Mapping OCL constraints into CTL-like logic and SML for UML validation

Miloud BENNAMA and Thouraya BOUABANA-TEBIBEL
Laboratoire de Communication dans les Systèmes Informatiques (LCSI)
Ecole Nationale Supérieure d'Informatique (ESI)
Alger, Ageria
m_bennama@esi.dz, t_tebibel@esi.dz

The UML (Unified Modeling Language) graphical models miss providing some pertinent elements of specification as constraints over objects and operations. To fill this lack, OCL (Object Constraint Language) has been developed by IBM and integrated to UML as a modern and formal modeling language, which is easy to learn and efficient to use. On the other hand, many works emerged providing a formal semantics to UML dynamic diagrams by using CP-nets (High-level Petri Nets). The latter are verified based on system properties written in temporal logic. The purpose of this paper is to assist the UML modeler, not necessarily familiar with temporal logics, by letting him expressing the properties in OCL language and proposing an automatic mapping of OCL invariants and pre/post-conditions into CTL-like logic (Computational Tree Logic) coupled with the functional programming language Standard ML. The obtained temporal logic formulas are verified over the state space of the CP-net models derived from UML diagrams by model-checking.

UML; OCL; CP-nets; CPN-ML; ASKCTL; CPNtools; Mapping; model-checking.

1. INTRODUCTION

UML [17] is the de facto standard for specifying both of the structural and behavioral aspects of systems. OCL (Object Constraint Language [16]), an integral part of UML, allows for specifying additional constraints on UML models in a more precise and concise manner. OCL has a mathematical definition based on set theory with a notion of object model and system states. UML and OCL are easy and familiar to users, but they do not support validation tasks and their semantics is defined in a semi-formal way.

To provide a rigorous semantics for UML models, CPN-nets [10] have been used by many studies [1,3,9,20] as an expressive semantics domain. Also, OMG (Object Management Group) has inspired many UML concepts from Petri Nets, particularly, in activity diagrams. CP-nets are widely-used for specifying and analysing behaviour of concurrent systems. They consist in a transition system that supports model-checking validation method.

Model-checking technique shows that a system satisfies its specification. It requires a formal representation (as CP-nets) of the system and a specification that is often expressed in terms of a temporal logic formula [2].

A formal tool, called CPNtools, has emerged for analyzing CP-nets. It uses the functional programming language Standard ML [14,21] and CTL-like temporal logic, called ASKCTL [5], for model description, data manipulation and properties specification.

We proposed in [3] an approach to translate the Interaction Overview Diagram (IOD) to CP-nets for simulation and state space analysis using CPNtools. To specify and check system properties, the modeler is not familiar with temporal logic and ML language that require many skills.

To assist the modeler in this phase of specification, we propose to allow him to express system properties in his usual language of specification, namely OCL, and to automate the translation of OCL properties to ASKCTL and ML. The resulting formulas are evaluated over the state space of the CP-net model derived from the IOD diagram.

Various works [2,4,7,12,15,23] have been undertaken to transform OCL into other formal languages. Our approach differs from works that use temporal logics to formalize OCL constraints in that it achieves a detailed mapping of basic and complex expressions of OCL into Standard ML.

This extends our previous works on the IOD diagrams mapping into CP-net models. Unlike other works, our approach translates the class diagram, IOD diagram, and OCL specifications into the input languages of CPNtools model-checker.

The remainder of this paper is organized as follows. Section 2 presents OCL language and its constraints whereas section 3 presents the CPNtools and its input formal languages. Section 4 describes our approach of mapping of OCL expressions into ML functions and OCL constraints into ASKCTL formulas. Section 5 illustrates the application of our approach over an ATM system. Section 6 recalls the related works. Finally, Section 7 concludes and presents future works.

2. OBJECT CONSTRAINT LANGAGE

The UML has been widely accepted as a standard for object-oriented modeling language and is supported by a great number of CASE tools. The Object Constraint Language (OCL) is an integral part of UML, and was introduced to express subtleties and nuances of meaning that diagrams cannot convey by themselves.

OCL has been introduced by IBM for business modeling and adopted by UML as a mean to specify invariants of classes and types in a class model, to specify type invariant of stereotypes, to describe pre- and post-conditions on operations and methods, to describe guards, and also as a navigation language. OCL is a language of typed expressions, where an expression can be universally and existentially quantified [13].

2.1 Invariants

The OCL expression can be part of an Invariant which is a Constraint stereotyped as an «invariant». When the invariant is associated with a Classifier, the latter is referred to as a “type” in this clause. An OCL expression is an invariant of the type and must be true for all instances of that type at any time [16]

context <TypeName> inv <InvName>:

<BooleanExpression>

Optionally, the name of the constraint may be written after the inv keyword, allowing the constraint to be referenced by name.

Integrity constraints in OCL are represented as invariants defined in the context of a specific type, named the context type of the constraint. Its body, the Boolean condition to be checked, must be satisfied by all instances of the context type.

2.1 Pre/post conditions

The OCL expression can be part of a Precondition or Postcondition, corresponding to «precondition» and «postcondition» stereotypes of Constraint associated with an Operation or other behavioral feature. The contextual instance self then is an instance of the type that owns the operation or method as a feature. The context declaration in OCL uses the context keyword, followed by the type and operation declaration. The stereotype of constraint is shown by putting the labels ‘pre:’ and ‘post:’ before the actual Preconditions and Postconditions [16].

context
<TypeName>::<OperationName>(<param1>: <Type1>, ...): <ReturnType>

pre <PreName>: <BooleanExpression>

post <PostName>: <BooleanExpression>

Optionally, the name of the precondition or postcondition may be written after the pre or post keyword, allowing the constraint to be referenced by name.

3. CP-NETS AND TEMPORAL LOGIC

3.1 CPN-ML language

The CPN-ML [11] programming language is a variation of functional programming language Standard ML (SML) used in CP-nets. CPN-ML embeds the Standard ML and extends it with constructs for defining colour sets and functions, declaring variables, and writing inscriptions in CP-net models. SML provides the user with the expressiveness required to model data and data manipulation of complexity found in industrial systems. SML is also used to implement simulation, state space analysis, and performance analysis of CP-net models.

(i) Colour sets: The CPN ML language provides a predefined set of basic types inherited from Standard ML that can be used as simple colour sets.
(ii) Expressions and Types: In CP-nets, relatively simple expressions have been used as arc expressions, guards, and initial markings. It is possible to use the complete set of Standard ML expressions.
(iii) Functions: Functions are similar to the procedures and methods known from conventional programming languages.
(iv) Recursion and Lists: Such loop statements are not available in a functional programming language, which instead relies on recursive functions to express iteration.

3.2 ASKCTL logic

The ASKCTL [5] is a CTL-like logic which is interpreted over the state spaces of CP-nets. The logic has been designed to express properties of both state and transition information over the CP-net state space. The logic is powerful enough to express many of the standard CP-net properties. Using ASKCTL logic implies that we get a well understood and easy to use framework for expressing a much wider range of properties. The models over which we interpret ASKCTL are state spaces of CP-nets. These graphs carry information on both nodes and edges.

ASKCTL is a branching-time modal logic and an extension of Computational Tree Logic (CTL [8]). An ASKCTL statement is defined to be a state or a transition formula. For more details about ASKCTL syntax see [6] and [22].

3.3 CP-nets

CP-nets [10] are high-level Petri nets widely-used formal method for system specification, design, simulation and verification. They provide a graphical oriented modeling language capable of expressing concurrency, synchronization, resources sharing and non-determinism at different levels of abstraction. They combine the mathematic primitives of Petri Nets [18] and the expressive power of SML. They support a variety of verification techniques such as state space analysis and model simulation.

3.4 CPNtools

CPNtools [19] is a tool for editing, simulating, and analyzing CP-nets. The tool features incremental syntax checking and code generation, which take place while a net is being constructed. A fast simulator efficiently handles untimed and timed nets. Full and partial state spaces can be generated and analyzed, and a standard state space report contains information, such as boundedness properties and liveness properties. CPNtools supports state space analysis and model-checking of ASKCTL logic.

4. MAPPING OF OCL

4.1. Mapping of OCL expressions to ML

In OCL language, a number of basic types are predefined to the UML modeler. These predefined types, such as Boolean, UnlimitedNatural, Integer, Real, Enumeration and String, are independent of any object model. In ML language, the same basic types are available with some difference in the syntax.

4.1.1 Numbers and arithmetical operations

There are three types to express numbers in OCL: Real, Integer and UnlimitedNatural, where UnlimitedNatural is a subtype of Integer and Integer is a subtype of Real. The basic arithmetical (+, -, *, =, *abs(), min(), max()*) and comparison operations (>, <, >=, <=) are defined for numbers. Two types of conversion from Real to Integer are provided (*floor(), round()*) and additionally, a conversion to string was introduced (*toString()*). There are two operations defined for the Integer and UnlimitedNatural types only: division quotient (*div()*) and remainder (*mod()*). Table 1 shows the mapping of OCL numeral operations into ML.

The OCL UnlimitedNatural type represents the set of non-negative integers. Its OCL declaration *a : UnlimitedNatural* is expressed in ML by *var a:INT with 0..maxINT;*. All integer operations are applied in the UnlimitedNatural subtype except the negation operation.

The OCL Real type represents numerals with a decimal point. All integer operations are applied in the Real supertype except the *div* and *mod* operations. Additionally, the *floor* and *around* operations are supported in Real type.

4.1.2 Boolean type mapping

Basically OCL users work with the Boolean values true and false that are the instances of the Boolean type. OCL provides the basic logical operations *and, or, not* as well as the derived operations *xor, implies*. The OCL declaration syntax of a Boolean *a: Boolean* is expressed in ML syntax by *var a: BOOL;*

In ML, the OCL Boolean conjunction '*a and b'* becomes '*a andalso b'*, disjunction '*a or b'* becomes '*a orelse b'*, implication '*a implies b'* becomes '*not a orelse b*' and negation '*not a'* is the same. The table 2 shows the mapping of OCL Boolean operations into ML.

OCL	**ML**	**Description**
A:Integer	var a: INT;	integer type declaration
A: UnlimitedNatural	var a:INT with 0..maxINT ;	natural type declaration
A:Real	A:Real;	real type declaration
A = B , A <> B	A = B, A <> B	equal, not equal
A > B , A >= B , A< B, A <= B	A > B , A >= B , A< B, A <= B	greater, greater or equal, less, less or equal
A + B , A – B , A*B	A + B , A – B , A*B	addition, subtraction, multiplication
- A	~A	Negation (only for real and integer)

A. div(B) , A.mod(B)	A div B , A mod B	division quotient, remainder (only for integer and natural)
A.min(B), A.max(B)	INT.min(A,B), INT.max(A,B)	minimum, maximum
A.abs	abs A	absolute value
A.floor()	floor A	the largest integer not larger than A (only for real)
A. round()	round A	the integer nearest to A (only for real)

Table 1. Mapping of Numeral operations

4.1.3 Strings and text operations

Strings are specified by sequences of printable ASCII characters surrounded with double quotes. The OCL declaration syntax of a string *a: String* is expressed in ML syntax by *var a: String;*. In both OCL and ML, the length of a string (size) can be determined, a string can be projected to a substring, and two strings can be concatenated (^). Also, it is possible to access single or all characters of a given string and to applied case conversion.

Table 3 shows the mapping of OCL string operations into ML.

OCL	ML	Description
A: Boolean	var a: BOOL;	Boolean type declaration
A = B , A <> B	A = B , A <> B	equality , inequality
A and B	A andalso B	conjunction
A or B	A orelse B	disjunction
A xor B	A <> B	exclusive disjunction
A implies B	not A orelse B	Implication (if A then B else true)
not A	not A	negation
if A then B else C endif	If A then B else C	If the expression A is true then B must be true else C must be true

Table 2. Mapping of Boolean operations

OCL	ML	Description
A: String	var A: STRING;	String type declaration
A = B , A <> B	A = B , A <> B	equality and inequality
A .size()	String.size A	String length
A.concat(B)	A^^B	Concatenate two strings
A.subString(i,i+len)	substring (A,i,len)	extract a substring of length len starting at position i in A, first position is 0
A.characters()	explode A	convert string A to list of chars
A.at(i)	sub (A, i)	returns the i(th) char of A, counting from zero.
A.toUpper()	map toUpper A	convert all chars to uppercase
A.toLower()	map toLower A	convert all chars to uppercase

Table 3. Mapping of string operations

4.1.4 Enumeration type

OCL enumeration types are user-defined types. An enumeration type is defined by specifying a name and a set of literals. An enumeration value is one of the literals used for its type definition. In ML, enumerated values are explicitly named as identifiers in the declaration. These values must be alphanumeric identifiers. Table 4 shows the mapping of OCL enumeration operations into ML.

OCL	ML	Description
A : Enum{v0,v1,..,vn}	Var Enum = with v0| v1|...|vn;	enumeration type declaration
A = #vi , A <> #vi	A = #vi , A <> #vi	equality and inequality with an enumeration value

Table 4. Mapping of enumeration operations.

4.1.5 TupleType

Informally known as record type. It combines different types into a single aggregate type. The parts of a TupleType are described by its attributes, each having a name and a type.

The OCL TupleType declaration *A: Tuple*(id_1:$type_1$, id_2:$type_2$,…, id_n:$type_n$) is expressed in ML by :
colset tuple = record id_1:$type_1$ * … * id_n:$type_n$;
var A:tuple;

Values of this color set have the form: {id_1=v_1,..., id_n=v_n} where vi are values of type $type_i$ for $1<=i<=n$.
To extract the ith element of a product the following operation is used: #id_i *tuple*
In ML, each component in the record color set may be a different type and each is identified by a unique label so that each field is position-independent.

4.1.5 CollectionType

It describes a list of elements of a particular given type. It is a concrete metaclass whose instances are the subclasses SetType, OrderedSetType, SequenceType, and BagType.

(v) BagType is a collection type that describes a multiset of elements where each element may occur multiple times in the bag. The elements are unordered.

(vi) SequenceType is a collection type that describes a list of elements where each element may occur multiple times in the sequence. The elements are ordered by their position in the sequence.

(vii) SetType is a collection type that describes a set of elements where each distinct element occurs only once in the set. The elements are not ordered.

(viii) OrderedSetType is a collection type that describes a set of elements where each distinct element occurs only once in the set. The elements are ordered by their position in the sequence.

The OCL CollectionType declaration *A: Collection(Type)* is expressed in ML by:
colset collection= list Type;
var A : collection;

In ML, the values of a list color set are a sequence whose color set must be the same type. Values of this color set have form *[v1, v2, ..., vn]* where vi has type *Type* for *i=1..n*.

The four kind of collection in OCL have the same declaration in ML as a list color set but the difference is shown in the treatment of their operations. Table 5 shows the mapping of standard operations while table 6 presents the mapping of iteration operations and table 7 describes the mapping of the collection operations.

OCL	ML	Description
Collection (T)	Colset Collect = List T Var C : Collect	Collection type declaration
C->size()	length C	Number of elements in the collection;
C->sum():Integer	foldr (fn (x,y) => x+y) 0 C	Sum of elements in the collection. Elements must be numbers or have a + operation defined
C->count(e)	foldr (fn (x,y) => if x=e then y+1 else y) 0 C	The number of times that e is in c.
	cf e C	
C->excludes(e): Boolean	not (mem C e)	C exclude the element e
C->includes(e): Boollean	mem C e	C include the element e
C1->excludesAll(C2)	((intersect C1 C2) = nil)	no element of C2 is in C1
C1->includesAll(C2)	contains_all C1 C2	All elements of C2 are in C1
C->isEmpty: Boolean	C=nil	Same as (c->size = 0)
C->notEmpty: Boolean	C<>nil	Same as (not c->isEmpty)

Table 5. Mapping of standard operations

OCL	ML	Description
C->select(x|p(x))	filter p C	The collection of those elements in c for which p is true.
C->reject(x|p(x))	filter (not p) C	The collection of those elements in c for which p is false.
C->any(x|p(x))	random (filter p C)	Returns any element for which p is true where C->notEmpty() is true
C->exist(x|p(x))	List.exists p C	returns true if p is true for some element in C
C->forAll(x|p(x))	(filter p C) = C	returns true if p is true for all element in C
C->isUnique(x|f(x))	isUnique f C fun isUnique _ [] = false | isUnique _ [x]=true | isUnique f [x,y] = f(x)=f(y) | isUnique f (x::xs) = (f(x)=f(hd(xs))) andalso (isUnique f xs);	Does f has unique value for all elements of C
C->sortedBy(x|f(x))	sort f C	Returns a collection containing all elements ordered by f
C->collect(x|f(x))	map f C	Returns a collection containing the result of applying f on all elements of C
C->iterate(x, r=v| f(x,r))	foldr f v C	returns f(e_1, f(e_2, …,f(e_n, v) …)) where C = [e_1, e_2,…, e_n]

Table 6. Mapping of iteration operations

OCL	ML	Description
C->asSet: Set	remdupl C	A set corresponding to the collection (duplicates are dropped).
C->asSequence: Sequence	sort T.lt C	A sequence corresponding to the collection. More useful: c->sortedBy (Comparator(T)).
C->asBag: Bag	C	A bag corresponding to the collection.
C->asOrdredSet: OrdredSet	sort T.lt (remdupl C)	An OrdredSet corresponding to the collection (duplicates are dropped, ordred).
C1=C2: Boolean	C1 = C2	Equality between two OrdredSets or two Sequence s
	contains_all C1 C2 andalso contains_all C2 C1	Equality between two Sets or two Bags

C1=C2: Boolean		C1 <> C2	Inequality between two OrdredSets or two Sequence s
C1=C2: Boolean		not (contains_all C1 C2 andalso contains_all C2 C1)	Inequality between two Sets or two Bags
C1 – C2		listsub C1 C2	Subtraction of two collections where C2->includesAll(C2) is true
C->excluding(e)		rmall e C	Remove all appearances of e from C
C->including(e)		ins C e	Add the element e at the end of C (C+e)
C1->intersection(C2)		intersect C1 C2	returns the intersection of C1 and C2
C1->union(C2)		union C1 C2	Union of C1 and C2 (C1+C2) (for BagType and SequenceType)
C1->union(C2)		remdupl (union C1 C2)	Union for SetType and OrdredType (no multiplicity)
Sequence and OrdredSet	C->at(i)	List.nth(C, i-1)	return the ith element of a collection where i>0
Sequence and OrdredSet	C->last()	List.nth(C, (length C)-1)	return the last element of a list where C->size()>0
Sequence and OrdredSet	C->first()	hd C	return the first element of a list where C->notEmpty() is true
Sequence and OrdredSet	C->append(e)	C^^[e]	The colection obtained by appending e to C
Sequence and OrdredSet	C->prepend(e)	e::C	The colection obtained by prepending e to C
C->subSequence (i, j)		List.take(List.drop(C, i-1), j-i+1);	The sequence from position i to j
C->subOrderedSet(i,j)		List.take(List.drop(C, i-1), j-i+1);	The ordredset from position i to j
C1->symmetric Difference(C2)		listsub (remdupl (union C1 C2)) (intersect C1 C2)	The set containing all the elements that are in C1 or in C2 but not in both
C->reverse()		rev C	Reverse the collection C
C->insertAt(i,e)		List.take(C,i-1)^^[e]^^ List.drop(C,i-1)	The collection consisting of C with element e inserted at position I (for OrdredSet and Sequence)
C->indexOf(e)		indexOf e C = if List.nth(C,0) = e then 0 else indexOf e (lt C) +1	Index of element e in the OrderedSet C

Table 7. Mapping of collection operations

4.1.6 Classes and objects

OCL expressions can refer to classes, attributes, assocaionEnds and operations of the class diagram. The class type *class_name (att$_1$:type$_1$, ….,att$_n$:type$_n$)* is declared in ML as an record type :
Colset class_type = record att$_1$:type$_1$ …. *att$_n$:type$_n$*
var class_name : class_type;
The attribute value *class_name.att$_i$* is expressed in ML by *#att$_i$ class_name;.*

4.2 Mapping of OCL Constraints to ASKCTL

OCL constraints consist of an OCL expression of type Boolean and some declaration connecting the OCL expression to an item in the class diagram. In the case of pre and post-conditions, the constraint is bound to an operation; invariants are bound to a class.
An OCL invariant is an OCL expression associated with a class. It must be true for all instances of that class type at any time. Its structure is:

context ClassType inv: ExpOcl.

An invariant is translated by the ASKCTL formula:

INV(NF("",ML(ExpOcl)))

where:

- ML() is a mapping function that gets an equivalent expression in ML code.
- NF() is the node function used as a state subformula. Its arguments are a string and a ML function which takes a state space node and returns a Boolean.
- INV(A) is a state formula witch is true if the argument A is true for all reachable states from the current state.

Figure 1 shows the verification of an OCL invariant on the CP-net state space where the ML Boolean expression (MLExp=ML(ExpOCL)) must hold in all reachable nodes.

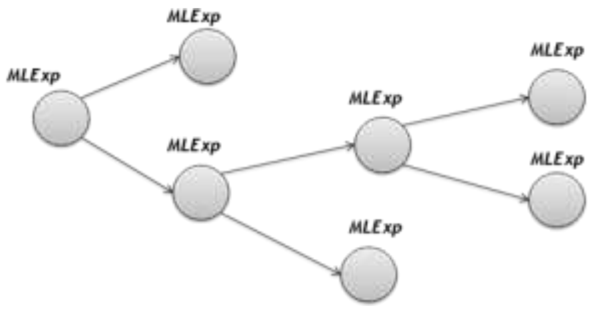

Figure 1. Mapping of OCL invariant

A pre/post condition OCL is associated with an operation of a class. The pre condition must be true before the operation call and the post condition must be true after the operation execution. Its structure is:

Context ClassType::Operation(Parameters:Types): ReturnType
Pre: ExpOcl1
Post: ExpOcl2

A pre/post condition is translated by an ASKCTL formula:

INV(AND(OR(NOT(NF("",Fir(t1))),NF("", ML(ExpOcl1))),OR(OR(NOT(NF("",Fir(t1))), NOT(NF("",ML(ExpOcl1)))), FORALL_NEXT(AND(NF("",Fir(t2)), FORALL_NEXT(NF("",ML(ExpOcl2))))))));

where:

- t1 and t2 are the derived transitions from the sending and receiving events of the message "operation call".
- Fir(t) indicates the firing of a transition t.
- FORALL_NEXT(A): used as a state formula, looks at immediate successors, is true if the argument, A, is hold for all immediate successors.

Figure 2 shows the verification of an OCL pre/post-condition on the CP-net state space where the ML Boolean pre-expression (MLExpPre = ML(ExpOCL1)) must be true just prior the operation execution (state i) and the ML Boolean post-expression (MLExpPost = ML(ExpOCL2)) must be true just after the operation execution (state i+2).

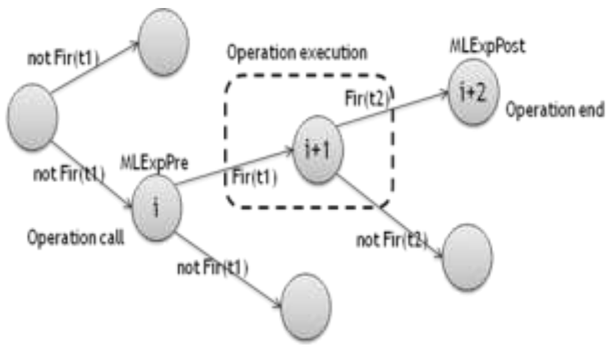

Figure 2. Mapping of OCL pre/post condition

5. CASE STUDY

Our approach of mapping and analysis is applied to an ATM system. To start the application, the client inserts his card in the dispenser. He then enters his personal identification number (PIN). In the absence of error, he chooses to withdraw money or view his balance. Otherwise, he starts again the identification phase. To withdraw money, he introduces the amount and recovers his money if the balance is sufficient. For account inquiry, only the balance is displayed. In all cases, the client gets his card at the operation end.

Thus, the static view of the ATM system is modeled by a class diagram (see figure 3), and an object diagram, see figure 4. The object diagram is used to initialize the model for a possible execution.

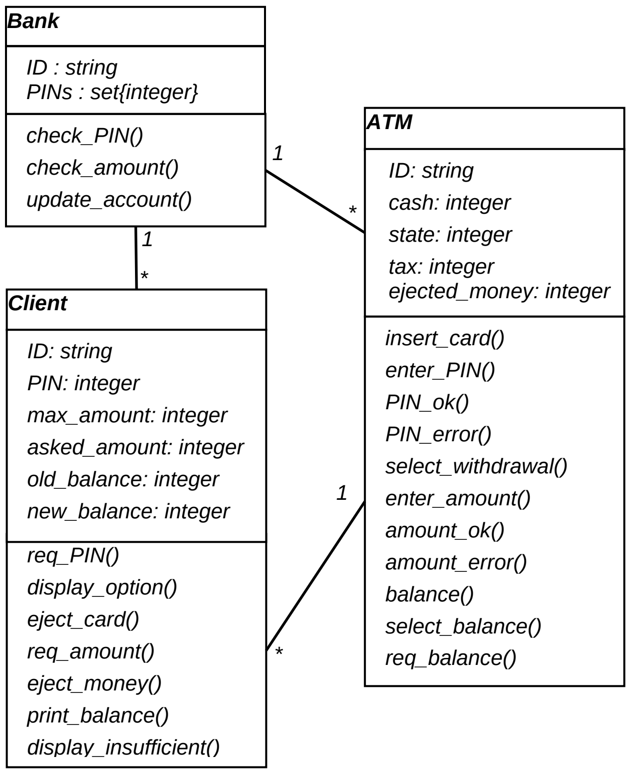

Figure 3. ATM Class Diagram

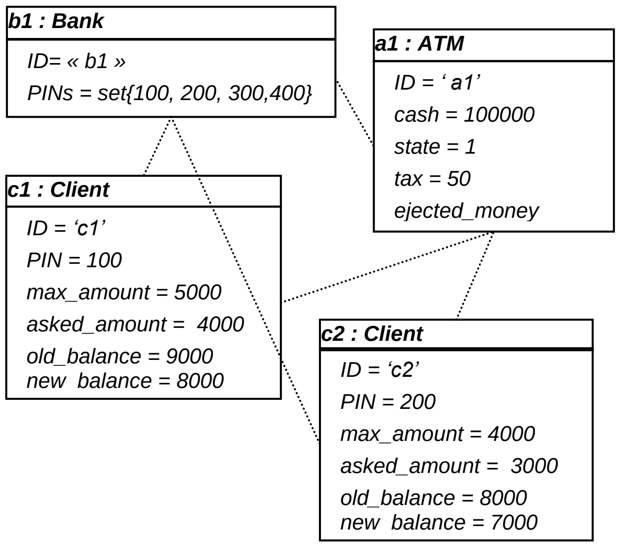

Figure 4. ATM Object Diagram

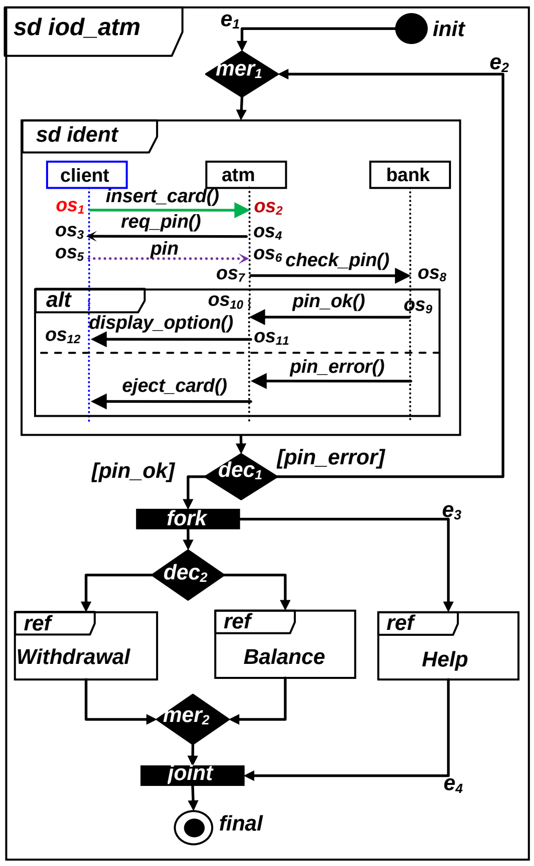

Figure 5. ATM IOD diagram

To illustrate the behaviour view of the ATM system, we present in figure 5 the Interaction Overview Diagram (IOD) of the ATM. The ATM IOD consists of three sequence diagrams: client identification, balance and withdrawal transaction; each of which models a part of the system interactions.

We limit ourselves to only show the identification SD. We use a TranslatorTool that implements the mapping rules developed in [3] for automatically generating a CP-net model from the ATM IOD in accordance with the ATM Class Diagram.

The obtained CP-net model (see figure 4) is initialized by the multi-set of tokens derived from the ATM Object Diagram, see figure 4. The initial multi-set of tokens is given as follows:

{("client", "c1", (100,5000,4000,9000,8000)), ("client", "c2", (200,4000,3000,8000,9000)), ("atm", "a1", [100000,1,50,0]), ("bank", "b1", (100,200,300,400))}

The resulting CP-net model is executed in CPNtools for simulation, state space analysis and system properties verification.

Using the simulation tool, we can examine different scenarios and explore the behaviour of the system. Simulation provides a partial validation of the model. It is often used to debug its dynamics. The simulation of a HCPN can be either interactive or automatic with graphical feedback showing visually the tokens movement, enabled transitions and places marking. The simulation feedback can be interpreted by a helpful sequence diagram for user facilities and errors detection.

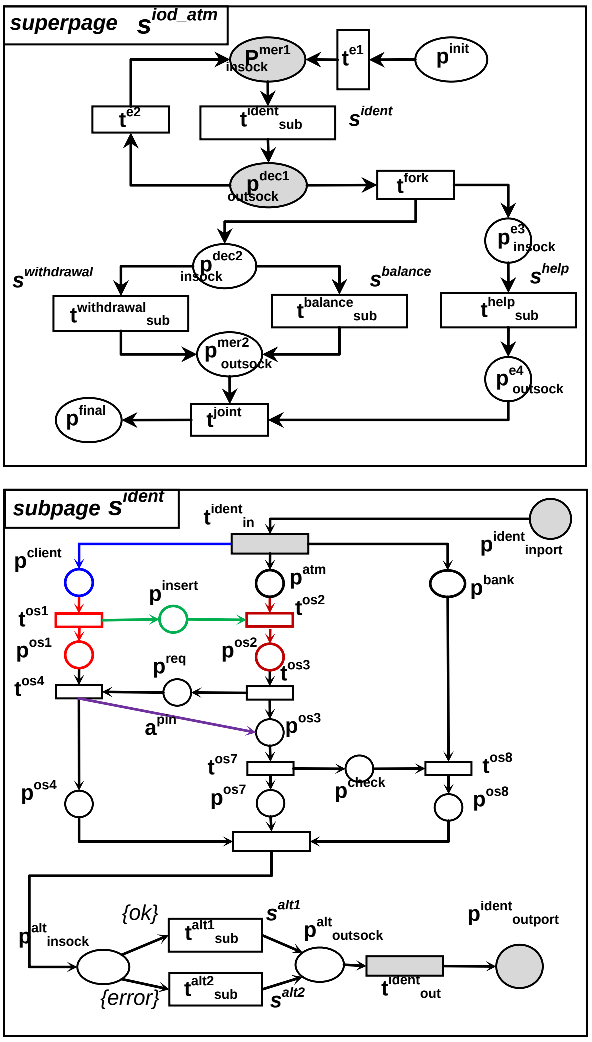

Figure 6. ATM CP-net model

As for the state space analysis, it is one of the main formal analysis methods of Petri Net. It has proven successful in the verification of systems. Once the state space is generated for the resulting CP-net model, we obtain a text file which contains a standard report providing information about generic properties such as state space statistics, boundedness properties, home properties and liveness properties.

ML standard queries available in CPNtools may also be evaluated. In the case of negative answers, the user is helped to investigate why an expected property does not hold. If an unexpected dead state is found a shortest path from the initial state to the dead state is helpful information, as a counterexample. This situation may be interpreted to UML user with both a sequence diagram describing the error trace (events sequence), and an object diagram describing the dead marking (object values).

However, as UML users are not necessary familiar with input languages of CPNtools (CP-nets, ASKCTL and CPN-ML). The specification of system properties, to check the model consistency with the expected properties of the real system, will be difficult for users to understand. So, we allow UML user to express system properties in OCL language, as invariants and pre/post conditions, on the class diagram, then we automatically map these constraints into ASKCTL formulas based on CPN-ML functions. Finally, we check OCL properties on CP-net state space trough ASKCTL formulas. Positive responses are shown to UML user and negative responses are interpreted by a counterexample through a sequence diagram and an object diagram.

We express in what follows four OCL properties checked over the state space of the resulting CP-net model in CPNtools environment.

Property 1: ATM machine does not eject money if the client asks for an amount higher than its balance.

OCL invariant:
Context c:Client
inv: (c.asked_amount > c.balance) implies (c.ATM.eject_money = 0)

ASKCTL formula (without detail for invariant condition):

```
use (ogpath^"/ASKCTL/ASKCTLloader.sml");
val CTLFormula1 = INV( NF("",MLInv));
eval_node   CTLFormula1   1
```

where INV() is a state formula which is true if its argument is true for all reachable states. *Eval_node()* is a function that allows to evaluate a state formula from a specified state node (*initial state node = 1*). It returns true or false, and in the case of false, it also prints out a diagnostic report. Thus, the first code line allows loading the ASKCTL library. The ASKCTL library has two parts: one which implements the language of the logic, and one which implements the model checker [6]. *MLInv* is a ML function which allows verifying the OCL invariant condition.

Property 2: after an "insufficient balance" message is returned by the machine, the client balance must be decreased by the tax value.

OCL pre/post-condition:
Context Client::insufficient()
let c:Client
POST: (a.new_balance = a.old_balance - a.ATM.cach)

ASKCTL formula (without detail for post-condition):

```
use (ogpath^"/ASKCTL/ASKCTLloader.sml");
Val  CTLformula2= INV(or(not NF("", fire(t1)),
FORALL_NEXT( and( NF( "",  fire(t2)),
FORALL_NEXT( NF("", MLpost))))));
eval_node  CTLformula2   1
```

where FORALL_NEXT() is used as a state formula. It is true if its argument is true for all immediate state successors. *t1* and *t2* are derived transitions from the sending and receiving events of the call operation message. *Fire(t)* indicates that the transition *t* is enabled. *MLpost* is a ML function which allows verifying the OCL post-condition.

Propriety 3: The machine does not eject money if the requested sum is greater than the cash machine or greater than the maximum or exceeds the client's balance amount.

OCL invariant:
Context c: Client
INV: (c.asked_amount+c.ATM.tax > c.balance) or (c.asked_amount+ c.ATM.tax > c.max_amount) or (c.asked_amount+a.ATM.tax > c.ATM.cash) implies c.ATM.eject_money=0

ASKCTL formula 3:

```
use (ogpath^"/ASKCTL/ASKCTLloader.sml");
val CTLFormula3 = INV(NF("",MLInv)) ;
eval_node   CTLFormula3   1;
fun MLInv3   n =
if (Mark.SubPageAmountError'P11  1  n) <> empty
then CheckEjctedMoney n
else true
fun CheckEjectedMoney  n =
let
val atm= List.nth(Mark.SubPageAmountError 'P11
1  n,0) :TOBJ;
val class=(#1 atm): STRING;
val ID=(#2 atm) : STRING;
val list=(#3 atm): INTlist;
in
```

```
(class="atm") andalso (ID="a1") andalso
(List.nth(list,3)=0)
end
```

Property 4: The machine rejects the user PIN if it does not appear in the bank data. The machine is thus in a state of reject.

OCL pre/post condition:
Context Bank :: pin_error()
Let b :bank
PRE : b.PINs→excludes(b.client.PIN)
POST : atm.state =0

ASKCTL formula 4:

```
use (ogpath^"/ASKCTL/ASKCTLloader.sml");
val CTLFormula4 = INV( AND( OR( NOT( NF("",
firt1)),
NF("",MLpre)),OR(OR(NOT(NF("",firt1)),NOT(NF(""
, MLpre))), EXIST_NEXT( AND( NF( "", firt2),
EXIST_NEXT( NF( "", MLpost)))))));
eval_node    CTLFormula4  1
fun firt1 n = ((Mark.SubPagePinError'P20  1  n) <>
empty)
fun firt2 n = ((Mark.SubPagePinError'P_msg1 1  n)
<> empty)andalso((Mark.SubPagePinError'P10  1
n) <> empty)
fun MLpre n =
let
val bank= List.nth(Mark.SubPagePinError'P20  1
n,0) :TOBJ;
val list2=(#3 bank): INTlist;
val client= List.nth(Mark.SubPagePinError'P00  1
n,0) :TOBJ;
val list0=(#3 client): INTlist;
in    (not(checkpin list0 list2)) end
fun MLpost n =
let  val atm= List.nth(Mark.SubPagePinError'P11 1
n,0) :TOBJ;
val list1=(#3 atm): INTlist; in (List.nth(list1,1)=0)
end
```

where MLpre is a ML function which allows verifying the OCL pre-condition.

6. RELATED WORKS

OCL has been formalized by various formal languages such as B, Z, CSP, PVS, mu-calculs and temporal logic. Many temporal extensions of OCL exist. Ziemann and Gogolla aim in [23] to expand the semantics of the language with a LTL-based extension. Bill et al. present in [2] an OCL extension with CTL-based temporal operators. Kanso and Taha propose in [12] a pattern-based extension of the OCL language to express temporal constraints on object-oriented systems. Distefano et al. provide in [7] a formal semantics to OCL by using OBTL (Object-Based Temporal Logic), which facilitates the specification of dynamic and static properties of object-based systems. They do not expand OCL with temporal operators, but provide a theoretical precise mapping of a part of OCL into OBTL. Cengarle and Knapp propose in [4] an extension of OCL, called OCL/RT, for modeling real-time and reactive systems. OCL/RT introduces a general notion of time and event to describe the temporal behavior of UML models. Mullins and Oarga provide in [15] an OCL extension, called EOCL, with CTL temporal operators. This extension is strongly inspired by BOTL, and allows model checking EOCL properties on UML models expressed as abstract state machines.

Theoretically, our approach of OCL formalization can be compared to [7] when translating invariants and pre/post conditions to a variant of CTL logic. But, practically, our work uses a specific logic strongly based on a functional programming language SML in CP-nets context. This allows detailed mapping of basic and complex types and operations of OCL language. Our approach is implemented and integrated in a validation framework of UML models by using CPNtools environment.

7. CONCLUSION

To help and assist UML modelers verifying their specification, we proposed to automatically translate OCL properties, specified on the class diagram, to CTL-like logic based on SML. We also present in details the translation of basic and complex expressions of OCL by exploiting the expressiveness of the functional programming language CPN-ML. We relied on the class diagram for the static view of the system and the IOD diagram for the behaviour view of the system. The CP-net model derived from the UML description is analysed by model-checking based on OCL constraints derived to ASKCTL logic. To the best of our knowledge, it is the first work that uses Standard ML to formulate OCL expressions in a CP-net context. The resulting formulas are succinct and of reduced execution time as ASKCTL logic is based on the functional and recursive aspect of ML as well as the Strongly Connected Component graph (SCC). ASKCTL formulas have been evaluated over the generated state space of CP-net model within CPNtools environment. In case of negative answers, we propose to help the user investigating why an expected property does not hold. For this purpose, a sequence diagram is returned to the user relating the property error trace.

For future works, we plan to improve our implementation with regard to efficiency and usability. We also plan to integrate the proposed approach of mapping in a CASE tool (Computer-Aided Software Engineering) of UML2 in order to generalize its application to other dynamic diagrams.

8. REFERENCES

1. Alhroob, A., Dahal, K., & Hossain, A. (2010, October). Transforming UML sequence diagram to high level Petri Net. In Software Technology and Engineering (ICSTE), 2010 2nd International Conference on (Vol. 1, pp. V1-260). IEEE.
2. Bill, R., Gabmeyer, S., Kaufmann, P., & Seidl, M. (2013). OCL meets CTL: Towards CTL-Extended OCL Model Checking. In Proceedings of the MODELS 2013 OCL Workshop} (Vol. 1092, pp. 13-22).
3. Bennama, M., & Bouabana–Tebibel, T. (2013). Validation environment of UML2 IOD based on hierarchical coloured Petri nets. International Journal of Computer Applications in Technology, 47(2), 227-240.
4. Cengarle, M. V., & Knapp, A. (2002). Towards ocl/rt. In FME 2002: Formal Methods—Getting IT Right (pp. 390-409). Springer Berlin Heidelberg.
5. Cheng, A., Christensen, S., & Mortensen, K. H. (1997). Model checking Coloured Petri Nets-exploiting strongly connected components. DAIMI Report Series, 26(519).
6. Christensen, S., & Mortensen, H.K. (1996) ‘Design/CPN ASKCTL Manual Version 0.9’, University of Aarhus.
7. Distefano, D., Katoen, J. P., & Rensink, A. (2000). On a temporal logic for object-based systems (pp. 305-325). Springer US.
8. Edmund M. Clarke, E. A. Emerson, and A. P. Sistla, “Automatic Verification of Finite State Concurrent System Using Temporal Logic”, ACM Transactions on Programming Languages and Systems, vol. 8(2), 1986, pp. 244-263.
9. Fernandes, J. M., Tjell, S., Baek Jorgensen, J., & Ribeiro, Ó. (2007, May). Designing tool support for translating use cases and UML 2.0 sequence diagrams into a coloured Petri net. SCESM'07: ICSE Workshops 2007. Sixth International Workshop on (pp. 2-2). IEEE.
10. Jensen, K. (1998) An Introduction to the Practical Use of Coloured Petri Nets. Lectures on Petri Nets II: Applications, Lecture Notes in Computer Science, 1492, 237-292, 1998.
11. Jensen, K., & Kristensen, L. M. (2009). Coloured Petri nets: modelling and validation of concurrent systems. Springer.
12. Kanso, B., & Taha, S. (2013). Temporal Constraint Support for OCL. In Software Language Engineering (pp. 83-103). Springer Berlin Heidelberg.
13. Mandel, L., & Cengarle, M. V. (1999). On the expressive power of the Object Constraint Language OCL. Available on the World Wide Web: http://www. fast. de/projeckte/forsoft/ocl.
14. Milner, R. (Ed.). (1997). The definition of standard ML: revised. The MIT press.
15. Mullins, J., & Oarga, R. (2007). Model checking of extended OCL constraints on UML models in SOCLe. In Formal Methods for Open Object-Based Distributed Systems (pp. 59-75). Springer Berlin Heidelberg.
16. OMG, Object Constraint Language 2.3.1, Doc Number: formal/2012-01-01, 2012.
17. OMG, UML Superstructure Specification 2.4.1, Doc Number: formal/2011-08-06, 2011.
18. Petri, C. A. (1962). Kommunikation mit Automaten. Bonn: Institut f¨ur Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.
19. Ratzer, A. V., Wells, L., Lassen, H. M., Laursen, M., Qvortrup, J. F., Stissing, M. S., ... & Jensen, K. (2003). CPN tools for editing, simulating, and analysing coloured Petri nets. In Applications and Theory of Petri Nets 2003 (pp. 450-462). Springer Berlin Heidelberg.
20. Staines, T. S. (2008, March). Intuitive mapping of UML 2 activity diagrams into fundamental modeling concept Petri net diagrams and colored Petri nets. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the (pp. 191-200). IEEE.
21. Ullman, J. D. (1998). Elements of ML programming.
22. Zaidi, A. K., & Levis, A. H. (2006). Verification of System Architectures Using Modal Logics and Formal Model Checking Techniques. In Conference on Systems Engineering Research (CSER).
23. Ziemann, P., & Gogolla, M. (2003, January). Ocl extended with temporal logic. In Perspectives of System Informatics (pp. 351-357). Springer Berlin Heidelberg.