A Model for Visual Specification of e-Contracts

Enrique Martínez, Gregorio Díaz, M. Emilia Cambronero
Department of Computer Science
University of Castilla - La Mancha
Albacete, Spain
{*emartinez, gregorio, emicp*}*@dsi.uclm.es*

Gerardo Schneider
Department of Applied IT
University of Gothenburg, Sweden
Department of Informatics
University of Oslo, Norway
gersch@chalmers.se

***Abstract*—In a web service composition, an electronic contract (e-contract) regulates how the services participating in the composition should behave, including the restrictions that these services must fulfill, such as real-time constraints. In this work we present a visual model that allows us to specify e-contracts in a user friendly way, including conditional behavior and real-time constraints. A case study is presented to illustrate how this visual model defines e-contracts and a preliminary evaluation of the model is also done.**

***Keywords*-contracts; deontic specifications; visual models;**

I. Introduction

Most of the research efforts spent on the theory of electronic contracts in service-oriented architectures have been oriented to the formal definition of a public service interface with which other services can interact [5]. However, services e-contracts not only refer to the interfaces provided by these services, they also refer to a certain number of clauses that must be satisfied by several parties. These clauses regulate how participants should behave, what are the penalties in case of misbehavior, and under which conditions such clauses must be enacted (e.g. time restrictions such as deadlines). When a clause is violated, the contract is breached. However, if the clause defines a reparation (secondary clauses that come into force when the main clause is not satisfied), and this reparation is fulfilled, then the clause is eventually fulfilled [1].

Recently some works about specifying services e-contracts in a formal manner have been released [2], [4], [7], [18]. These approaches consist of formal languages which are hard to study and manipulate by untrained final users of this technology, as business process developers.

The goal of this work is to introduce a new approach for the specification of e-contracts in a user friendly way. E-contracts may be complex, consisting of composite clauses making reference to other clauses in the same or in another contract. Furthermore, we consider contracts with timed restrictions and conditions under which the contract clauses must be applied. Hence, our approach is based on a visual model, since it is well-known that the use of visual models makes easier the perception of knowledge, and in this way, the intuitive understanding, reading and maintenance of complex problems [8], [9]. This approach can be useful not only in service-oriented architectures but also in component-based systems, requirements acquisition, software product lines, etc.

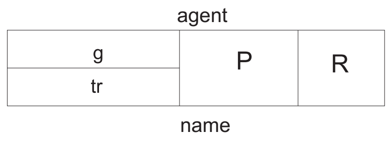

Figure 1. Box structure

The contribution of this work is twofold. First, we define a visual model to deal with the acquisition and elicitation of requirement and restrictions. This visual model allows us to specify the notions of obligation, permission and prohibition [14] as elements of a hierarchical diagram. In this way, these elements are clauses that can be refined hierarchically and can include (real-time) constraints and a reparation that must be performed when the main norm is not fulfilled. Second, a preliminary evaluation of the model is presented, based on user-based tests and the principles defined in [16].

The rest of the work is structured as follows: Section II describes the visual model we have developed, showing in Section III a case study where this model is applied. In Section IV we present the results of the preliminary evaluation and Section V is concerned with related work. Finally, in Section VI, we present the conclusions and future work.

II. Visual Model

In our visual model we define a hierarchical tree diagram used to specify the contract clauses. We call this diagram *Contract-Oriented Diagram* or *C-O Diagram* for short.

In Figure 1 we show the basic element of our *C-O Diagram*. It corresponds to a contract clause and we call it **box**. This box consists of four fields, allowing us to specify normative aspects or simple norms (**P**), reparations (**R**), conditions (**g**) and time restrictions (**tr**). Each box has a name and an agent. The *name* is useful both to describe the clause and to reference the box from other clauses, so it must be unique. The *agent* indicates who is the performer of the action.

On the left-hand side of the box we specify the conditions and restrictions. The *guard* **g** specifies the conditions under which the contract clause must be taken into account. The

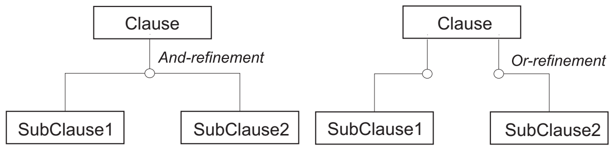

Figure 2. AND/OR refinements

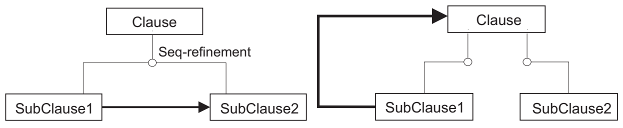

Figure 3. SEQ refinement and repetition in *C-O Diagram*

time restriction **tr** specifies the time frame in which the contract clause must be satisfied.

The *propositional content* **P**, on the center, is the main field of the box, and it is used to specify normative aspects (obligations, permissions and prohibitions) that are applied over actions, and/or the actions themselves.

The last field of these boxes, on the right-hand side, is the *reparation* **R**. This reparation, if specified by the contract clause, is another contract that must be satisfied in case the main norm is not satisfied, considering the clause eventually satisfied if this reparation is satisfied.

These basic elements of a *C-O Diagram* can be refined by using AND/OR refinements, as shown in Figure 2, in the same way that we refine goals into subgoals in goal model diagrams [21]. It is also possible to use another refinement to specify a temporal relationship of sequence (SEQ) between the subclauses, as shown in the left part of Figure 3. The aim of these refinements is to capture the hierarchical clause structure followed by most contracts. An **AND-refinement** means that all the subclauses must be satisfied in order to satisfied the parent clause. An **OR-refinement** means that it is only necessary to satisfy one of the subclauses in order to satisfy the parent clause, so as soon as one of its subclauses is fulfilled, we conclude that the parent clause is fulfilled as well. A **SEQ-refinement** means that the norm specified in the target box (*SubClause2* in Figure 3) must be fulfilled after satisfying the norm specified in the source box (*SubClause1* in Figure 3). In this way, we can build a hierarchical tree with the clauses defined by the contract, where the leaf clauses correspond to the atomic clauses, that is, to the clauses that cannot be divided into subclauses.

Finally, there is another structure that can be used to model **repetition**, apart from the refinements previously defined. This structure is represented as an arrow going from a subclause to one of its ancestor clauses (or to itself), meaning the repetitive application of all the subclauses of the target clause after satisfying the source subclause. For example, in the right part of Figure 3, we have an **OR-refinement** with an arrow going from *SubClause1* to *Clause*. It means that after satisfying *SubClause1* we apply *Clause* again, but not after satisfying *SubClause2*.

In the next paragraphs we describe each one of the fields of a box in more detail.

Propositional Content P: This is the main field of a box. It allows us to specify the *obligations*, *permissions* and *prohibitions*, as defined in deontic logic [15], that the contract must be satisfied. In this work, we follow an *ought-to-do* approach [22], i.e., these normative aspects are applied over *actions* performed by the participants in the contract.

Although we will see later that it is possible to specify *compound actions*, we only consider the specification of *atomic actions* in the **P** field of the leaf clauses of our diagrams. These actions are denoted by lower case Latin letters ("a","b","c", $\dots$). We use a dash ("-") to denote that there is no action specified in the no leaf clauses.

The composition of actions can be achieved by means of the different kinds of refinement. In this way, an AND-refinement can be used to model *concurrency* "&" between actions, an OR-refinement can be used to model a *choice* "+" between actions, and a SEQ-refinement can be used to model *sequence* ";" of actions. In Figure 4 we can see an example about how to model these compound actions through refinements, given two atomic actions a and b.

The *deontic norms* (obligations, permissions and prohibitions) that are applied over these actions can be specified in any clause of our *C-O Diagrams*, affecting all the actions in the leaf clauses that are subclauses of this clause. If it is the case that the clause where we specify the deontic norm is a leaf clause, the norm only affects the atomic action we have in this clause. We use an upper case "*O*" to denote an obligation, an upper case "*P*" to denote a permission, and an upper case "*F*" to denote a prohibition (forbidden). These letters are written in the top left corner of field **P**.

The composition of deontic norms is also achieved by means of the different refinements we have in *C-O Diagrams*. Thus, an AND-refinement corresponds to the *conjunction* operator "$\wedge$" between norms, an OR-refinement corresponds to the *choice* operator "$+$" between norms, and a SEQ-refinement corresponds to the *sequence* operator ";" between norms. For example, we can imagine having a leaf clause specifying the obligation of performing an action a, written as $O(a)$, and another leaf clause specifying the obligation of performing an action b, written as $O(b)$. These two norms can be combined in the three different ways mentioned before through the different kinds of refinement (Figure 5).

However, the specification of obligations, permissions and prohibitions in our diagrams must fulfill the following rules:

1. At least one deontic norm must be specified in each one of the branches of our hierarchical tree of clauses, i.e., we cannot have an action without a deontic norm applied over it.
2. No more than one deontic norm can appear in each one of the branches of our hierarchical tree of clauses, i.e., we cannot have deontic norms applied over other deontic norms.

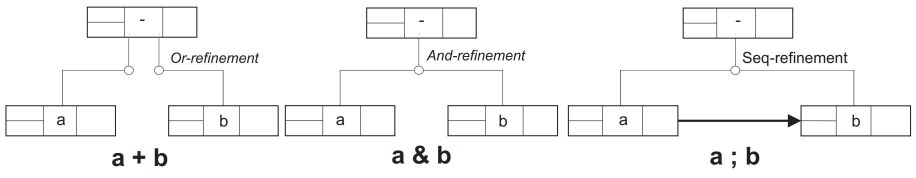

Figure 4. Compound actions in *C-O Diagrams*

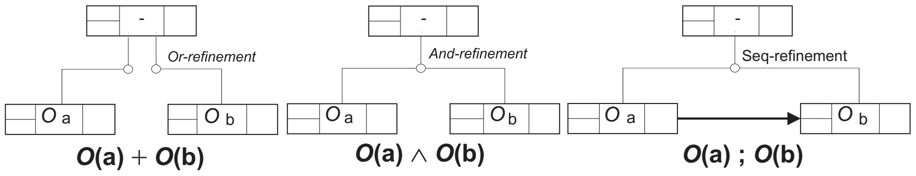

Figure 5. Composition of deontic norms in *C-O Diagrams*

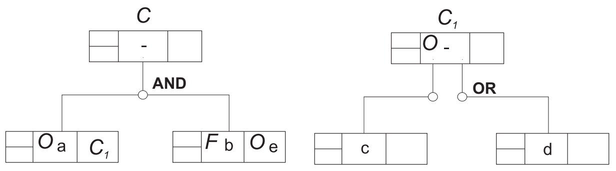

Figure 6. Reparations in *C-O Diagrams*

3. The deontic norms we take into account to check restrictions *1.* and *2.* can be shared by several branches, i.e., when we have a deontic norm applied over a compound action, this norm is part of several branches of our diagram.

The *repetition* of both, actions and deontic norms, can be achieved by means of the repetition structure we define in *C-O Diagrams*. The meaning of this structure is similar to the *Kleene's star* operator "$*$" [10] applied over the elements of the target clause of the arrow, but it is richer in the sense that the repetition can be conditioned to the satisfaction of the source clause of the arrow and not other alternative clause.

Reparation R: This field of a box can state a *new contract* that must be satisfied when the main field **P** is not satisfied (a *prohibition* is violated or an *obligation* is not fulfilled, there is not reparation for *permission*). This new contract can be just a new norm, but it can also be a new hierarchical tree of clauses, including their own reparations. In this way, we are able to specify nested reparations in our *C-O Diagrams*.

The field **R** is only allowed in the clauses of our diagrams where we specify a deontic norm of obligation or prohibition in field **P**, being forbidden in the other clauses. E.g., we can imagine a main contract C stating that we have the obligation of performing an atomic action a and the prohibition of performing an atomic action b. However, if we do not perform the obligatory action a, we can compensate it by fulfilling another contract called C_1, consisting of performing an action c or an action d, and if we perform the forbidden action b, we can compensate it just by performing an action e. This situation can be modeled in our diagrams as shown in Figure 6.

At this point we can see clearly the difference between having a composition of obligations over atomic actions and having an obligation over a compound action. While the former allows us to specify a different reparation for each one of the atomic actions we are obliged to do, the latter only allows us to specify one reparation for the compound action that is under the obligation operator. For the first case, we can consider the diagrams we have in Figure 5, where it is possible to specify a different reparation in each one of the leaf clauses of the diagrams. For the second case, we can imagine having the diagrams shown in Figure 7, where we can only specify reparations in the no leaf clauses where we have the obligations, affecting these reparations the whole composition of actions.

This difference is a bit trickier if we consider prohibitions or permissions, because it not only concerns to the specification of reparations [17]. For example, given two atomic actions a and b, the meaning of prohibiting the sequence of these two actions, written as $F(a.b)$, is different from the meaning of prohibiting action a, and next prohibiting action b, written as $F(a).F(b)$. In the first case, the sequence of actions starting with a and continuing with any action different from b is allowed, while in the second case any sequence of actions starting with a is forbidden. Similar distinctions exist when we consider permissions instead of prohibitions.

Guard g: This field of a box is a *boolean expression* that evaluates some information provided by the clause specification, telling us under which conditions the clause must be taken into account. E.g., in a car insurance contract we can have a clause that is only applied to people under the age of 21. In that case, we must include in the box modeling this clause a guard like **age** $<$ **21**.

Basically, a guard is a set of expressions that evaluate to a boolean (*true* or *false*) combined by means of conjunctions (*and*), disjunctions (*or*), and negations (*not*). These expressions can include constant values, variables, and equality and inequality operators ($==, !=, <, >, \ldots$).

When the guard condition corresponding to a subclause of

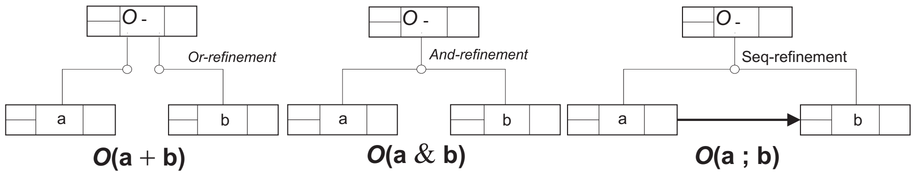

Figure 7. Obligations over compound actions

an **AND/SEQ refinement** evaluates to *false*, the subclause is trivially satisfied, so we only must check the subclauses with a *true* guard (or without guard). However, when the guard condition corresponding to a subclause of an **OR-refinement** evaluates to *false*, we cannot satisfy that subclause in order to satisfy the parent clause, so it is necessary to satisfy one of the other subclauses with a *true* guard (or without guard). E.g., in a payment system we can have the obligation of paying by cash or by credit card, but the second option is conditioned to be of legal age, so in case it is not satisfied we have only the possibility of paying by cash.

Time restriction tr: Each clause can have associated a time restriction, e.g., deadlines, timeouts, etc. These real-time aspects are expressed in the boxes of our diagram by means of *intervals* within the field **tr**. These intervals indicate the period of time in which the clauses must be satisfied.

The time restrictions can be specified in two different ways within the boxes: we can specify the *dates* binding the beginning and the end of the time frame corresponding to the clause (*absolute time*), or we can specify a deadline saying the number of *time units* that can elapse before the clause is satisfy from the moment at which another clause is satisfied or from the moment at which the contract comes into effect (*relative time*). For example, in the case of absolute time, we can model a contract stating that a clause must be satisfied in the **first five days of October** (interval [10/1 00:00, 10/5 23:59]), whereas in the case of relative time a contract can state that a clause must be satisfied not later than **five days after satisfying another clause C** (interval [C, 5days]).

When we have a time restriction specified in a clause that is refined into subclauses, this restriction affects all the subclauses, i.e., all the subclauses necessary to satisfy the parent clause must be satisfied in the time frame specified in this parent clause. Otherwise, the parent clause is considered unfulfilled.

III. Case Study: A Software Provision System

In this case study we present a contract regulated composition of services between a client and a software provider. The case study is inspired by the one published in [13], but with some modifications in the composition and the contract specification, including the definition of real-time constraints. The parties involved in this contract are the *client*, the *software provider*, and the *testing agency*.

The scenario we are considering in this case is the following: everything starts when the *client* asks the *software provider* to develop a new software. The *software provider* develops the different components needed to implement this new software. The *software provider* informs twice the *client* about the progress of the software development. If the *client* wants any changes in the software, he can request them after the first update. Any changes suggested after the second update are considered a violation. The *client* can recover from this violation by paying a penalty to the *software provider* or by withdrawing the suggested changes. Every update is followed by a payment from the *client* to the *software provider*. If the *software provider* does not send the updates to the *client* at the schedule time (**three months** for the first update and **nine months** for the second update), it is also considered a violation of the contract that can be repaired by paying a penalty charge to the *client*.

Once all the software components are implemented, the *software provider* integrates these components and sends the final product to the *testing agency* for testing. Then this *testing agency* sends testing reports to the other parties. If the tests fail, the components are revised by the *software provider* and then tested again. Finally, if the tests succeed, the *software provider* delivers the final product to the *client*.

In Table I we show a list of the obligations, permissions and prohibitions we can deduce from the scenario described above. We can see here that there are two clauses specifying real-time constraints, **Clause 1** and **Clause 4**. If we consider the moment when the *client* asks the *software provider* to develop a new software as the moment when the contract comes into effect (denoted as *0*), the time *software provider* has to update *client* for the first time after being asked to develop the new software is three months (denoted as *3M*) according to **Clause 1**. Moreover, from the above scenario we can deduce that there is a second time constraint, that the time *software provider* has to update *client* for the second time after updating him for the first time is nine months (denoted as *9M*) according to **Clause 4**. We can also see in Table I that **Clause 9** and **Clause 10** are conditioned to the result of the tests, so we consider a boolean variable we call *Tests_Ok* specifying if the tests have succeeded.

The main problem we have with this textual specification of the contract is that it is no clear the relationship existing between the different clauses, which makes difficult any kind of analysis of the contract. Therefore, we aim at a specification language that clearly defines the relationship between the different clauses, allowing the analysis of the contract, but not so formal that an expert is needed. *C-O Diagrams* satisfies the above requirements. In what follows

Clause	Agent	Modality	Action	Reparation
1	*Software provider*	*Obligation*	Updates *client* before **three months** (first update).	Pays penalty, eventually updating *client* (*Obligation*).
2	*Client*	*Obligation*	Sends the first payment to *software provider*.	∅
3	*Client*	*Permission*	Requests changes to *software provider* after first update.	∅
4	*Software provider*	*Obligation*	Updates *client* before **nine months** after first update (second update).	Pays penalty, eventually updating *client* (*Obligation*).
5	*Client*	*Obligation*	Sends the second payment to *software provider*.	∅
6	*Client*	*Prohibition*	Requests changes to *software provider* after second update.	Pays penalty or withdraws changes (*Obligation*).
7	*Software provider*	*Obligation*	Sends the integrated components to *testing agency*.	∅
8	*Testing agency*	*Obligation*	Sends testing reports to *client* and *software provider*.	∅
9	*Software provider*	*Obligation*	If tests fail, revises components, repeating after that the testing process.	∅
10	*Software provider*	*Obligation*	If tests succeed, delivers the final product to *client*.	∅

Table I
NORMS OF THE *Software Provision System* CONTRACT

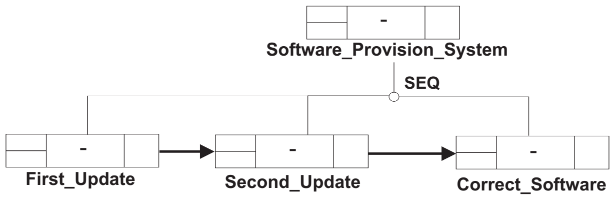

Figure 8. Top-level of the *C-O Diagram* for the *Software Provision System*

we explain how to model the contract with these diagrams, taking into account the information provided in Table I. We use a_n to denote the action performed by clause number n and r_m to denote the reparation defined for clause number m.

In Figure 8 we show the top-level of the *C-O Diagram* we specify for the contract, where we have grouped the clauses in Table I into three more general clauses with a sequence relationship between them: *First_Update* (**Clause 1**, **Clause 2** and **Clause 3**), *Second_Update* (**Clause 4**, **Clause 5** and **Clause 6**), and *Correct_Software* (**Clause 7**, **Clause 8**, **Clause 9** and **Clause 10**). These three clauses cover the three different phases we can distinguish in the contract.

The decomposition of clause *First_Update* into subclauses can be seen in Figure 9. In this case, we first have the specification of the obligation contained in **Clause 1**, including the deadline (**three months** from the beginning) and the

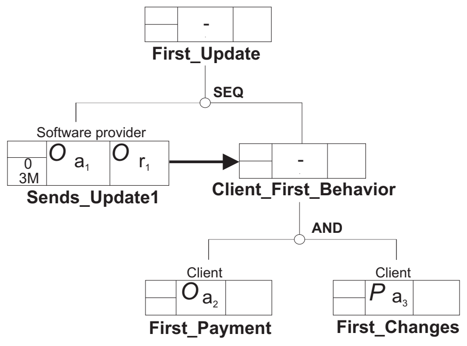

Figure 9. Decomposition of clause *First_Update*

reparation, and after that we have the specification of **Clause 2** and **Clause 3**, both affecting the behavior of the *client* and with an $\wedge$ relation between them, composing the parent clause *Client_First_Behavior*.

The decomposition of clause *Second_Update* into subclauses can be seen in Figure 10 and it is very similar to the previous one. We first have the specification of the obligation contained in **Clause 4**, including the deadline (**nine months** from the fulfillment of **Clause 1**, abridged as *C1*) and the reparation, and after that we have the specification of **Clause 5** and **Clause 6**, both affecting the behavior of the *client* and with an $\wedge$ relation between them, composing the parent clause *Client_Second_Behavior*. The main difference is that the *client* is not allowed to request any change, so instead

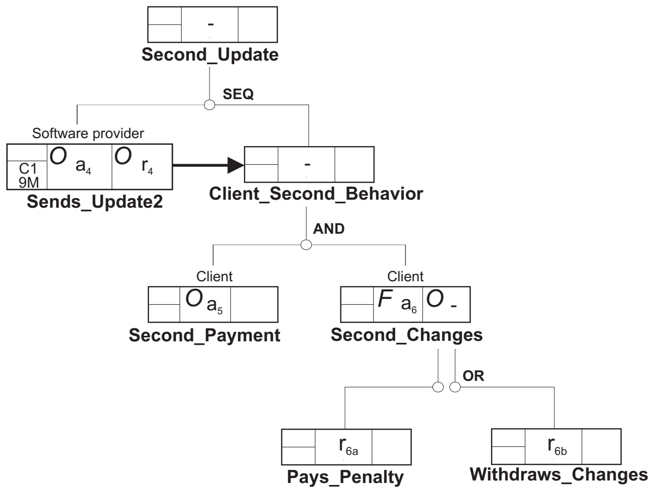

Figure 10. Decomposition of clause *Second_Update*

of a permission we have now a prohibition in **Clause 6**. This prohibition specifies two possible reparations, so we use r_{6a} to represent the payment of a penalty and r_{6b} to represent the withdrawal of the changes by the *client*. We write this complex reparation in the same diagram instead of referencing another diagram just to save space.

Finally, the clause *Correct_Software* is decomposed as shown in Figure 11. We first have the obligation contained in **Clause 7**, about sending the software to the *testing agency*. After that we have the obligation contained in **Clause 8**, about sending the reports to the *software provider* and to the *client*. These two actions can be performed concurrently, so we call a_{8a} the action of sending the testing reports to the *software provider* and a_{8b} the action of sending these reports to the *client*. Last, we have the specification of **Clause 9** and **Clause 10**, both affecting the *software provider* and with a $+$ relation between them, composing the parent clause *Tests_Result*. We notice that the selection of one subclause or the other is related to the result of the tests, so we use the variable *Tests_Ok* to model that situation. Each time we apply **Clause 9** (*software provider* revises the components), we repeat the sequence of software testisng, going back to the application of **Clause 7**. If **Clause 10** is applied, the *software provider* delivers the final product to the *client* and the contract finishes.

IV. Evaluation of the Model

In this section we present an evaluation of the visual model described above, divided into a qualitative and a quantitative evaluation. For the former evaluation we discuss how the model fits some of the most important principles for designing effective visual notations defined in [16]. First, the principle of *semiotic clarity* is accomplished by the model, as there is only one graphical structure corresponding to each semantic concept and vice versa. Second, the principle of *perceptual discriminability* is taken into account to differentiate between refinements, having each kind of refinement a clearly distinct shape and using the text with the name of the refinement to complement the graphics (principle of *dual coding*). We also have that the number of different graphical symbols in the model is under the upper limit of six categories for graphics complexity, so the principle of *graphic economy* is accomplished. Finally, the principle of *complexity management* is covered by the modularization of the diagrams, as we have done in the case study. As we have seen in the previous section, modules are combined by having the same box appearing in several diagrams (principle of *cognitive integration*).

	Error Rate (%)		Time (sec)	
	Textual	Visual	Textual	Visual
Basic	1.8%	1.8%	26	26
Composition	22.0 %	21.3%	53	45
Temporal	12.1%	9.2%	42	37
Reparation	54.5%	33.6%	98	83

Table II
Results obtained for the visual and textual notations

The quantitative evaluation of the model is done by means of user-based tests. Our purpose with these tests is to compare *C-O Diagrams* with textual notations for e-contracts, so in the tests we use a textual version of the diagrams very similar to $\mathcal{CL}$ language [18], as it is closely related to the approach followed by *C-O Diagrams*, but adding agents and time aspects that are not currently supported by $\mathcal{CL}$. We have designed two tests to compare the understandability of both, textual and visual representation, where we have that the same semantic concept is represented by a *C-O Diagram* in one test and by the textual language in the other test, asking the same questions in both cases. These tests have been done by 20 students of our university who have previously attended two lectures about e-contracts and *C-O Diagrams*, and we have obtained the results shown in Table II. In the table we show the average error rate and time taken by students in both cases, for textual representation and for visual representation. The results are divided into four rows depending on the kind of contracts that the questions correspond to: contracts with only a basic deontic norm, contracts with a composition of deontic norms, contracts with temporal restrictions, and contracts including reparations. As we can see, in the simplest case there is no difference between textual and visual representation, but in all the other cases we obtain better results for visual representation than for textual representation.

We also have defined another test to rank the subjective opinion of the users based on the *System Usability Scale (SUS)* [3]. It is a simple, ten-item scale giving a global view of subjective assessments of usability. In our case, we use this text not only to asses the usability of *C-O Diagrams*, but also to compare these diagrams with the textual notation. SUS scores have a range of 0 to 100, where 0 corresponds to the worst evaluation of usability and 100 corresponds to the best one. We have obtained an average score of **77.83** in

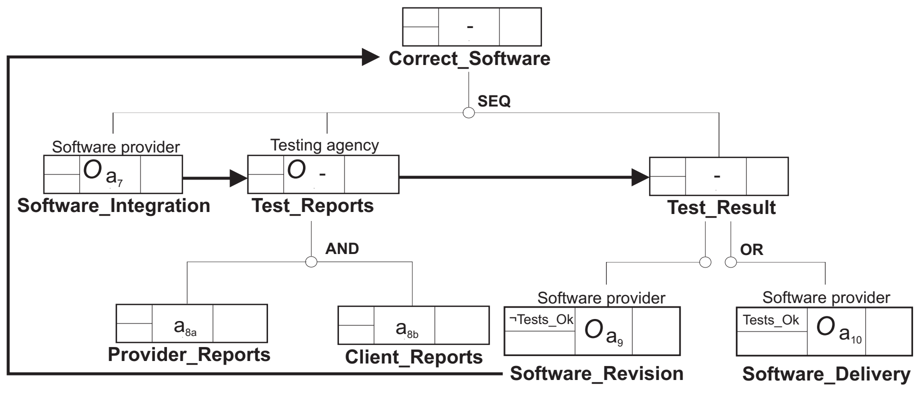

Figure 11. Decomposition of clause *Correct_Software*

this scale for our model. This is a score that clearly shows the user's preference to use the visual model instead of the textual notation.

All these tests can be accessed via the Moodle course *"C-O Diagrams"* in http://moodle.retics.uclm.info/. Anyone can access this course login as a guest. The tests are available at the *Social Activities* box, through the preview option.

V. Related Work

To the best of our knowledge, there is not any other visual model specially created for the definition of e-contracts. However, several works in the literature define a meta-model for the specification of e-contracts whose purpose is the enactment or the enforcement of this e-contract. For instance, in [6] Chiu et al. present a meta-model for e-contract templates written in UML, where a template consists of a set of contract clauses of three different types: obligations, permissions and prohibitions. These clauses are later mapped into event-condition-action (ECA) rules for contract enforcement purposes, but the templates do not include any kind of reparation or recovery associated to the clauses, and the way of specifying the different possible relationships between clauses is not clear. In [11] Krishna et al. propose another meta-model of e-contracts based on entity-relationship diagrams that they use to generate workflows supporting e-contract enactment. This meta-model includes clauses, activities, parties and the possibility of specifying exceptional behavior, but this approach is not based on the deontic notions of obligation, permission and prohibition, and says nothing about including real-time aspects natively. Another approach can be found in [20], where Rouached et al. propose a contract layered model for modeling and monitoring e-contracts. This model consists of a business entities layer, a business actions layer, and a business rules layer. These three layers specify the parties, the actions and the clauses of the contract respectively, including the conditions under which these clauses are executed. However, real-time restrictions are not included and the specification of the clauses follows an operational approach, not a deontic approach.

The approach followed in *C-O Diagrams* for the specification of e-contracts is close related to the formal language $\mathcal{CL}$ [18]. In this language a contract is also expressed as a composition of obligations, permissions and prohibitions over actions, and the way of specifying reparations is the same that in our visual model. The main difference with *C-O Diagrams* is that $\mathcal{CL}$ does not support the specification of agents nor timing constraints natively, so they have to be encoded in the definition of the actions. Also, in $\mathcal{CL}$ there is no sequence operator to combine the different clauses, so the notion of sequence has to be expressed always by means of specifying the application of a clause after performing a certain action (denoted as $[\alpha]\mathcal{C}$, where α is a compound action and $\mathcal{C}$ is a general contract clause), like in propositional dynamic logic. Refer to [15] for a general description of deontic logic.

In [14] Marjanovic and Milosevic also defend a deontic approach for formal modeling of e-contracts, paying special attention to the modeling of time aspects. They distinguish between three different kinds of time in e-contracts: absolute time, relative time and repetitive time. The two first kinds are supported by *C-O Diagrams*, but repetitive time is not included yet in our model. Nevertheless, with the combination of the other two kinds of time and the repetition structure, we can achieve some repetitive time behaviors in our model. In [13] Lomuscio et al. present an approach to verify contract-regulated service compositions. They use the orchestration language WS-BPEL to specify all the possible behaviors of each service and the contractually correct behaviors. After that, they translate these specifications into timed automata supported by the MCMAS model checker to verify the behaviors automatically. In this work we have that the scope of the e-contracts is limited to web services compositions, specifying the e-contract corresponding to each one of the services separately. The specification of real-time constraints is not allowed because they are not supported by MCMAS and the deontic norms are restricted to only obligations.

VI. Conclusions and Future Work

In this paper we have presented *C-O Diagrams*, a new visual formalism for electronic contracts. Though we have shown the applicability of such diagrams on a case study taken from SOA, their usefulness go beyond services. They can be used for negotiation via email, contracting on the Internet, performing an Electronic Data Interchange Agreement (EDI), just to mention a few. Indeed, *C-O Diagrams* may also be used as an intermediate (formal) language for other applications. We are currently working on their use in the context of requirements engineering (to formalize requirements), product software families (as an extension of feature diagrams), and as a visual representation of $\mathcal{CL}$ contracts.

We are also working on a formal semantics based on automata, making it possible to formally analyze *C-O Diagrams* using for instance UPPAAL [12]. Besides the application of *C-O Diagrams* to the mentioned domains, we intend to explore how we can automatically obtain diagrams from a controlled (structured) English, by using for instance the Grammatical Framework (GF) [19]. We also envisage the possibility of specifying a different diagram for each one of the parties involved in an e-contract instead of having a global *C-O Diagram* with multiple agents. This compositional approach can be useful if we define a composition operator, specifying when two of these new *C-O Diagrams* can be composed and the result of the composition.

Acknowledgment

Partially supported by the Spanish government (cofinanced by FEDER founds) with the project TIN2009-14312-C02-02, the JCCLM regional project PEII09-0232-7745, and the Nordunet3 project "COSoDIS". The first author is supported by the European Social Fund and the JCCLM.

References

[1] A. Boulmakoul and M. Sall. Integrated Contract Management. Technical Report HPL-2002-183, Hewlett-Packard, 2002.

[2] M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. *Proceedings of the Sixth International Symposium on Software Composition*, pages 34–50, 2007.

[3] J. Brooke. SUS - A "quick and dirty" usability scale. *Usability Evaluation in Industry*, pages 189–194, 1996.

[4] M. G. Buscemi and U. Montanari. Cc-Pi: A Constraint-Based Language for Contracts with Service Level Agreements. *Proceedings of Second International Workshop on Formal Languages and Analysis of Contract-Oriented Software*, pages 1–8, 2008.

[5] S. Carpineti, G. Castagna, C. Laneve, and L. Padovani. A formal account of contracts for Web services. *Proceedings of Third International Workshop on Web Services and Formal Methods*, pages 148–162, 2006.

[6] D. Chiu, S. Cheung, and S. Till. A Three-Layer Architecture for E-Contract Enforcement in an E-Service Environment. *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS-36)*, pages 74–83, 2003.

[7] H. Davulcu, M. Kifer, and I. V. Ramakrishnan. CTR–S: A Logic for Specifying Contracts in Semantic Web Services. *Proceedings of the Thirteenth international World Wide Web conference*, pages 144–153, 2004.

[8] E. M. Haber, Y. E. Ioannidis, and M. Livny. Foundations of Visual Metaphors for Schema Display. *Journal of Intelligent Information Systems*, 3(3–4):263–298, 1994.

[9] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[10] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, pages 3–41, 1956.

[11] P.R. Krishna, K. Karlapalem, and A.R. Dani. From Contract to E-Contracts: Modeling and Enactment. *Information Technology and Management*, 6(4):363–387, 2005.

[12] K. G. Larsen, Z. Pettersson, and Y. Wang. UPPAAL in a Nutshell. *STTT: International Journal on Software Tools for Technlogy Transfer*, 1(1–2):134–152, 1997.

[13] A. Lomuscio, H. Qu, and M. Solanki. Towards verifying contract regulated service composition. *Proceedings of IEEE International Conference on Web Services (ICWS 2008)*, pages 254–261, 2008.

[14] O. Marjanovic and Z. Milosevic. Towards formal modeling of e-Contracts. *Proceedings of 5th IEEE International Enterprise Distributed Object Computing Conference*, pages 59–68, 2001.

[15] P. McNamara. Deontic Logic. In *Gabbay, D.M., Woods, J., eds.: Handbook of the History of Logic*, volume 7, pages 197–289. North-Holland Publishing, 2006.

[16] D. L. Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009.

[17] G.J. Pace and G. Schneider. Challenges in the specification of full contracts. *Proceedings of 7th International Conference on integrated Formal Methods*, pages 292–306, 2009.

[18] C. Prisacariu and G. Schneider. A formal language for electronic contracts. *Proceedings of 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 174–189, 2007.

[19] A. Ranta. Gramatical Framework. *Journal of Functional Programming*, 14(2):145–189, 2004.

[20] M. Rouached, O. Perrin, and C. Godart. A Contract Layered Architecture for Regulating Cross-Organisational Business Processes. *Proceedings of Third International Conference on Business Process Management*, pages 410–415, 2005.

[21] A. van Lamsweerde, A. Dardenne, and S. Fickas. Goal-directed requirements acquisition. *Selected Papers of the Sixth International Workshop on Software Specification and Design. Science of Computer Programming*, 20(1–2):3–50, 1993.

[22] G. H. V. Wright. Deontic logic: A personal overview. *Ratio Juris*, 12(1):26–38, 1999.