Writing Use Cases Modelled with Situation Theory

Francisco A. C. Pinheiro[1], Isabel Díaz López[2,3]

[1]Universidade de Brasília
Departamento de Ciência da Computação
facp@cic.unb.br
[2]Universidad Central de Venezuela
[3]Universidad Politécnica de Valencia
idiaz@dsic.upv.es

Abstract. This article extends a framework intended to simplify the generation of use cases. The framework allows a concrete and systematic way of describing actions by means of a use case pattern and simple natural language sentences. The framework is composed of a pattern for specifying a use case and a set of guidelines for writing the sentences describing it. These guidelines help the specifier to write natural language sentences that may be easy and uniformly understood. The extension presented in this article uses situation theory to formalise the use cases as situations and to translate the simple sentences into situation theoretic concepts. This article refers to **PROSIT**, a computational implementation of situation theory, to show how use cases can be described in terms of situation theory.

1 Introduction

The use of natural language is one of the most common ways to express the contents of a use case in the initial phases of the development process. The use case specification is a document in which one describes some functionality of a system, a software component or another modelling element [10]. Normally, the functionality described in a use case is written as a sequence of sentences, in which each sentence represents the interaction between the system and its environment. The entities in the environment are modelled by actors and the system maybe the whole system or part of it, including subsystems and even individual classes and interfaces.

The use of natural language makes it easy for any person to understand what is described by a use case. Assuming an acquaintance with the domain being described, it is not necessary training to understand what is expressed by a use case. The designers can write the requirements and communicate with the domain specialists, users, and other developers using the terminology known by the majority. On the other hand, the benefits of using natural language can soon become negative points, when it causes ambiguities, inconsistencies, and redundancies in the description of the requirements [2, 4].

The clarity of a text is affected by the different styles of writing and the terminological diversity. Two factors decisively contribute this situation: the style and structure of the text (that we generally refer to as writing) and the terminology. The writing is determinant for the understanding of a use case and the terminology is related to the identification of the elements used to describe the functionality of a system.

Writing and terminology depend mainly on people in charge of the specification, and their different skills and styles of writing may hinder a uniform understanding of what has been written. A confuse, incorrect, complicate, or simply strange way of writing may generate doubtful interpretations of requirements. The diversity of vocabulary may also be the source of inconsistencies in the design of a system, with one word used to name different elements of the design, or different names used to name a single element.

To reduce the difficulties related to the writing of use cases Díaz and Matteo [6] proposed a framework that can be used to systematically write use cases. The framework is composed of a pattern for specifying a use case and a set of guidelines for writing the sentences describing it. These guidelines help the specifier to write natural language sentences that may be easy and uniformly understood.

In this article we investigate the use of situation theory to formalise part of the framework proposed by Díaz and Matteo [6]. There are two reasons guiding us. First, the natural language sentences proposed by them are of a very simple form and amenable to formalisation using situation theory concepts. Second, their proposed pattern may be described by situations in the way discussed by Pinheiro [9].

In Section 2 we present the main concepts of situation theory and **PROSIT**, the software that implements it. In Section 3 we present the framework proposed by Díaz and Matteo and show how it can be described using **PROSIT**. In Section 4 we discuss some uses of situation theory to deal with use cases descriptions. Section 6 contains our conclusions.

2 Situation Theory

Situation theory is a mathematical theory of meaning introduced by Barwise and Perry [1]. The description here is an informal account of the theory based mainly on [5] and [1].

In situation theory *infons* are the basic unity of information. The simple (or basic) infons convey information about *individuals* related in some way. The information conveyed by the statement "The client inserts the card into the ATM" is an infon about the individuals client, ATM, and the card related in a way that makes true the information that the client inserts the card into the ATM. Observe that the same infon appears when someone, pointing to the client, says: "Look, he inserts the card into the ATM". Infons are also called *state of affairs* or *possible facts*, suggesting that an infon may or may not be supported in the world, i.e., they may or may not hold in some real situation.

Individuals and *relations* are primitive concepts of Situation Theory. Individuals may be anything we may refer to as being related to other individuals or to other objects of the theory. The term *object* or *situation theoretic object* is used to encompass the individuals and other (more complex) objects of the theory.

The structure of an infon consists of a relation, the related individuals, and the roles each one plays in the relationship. The individuals of an infon are called *arguments* of the infon. Relations may be seen as parametric objects that need an assignment associating individuals to specific roles. One notational convention is to represent an infon placing its relations and the corresponding assignments between double angle brackets.

<< insert_into, agent → client, object → card, place → atm; 1 >>

Sometimes, for convenience purposes infons are represented with no explicit reference to the assignments. This is to make them more readable as we assume that the roles played by the individuals are clear. The above infon could also be described as:

Insert_into(client, card, atm) : yes.

The number 1 in the first infon notation and the word "yes" in the second indicate that this is a positive infon, carrying positive information. There are also negative infons like "The client doesn't kick the ATM". These are indicated by the number 0 or the word "no", as in:

kick(client, atm) : no.

Relations have appropriateness conditions specifying the type of individuals that can play each role. It would make no sense to say that the colour blue admires honesty. Admiration is not something we expect from the abstract concept of colour. So, the relation admire would have appropriateness conditions ruling out that kind of assignment. In fact only relations with appropriate assignments are considered infons.

2.1 Situations

Intuitively, *situations* are parts of the world perceived by a cognitive agent. Infons are said to hold in a situation if the information conveyed by the infon is perceived as part of that situation. We also say that a situation *supports* an infon to mean the same thing. In symbols we indicate that a situation **s** supports an infon δ by $s \models \delta$.

We describe a situation using every infon we want to be held in that situation. The situation **s** below is described by three infons saying that a user types his password and, after it has been validated by the system, he types his account number. The notation used should be evident.

S at t_0 : type_in(user, password): yes
at t_1 : validate(system, password): yes
at t_2 : type in(user, account number): yes
$t_0 \pi t_1 \pi t_2$

Every situation of the world where the information conveyed by the three infons can be hold as a fact is said to support the conjunction of these infons.

2.2 PROSIT

The **PROSIT** language [8, 11] is one implementation of situation theory. We use **PROSIT** to build our representations of use cases. We are aware of two other systems implementing situation theory: **ASTL** [3] and **BABY-SIT** [12]. Our choice for **PROSIT** is based on reports of its adequacy to general problems of knowledge representation [7].

PROSIT is implemented in **LISP** as a logic programming language and, therefore, shares many features with other languages of this kind: statements are either queries or assertions, goals are solved using unification, backtracking, and depth-first search.

In **PROSIT** an infon inf is asserted in a situation **S** using the assertion predicate ! as in (! **S** inf). The action that a user has inserted his card in the ATM may be represented by the relation insert_into and the individuals client, card, and atm: (insert user card atm). To represent that action occurring in a situation named withdraw we write:

```
(! withdraw (insert-into client card atm))
```

Every assertion occurs in a situation. In **PROSIT** there is one universal situation called top that holds all infons. To create a new situation **S** we use the command (in **S**). This also has the effect of making **S** the current situation. The result is that every subsequent command, unless explicitly stated, refers to that current situation.

```
(in withdraw)
(! (insert-into client card atm))
```

We can have several infons asserted with a single assertion command.

```
(in withdraw)
(! (insert user card atm)
   (check atm card))
```

We assert negative information using the predicate no. The following assertions tell us that in situation withdraw_ free the ATM does not check the card the user has inserted into it.

```
(in withdraw_free)
(! (insert user card atm)
   (no (check atm card)))
```

All examples in the next sections were run on **PROSIT** and the answers are those given by the system. To reproduce the examples one should start a **LISP** environment and load the **PROSIT** program[1].

3 Writing Uniform Use Cases

In [6] Díaz and Matteo established, from a grammatical perspective, some restrictions on writing use cases. These restrictions allow designers to overcome the difficulties with the unconstrained use of natural language in the specification of use cases.

[1] The software is available from www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/impl/other/ PROSIT/0.html

Particularly, they treat the terminological control necessary to avoid ambiguity, incoherence, and redundancy derived from the indiscriminate use of terms (words) when describing use cases. Their framework consists of the use of simple sentences and a pattern to describe use cases. Their work is based on a communication model consisting of the following elements:

– the speaker (or sender),
– the hearer (or receiver),
– the channel, and
– the message.

Applied to use cases the speaker can be one of the actors in a use case or an entity of which some functionality is being modelled. The hearer is the actor or entity receiving the message, which represents the information conveyed from the speaker to the hearer. For modelling purposes the channel is considered to be irrelevant.

Communications in this context happen through the use of natural language sentences. We relate the hearer and speaker to the subject and the object of the sentence expressing the interaction. The message is related to the predicate of the sentence.

3.1 Simple Sentences and Use Case Pattern

We consider that all sentences in a use case description can be written as a simple sentence, expressing one action performed by a single subject. It is our experience that this accounts for the majority of the cases. We only take simple sentences composed of a subject and a predicate, according to the structure below.

[pre-condition] <actor> | <entity> <activity> [restriction]

The grammar allowing the construction of such sentences defines <actor> and <entity> to be a single nouns, optionally preceded by an article. The term <activity> is defined to be the predicate of the sentence consisting of a verb and its objects. The restriction is an expression limiting the activity and the pre-condition expresses what should be true before the activity happen. These elements are illustrated in the following sentence:

$$\underbrace{\text{After the delivery of the money,}}_{\textbf{Pre-condition}}\ \underbrace{\text{the ATM}}_{\textbf{Entity}}\ \underbrace{\text{returns the card to the client}}_{\textbf{Activity}}\ \underbrace{\text{in a time no longer than 10 seconds}}_{\textbf{Restriction}}$$

Our use cases are specified according to the pattern illustrated in Figure 1. The events describing the basic and alternative flow of actions are expressed as simple sentences.

USE CASE	<use case identification>
ABSTRACT	<short description>
ACTOR	<actors involved in the use cases>
PRE-CONDITION	<condition that should be true for the use case to happen>
DESCRIPTION	*Normal flow the events* 1) [precondition]<actor>|<entity><activity><restriction> 2) [precondition]<actor>|<entity><activity><restriction> ⋮ n) [precondition]<actor>|<entity><activity><restriction> *Alternative flow the events* <event> [precondition]<actor>|<entity><activity><restriction> ⋮
POST-CONDITION	<condition that should be true after the actions in the use case has been performed>

Fig. 1. Basic Form of a Use Case

3.2 Using Situation Theory

The use of situation theory to describe use cases according to the framework just described is straightforward. The use case pattern is similar to the one discussed in [9] and the simple sentences are amenable to translation to situation theory concepts.

In terms of situation theory the actions in the flow of events are infons. Each simple sentence express an action performed by the subject, possibly acting on some objects. The actions are translated to relations and their subject and objects to arguments. For example the sentence "The client inserts his card into the ATM" may be translated to the infon

```
(insert client card atm)
```

The set of infons, i.e. all actions in the flow of events, describe a situation — the situation depicted by the use case. The pre and post conditions are also situations describing facts that should be true before and after the flow of events of the use case.

The use of situation theory to describe use cases is illustrated in Figure 2 that shows a use case describing a withdraw operation in a automatic teller machine. In this use case we only show the normal flow of events. For each action we show their expression as a simple natural language sentence and as a infon. Observe that the infons in the right side of the normal flow of events are expressed as commands of **PROSIT**. In fact all infons were written to a file preceded by the command (in withdraw) and loaded into a **PROSIT** session. The next examples shows the real answers we get from **PROSIT** regarding this situation.

4. Exploring Situation Theory Formalism: Making Queries

The framework proposed by Díaz and Matteo already facilitates the systematic construction of use cases that can be understood in a uniform way by all interested parties. This is achieved mainly by the use of simple sentences to express the actions in a use case.

As the example in Figure 2 shows the translation from simple sentences to infons retains some of the sentences' expressiveness. Also, using situation theory to describe use cases makes possible to explore them in richer ways. We can for example make queries and investigate inconsistencies and goal accomplishment.

USE CASE	Withdraw	
ABSTRACT	If Client is a member of the network, the ATM allows him to withdraw money from any of his accounts.	
ACTOR	Client	
PRE-CONDITION	ATM ready for operation	
DESCRIPTION	*Normal flow the events*	
	1) The Client inserts his card into the ATM.	(! (insert client card atm))
	2) The ATM checks card validity.	(! (validate atm card))
	3) The ATM displays a prompt for password.	(! (prompt atm password))
	4) The Client types his password.	(! (typein client password))
	5) The ATM checks password validity.	(! (validate atm password))
	6) The ATM displays a list of options.	(! (display atm options))
	7) The ATM displays a prompt for option.	(! (prompt atm option))
	8) The Client chooses the withdraw option.	(! (choose client withdrawOption))
	9) The ATM displays a list of accounts.	(! (display atm listAccount))
	10) The ATM displays a prompt for account.	(! (prompt atm account))
	11) The Client chooses an account from the list.	(! (choose client account))
	12) The ATM displays a prompt for amount.	(! (prompt atm amount))
	13) The Client enters the amount.	(! (typein client amount))
	14) The ATM checks if the account balance is sufficient for debiting the amount.	(! (check atm balance amount))
	15) The ATM delivers the money to the client.	(! (deliver atm money client))
	16) The ATM registers the transaction.	(! (register atm transaction))
	17) The ATM updates the account balance.	(! (update atm balance))
	18) The ATM prints a receipt.	(! (print atm receipt))
	19) The ATM ejects the card.	(! (eject atm card)
POST-CONDITION	The Client gets his money and his card	

Fig. 2. A Withdraw Scenario expressed using Natural Languages and **PROSIT** Commands

Once an infon is asserted in a situation s, the situation supports that infon. In **PROSIT** we can make queries just typing the infon. If the infon is supported by the current situation the answer is yes. With respect to the situation of Figure 2, we have

(insert client card atm)

resulting in the answer yes because the infon (insert client card atm) is explicitly stated in situation withdraw. For the query

(check atm card)

the answer is unknown because the infon we have in withdraw is (validate atm card) and not (check atm card). In situation theory there is no closed-world assumption. An answer is no with respect to a query inf only if the negative information about inf is supported by the situation.

Queries can be made using variables. In **PROSIT** variables are indicated by a preceding asterisk. For the following query:

(insert user *thing atm)

the answer is yes, if *thing = card because there is an infon (insert user card atm) supported by withdraw. Variables can be used over relations and several variables can be used in a single query. For example, the query

```
(insert *who *thing atm)
```

is answered by yes, if *who = user, *thing = card.

The general query (! = *x) will be matched by every infon supported by the current situation. This includes all the infons asserted in the situation and also those that can be inferred from it.

It is possible to make larger and more complex queries using the connectives and, or, and not.

There is a possibility to say that an infon, and more generally, a situation implies another one. So, we could have an assertion saying that the validation of something implies the checking of this thing.

5. Dealing with Inconsistencies

Maybe one of the most interesting characteristics of situation theory is the ability to deal with inconsistency. A situation is incoherent is it supports inf and (no inf). When analysing scenarios this may be a valuable feature. Suppose that two scenarios about the same situation have been elicited, maybe by two different analysts interacting with different users. In one scenario there is an assertion that the ATM prompts the user to type in his password

```
(prompt atm typein pw user)
```

and the other scenario explicitly says the ATM should not prompt the user to type in his password.

```
(no (prompt atm typein pw user))
```

When these two situations are loaded, the two assertions become supported by the same situation. It is possible to investigate the existence of inconsistencies with a explicit query like

```
(and *x (no *x))
```

The answer will be yes, if *x = (prompt atm typein pw user) and yes, if *x = (no (prompt atm typein_pw_user)).

It is also possible to catch inconsistencies with simple queries. When the predicate (duals) is on, **PROSIT** tries to prove the duals of every assertion. In duals mode if a situation supports both the infon i and its dual (no i), the answer to a query (i) is both yes and no. For example,

```
(duals)
(prompt atm typein pw user)
```

results in the answer yes and no.

5.1 Constraints

A constraint is a special kind of infon which uses variable to express rules. The general form of a constraint is (<= head gol1 ... goaln). A constraint says that its head is true in a situation if and only if all goals are true in that situation.

When the constraint below is asserted in a situation sit, the the infon (end-ok) is supported in sit if the situation supports the infons (get-money *x) and (get-card *x)

```
(<= (end-ok) (get-money *x) (get-card *x))
```

There may be several interconnected constraints in a situation.

```
(! (<= (get-money user) (deliver atm money client)))
(! (<= (get-card *x) (eject atm card)))
(! (<= (end-ok) (get-money *x) (get-card *x)))
```

These constraints, if asserted in the situation withdraw, cause the infon (end-ok) to be true since the infon (get-card *x) is supported (by a direct application of the second constraint because the infon (eject atm card) is explicitly stated) and the infon (get-money *x) is also supported (making *x = user and applying the first constraint). The system answers yes to the query (end-ok).

Constraints can be used to verify post-conditions and the state of affairs after the actions in the scenario had happen. They can also be used to check the goals of a situation.

In **PROSIT**, constraints of this type cause a backward-chaining computation in which the system tries to satisfy each goal in turn and backtracks to attempt another path if the satisfaction fails. There is another type of constraint that uses forward-chaining computation. It general form is (=> (head tail1... tail2)). This constraint says that whenever the head is asserted in a situation all of the tails are automatically asserted as well. This is a very useful feature to describe the environment, and its expected laws, of a scenario.

5.2 Subtype and Implication

PROSIT allows the declaration of a subtype relationship between situations in the form of an assertion (← sit1 sit2). This causes sit1 to be a subtype of sit2. The situation sit1 now supports every infon and every constraint supported by sit2**.** In fact the predicate symbol ← was chosen to be an arrow pointing in the direction of the flow of information: what is asserted and supported in sit2 flows to sit1. This is very useful to construct scenarios composed of parts or to allow the composition of different scenarios.

The subtype relationship can be queried. A query like (← sit1 sit2) will be answered yes if and only if the situation sit1 is a subtype of situation sit2. This is a very powerful mechanism to verify if everything that is supported by a situation is also supported by another one. This feature can be used to see if the execution of a scenario indeed satisfies its goals.

Note that if in the supersituation sit1 we make assertions about sit1, like (! (!= sit1 inf)) saying that inf is supported by sit1, this becomes only an assertion of something in sit1 as seen by sit2. It has no binding effect on the real situation sit1, that is there is no guarantee that inf holds in sit1 just because sit2 *thinks* it holds. This is very useful to model assumptions about situations that may or may not be true.

If we want to assure that every assertion made in a situation sit2 about a situation sit1 is actually true in sit1, then we need a stronger kind of the subtype relationship. In

PROSIT this is called *subchunk* and is expressed in the form (c< sit1 sit2) meaning that sit1 is a subchunk of sit2.

The assertion of (c< sit1 sit2) causes sit1 to be totally described by the infons in sit2. If (!= sit1 inf**)** holds in sit2, then inf holds in sit1. Conversely, if inf holds in sit1, then (!= sit1 inf) holds in sit2. Using this kind of relationship we can assert things about a situation and test if the situation was constructed with inconsistencies.

6. Conclusions

The main idea in the work of [6] is to simplify the generation of use cases. They present a concrete and systematic way of describing actions by means of a use case pattern and simple sentences. In this article we reaffirm their conclusions that this way of describing use cases facilitates their uniform understanding.

For practical use, we suppose the existence of a tool to automatically establish standards for using the terms (words): defining, characterising, classifying, and relating them in the same way (uniform) to all use cases describing the functionality of the system. This automatic help is called term's management and should facilitate the construction of an initial object model of the system, describing the problem domain.

The term's management is based on the use of ontologies. The business ontology describing the semantics of the terms characterising the problem domain and their relationships, and the domain ontology describing the terms of the domain in the context of the use case.

The business ontology may not exist before the specification of use cases. Nevertheless, it can be incrementally constructed with those terms perceived as relevant to the domain by a domain expert. The relevant terms are identified by a domain expert with the help of a syntactic analyser.

The domain ontology has two sources: the textual specification of the use cases (to which we use the structural patterns and the rules of style described in [6]), and the business ontology. The information describing the elements and relationships proper to the use case model are taken from the textual specification of the use case. The business ontology gives complementary information that makes it easier to construct the object model of the problem domain.

For medium to large projects with dozens of use cases and a greater number of scenarios it is advisable to have some automatic way to deal with use cases, querying them and finding inconsistencies. To this end, we showed how to use situation theory to describe use cases while retaining some of the natural language expressiveness.

The next step would be to develop the environment to be used in larger experiments and to investigate the use of situation theory to deal with inconsistencies in use cases and scenarios.

References

1. J. Barwise and J. Perry. *Situations and Attitudes*. Bradford Books, The MIT Press, 1983.
2. E. Berard. *Be careful with use cases*, 1996. Available at www.toa.com/pub/usecase.txt.

3. A. W. Black. *Using a computational situation theoretic language to investigate contemporary semantic theories*. Schloss Dagstuhl Seminar IBFI, Report 57, 1993. Available at www-cs.cmu.edu/~awb/.
4. A. Cockburn. *Goals and use cases*. Journal of Object-Oriented Software Engineering, 10(5):35-40, 1997.
5. R. Cooper. *A working person's guide to situation theory*. In Steffen Leo Hansen and Finn Soeressen, editors: *Topics in Semantic Interpretation*. Samfundslitteratur, Frederiksberg, 1992. Available at www.ling.gu.se/~cooper/papers.html.
6. I. Díaz and A. Matteo. *Alternativas gramaticales del idioma español y la Especificación de Casos de Uso*. In Conferencia Latinoamericana de Informática (CLEI'2001), Mérida, Venezuela, September 2001.
7. M. Ersan and V. Akman. *Situated modelling of epistemic puzzles*. Technical report BU-CEIS-94-30, Bilkent University, Department of Computer Engineering and Information Science, 1994.
8. H. Nakashima, H. Suzuki, P.K. Halvorsen, and S. Peters. *Towards a computational interpretation of situation theory*. In International Conference on Fifth Generation Computer Systems (FGCS'88), p. 489-498, Tokyo, 1988.
9. F. A. C. Pinheiro. *Preliminary thoughts on using situation theory for scenario modelling*. In V Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes de Software, p. 272-280, La Habana, Cuba, 23-26, April 2002.
10. Rational Software Corporation, OMG Inc. *Unified Modeling Language Documentation Set*, version 1.3, June 1999. (available at www.rational.com/uml).
11. H. SchÄutze. *The* ***PROSIT*** *language, version 0.4*. Csli memo, Center for the Study of Language and Information, Stanford University, Stanford, CA, 1991.
12. E. Tin and V. Akman. *Computing with causal theories*. International Journal of Pattern Recognition and Artificial Intelligence, 6(4):699-730, 1992.