Checking Semantics in UML Models: Use Cases Diagrams

Rodolfo Gómez and Pablo Fillottrani
Grupo de Investigación en Sistemas Orientados a Objetos
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur, Bahía Blanca, Argentina
[rgomez,prf]@cs.uns.edu.ar

Abstract

Constraints add to the semantics to UML models in the form of statements which are expected to hold for the model to be considered correct, i.e., to satisfy system requirements. Constraints are considered in the UML metamodel as adornments attached to model elements, and languages like OCL allow for different statements to be expressed.

Despite several CASE-like tools are currently supporting UML diagram and constraint definition, as far as we know none of them provides support for verification and maintainance of constraint consistency. Having this kind of facility helps users in the design of UML models, specially when these models are complex and the impact of model evolution is difficult to trace.

This work studies how constraint consistency may be compromised when model elements are introduced or modified during system development. Focus is over use case diagrams, which suffice to highlight a number of important issues related with consistency maintenance. We introduce procedures for verifying and maintaining constraint consistency, and illustrate them using a simple constraint specification language. Our goal is to make a first step in the development of tools for automatic verification and mantainance of model semantics.

Keywords: Object-Oriented Modeling, Constraints, Semantic Checking, UML

1 Introduction

Modeling is one of the most crucial activities during all the phases in any software development life cycle. Typically, system development is a very complex task, and working with models helps to handle this complexity in an organized way, allowing us to reason about the properties that entities possess. Models have a broad acceptance among all engineering disciplines, mainly because it supports such general principles as abstraction, decomposition, and generality. In software engineering, notation, techniques and methodology for object-oriented model building has been lately the focus of active research work [11, 2].

In the case of object-oriented software development, models are composed by a number of communicating and well delimited elements. Although such models are sometimes harder to develop, they are easier to understand, and simpler to maintain and modify[8]. Thus, reusability of elements among models is enhanced.

In order to maximize these properties, we need tools that support the process of object-oriented development by serving as repository of previously asserted knowledge, checking the integrity of the model and maintaining the different views that form each model. The Object-Oriented Systems Research Group at the Universidad Nacional del Sur is currently working on such a tool, called HEMOO (Herramienta para Edicin de Modelos Orientados a Objetos, Object-Oriented Model Editing Tool), for supporting the development of object-oriented models using UML[4]. The objective is to assist the user in the management of the semantic aspects of a model, as well as the notational ones. In this sense the tool supplies flexible semantic checks that can be invoked on demand. Checking on demand means allowing the user to set those defined properties that will be checked, and the time when this control is done. Flexibility is achieved by providing the user the possibility to define his/her own checks, guaranteeing their treatment on an equal basis as predefined ones. Consequently, it is expected this automatic control will reduce the manual, error-prone work of maintaining model consistency across all life cycle phases.

UML, *Unified Modeling Language* is a graphical language for visualizing, specifying, constructing and documenting the elements of a software intensive system [9]. UML provides notation for expressing the model, in the form of graphic and text elements. Attached to these elements, there is a semantic interpretation that attempts to capture the meaning of the model, and it is represented by the `constraint` mechanism[13]. Constraints are one of the three extensibility mechanisms that UML introduces, although currently the language for expressing them is not standard (natural language, or OCL[15] can be used for this purpose).

Supporting tools for object oriented development in UML should not only provide a graphic editor for the notation, but also help to ensure the coherence of its semantic aspects. The design of the latter is a very difficult task because the intended meaning depends heavily on the purpose of the model in a specific moment. For example, sometimes it may be desirable to check the consistency of the relationships among classes, but later it may be more important to assure the type consistency of every method invocation.

In this paper, we analyze the impact on asserting constraints for different elements in use case diagrams, showing the conditions these constraints must satisfy in order to be consistent with respect to standard UML semantics and other, user-defined constraints in the model. In order to do so, we propose a small formal constraint specification language. It is important to note that constraint consistency does not guarantee constraint satisfaction, so our tool does not support a formal methodology for system development. Instead, its objective is to evaluate all the formal knowledge represented in the model in order to make an earlier detection of

design problems. Such evaluation should be done when the model is considered "stable", that is between two consecutive major changes.

The structure of this paper is as follows. Section 2 introduces a small constraint specificacion language and basic concepts. Section 3 describes the kind of constraints appearing over use case diagrams, and presents a procedure for checking and maintaining consistency among these constraints. Section 4 shows those actions that are required to be performed for some common model evolutions, e.g., design changes applied over actors or use cases. Section 5 highlights a number of issues which have arisen during this first research stage. Future research directions are also addressed.

2 Defining constraints and constraint checks

Constraints may be thought as statements which add to the semantics of UML model elements. These statements are expected to hold for the model to be considered correct, i.e., to satisfy the system requirements. Constraints are modelled by UML metamodel class `Constraint`, and can be attached to elements like classes, actors, use cases, associations, operations, etc. In this paper we will only consider the following: preconditions and postconditions attached to actor operations, use case operations or use cases as a whole; and invariants attached to actors. Preconditions (postconditions) are required to hold before (after) a given set of actions could be performed. On the other hand, invariants are required to hold permanently in every stable state of the object.

Let A be an atom, i.e., an undecomposable UML model element where constraints may be enforced such an attribute, an operation parameter, an operation result, etc. Let $z \in \mathbb{Z}$ be a constant value, $=, <, >, \leq, \geq, \neq$ the relational operators (as usually interpreted over $\mathbb{Z}$) and $\neg$ the negation operator (in the clasic first-order logic sense). The syntax for the constraint specification language is formally provided by the following BNF grammar:

$$\begin{array}{lll} \textit{Constraint:} & C & ::= BC \mid IC \\ \textit{Boolean Constraint:} & BC & ::= A \mid \neg A \\ \textit{Integer Constraint:} & IC & ::= A = z \mid A < z \mid A > z \mid A \leq z \mid A \geq z \mid A \neq z \end{array}$$

It is important to mention that we assume that constraints are well-formed. For instance, boolean (integer) atoms cannot appear in integer (boolean) constraints, operation results cannot appear in operation preconditions, etc. Following definitions introduce the concept of constraint consistency based on the previous language. Although more expressive constraint languages would require deeper considerations, our consistency definition serves as a ground for explaining verification procedures and illustrating kew concepts through a number of small examples.

DEFINITION **1** Let c be an integer constraint. We define the *interval* of c as the set of integers denoted by c. Formally, the interval for c, $\mathcal{I}_c$, is defined as follows:

$\mathcal{I}_{A=z} = \{z\}$ $\quad \mathcal{I}_{A<z} = \{i | i \in \mathbb{Z}, i < z\}$ $\quad \mathcal{I}_{A>z} = \{i | i \in \mathbb{Z}, i > z\}$
$\mathcal{I}_{A\leq z} = \{i | i \in \mathbb{Z}, i \leq z\}$ $\quad \mathcal{I}_{A\geq z} = \{i | i \in \mathbb{Z}, i \geq z\}$ $\quad \mathcal{I}_{A\neq z} = \mathbb{Z}/\{z\}$ □

DEFINITION **2** Let c be a constraint. The *complement* of c, denoted as $\bar{c}$, is defined as follows:

$\overline{A}$ is $\neg A$ $\quad \overline{\neg A}$ is A $\quad \overline{A = z}$ is $A \neq z$ $\quad \overline{A < z}$ is $A \geq z$
$\overline{A > z}$ is $A \leq z$ $\quad \overline{A \leq z}$ is $A > z$ $\quad \overline{A \geq z}$ is $A < z$ $\quad \overline{A \neq z}$ is $A \neq z$ □

DEFINITION **3** Two constraints c_1, c_2 are said to be *comparable* if both refer to the same atom. Two comparable constraints c_1, c_2 are said to be *inconsistent* if: a) both are boolean constraints,

and c_1 and c_2 are complementary literals, or b) both are integer constraints, and $\mathcal{I}_{c_1} \cap \mathcal{I}_{c_2} = \emptyset$. Consequently, a set of constraints is said to be inconsistent if it includes at least a pair of inconsistent constraints, and it is regarded as consistent otherwise. □

EXAMPLE **1** Suppose an actor `Customer` being part of the use case diagram modeling a video rental system. Let `isVIP` and `rentalsHeld` be a boolean and an integer constraints, respectively. They may appear in operation preconditions like {`isVIP`} `rentSpecialOffers()` or in actor invariants like {`rentalsHeld`$\leq$`3`}. Then, for instance, a) `isVIP` and $\neg$ `isVIP` are complementary literals, and yet inconsistent constraints, b) `rentalsHeld`$\leq$`3` and `rentalsHeld>5` are inconsistent but not complementary, c) `rentalsHeld`$\leq$`3` and `rentalsHeld>3` are complementary and d) `rentalsHeld`$\leq$`3` and `rentalsHeld`$\leq$`1` are consistent constraints. □

Constraint checking comprises two stages: *consistency checking* and *satisfaction checking*. Consistency checking is a general verification process where constraints attached to model elements are partitioned into sets (called *consistency bases*) which are expected to be consistent. For instance, let a given actor A be attached a set of invariants INV, and a given operation of A be attached a set of preconditions, PRE. Consistency bases include, among others, the sets INV, PRE and $INV \cup PRE$. One of the objectives of this work is to present a precise characterization of the consistency bases that are checked, and the verification process itself.

On the other hand, satisfaction checking is a specific verification process which only applies to certain model elements where postconditions can be attached. This process verifies that the element definition provides at least a way for the postconditions to be satisfied. For instance, postconditions attached to use cases as a whole should be satisfied by some of their courses of events. Usually, this process will be performed when the model is considered stable, i.e., between major model-evolution periods. Of course, constraints should be defined on the same shared ontology[7, 1] for consistency and satisfaction checks to be feasible. For the sake of brevity, this work will not consider this kind of constraint checking. Nevertheless, it is an interesting subject of further research.

Before discussing how consistency checking is applied to use case diagrams, there is still an issue deserving further comments. Every kind of model element has a proper set of consistency bases, but the kind of model evolution considered, i.e., whether an element is added, removed or modified, and the time when the check is done may require only some of them to be constructed or verified. For instance, if a new operation is added to an existing actor, and consistency has been already verified for that actor, then the only sets to verified are operation preconditions in isolation, PRE, and against actor invariants, $INV \cup PRE$. Clearly, neither the consistency of actor invariants nor the consistency of preconditions attached to other existing operations are compromised by these kind of actor evolution. Note, also, that there is a subtle difference between a consistency base like INV and others like $INV \cup PRE$. The former should be built, checked, and maintained with actor A. The latter, however, is based on two previously-built and previously-checked consistency bases. So it only stands for the fact that invariants should consistent against preconditions . There is no need to maintain this base with the actor, nor to re-check invariants and preconditions in isolation.

DEFINITION **4** A given number of consistency bases $B_1, B_2, \ldots, B_n$ are said to be *cross-checked*, an denoted as $B_1|B_2|\ldots|B_n$, if the set $B_1 \cup B_2 \cup \ldots \cup B_n$ is expected to be consistent provided each B_i, $1 \leq i \leq n$ is known to be consistent in isolation. □

Finally, we should mention that no particular procedure is assumed to be used for check consistency on a given set, nor for cross-checking consistency bases despite they depend on the

Model Element	*Checked in isolation*	*Cross-checked*
Actor A	INV_A, PRE^i_A and $POST^i_A$ for all $1 \leq i \leq n$	$INV_A \vert PRE^i_A$, $INV_A \vert POST^i_A$, $AncINV \vert INV_A$, $AncINV \vert PRE^i_A$ and $AncINV \vert POST^i_A$ for all $1 \leq i \leq n$
Use case U	INV_U, PRE_U, $POST_U$, PRE^i_U and $POST^i_U$ for all $1 \leq i \leq m$	$INV_U \vert PRE_U$, $INV_U \vert POST_U$, $INV_U \vert PRE^i_U$, $INV_U \vert POST^i_U$ for all $1 \leq i \leq m$

Figure 1: Consistency bases in structural checking

expressiveness of the constraint specification language. Research on this subject can be found on works like [14, 5, 6].

3 Constraint checking on Use Case Diagrams

In use case diagrams, constraints may appear as invariants attached to actors or use cases as a whole, postconditions and postconditions attached to actor operations, use case operations or use cases as a whole. The two former types defines the properties that model a course of event. However, if attached to use cases as a whole, postconditions apply only over the basic course of events [12], i.e., others such as exceptional paths may have different outcomes and no postconditions are attached to them. Use case operations model actor interactions, system responses or internal actions such as state changes. When modeling actor interactions, they match operations defined in one of the several possible actor interfaces. Other operations are regarded private to the use case.

Consistency checking is performed on two different grounds. *Structural checking* verifies the consistency of actors and use cases in isolation. *Behavioral checking* verifies the consistency of courses of events, and it is expected to be performed after the structural check. In the context of a CASE-like tool, both types of checks are expected to be performed on demand and after major model evolutions. Typically, the user will complete the use case diagram until a certain extent, perform structural checking on isolated actors and use cases and finally perform behavioral checking on particular use-case courses of events. Despite this whole process may be entirely repeated after the occurrence of major changes over actors and use cases, advantages may be taken from previous checks (see section 4). This section is devoted to explain how consistency checking is applied to use case diagrams, and the structures that are involved in the process and maintained for future checks.

Structural checking

Let A be an actor with operations $O_A = \{a_1, \ldots, a_n\}$, and U a use case with operations $O_U = \{u_1, \ldots, u_m\}$. Let INV_A, INV_U be the invariants attached, respectively, to A and U, PRE_U and $POST_U$ the preconditions and postconditions attached to U as a whole, PRE^i_A and $POST^i_A$ the preconditions and postconditions attached to $a_i \in O_A$, PRE^i_U and $POST^i_U$ the preconditions and postconditions attached to $u_i \in O_U$. In the general case, actors may be part of an inheritance hierarchy. Let $AncINV$ be the set of invariants attached to all ancesters of actor A in such a hierarchy. Figure 1 shows the consistency bases which are relevant to structural checking.

Example **2** Following example 1, suppose in the earlier development stages there was no

policy regarding the number of rentals a customer may currently held. In addition, consider the customer may a) request a delayed devolution of his rentals provided he currently hold more than 3 videos, and b) rent a 10-video pack at promotional price. This is modelled by the following `Customer` operations, with properly attached preconditions and postconditions:

{`rentalsHeld>3`} requestDelayedDevolution()

rent10VideoPack() {rentalsHeld≥10}

It is possible that later stages introduce changes that make actor `Customer` structurally inconsistent. For instance, if a new policy imposes that no more than 5 videos may be currently held, then inconsistences arise on previous operations. If we assume the new policy is modelled as actor invariant $INV = \{\texttt{rentalsHeld} \leq 5\}$, the structural check finds inconsistencies when cross-checking $INV|PRE$ and $INV|POST$, where $PRE = \{\texttt{rentalsHeld} > 3\}$ and $POST = \{\texttt{rentalsHeld} \geq 10\}$ are, respectively, the preconditions and postconditions attached to operation `requestDelayedDevolution()`. □

Behavioral checking

Use cases model interactions between the system and external entities, i.e., the actors. Use cases always comprise at least a basic course of events, and possible many alternative or exceptional paths. A basic course describes a simple, correct and most common interaction in normal conditions; an alternative path describes a possible correct interaction which usually does not occur; and an exceptional path describes uncommon, error-handling interactions. All type of paths will be generally treated, unless otherwise stated , as courses of events.

A course of event is assumed to be expressed as a sequence of operations. Operations may be one of the following: a) an actor operation, modeling the fact that some actor interaction is requested; b) a use case internal operation, modeling system responses or some kind of internal system processing; c) an inclusion point, stating that the basic functionality of a given use case is requested at this point; d) an extension point, denoting this course may conditionally be extended with the basic functionality of a given use case; or e) a conditional branch, denoting a possible bifurcation of this course in others like alternative or exceptional paths. While others, richer classifications may be provided for the operations appearing on a course of event, this is broad enough to model the interactions appearing in most of the systems and considers <<*include*>> and <<*extend*>> associations. It also suffices to describe the consistency checks over use case diagrams. For more details about use case diagrams see, e.g., [12], [8] and [10].

Behavioral checking verifies that semantics of individual model elements holds when they interact in a given course of events. Semantically, a course of events may be thought as an interleaved sequence of stable and transient periods. Transient periods are represented by operation execution, and interaction is considered stable a) before it begins, b) between two consecutive transient periods and c) after it finishes. As constraints are attached to model elements interacting in the course of events, it is expected a given set of constraint holds at every stable period as a result of all previous transient periods. Behavioral checking verifies that constraints remain consistent during stable periods. For instance, postconditions attached to a given operation may be inconsistent with the preconditions attached to the immediately following operation appearing in the course of events.

Different kind of constraints are expected to hold during different stable periods. For instance, invariants attached to a use case as whole are expected to hold during all stables periods in the interaction. On the other hand, preconditions attached to a given operation are only expected to hold during the stable period before the operation is performed, i.e., it may be the case they do not hold in next stable periods. Finally, postconditions are expected to hold after the operation has been performed, and until postconditions of future operations change them.

Behavioral checking comprises two stages. First, the outcome of all operations on a given course of event is calculated as the sets of postconditions which hold at every stable period in the course. These sets, called *states*, are checked for consistency and then maintained as a *state sequence* attached to the course of events. The second stage collects, for each operation in the course of events, the set of constraints that are expected to hold before that operation is performed, and then this set is verified to be consistent with the stable period currently holding at that point. State sequences are built as follows:

DEFINITION **5** Let U be a given use case. Let the operation sequence $O_U =< o_1, \ldots, o_n >$ represent a given course of events in U. Then $SEQ_U =< S_0, \ldots, S_n >$ denotes the state sequence defined over O_U, where S_i, $0 \leq i \leq n$ is defined as:

$S_0 = \emptyset$

$S_i = (S_{i-1} \cup P_i)/\{c | c \in S_{i-1}$ *and exists* $c' \in P_i$ *such that* c *and* c' *are inconsistent*$\}$

where P_i depends on operation o_i as follows:

1. If o_i denotes a call for a given operation o_j defined in actor A, then $P_i = POST^j_A$, i.e., the postconditions attached to o_j.
2. If o_i denotes a call for an internal operation o_j defined in use case U, then $P_i = POST^j_U$, i.e., the postconditions attached to o_j.
3. If o_i denotes the inclusion of a given use case IU is included, then $P_i = POST_{IU}$, i.e., the postconditions attached to IU as a whole.
4. If o_i denotes an extension point related to a given extending use case EU, and the extension is regarded as performed over O_U, then $P_i = POST_{EU}$, i.e., the postconditions attached to EU as a whole. If o_i is an extension point but it is not related to any extending use case, or the extension is regarded as not performed over O_U, then $P_i = \emptyset$.
5. If o_i denotes a branching operation then $P_i = \emptyset$. □

Based on the state sequence built during the first stage, the second stage of the process relates every operation in the course of events with a set of cross-checked consistency bases, collectively known as the *previous base* attached to that operation. This ensures that stable periods maintain a consistent set of constraints. Previous bases are defined next.

DEFINITION **6** Let U be a given use case, $O_U =< o_1, \ldots, o_n >$ a given course of events over U, and $SEQ_U =< S_0, \ldots, S_n >$ the state sequence over O_U. Let INV_{AU} be the set of all invariants attached to any actor involved during the course O_U, plus all invariants attached to children of these actors in the possible inheritance hierarchy, plus the invariants attached to U as a whole. This set must be verified for consistency and maintained with O_U. The previous base related to operation $o_i \in O_U$, $PREV_i$, is defined as follows:

$PREV_i = INV_{AU}|P_i|S_{i-1}$

where P_i depends on o_i as follows:

1. If o_i denotes a call for a given operation o_j defined in actor A, then $P_i = PRE^j_A$, i.e., the preconditions attached to o_j.
2. If o_i denotes an internal operation o_j defined in use case U, then $P_i = PRE^j_U$, i.e., the preconditions attached to o_j.
3. If o_i denotes the inclusion of a given use case IU is included, then $P_i = PRE_{IU}$, i.e., the preconditions attached to IU as a whole.

4. If o_i denotes an extension point related to a given extending use case EU, depending on condition c, and the extension is regarded as performed over O_U, then $P_i = PRE_{EU} \cup \{c\}$, i.e., the postconditions attached to EU as a whole plus the condition c that must be true so the extension could be performed. If o_i denotes an extension point but it is not related to any extending use case, or the extension is regarded as not performed over O_U, then $P_i = \overline{c}$.
5. If o_i denotes a branching operation depending on condition c, and c is regarded to hold over course O_U, then $P_i = \{c\}$. Otherwise $P_i = \{\overline{c}\}$.

Finally, the special cases for the first and last operations are considered

1. If o_i denotes the first operation on O_U, then the preconditions attached to U as a whole, PRE_U, must also be cross-checked against $PREV_i$.
2. If o_i denotes the last operation on O_U, then the postconditions attached to U as a whole, $POST_U$ must be cross-checked against the last state of the sequence, S_n. □

EXAMPLE **3** Suppose, in the video-rental system, a use case modeling a customer registration with the following (simplified) basic course of events:

1. `The system initializes customer's historic info. The number of rented videos, and the number of years as a member are set to 0.`
2. `The customer enters the name of up to three people who may rent on his behalf.`

The following policy currently holds, where *"allowed people"* stands for the people who may rent on behalf of the customer: *"Allowed people may only be added during the first year of membership"*. In addition, Customer is attached the invariant {`rentalsHeld` $\leq$ `5`}.
The course of events may be structured as follows, where operations are adorned with preconditions and postconditions.

1. `Customer.init()` {`totalOfRentals = 0, yearsAsMember = 0`}
2. {`yearsAsMember = 0`} `Customer.addAllowedPerson()`

Now consider the effect of:
a) A change in the policy: *Allowed people may only be added during the first year of membership and after the Customer has rented more than 20 videos.*
b) A change in the course of events: The step `'When a customer registers, he is given a video for free.'` is inserted in the course between steps 1 and 2.
The modified course is shown next:

1. `Customer.init()` {`totalOfRentals = 0, yearsAsMember = 0, rentalsHeld = 0`}
2. `Customer.rentVideoForFree` {`totalOfRentals = 1, rentalsHeld = 1`}
3. {`yearsAsMember = 0, totalOfRentals` $\geq$ `20`} `Customer.addAllowedPerson()`

We can see that preconditions of `addAllowedPerson()` are now inconsistent with the previous stable period. Indeed, conflict arises between constraints {`rentalsHeld = 1`} and {`totalOfRentals` $\geq$ `20`}. This kind of conflicts can be detected by behavioral checking (shown in boldface). Next we show only the relevant steps of this verification process.

	State	*Operation preconditions*	*Other constraints*
1.	{`totalOfRentals = 0, yearsAsMember = 0, rentalsHeld = 0`}	{}	{`rentalsHeld` $\leq$ `5`}
2.	{**`totalOfRentals = 1`**, `yearsAsMember = 0, rentalsHeld = 1`}	{`yearsAsMember = 0`, **`totalOfRentals` $\geq$ `20`**}	{`rentalsHeld` $\leq$ `5`}

We can also see, for instance, that customer's invariant {`rentalsHeld` $\leq$ `5`} is not compromised on any state, and that postconditions of attached to different operations may be contradictory without being inconsistent, such as {`totalOfRentals = 0`} and {`totalOfRentals = 1`}. □

4 Impact of model evolutions on consistency checks

Use case diagrams may change in different ways, and in different times during system development. Model elements like actors, use cases, operations, associations and constraints may be added, removed or modified during this process. In this dynamic environment, consistency may be maintained by means of a simple strategy: just perform structural and behavioral checks after every change in the model. However, a change may usually compromise only the consistency of certain elements. Furthermore, new consistency checks may benefit from previous checks and previously-built structures like state sequences. This section traces the effects of changes on model-element consistency, and describes what actions are required to check and maintain consistency after a group of changes have been applied to the model.

Additions

Figure 2 shows the actions required to maintain consistency when additions are performed inside an actor. Notation is as follows. A denotes an actor and o_i, $1 \leq i \leq n$ denotes a given operation defined on A. $NewINV_A$, $NewPRE^i_A$ and $NewPOST^i_A$ respectively denote a new set of invariants added to A, a new set of preconditions and a new set postconditions attached to o_i. INV_A, PRE^i_A and $POST^i_A$ respectively denote pre-existent sets of invariants in A, preconditions and postconditions attached to o_i. All bases $NewINV_A$, $NewPRE^i_A$ and $NewPOST^i_A$ are also required to be checked for consistency.

Figure 3 shows the actions required to maintain consistency when additions are performed inside a use case. Notation is as follows. U denotes a use case and o_i, $1 \leq i \leq n$, denotes a given operation defined on U. $NewINV_U$, $NewPRE_U$, $NewPOST_U$, $NewPRE^i_U$ and $NewPOST^i_U$ respectively denote a new set of invariants, preconditions and postconditions attached to U as a whole, and preconditions and postconditions attached to o_i. PRE_U, $POST_U$, PRE^i_U and $POST^i_U$ respectively denote a pre-existent set of preconditions and postconditions attached to U as a whole, and preconditions and postconditions attached to o_i. $NewCOND^i_U$ and $COND^i_U$ respectively denote a new and a pre-existent set of conditions attached to a branching operation o_i. All bases $NewPRE_U$, $NewPOST_U$, $NewPRE^i_U$, $NewPOST^i_U$ and $NewCOND^i_U$ are required to be checked for consistency.

For the sake of brevity we have not described the consequences of adding new operations to courses of events. The impact of actor additions over an inheritance hierarchy is analyzed in other work. Nevertheless, these changes are not hard to trace to the consistency checks and structures that we have defined, and many of them may be decomposed and treated as a sequence of simpler additions depending on the attached constraints.

Removals and modifications

The removal of model elements does not affect consistency for existing elements, but it may relax the consistency requirements of further evolutions. For instance, when removing an actor operation from a course of events the course itselfs remains consistent, but it is possible that operation postconditions remain part of future states. If this is so, this may prevent future operation inclusions if their preconditions are not consistent with the now obsolete postconditions.

Element added	*Consequences*
invariants $NewINV_A$	1. Bases $INV_A \vert NewINV_A$ and $PRE^i_A \vert NewINV_A$, $POST^i_A \vert NewINV_A$ for all $1 \leq i \leq n$ must be cross-checked. 2. All previous bases related to any course of events in any use case where A is involved must be cross-checked against $NewINV_A$. 3. All consistency bases INV_{AU} related to any course of events in any use case where A is involved must be re-built.
preconditions $NewPRE^i_A$ attached to o_i	1. Bases $PRE^i_A \vert NewPRE^i_A$ and $INV_A \vert NewPRE^i_A$ must be cross-checked. 2. All previous bases related to calls for o_i in any course of event in any use case where A is involved must be cross-checked against $NewPRE^i_A$.
postconditions $NewPOST^i_A$ attached to o_i	1. Bases $POST^i_A \vert NewPOST^i_A$ and $INV_A \vert NewPOST^i_A$ must be cross-checked. 2. The state sequence related to any course of events in any use case where A is involved must be re-built from the point where o_i is called in that course. Consequently, all previous and related to operations following o_i in that course must be re-built and cross-checked.

Figure 2: Adding new elements to actors.

The only removals requiring some maintenance actions are those concerned with postconditions and invariants, whether they are caused by whole actor or use case removals, or by the removal of pre-existent postconditions or invariant subsets. In the case of postconditions the state sequences related to courses of events where these elements are involved must be re-built. In the case of invariants, the consistency base INV_{AU} must also be re-built. As we said before, there is no need for any kind of consistency check. Modifications are regarded as decomposed in removals and additions. For instance, if a given postcondition p is changed to p', consequences are the same as if p were removed and then p' were added.

5 Conclusions and further work

We have discussed how semantics assigned to UML use case diagrams may be verified and maintained through model evolutions. Our analysis relies on constraint consistency. Procedures to check and maintain such consistency were developed and illustrated trough a small, structured constraint specification language. We have also described how model changes, e.g., modifying actor invariants, may compromise constraint consistency, and how previous checks helps to avoid unnecessary verifications.

This research has been done in the context of development of the HEMOO tool, but it is worthy to be compared with related works. TCM (Toolkit for Conceptual Modeling) [3] is a tool supporting edition and verification of UML diagrams, and as far we know it is the only tool with such verification capabilities. However, TCM maintains consistency only over UML-predefined constraints, like well-formedness rules and predefined semantics of model elements. On the other hand, HEMOO is also intended to provide controls over use-defined semantics.

Element added	*Consequences*
invariants $NewINV_U$	1. Bases $INV_U \vert NewINV_U$, $PRE^i_U \vert NewINV_U$, $POST^i_U \vert NewINV_U$ for all $1 \leq i \leq n$ and $PRE_U \vert NewINV_U$, $POST_U \vert NewINV_U$ must be cross-checked. 2. All previous bases related to any course of events over U must be cross-checked against $NewINV_U$. 3. The consistency base INV_{AU} related to courses of events over U must be re-built.
preconditions $NewPRE_U$ attached to U as a whole	1. Base $PRE_U \vert NewPRE_U$ must be cross-checked. 2. All previous bases related to inclusions of U (or extensions by U) on any course of event must be cross-checked against $NewPRE_U$.
postconditions $NewPOST_U$ attached to U as a whole	1. Base $POST_U \vert NewPOST_U$ must be cross-checked. 2. The state sequence related to any course of events must be re-built from the point where U is included (or extends a base case). Consequently, all previous and next bases related to operations in the course following the inclusion of U (or the extension by U) must be re-built and cross-checked.
preconditions $NewPRE^i_U$ attached to o_i	1. Base $PRE^i_U \vert NewPRE^i_U$ must be cross-checked. 2. All previous bases related to calls for o_i on any course of event must be cross-checked against $NewPRE^i_U$.
postconditions $NewPOST^i_U$ attached to o_i	1. Bases $POST^i_U \vert NewPOST^i_U$ must be cross-checked. 2. The state sequence related to any course of events over U must be re-built from the point where o_i is included in the course. Consequently, all previous and next bases related to operations in the course following o_i must be re-built and cross-checked.
conditions $NewCOND^i_U$ attached to a branching operation o_i	1. Base $COND^i_U \vert NewCOND^i_U$ must be cross-checked. 2. All previous bases related to calls for o_i on any course of event must be cross-checked against $NewCOND^i_U$.

Figure 3: Adding new elements to use cases.

In this first stage of research our verification procedures heavily depends on two key aspects. First, it was assumed a shared ontology supporting constraint definition: it is not possible to find inconsistences unless two or more constraints with similar or contradictory meanings can be compared. Second, use cases are required to be more structured than a simple textual description. Courses of events are treated as a sequence of operations defined over actors or use cases, with preconditions and postconditions modelling their intended meaning. Although operations are classified according to a fixed set, they are flexible enough to express commonly-found interactions.

Further research should be devoted to consider the consequences of more expressive constraint specification languages, the support for a coherent user-defined ontology and the automatic generation of structured courses of events from use case textual descriptions[16]. Once these issues had been studied, semantic verification of other UML diagrams and constraint satisfaction are expected to be addressed.

References

[1] Everett, J., Borrow, D., Stolle, R., Crouch, R., de Paiva, V., Condoravdi, C., van den Berg, M., and Polanyi, L. Making Ontologies Work for Resolving Redundancies Across Documents. *Communications of the ACM 45*, 2 (February, 2000), 55–60.

[2] Falbo, R., Guizzardi, G., Natali, A., Bertollo, G., Ruy, F., and Mian, P. G. Towards Semantic Software Engineering Environments. *Proceedings of the 14th. International Conference on Software Engineering and Knowledge Engineering (SEKE'02)* (2002).

[3] F.Dehne, and R.J.Wieringa. Toolkit for conceptual modeling, user's guide for tcm 1.2.0. Tech. Rep. IR-401, Faculty of Mathematics and Computer Science, Vrije Universiteit Amsterdam, April,1996. url: http://euklid.mi.uni-koeln.de/b/tcm/doc/usersguide/User.html.

[4] Fillottrani, P., Estevez, E., and Kahnert, S. Applying Logic Programming Techniques to Object-Oriented Modeling in UML. *Proceedings of the 14th. International Conference on Software Engineering and Knowledge Engineering (SEKE'02)* (2001), 228–235.

[5] Freuder, E. C., and Elfe, C. Neighborhood Inverse Consistency Preprocessing. In *Proceedings AAAI'96* (1996), pp. 4–9.

[6] Georget, Y., Codognet, P., and Rossi, F. Constraint Retraction in CLP(FD): Formal Framework and Performance Results. *CONSTRAINTS: An international journal, 4*, 1 (1999). Kluwer.

[7] Gruninger, M., and Lee, J. ONTOLOGY: Applications and design. *Communications of the ACM 45*, 2 (February, 2000).

[8] I. Jacobson and M. Christerson and P. Jonsson and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1997.

[9] Ivar Jacobson, G. B., and Rumbaugh, J. *The Unified Modeling Language Reference Guide.* ACM Press, Addison-Wesley, 1999.

[10] Ivar Jacobson, G. B., and Rumbaugh, J. *The Unified Modeling Language User Guide.* ACM Press, Addison-Wesley, 1999.

[11] Ivar Jacobson, G. B., and Rumbaugh, J. *The Unified Software Development Process.* ACM Press, Addison-Wesley, 1999.

[12] Kulak, D., and Guiney, E. *Use Cases, Requirements in Context.* ACM Press, Addison-Wesley, 2000.

[13] OMG. *Unified Modeling Language Specification Version 1.4.* Object Management Group, Inc., September,2001.

[14] Prosser, P., Stergiou, K., and Walsh, T. Singleton Consistencies. In *Proceedings CP'2000* (2000), Springer, pp. 353–368.

[15] Warmer, J., and Kleppe, A. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1999.

[16] Wouters, B., Deridder, D., and Paesschen, E. V. The Use of Ontologies for as a backbone for use case management. In *Proceedings of the 14th. European Conference on Object-Oriented Programming (ECOOP 2000)* (2000).