[10] S. Liaskos, M. Litoiu, M. D. Jungblut, and J. Mylopoulos, "Goal-based behavioral customization of information systems," in *Advanced Information Systems Engineering*, 2011, pp. 77–92.

[11] S. Liaskos, S. A. McIlraith, S. Sohrabi, and J. Mylopoulos, "Integrating preferences into goal models for requirements engineering," in *RE'10*, 2010, pp. 135–144.

[12] S. Liaskos and J. Mylopoulos, "On temporally annotating goal models." in *iStar*. CEUR-WS.org, 2010, pp. 62–66.

[13] R. Sebastiani, P. Giorgini, and J. Mylopoulos, "Simple and minimum-cost satisfiability for goal models," in *Advanced Information Systems Engineering*, 2004, pp. 20–35.

[14] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani, "Reasoning with goal models," in *ER '02: International Conference on Conceptual Modeling*. Springer-Verlag, 2002.

[15] G. Guedes, C. T. L. L. Silva, and J. Castro, "Goals and scenarios for requirements engineering of software product lines." in *iStar*, ser. CEUR Workshop Proceedings, 2011, pp. 108–113.

[16] E. Morales, X. Franch, A. Martnez, H. Estrada, and O. Pastor, "Technology representation in i* modules." in *iStar*, ser. CEUR Workshop Proceedings, 2011, pp. 78–83.

[17] A. Rifaut, "Intentional models based on measurement theory." in *iStar*. CEUR-WS.org, 2011, pp. 144–149.

[18] S. Ingolfo, J. Mylopoulos, A. Perini, A. Siena, and A. Susi, "Nmos: from strategic dependencies to obligations." in *iStar*, ser. CEUR Workshop Proceedings, 2011, pp. 72–77.

[19] P. Giorgini, J. Mylopoulos, and R. Sebastiani, "Goal-oriented requirements analysis and reasoning in the tropos methodology," *Engineering Applications of Artificial Intelligence*, vol. 18, no. 2, pp. 159–171, 2005.

[20] A. Sutcliffe, *The Domain Theory: Patterns for knowledge and software reuse*. CRC Press, 2002.

[21] A. Sutcliffe and N. Maiden, "The domain theory for requirements engineering," *IEEE TSE*, vol. 24, no. 3, pp. 174–196, 1998.

[22] L. Chung and S. Supakkul, "Capturing and reusing functional and non-functional requirements knowledge: a goal-object pattern approach," in *IRI'06*, 2006.

[23] A. Lapouchnian and J. Mylopoulos, "Modeling domain variability in requirements engineering with contexts," in *ER'09*, 2009.

[24] K. Chen, W. Zhang, H. Zhao, and H. Mei, "An approach to constructing feature models based on requirements clustering," in *RE*, 2005.

[25] N. Niu and S. Easterbrook, "Extracting and modeling product line functional requirements," in *RE'08*, 2008.

[26] R. Ali, F. Dalpiaz, P. Giorgini, and V. E. S. Souza, "Requirements evolution: from assumptions to reality," in *Enterprise, Business-Process and Information Systems Modeling*. Springer, 2011, pp. 372–382.

[27] V. E. S. Souza, A. Lapouchnian, K. Angelopoulos, and J. Mylopoulos, "Requirements-driven software evolution," *Computer Science-Research and Development*, vol. 28, no. 4, pp. 311–329, 2013.

[28] N. A. Ernst, J. Mylopoulos, Y. Yu, and T. Nguyen, "Supporting requirements model evolution throughout the system life-cycle," in *International Requirements Engineering, 2008. RE'08. 16th IEEE*. IEEE, 2008, pp. 321–322.

[29] T. Levendovszky, G. Karsai, M. Maroti, A. Ledeczi, and H. Charaf, "Model reuse with metamodel-based transformations," in *Software Reuse: Methods, Techniques, and Tools*, ser. LNCS, 2002, vol. 2319, pp. 166–178.

[30] A. Agrawal, G. Karsai, and F. Shi, "A uml-based graph transformation approach for implementing domain-specific model transformations," *International Journal on Software and Systems Modeling*, pp. 1–19, 2003.