Automatic Generation of Use Cases from Workflows: A Petri Net Based Approach

Oscar López[1], Miguel A. Laguna[2], and Francisco J. García[3]

[1] Technological Institute of Costa Rica, San Carlos Regional Campus, Costa Rica
olopez@infor.uva.es
[2] Department of Informatics, University of Valladolid, Spain
mlaguna@infor.uva.es
[3] Department of Informatics and Automatics, University of Salamanca, Spain
fgarcia@usal.es

Abstract. This paper presents automatic generation of use cases as an alternative both to speeding up requirements elicitation and formalizing the obtained use cases to approach the requirements reuse. We propose a framework for requirements documentation as use cases that might be included in coarse grain reusable structures. In order to effectively integrate the software requirements in reusable components, adequate models promoting reusability are required. Hence, we accomplish the requirement elicitation through a process using Workflows and Petri nets. This process gives an analytical treatment to system requirements which are stored in a repository.

Keywords: Requirement engineering, use cases, workflow, Petri nets, requirements reuse.

1 Introduction

Requirements engineering triggers the software development process by producing a document containing both the necessities of stakeholders and a characterization of the software that is going to be created in a specific domain [17,13]. However, it seems that the activities of requirements engineering take too much time, thus postponing the code production. Therefore, nowadays, research is aimed at developing methods and tools to adequately document the system requirements as well as to shorten the requirements process.

Requirements reuse is an approach which can contribute to improve and quicken the requirements engineering process by systematically using existing requirements documents. Although it has received little attention [21,13], reusing early software products and processes can improve the requirements engineering process [5,20]. If the developers can benefit from requirements reuse then it is possible both to increase the productivity and to reduce the error probability in requirements specifications.

Besides its potential benefits in software engineering, requirements reuse faces as principal trade-off its difficulties to enact, to process and hence to reuse, the requirements. The documentation of the requirements is originally oriented to being a means of communication between users and analysts. For this reason, it is represented with diverse notations and formats. This diversity implies the need for particular actions to analyze requirements documents and their organization in a repository of reusable artifacts [5]. The systematic requirements reuse requires two specific actions. First, to define the adequate way to model and store specifications. Second, to define a process for selecting and adapting the reusable requirements.

In the Research Group in Reuse and Object Orientation (in Spanish, GIRO), at the University of Valladolid, Spain, we have proposed a component model called *mecano* [8]. A mecano is a coarse grain reusable element consisting of a set of fine grain elements which correspond to distinct levels of abstraction and are associated by inter-level and intra-level relations. We shall integrate the requirements documentation as an essential part of the mecano structure. System requirements documents give a characterization of the software component to be applied to a specific domain. Thus this documentation should cover representation and comprehension of the environment and the essential functions of the software product [19] in a traceable format and without ambiguity [10]. Software requirements based on natural language give us poor results in requirements specification because of its shortages, such as ambiguity, poor scalability and traceability [14]. So, we aim to establish a method to formally define requirements that promote reusability at the requirement level of abstraction.

In this paper we supply an approach to model system requirements and to store them as reusable elements (assets) on the analysis level. We look for an adequate representation for requirements facilitating comprehension, retrieval, and adaptation. From the initial system functionality based on user job we collect system requirements as scenarios for interaction between the users and the system. These scenarios are expressed as use cases looking on a syntactic and semantic formalism. The starting point is an administrative workflow [9] represented as a Case Graph (CG) that leads to the automatic generation of use cases. These use cases are stored in a repository of reusable components.

The rest of the paper is arranged as follows: Section 2 presents a reference framework for the generation of use cases and assets. In section 3 we specify the modeling process applying workflows as starting point to automate software requirements elicitation with use cases. Section 4 relates our work to other known studies. Section 5 concludes the paper and focuses on future work.

2 A General Framework for Use Cases Generation

When defining software requirements, users and analysts are related to each other within a complex communication scheme. Users are not usually sure enough about the required functionality. Therefore, system analysts or requirement engineers should correctly specify functionality through an iterative process [19]. A strong approach is needed to support user-analyst interaction and to make sure user requirements are discovered and expressed in a correct, precise, unambiguous, verifiable, traceable and modifiable form.

In addition to the difficulties of correctly obtaining user requirements from system analysis, it is accepted that the best performance of software reuse is associated with an early comprehension of system functionality. Hence, it is strongly recommended to have a process model for requirement elicitation as presented in Figure 1. This model is situated in the discrete event systems, according to Silva [18], because we are interested in the evolution of the states regardless of when the system reaches a particular state, or how long it remains in that state. In other words, requirement modeling is based on state sequences within the system. As a result, we use Petri nets in both process modeling and system requirements modeling to supply formal support for verification of the system, as has been proved in [6,14].

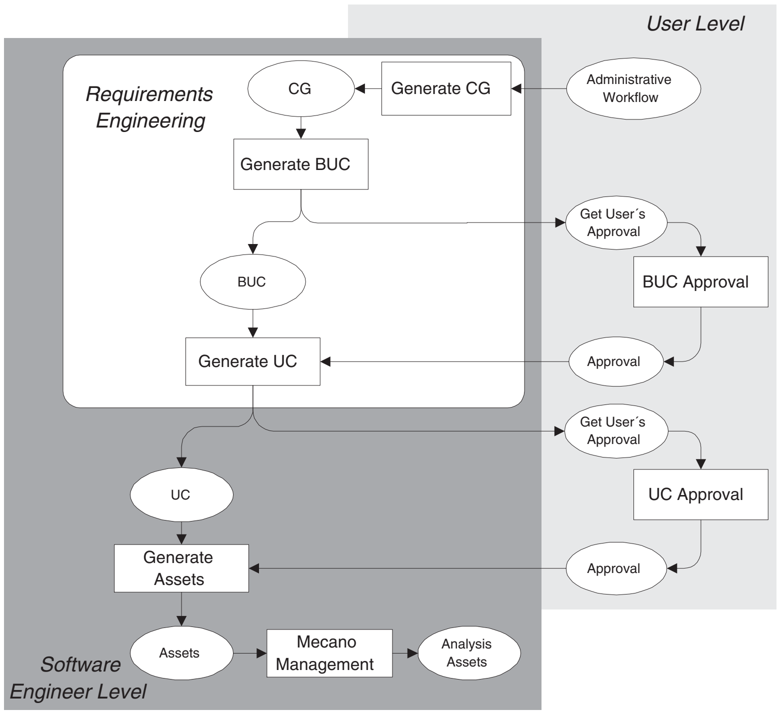

Fig. 1. General framework, arranged as a Petri net, for deriving analysis assets from a workflow

The process model includes two levels in requirement elicitation: The user level, and the software engineer level. The former has an external view (black box) of the system. The latter has an internal view (white box) of the system. Inside the Software Engineer Level one finds the Requirement Engineer View, which acts as an interface between user and engineer levels. In this way, we have arranged a general frame for the normalization of requirements elicitation.

The initial point is an administrative workflow containing the information flow. This workflow reflects the information changing from its entry into the system until it is outputted with the corresponding modifications. Workflows have been used to express business logic as recommended by the Workflow Management Coalition (WfMC) [22]. We obtain a preliminary description from the user job by a specific requirements documentation technique known as Document-Task diagram (DTd) [4]. When rigorously applied, the DTd satisfies WfMC standards indicating which tasks have to be performed in what order to transform the relevant information. In this way, our proposed methodology gives us a preliminary definition of system requirements.

The Requirements Engineering job consists of modeling the system functionality as a CG which is described in Section 3. From this CG the requirement engineer gets a set of business use cases (BUC Graphs), which require user approval to generate a set of use cases (UC Graphs). Again, use cases must be approved by the user in order to be used to generate assets. Although this is not shown, validation and verification activities gives the possibility of correcting the initial version of the CG, and if required, the process can ask for a new release of the CG and, consequently, new BUC Graphs and UC Graphs.

The verified CG can be used by the software engineer to obtain assets as templates of use cases, as in Durán [7]. These assets are sent to a mecano manager to iterate with the repository, and to produce the corresponding mecanos for storing in a repository. Obtaining use cases as templates, as well as the iteration process with the repository, is beyond the scope of this paper.

3 Problem Modeling

3.1 The Case Graph Definition

A DTd is a particular system documentation technique which models business logic. By adding information, the DTd becomes a CG which specifies the tasks to be done in a given order. Both tasks and documents, joined by arcs, are the foundation of a CG.

Definition 1. *Case Graph: A Case Graph is a four-tuple (D,T,A,E), where:*

- *D is a finite set of documents*
- *T is a finite set of tasks,* $(D \cap T = \emptyset)$, $T^p = \{T_{(AA)}, T_{(AO)}, T_{(OA)}, T_{(OO)}\}$ *is a disjointed partition of T. That is, in a Case Graph there are four types of tasks.*
- *A is a set of arcs,* $A \subseteq ((D \times T) \cup (T \times D))$
- $E : D \cup T \rightarrow \Sigma^+$ *is a label function which relates a distinct label to each document and to each task.* Σ^+ *is a finite set of labels.*

Tasks are the transition points between documents, and there are four different kinds of tasks (AA, OA, AO, OO). Each task contains a list of internal sub-tasks. Graphical icons of tasks are shown in Figure 2.A. We have chosen a representation showing requirements for a task to be executed in an organizational context. To execute a task, both required inputs (input documents) and pre-conditions must be met. Carrying out a task leads to both output documents and holding the post-condition. Sometimes a temporal event is also required in order to execute a task. Additionally, each task has an associated operator who is directly responsible and in charge of the same task. Graphical representation of tasks is shown in Figure 2.B.

The different kinds of tasks have distinctive behavior. The AA type requires all its input documents to be enabled, and its triggering leads to all its output documents. The OA is enabled with one of its input documents, and its triggering leads to all its output documents. The AO requires all of its input documents to be enabled, and its triggering leads to one of its output documents. The OO type is enabled with one of its input documents, and its triggering leads to one of its output documents.

We have represented the four basic types of routing in CGs. These types of routing are shown in Figure 2.C. Sequential actions and parallel actions may be described using AA tasks. Nevertheless, conditional actions and iterative actions require different arrangements of AA, OA, AO, and OO tasks.

To formally treat the resulting CGs, we need to standardize the behavior of the tasks. The Petri nets, which have been applied to express the workflow semantics [6], can help us to supply formal support for the analysis of CGs. In this way, the tasks are visualized as Petri net transitions. Documents are represented by Petri net places. The Petri net marking represents the sequence of states. As OA, AO, and OO tasks become AA tasks, the CG becomes a Petri net. Figure 2.D graphically shows this transformation of OA, AO and OO tasks into AA tasks, and next in algorithm 2 we give its formal expression.

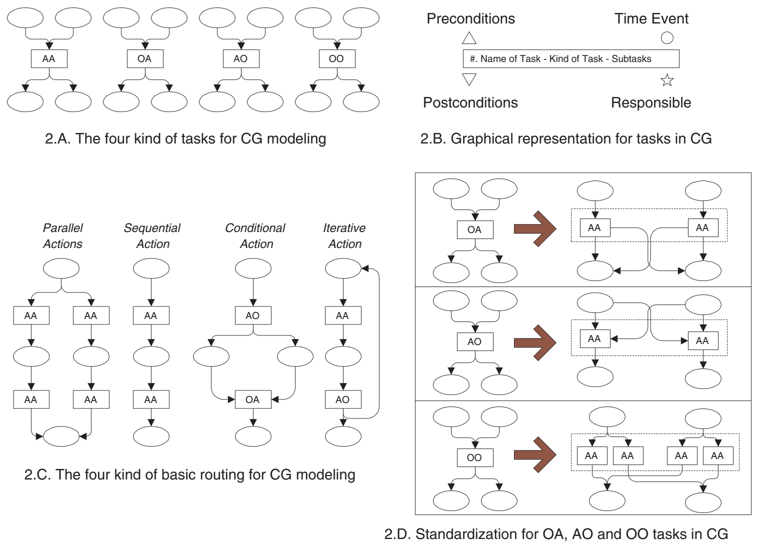
AA
OA
AO
OO
2.A. The four kind of tasks for CG modeling
Preconditions
Time Event
#. Name of Task - Kind of Task - Subtasks
Postconditions
Responsible
2.B. Graphical representation for tasks in CG
Parallel Actions
Sequential Action
Conditional Action
Iterative Action
2.C. The four kind of basic routing for CG modeling
2.D. Standardization for OA, AO and OO tasks in CG

Fig. 2. Representation for tasks, transformations and action flows for Case Graph modeling

To illustrate how to apply our approach for deriving use cases, we have taken a DTd from a real case of a Spanish Electrical Enterprise. It is represented as a CG as shown in Figure 3. We take four general processes - Request Process, Resolution Process, Execution Process, and Refund Process - taking place in two organizational units represented by two different internal actors - Control Centre (CC), and Local Operator (Op)- which are responsible for the actions. There are seven external actors: Particular Customer, Area Agent, Requester, Security Committee, Affected Customer, Enterprise Organizations, and Tax Organizations.

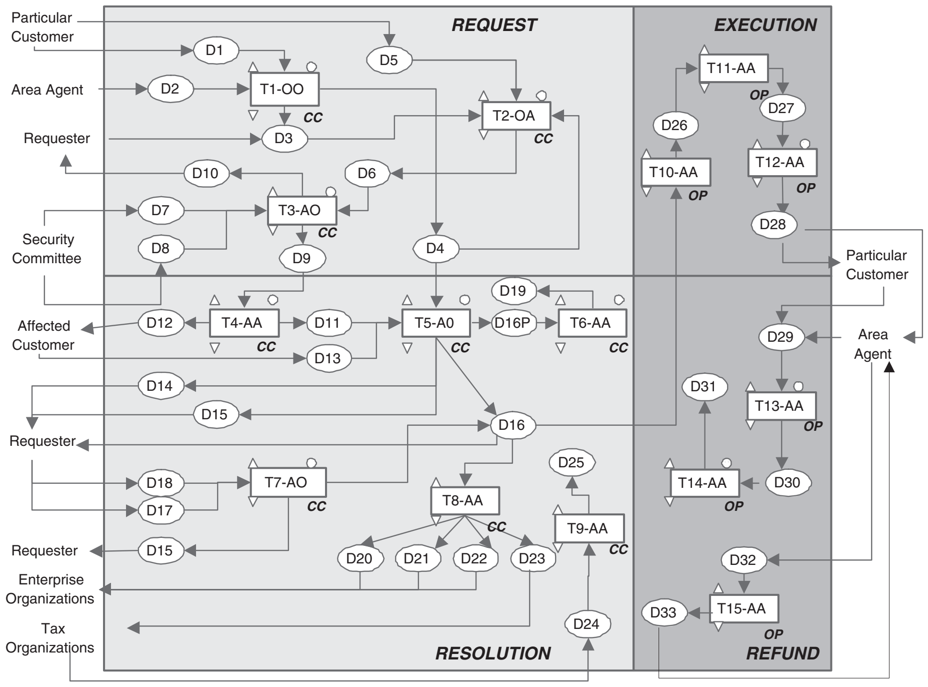

Fig. 3. Graphs of Cases for four processes in two departments of an organization

3.2 BUC Graphs and UC Graphs

Because the Case Graph represents system functionality, it might indicate what the interaction between users and the system is. Jacobson [12] differentiates two types of interaction as use cases: Business Use Cases (BUC), and Use Cases (UC). According to our framework, BUC and UC are generated from a CG as BUC Graphs and UC Graphs, respectively.

Both BUC Graphs and UC Graphs reflect possible sequences of interaction between actors and system. A BUC Graph corresponds to an external actor. A UC Graph corresponds to an internal actor. Both BUC Graphs and UC Graphs contain paths in which the action flow can follow.

Definition 2. Path: *Let G= (D,T,A,E) be a Case Graph,* $N = \{n_i, i = 1..n,$ *such that* $n_i \in D \cup T\}$. *The R path from a node* n_1 *to* n_k *is a sequence* $(n_1, n_2, .., n_k)$ *such that* $(n_j, n_{j+1}) \in A, \forall j \in \{1..k-1\}$.

This definition means that a path is any logical sequence of action in the current system. This sequence is formed by documents and tasks that should be joined by arcs inside the CG. For example, in Figure 3, the sequence D3,T2,D6,T3,D10 is a path.

Tasks are the transition points in documents. Each task has two associated sets of documents, previous documents and post documents. Formally, these two sets are defined as follows:

Definition 3. Previous Documents and Post Documents: *Let G= (D,T,A,E) be a Case Graph, the set of previous documents of task t* $(t \in T)$ *is defined by* $^{o}t = \{d_i \in D \mid \exists x \in A$*, x connects* d_i *to t*$\}$. *Analogously, the set of post documents of task t* $(t \in T)$ *is defined by* $t^{o} = \{d_i \in D \mid \exists x \in A$*, x connects t to* $d_i\}$. *Similarly, the notations* ^{o}p *and* p^{o} *mean the set of previous transitions and post transitions of place p* $(p \in P)$.

To assure the consistency of the BUC Graphs and the UC Graphs, we should guarantee that all nodes are achievable. It is a strongly connected Case Graph when a path exists that connects any two points in the graph.

Definition 4. Strongly connected: *Let G= (D,T,A,E) be a Case Graph. G is strongly connected if* $\forall x \in N, \forall y \in N, N = \{n_i, i = 1..n,$ *such that* $n_i \in D \cup T\}$, *then a path exists leading from x to y.*

According to Cockburn [3], use cases describe how users use the system. Our Case Graph should describe possible interaction flows between users and the system. Consequently, the term *Case Sequence* needs to be defined.

Definition 5. Case Sequence: *Let G= (D,T,A,E) be a Case Graph. G is a Case Sequence (CS) if:*

1. *The set D has two special documents* ***i*** *and* ***o****. The place* ***i*** *is a source place,* $^{o}i = \emptyset$. *The place* ***o*** *is a sink place,* $o^{o} = \emptyset$.
2. *If a task t' is added to T, t' connects the documents o and i (i.e.* $\{$*o,t',i*$\}$ *is the path from o to i), then the resulting Case Graph is strongly connected.*
3. *No symmetric associations exist between documents and tasks. In other words,* $\forall t_i \in T, {^{o}t_i} \cap {t^{o}}_i = \emptyset$ *is satisfied.*

This definition of Case Sequence is coincident with the Workflow Net definition given by Van der Aalst [6], specifically regarding conditions 1 and 2. We have added the condition 3 in order to avoid potential deadlocks and/or livelocks in our Case Sequence definition. Adopting this Workflow Net definition we are sufficiently sure about the soundness property of a procedure modeled by a Case Sequence.

We can now define a BUC Graph and a UC Graph. A BUC Graph is a Case Graph that contains all the possible case sequences for an input document from an external actor. A UC Graph is a CG containing all possible case sequences for an internal actor inside a BUC Graph.

Definition 6. BUC Graph: *Let G= (D,T,A,E) be a Case Graph. G is a BUC Graph if it contains all the possible case sequences for an input document from an external actor.*

Definition 7. UC Graph: *Let G= (D,T,A,E) be a Case Graph. G is an UC Graph if:*

1. *It contains case sequences which correspond to an internal actor inside a BUC Graph.*
2. *All the tasks of G are of the AA kind. That is,* $T = T_{(AA)}$ *so* $(T_{(AO)} \cup T_{(OO)} \cup T_{(OA)}) = \emptyset$ *is satisfied.*

Twelve BUC Graphs have been identified in our case study, one for each input document, and are presented in Figure 4. This figure also represents 10 UC Graphs (the different shadowed areas). The CG contains a set of BUC_i Graphs. Each BUC_i Graph is a set of UC_i Graphs. Both a BUC_i Graph and a UC_i Graph consist of internal structures and external interfaces. The shared documents and the shared tasks are considered part of the external interfaces. The internal structures are the same as in a Case Graph, according to definitions 1 to 7.

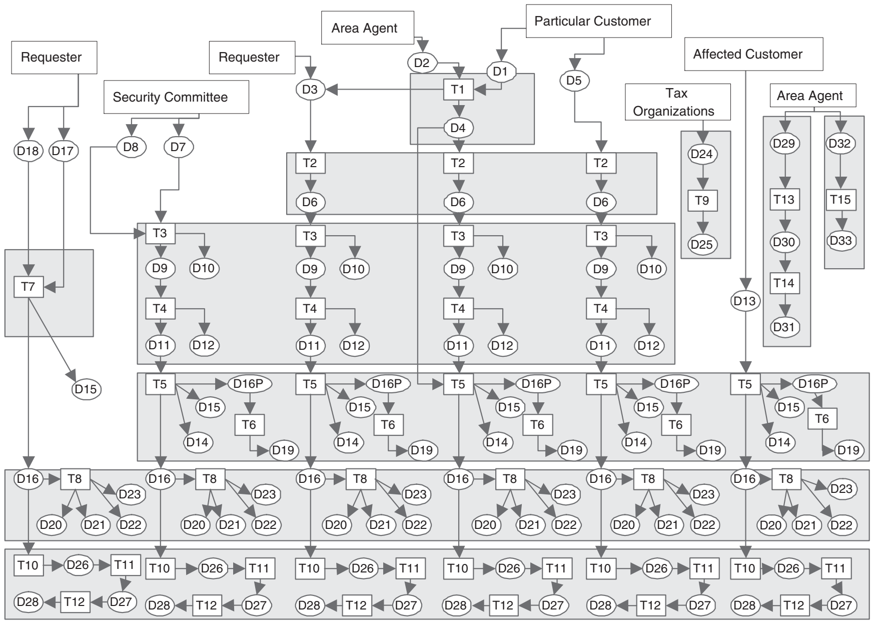

Fig. 4. Case Sequences for four processes in two departments of an organization

3.3 The Modular Case Graph

Since the same document could be directed to, or directed from, distinct tasks, the documents and tasks can be shared by different sequences of case. This means that different sequences of case can be intersected.

While BUC Graphs can share documents and tasks, UC Graphs can only share documents. Each UC Graph indicates the way the information is handled through different case sequences inside a BUC Graph. Because information is handled by tasks, these are not sharable between UC Graphs.

Shared documents acting as connection points between UC Graphs leads the Case Graph to be viewed as a modular structure.

Definition 8. Modular Case Graph: *A Modular Case Graph (MCG) is a set* $\{G_i = (D_i, T_i, A_i, E_i), i = 1 \ldots n\}$ *where:*

- *Each G_i is a UC Graph*
- *All T_i must be disjointed for every G_i*
- *The same label must not be used for documents and tasks, thus, $\forall G_i, \forall G_j, \neg\exists d \in D_i, \neg\exists t \in T_j$ such that $E_i(d) = E_j(t)$*

A process must be carried out on the CG in order to obtain the MCG. This process is composed of factorization of the GC and refining and transforming tasks. It leads to the discovery of common blocks across the BUC Graphs.

Factorization of the CG. The factorization leads to the specification of the common blocks of CG. Identifying these common blocks leads to the factorized expression of the CG without changing the particular structure of each BUC Graph. Consequently, only strictly necessary abstractions of use cases can be done to specify the system functionality.

Algorithm 1. Factorization of the CG: *Let M= (D,T,A,E) be a Case Graph.*

```
1. Let S be $\{CS_d, d \in D$ such that $CS_d$ is a Case Sequence corresponding to the external document d $\}$.
2. For every $CS_d \in S$
   (a) Let $T_{cs_d}$ become $\bigcup_{i=1}^{n} T_{cs_{di}}$ such that $T_{cs_{di}}$ is the task set of the CS corresponding to document d and internal actor i
3. Let $T^{\sim}_{cs_{di}}$ be recursively defined as follows:
   (a) $T^{\sim}_{cs_{d1}} = T_{cs_{d1}}$
   (b) $\forall i = 2..n, T^{\sim}_{cs_{di}} = T_{cs_{di}} \setminus \bigcup_{j=1}^{i-1} T^{\sim}_{cs_{dj}}$
4. Let every $T^{\sim}_{cs_{di}}$ and the associated documents, arcs and labels be a preliminary UC Graph
```

This algorithm allows a factorization of CG to be obtained. Every CS belonging to a specific CG is divided into fragments. These fragments are formed by the $T^{\sim}_{cs_{di}}$ and the associated documents, arcs and labels. Each fragment is taken as a preliminary UC Graph because each of these blocks may contain different kinds of tasks. In this way a factorized form of the initial CG is obtained.

Following our example, we obtain the common blocks to different BUC Graphs which are shown in Figure 4. It can be BUC Graphs without common blocks. For example, the ones corresponding to D24, D29, and D32 documents. The other BUC Graphs have common structures. The factorized form of the CG is shown in Figure 5.

The distinct blocks from Figure 5 are only previous to every UC Graph because each of these blocks has to be transformed until it contains only AA tasks. The transformation process from OA, AO, and OO tasks to AA tasks implies refining tasks combining only AA tasks. We propose this job be done automatically. Modular Case Graphs (MCG) will result from this transformation. We must point out that the model of Figure 5 is only a base for the MCG.

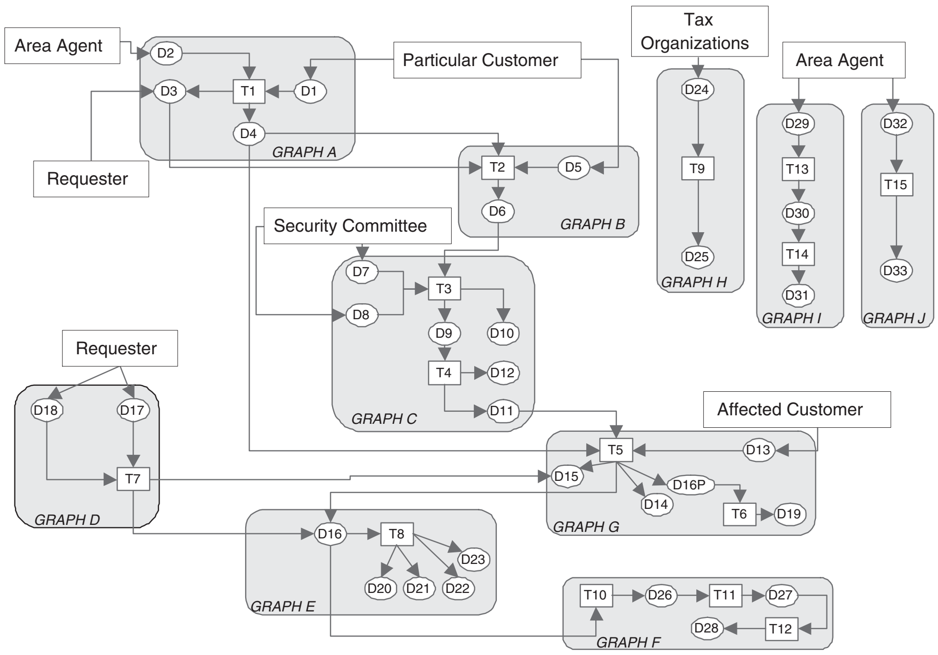

Fig. 5. Factorization of BUC Graphs for the processes of the organization

Refining and Transforming Tasks. Use cases can be extracted automatically from a factorized Case Graph. Hence we propose an algorithmic process for refining and transforming the tasks if required. This process should lead to the expression of factorized Case Graphs by combining UC Graphs. This must take the kind, precondition, and post condition of each task into account. Moreover, it is necessary to know how many documents are being input, and how many are being output in each task.

Algorithm 2. Refining and Transforming Tasks: *Let M= (D,T,A,E) be a Case Graph.*

```
1. For every $t \in T$ do
   (a) Case of $t \in T_{(OA)}$
       i. Let t become $\{t_{(AA)s}\}$ such that $s = 1, 2, \ldots, j$; where $j = |{}^{o}t|$, $t_{(AA)s}$ is an AA task
       ii. Every $d \in {}^{o}t$ is connected to a distinct $t_{(AA)s}$ such that ${}^{o}t_{(AA)s} = 1$
       iii. Every $d \in t^{o}$ is connected to each $t_{(AA)s}$ such that ${t_{(AA)s}}^{o} = |t^{o}|$
       iv. Let $T = T \backslash \{t\} \bigcup \{t_{(AA)s}\}$
   (b) Case of $t \in T_{(AO)}$
       i. Let t become $\{T_{(AA)s}\}$ such that $s = 1, 2, \ldots, j$; where $j = |t^{o}|$, $t_{(AA)s}$ is an AA task
       ii. Every $d \in {}^{o}t$ is connected to each $t_{(AA)s}$ such that ${}^{o}t_{(AA)s} = |{}^{o}t|$
       iii. Every $d \in t^{o}$ is connected to a distinct $t_{(AA)s}$ such that ${t_{(AA)s}}^{o} = 1$
       iv. Let $T = T \backslash \{t\} \bigcup \{t_{(AA)s}\}$
```

(c) *Case of $t \in T_{(OO)}$*
   i. *Let t become $\{T_{(AA)s}\}$ such that $s = 1, 2, \ldots, j$; where $j = |^o t| \times |t^o|$, $t_{(AA)s}$ is an AA task*
   ii. *Every $d \in {}^o t$ is connected to $|t^o|$ distinct $t_{(AA)s}$ such that ${}^o t_{(AA)s} = 1$*
   iii. *Every $d \in t^o$ is connected to $|^o t|$ distinct $t_{(AA)s}$ such that ${t_{(AA)s}}^o = 1$ and $\forall t_{(AA)s1}, t_{(AA)s2}$ if ${}^o t_{(AA)s1} = {}^o t_{(AA)s2}$ and ${t_{(AA)s1}}^o = {t_{(AA)s2}}^o$ then $t_{(AA)s1} = t_{(AA)s2}$*
   iv. *Let $T = T \backslash \{t\} \bigcup \{t_{(AA)s}\}$*

2. *Take every $t \in T$ as a chain of generic tasks*

This algorithm 2 allows the OA, AO, and OO tasks to be transformed following the pattern given in Figure 2.D. The behaviour of the algorithm depends on the type of task to be refined or transformed. The OA tasks are refined to as many sub-tasks as there are inputs, each input is connected to one sub-task, each sub-task is also connected to all outputs. The AO tasks are refined to as many sub-tasks as there are outputs, each input is connected to all sub-tasks, each sub-task is also connected to one output. The OO tasks are refined to as many sub-tasks as the product of the amount of inputs and outputs, each input is connected to as many sub-tasks as outputs in the mother task, each sub-task is also connected to an output from every input document.

In this way, the refining and transforming process for each task leads to a Case Graph expressed in terms of the AA standard. If all blocks are expressed with only AA tasks then the definition of a UC Graph is satisfied. If each UC Graph corresponds to a Petri net then the entire Modular Cases Graph is also a Petri net. On the other hand, the validation and verification activities lead to the adjustment of the tasks if required.

The algorithm 2 establish also each task is formed by a chain of generic sub-tasks. If we consider each task as a chain of sub-tasks then we refine each one as this sequence of sub-tasks. Once again, following the example, let us consider data from table 1. We consider each task to be formed by the chain: *Verify, Process, and Generate.*

Table 1. Data for refining the T1 and T2 tasks of the Case Graph of an organization

Task Name	*Details*	*Kind of Task*	*Responsible*
T1: Fill out unloading form	*Verification of* precondition *Processing* data and establish postcondition *Generate* D3 or D4	OO	Control Centre
T2: Receive unloading form	*Verification of* precondition *Processing* data and establish postcondition *Generate* D6	OA	Control Centre

These generic sub-tasks allow the set of steps for use cases to be obtained. The *Verification* sub-task is directly related to precondition and to inputs. The *Process* sub-task is in charge of establishing the post-condition. The *Generate* sub-task is in charge of producing the output of the mother task.

3.4 Obtaining Use Cases

As established in definition 6, the BUCs are obtained as graphs containing all case sequences. An external actor inputting a document starts each sequence. Then, in definition 7, we afirm that obtaining the use cases requires transforming OA, AO, and OO tasks to AA tasks. In algorithms 1 and 2 we establish how to process the CG to obtain an MCG. Finally, having refined and transformed tasks, use cases may be derived and expressed as a template. To do this, we can follow the marking of the Petri net. Some standards for naming intermediate documents must be defined. It is also necessary to decide upon the general naming of use cases to write them as a template, as proposed by Durán [7].

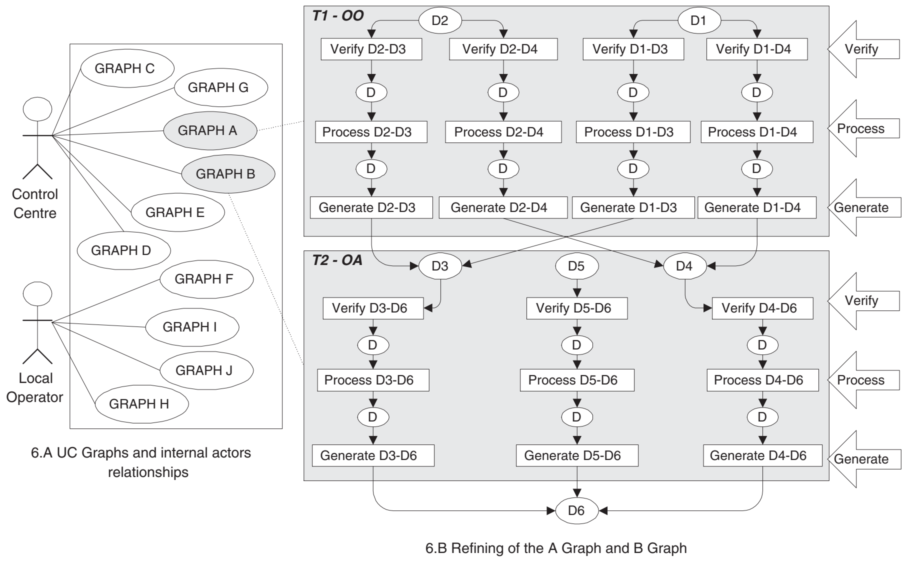

Fig. 6. Relationship between UC Graphs and actors, and refining of the A and B Case Graphs

Figure 6.A shows the relationships between the distinct UC Graphs and the internal actors. Figure 6.B shows A and B cases containing T2 and T3 transformed tasks according to what was established in algorithm 2. One can see that figure 6.B contains all different flows of sequential actions corresponding to the A Graph and B Graph, after refined the T1 and T2 tasks.

Use cases, written as templates, or expressed as a Petri net, have to be sent to a Mecano Manager to finally become adjust assets under the corresponding semantic of the repository. In this way, it is possible to conform the mecanos for reuse, as established by García [8]. On the other hand, expressing use cases under Petri net semantics should lead to establishing a way to organize the requirements of a domain. The definition of these mechanisms both to organize the requirements and write them as templates are being addressed in our future work.

4 Related Work

Few studies have looked at reusing software at requirements level. Besides it is known that requirements from domains and similar tasks are more likely to show similarities than other software elements [21], the requirements reuse has received relatively little attention. A tool and methodological support is necessary to reuse the software requirements [1]. Complex structures to accomplish the requirements related to design and code have been proposed [8]. Nevertheless, to integrate requirements assets and design assets and code assets, it is necessary to find out how to adequately model them. Requirements are modeled with a diversity of techniques so we aimed at normalization to ensure requirements reusability when stored in a repository. The normalized requirements when related to assets of different levels of abstraction provide a large grain interface to increase the abstraction level of the reuse process.

Software requirements have to be treated adequately inside a reuse strategy. Requirements have to be previously classified to be retrieved when developers send their queries. Classification and retrieval of general assets have been approached with schemes and methods based on text, ontology or facets. However, requirements contain knowledge from both the domain and the development process. The complexity of this knowledge forces the application of sophisticated techniques for requirements classification and retrieval [5]. To efficiently answer the queries by selecting, composing or generating elements, a robust strategy based on software requirements reuse is required.

Common classification and retrieval techniques show limited utility in representing requirements for reuse [5]. Some different alternatives based on knowledge representation [15] and analogical reasoning [16] to reuse the requirements from a knowledge base have been proposed. These techniques place emphasis on the semantics of requirements documents and it demands artificial intelligence applications to acquire and manage the encoded knowledge in requirements documentation. Other techniques are based on meta-models [17], evolutionary development [2] and formal methods [23] all of which emphasize the process for development and maintenance of the reusable requirements. Cybulski [5] has proposed a set of techniques based on structural properties of documents and tasks emphasizing the way to enact and use the requirements in the life cycle of requirements engineering.

Scenarios approaches like use cases (UC) and business use cases (BUC) [12,11] are helpful and widely accepted in requirements elicitation. They offer several advantages in overcoming natural language shortcomings. Particularly, the UC approach solves scalability and traceability shortcomings, and it supplies facilities for description. Nevertheless, according to Lee, Cha and Kwon [14], the UC approach does not correct natural language ambiguity. We also believe that use cases do not supply facilities for organizing software requirements to software development with reuse.

5 Conclusions and Future Work

To integrate system requirements in complex reuse structures, adequate models to promote requirements reusability are required. In the present paper we have proposed a method to automatically obtain analysis assets as use cases from an administrative workflow. We have shown that use cases can be generated from a workflow modeling the information flow in a business domain.

We have applied an elicitation process of system requirements through a Case Graph and a Petri net. The established process leads to formalized analysis assets taking part in a reusable component called mecano. With formalized functionality we are in a position to support different analysis to systematically investigate the component behaviour. Furthermore, from sets of use cases we hope to establish ways to organize the information about system requirements.

Proposed CG is an unambiguous alternative to model business dynamics with little information. The CG represent activity flows by a control scheme formalizing the requirements elicitation process and accomplishing scalability and traceability. However, this CG based approach should be proven and validated in the definition of system requirements. We expected their functionality, representing a domain and founded on Petri net theory, will be applicable for obtaining good performance in managing the requirements information.

Our middle term objective is to integrate different requirements notations within requirements reuse strategy. We have found the Petri net approach useful to obtain a precise expression of system requirements from workflow. However, classical Petri nets have shown such net explosion in system modeling as we look for a more compact representation. We are working on a high level Petri net based approach, specifically coloured Petri nets, to represent analysis assets. Our next job will also supply:

- A tool to automatically generate use cases from Case Graphs.
- A model to support organization of system requirements in a domain.

Acknowledgments

This work is sponsored by the DOLMEN Project within CICYT-TIC2000-1673-C06-05, Ministry of Technology and Science, Spain. Oscar López wishes to thank the Spanish Agency for International Cooperation (AECI) and the Ministry of Technology and Science of Costa Rica.

References

1. K. Barber, S. Jernigan, and T. Graser. Increasing opportunities for reuse through tool and methodology support for enterprise-wide requirements reuse and evolution, 1999.
2. Roberto Bellinzona, Maria Grazia Fugini, and Vicki de Mey. Reuse of specifications and designs in a development information system. In N. Prakash, C. Rolland, and Barbara Percini, editors, *Information System Development Process*, pages 79–96, Amsterdam, 1993. North-Holland.
3. A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, Boston, 2000.

4. Collonges, Huges, and Laroche. *Merisse. Methode de conception*. Dunod, France, 1987.
5. Jacob L. Cybulski. Patterns in software requirements reuse. Technical report, Department of Information Systems. University of Melbourne, July 1998.
6. V. d. Aalst. The Application of Petri Nets to Workflow Management. Eindhoven, University of Technology, Netherland, URL: `http://wwwis.win.tue.nl/~wsinwa/jcsc/jcsc.html`, 1997.
7. A. Durán, B. Bernárdez, M. Toro, R. Corchuelo, and J.A. Pérez. Expressing customer requirements using natural language requirements templates and patterns. In *Modern Applied Mathemathics Techniques in Circuits, Systems and Control*, pages 337–342, Atenas, Greece, 1999. World Scientific and Engineering Society Press.
8. Francisco José García, Antonio Barras, Miguel Ángel Laguna, and José Manuel Marqués. Product line variability support by FORM and mecano model integration. In *ACM Software Engineering Notes*, To appear 2002.
9. Dimitrios Georgakopoulos, Mark F. Hornick, and Amit P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.
10. IEEE. *IEEE Software Engineering Standard Collection. 1999 Edition*. IEEE Computer Society Press, 1999.
11. Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, 1999.
12. Ivar Jacobson, M. Christerson, Patrik Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992. (Revised 4th printing, 1993).
13. I. Kotonya, G.; Sommerville. *Requeriments Engineering: Processes Techniques*. USA Wiley, 1997.
14. Woo Jin Lee, Sung Deok Cha, and Yong Rae Kwon. Integration and analysis of use cases using modular Petri nets in requirements engineering. *IEEE Transactions on Software Engineering*, 24(12):1115–1130, December 1998. Special Issue: Scenario Management.
15. M. Lowry and R. Duran. Knowledge-based software engineering. In *The Handbook of Artificial Intelligence*, pages 241–322, Massachusetts, 1989. A. Barr, P.R. Cohen, and E.A. Feigenbaum, Editors. Addison-Wesley.
16. N. Maidenand and A. Sutcliffe. Exploting reusable specification through analogy. *Communications of ACM*, 35(4):55–64, 1993.
17. Colette Rolland and Naveen Prakash. From conceptual modelling to requirements engineering. Technical Report Series 99-11, CREWS, 1999.
18. Manuel Silva. *Las Redes de Petri en la Automática y la Informática*. Editorial AC, Madrid, España, 1985.
19. I. Sommerville. *Software Engineering*. Addison-Wesley, USA, 6th edition, 2001.
20. A. Sutcliffe, N. Maiden, S. Minocha, and D. Manuel. Supporting scenario-based requirements engineering. *IEEE Transactions on Software Engineering*, 24(12), December 1998.
21. Axel van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *22nd. International Conference on Software Engineering*, Limerich, June 2000. ACM Press.
22. WfMC. The workflow management coalition. Terminology and Glossary. Document Number WFMC-TC-1011. United Kingdom, 65 pages. feb 99.
23. M.R. Wirsing, R. Hennicker, and R. Stabl. Menu - an example for the systematic reuse of specifications. In *2nd European Software Engineering Conference*, pages 20–41, Coventry, England, 1989. Springer-Verlag.