An Accessible Formal Specification of the UML and OCL Meta-Model in Isabelle/HOL

Tamleek Ali, Mohammad Nauman, Masoom Alam
International Islamic University, Islamabad, Pakistan
City University of Science and Information Technology, Peshawar, Pakistan
Institute of Management Sciences, Peshawar, Pakistan
tamleek@iuiu.edu.pk, recluze@gmail.com, masoom.alam@gmail.com

Abstract—UML is the de-facto standard for system modeling. Due to its visual syntax and expressiveness, it is widely accepted and used in the industry. However, it is a semi-formal means of system specification and thus prone to inconsistencies. We believe that UML models need to be thoroughly verified because verification of UML models helps to find errors in the early stages of the system design. Object Constraint Language (OCL) somewhat alleviates this problem but is not always enough. Past attempts at formally specifying UML for verification include those based on simplistic Z specifications and the much more complex ones based on shallow embedding of UML and OCL in Higher Order Logic (HOL). All these approaches are either too simplistic or too complex for the software industry’s purposes. In this paper, we formalize UML’s class diagram and OCL constraints in the highly successful automated/interactive theorem prover Isabelle using one of its built-in logics, HOL. The aim is to create a formalization, which is accessible to the average software engineer while still being powerful enough to be able to prove consistency and other useful properties. The formalization – based on UML2.0 and OCL2.0, addresses all concepts related to class diagrams such as type definitions, attributes, operations, aggregation and association along with the syntax and semantics of OCL expressions in the context of UML class diagrams.

Index Terms—Algorithms, Tools & Applications, Formal Methods, System Design, Automated Theorem Prover

I. INTRODUCTION

The Unified Modeling Language (UML) [1] is a visual language highly accepted and used in the industry. It is domain independent and thus applicable in a wide variety of domains [2]. The basic diagram in UML is the Class Diagram, which is composed of classes, attributes, operations, parameters and different types. Figure 1a shows a simple class diagram.

UML models are easy to use but they lack in specification of information which cannot be depicted visually. The Object Constraint Language (OCL) [3] was developed to alleviate this problem [4]. OCL is a powerful language which can be used not only to specify complex constraints on UML models but also as a query language [2]. Figure 1b shows a collection of constraints written in OCL.

While OCL manages to bring some formality to the UML models, it is only a specification language and requires some sort of verification technique for proving properties. Verification of certain properties such as correctness, security, and well-formedness are especially important in systems of critical nature. This importance of formal analysis is generally well accepted for enhancing correctness and quality of software [5]. The problem is the time and effort required to learn and apply these techniques.

Some common verification methods are the usual testing techniques, model checking and formal proof construction. In this paper, we use the latter by embedding the UML Class Diagram and the OCL expressions in the interactive and automated theorem prover Isabelle using its built-in Higher Order Logic (HOL). HOL itself is built on typed lambda calculus and is extremely useful for embedding languages. The novelty of our specification is the ease of use. Since the majority of software practitioners are unwilling to invest time and efforts in learning the complexities of mathematical techniques such as those used in earlier formalization, we believe that by using simpler techniques, the formalization can be made more accessible to a wider audience.

The meta models of UML and OCL are detailed specifications of these two languages. By modeling the meta models, formal proofs can be developed which can be reused for any system specification described through UML and OCL. This gives a high reusability value to the efforts put in the formal specification and verification of the meta model.

The rest of the paper is organized as follows: First, we give a brief introduction to past attempts at formalizing the UML and OCL. We describe the Isabelle theorem prover in brief giving a short introduction to the constructs used in this formalization. In Section III, we formalize the UML Class Diagram along with all the required components. OCL Expressions are specified in Section IV. We give examples of usage in parallel to the specification of constructs. Finally, we describe the open issues in this formalization and the future directions for this formalization.

II. BACKGROUND WORK

A. Past Formalizations of UML/OCL

Several attempts have been made in the past at formalizing the UML meta model. These have been, for example, formal specifications in Z notation [6], [7]. These proved to be only specifications with no way to automatically verify

1The complete Isabelle theory source for our formalization is available from: http://recluze.wordpress.com/publications/
theorems automatically thus saving time and efforts. Isabelle's simplifier based on term rewriting, can be used to prove simple results. For more complex proofs, Isabelle provides theorem proving tools that are powerful and complex. These tools are designed to ensure that each step, and thus the whole proof, is logically correct. This gives a very high level of credibility to the proofs created using Isabelle.

Another, more detailed formalization of UML and OCL has been done by Brucker et al. It is based on shallow embedding of the OCL in Isabelle and is called HOL-OCL [9]. This technique has proved to be extremely powerful for verification of UML and OCL specifications. However, it requires the use of very complex operations and techniques which are difficult even for the experienced software specification engineers. We believe that this is a work in the right direction but a generic theorem prover. This means that it is possible to encode new logics in the meta-logic of Isabelle which is very helpful since one doesn’t have to learn different tools for new logics. It also allows for the checking of the consistency of logics themselves.

Model checking is another technique, which has been used extensively in the past for the purpose of verification. Construction of formal proofs, however, is preferable over model checking because formal proofs do not require limiting the number of executions or other assumptions such as finiteness. They can also be used to prove certain properties, which can not be proved through model checking [13].

A complete discussion of Isabelle theory and syntax is well outside the scope of a single paper. However, in the next section, we give a brief account of the syntax of constructs used in the formalization of UML and OCL meta-model. Please refer to the appropriate section of the Isabelle/HOL tutorial [10] for detailed discussion of these constructs.

C. Basic Isabelle Syntax

All formalizations in Isabelle are done in the context of a theory. A theory is composed of datatypes, constants, definitions, functions, lemmas and theorems etc. A formalization normally begins with the definition of datatypes required by the theory. It is important that its data types be defined most accurately. Isabelle has several constructs for defining types, each with its pros and cons. Following is a brief discussion regarding these constructs. See [10] Chapter 8 for more details about these concepts.

typedel: is the most basic way of defining types. It only defines a type name without any details of the type. This is used for types which are parameters to the theory and are not defined by the theory itself.

typedef: defines a non-empty subset of an existing type. This is the basic construct and most other type definition constructs are built on top of it. The only requirement for a datatype defined using typedef is that it must have at least one element.

records: are not defined as primitive constructs of Isabelle but are built on top of other primitives. They are essentially represented as n-tuples but the different elements are assigned a unique name thus making access easier. The syntax for defining records is this:

\[
\text{record \hspace{1em} RecordName =} \]

B. Isabelle Theorem Prover

Isabelle [10] is an interactive/automated theorem prover extremely popular in the mathematics and research communities [11]–[13]. It is interactive in that the user specifies a theorem and tries proving the theorem in steps. Isabelle ensures that each step, and thus the whole proof, is logically correct [14]. This gives a very high level of credibility to the proofs created using Isabelle.

Model checking is another technique, which has been used extensively in the past for the purpose of verification. Construction of formal proofs, however, is preferable over model checking because formal proofs do not require limiting the number of executions or other assumptions such as finiteness. They can also be used to prove certain properties, which can not be proved through model checking [13].

A complete discussion of Isabelle theory and syntax is well outside the scope of a single paper. However, in the next section, we give a brief account of the syntax of constructs used in the formalization of UML and OCL meta-model. Please refer to the appropriate section of the Isabelle/HOL tutorial [10] for detailed discussion of these constructs.

C. Basic Isabelle Syntax

All formalizations in Isabelle are done in the context of a theory. A theory is composed of datatypes, constants, definitions, functions, lemmas and theorems etc. A formalization normally begins with the definition of datatypes required by the theory. It is important that its data types be defined most accurately. Isabelle has several constructs for defining types, each with its pros and cons. Following is a brief discussion regarding these constructs. See [10] Chapter 8 for more details about these concepts.

typedel: is the most basic way of defining types. It only defines a type name without any details of the type. This is used for types which are parameters to the theory and are not defined by the theory itself.

typedef: defines a non-empty subset of an existing type. This is the basic construct and most other type definition constructs are built on top of it. The only requirement for a datatype defined using typedef is that it must have at least one element.

records: are not defined as primitive constructs of Isabelle but are built on top of other primitives. They are essentially represented as n-tuples but the different elements are assigned a unique name thus making access easier. The syntax for defining records is this:

record RecordName =
To define new records as extensions to a previously defined record, we use the following syntax:

\[
\text{record } \text{NewRecordName} = \text{OldRecord} + \text{Name} :: \text{Type}\\
\]

**types**: simply defines a synonym for another type.

**datatype**: is the most powerful way of defining datatypes. It is safer since all definitions are checked for consistency and is still easy enough to define very complex types. It is also possible to define more than one inter-dependent types using datatype. These are called **mutually defined types**. In datatype definitions, certain function can be represented using symbols defined as infix operators. The syntax for doing this is \((\text{infix \ op \ priority})\). Several definitions in our formalization use this syntax. See below for examples.

Another important aspect of Isabelle’s syntax is the theorem proving methods and logical rules. When in **proof state**, we can use the following syntax to apply a certain theorem proving method using a rule:

\[
\text{apply(method rule)}\\
\]

For defining the datatype \text{Integer} in Section III-A, we have used the method \text{rule-tac} and the rule \text{exI}. \text{rule-tac} applies a logic rule in backward direction, instantiating a variable with a value, and the rule \text{exI} is defined internally in Isabelle as:

\[
exI: P\ x = \Rightarrow \exists x. P\ x\\
\]

This is the internal representation of Isabelle. In logic texts, this rule is normally written as:

\[
P\ x \vdash \exists x. P\ x\\
\]

In Isabelle, syntactic shortcuts can be defined using the keywords **syntax** and **translations**. These are, for the most part, self-explanatory.

Axioms can be defined in Isabelle using the keyword **axioms**. Each axiom is composed of a name and an expression. Defining axioms requires great care. Since axioms are not proved, a faulty axiom may lead to contradictions in the theory built on top of it.

Constants are declared using the keyword **consts** and can be given values using the keyword **defs**. Constants declaration and definition can be combined in a single command using the keyword **constdefs**.

### III. Formalizing UML Class Diagrams

#### A. Defining Basics of UML Class Diagram

This formalization of the class diagram is based on the meta-model of a class diagram described in the UML Infrastructure document [1]. Before we start with the formalization, we define a basic type called \text{String} which is currently represented by \text{nat}. This representation of \text{String} as \text{nat} is simply because we want to keep the specification simple. This does not lead to any deficiency in the formalization or verification of the meta-model. We base this technique on Foster and Vytiniotis’ formalization of *Featherweight Java* [15].

**types** \text{String} = \text{nat}\
**typedef** \text{Integer} = \{0 :: \text{nat}, 1\}

We define \text{Integer} to be a subset of \text{nat} containing only 0 and 1. When defining types with \text{typedef}, we have to prove that the type is non-empty. This is formulated by Isabelle in the form of a proof goal:

\[
\exists x. x \in \{0, 1\}\\
\]

We solve this goal and complete the proof by providing a witness.

\[
\text{apply(rule-tac } x = 0 \text{ in exI) by simp}\\
\]
We still have to define what 0 and 1 are called for type `Integer`. We can use the same symbols i.e. 0 and 1 for `Integer` type as we do for `nat` type. Please refer to the theory source file for explicit definition of this overloaded meaning of the constants 0 and 1.

Two other datatypes required by the UML specification are also defined.

```plaintext
types UnlimitedNatural = nat

The basic building block of a UML Class Diagram is the Element. It does not contain any attributes or operations but is used as the parent of all other parts of the class diagram. Since it does not contain any information, we choose to leave it out of our specification and define the elements that do contain some information. Also note that UML Specification requires the ability of multiple inheritance of types. This is not directly possible in Isabelle records. We use single inheritance/extension of records to emulate this feature.

First we define the basic element of interest to us – `Type`, which is an element that contains a name. It only has one attribute – `name`. It is used to define types of other elements – attributes, operations etc.

```plaintext
record Type =
  name :: String
```

When we define `Type`, Isabelle defines and proves several helpful theorems, for example:

```plaintext
thm Typedefs

make ?name ≡ (name = ?name)

fields ?name ≡ (name = ?name)

extend ?r ?more ≡ (name = name ?r, . . . = ?more)

truncat ?r ≡ (name = name ?r)
```

After this definition, it is possible to define new types. For example, if we were to define a new type we could do it using `constdefs`. Below, we define two new types: `int` and `cls` – type of integers and class:

```plaintext
constdefs
  int :: Type
  int ≡ (name = 0)

  cls :: Type
  cls ≡ (name = 1)
```

A `MultiplicityElement` is required by associations, properties and parameters. It too is defined using records. Again, several theorems are proved by Isabelle automatically but we exclude them from the text for the sake of brevity.

```plaintext
record MultiplicityElement =
  isOrdered :: Boolean
  isUnique :: Boolean
  lower :: Integer
  upper :: UnlimitedNatural
```

Similar to the definition of a basic type, we can define a multiplicity element as follows:

```plaintext
constdefs
  m₁ :: MultiplicityElement
  m₁ ≡ (isOrdered=False, isUnique=False, lower=0, upper=1)
```

B. Defining Properties and Operations

Properties are the attributes of a class. They may contain any type, be read-only and/or have a default value. The Official UML Specification defines properties using multiple inheritance from `TypedElement` and `MultiplicityElement`. Since Isabelle does not support extension of records from multiple bases, we derive properties only from `MultiplicityElement` and include the fields of `TypedElement` manually. The record `Property` is defined as an extension to the `MultiplicityElement` record type.

```plaintext
record Property = MultiplicityElement +
  name :: String
  proptype :: Type
  isReadOnly :: Boolean
  default :: String
  isComposite :: Boolean
  isDerived :: Boolean
```

We define a simple property `age` of type `int`. Since the property is extended from a multiplicity element, we use the `extend` and `fields` functions of both records to construct the property.

```plaintext
constdefs
  age :: Property
  age ≡ MultiplicityElement.extend m₁ (Property.fields 1 int False 0 False False)
```

Similarly, we may define `Operations` but before defining them, we need to define what `Parameters` are. A `Parameter` is also a `MultiplicityElement` but it is typed.

```plaintext
record Parameter = MultiplicityElement +
  name :: String
  paramtype :: Type
```

Now, we can define `Operations` using records. An operation requires an ordered collection of `Parameters`. Therefore, we use Isabelle’s `list` construct to define `ownedParameters` as a list of parameters.

```plaintext
record Operation = MultiplicityElement +
  name :: String
  rettype :: Type
  ownedParameter :: Parameter list
```

We define an operation `getAge` of no parameters as an example:

```plaintext
constdefs
  getAge :: Operation
  getAge ≡ MultiplicityElement.extend m₁ (Operation.fields 2 int Nil)
```

C. Defining Classes

Now that we have formalized the concepts of operations, parameters and properties, we can define the element `Class`. A `Class` extends `Type` and has several additional attributes.

```plaintext
record Class = Type+:
  isAbstract :: Boolean
  properties :: Property list
  operations :: Operation list
```

An example of a class with the operation and attribute defined as example earlier is shown below. Notice that
attributes and operations can be represented as comma separated elements enclosed in square brackets as is the usual notation of Isabelle.

\begin{verbatim}
consts subclass \quad \textbf{inheritance too: from which. Using relation, we can implement multiple}
This relation contains information about which class inherits
\end{verbatim}

\textbf{D. Formalizing Class Hierarchy}

Class hierarchy is maintained in a relation called \textit{subclass}. This relation contains information about which class inherits from which. Using relation, we can implement multiple inheritance too:

\begin{verbatim}
constdefs subclass :: \textbf{(Class \times Class)\set}
Working with a relation becomes tedious during reasoning. We need a shorthand for describing the class inheritance. We say that if a class \textit{x} is a sub-class of another class \textit{y} then \((x,y) \in \text{subclass} \text{ or } x \prec y\). We define this short-hand using a dummy function -\textit{subclass} along with a syntax translation. We also declare the transitive closure of the relation represented by the symbol \(-\prec^+\):

\begin{verbatim}
syntax
-\textit{subclass} :: \textbf{Class \Rightarrow Class \Rightarrow bool} \quad \textbf{(infix \prec 50)}
-\textit{subclassp} :: \textbf{Class \Rightarrow Class \Rightarrow bool} \quad \textbf{(infix \prec^+ 50)}
\end{verbatim}

\begin{verbatim}
translations
x \prec y \overset{\text{def}}{=} (x, y) \in \text{subclass}
\end{verbatim}

\begin{verbatim}
x \prec^+ y \overset{\text{def}}{=} (x, y) \in \text{subclass}^+
\end{verbatim}

axioms

\begin{verbatim}
no-circ: \forall x \in \text{Class}. \neg(x \prec^+ x)
\end{verbatim}

The axiom \textit{no-circ} declares that no class can be in the transitive closure of inheritance of itself – in other words, it makes the presence of circular class hierarchy impossible.

\textbf{IV. OCL Expressions}

The core of OCL are the OCL Expressions. They define a complete specification/query language and are thus quite complex. We model the OCL Expressions using Isabelle's \textbf{datatype} construct to ensure consistency in the specifications. Please refer to [3] for a detailed discussion regarding the syntax and semantics of OCL constructs.

First, we define some basic types required by OCL. Notice that we have used \textit{typedef} to define the \textit{state}. This means that we are not going to model the state and it will be considered as a parameter to our theory.

\begin{verbatim}
typedef state = String
\end{verbatim}

A \textit{Classifier} defined in the UML represents an OCL type. So, the following definition can be taken as being synonymous to OCL types:

\begin{verbatim}
datatype Classifier = OCLAny | OCLMessage String | OCLVoid
| OCLInvalid | OCLReal real | OCLInt nat | OCLString String
| OCLBoolean Boolean | OCLClass Class | OCLType Type
| OCLSet | OCLOrdSet | OCLBag | OCLSeq
\end{verbatim}

\textbf{example, we cannot model oclexpr without modeling ifexpr and vice versa. To tackle this problem, we use \textit{mutually defined types} of Isabelle. Each definition is separated using the keyword \textit{and}. Isabelle processes all definition together and proves consistency and other properties for the combined definition of all types.}

\begin{verbatim}
datatype oclexpr = IFExpr ifexpr
| VAR varname
| LETExpr letexpr
| MSGExpr String
| STATEExpr state
| LITEExpr Classifier
| ITERExpr iterexpr
| BOOLExpr boolean
| LOOPExpr loopexpr
| TYPEExpr Type
\end{verbatim}

\textbf{Fig. 3: Theorems proved by Isabelle for oclexpr}

\begin{verbatim}
| less oclexpr oclexpr (infix < 65)
| greater oclexpr oclexpr (infix > 65)
| leq oclexpr oclexpr (infix \leq 65)
| geq oclexpr oclexpr (infix \geq 65)
| eq oclexpr oclexpr (infix = 65)
| loopexpr = LOOP varname oclexpr list
\end{verbatim}

Isabelle proves many helpful theorems when we define a new type using \textbf{datatype}. See Figure 3 for the output generated by Isabelle for the definition of \textit{datatype oclexpr}. The most important of these from the point of view of verification are the induction rules. The induction rule for abstraction of \textit{ifexpr} from underlying set, for example, is:
As an example, we define a very simple OCL expression:
\[ cstage \geq 15 \]

We define a variable cstage (assigning it a \textit{nat} type) and then build an OCL expression using this variable:

\[
\begin{align*}
\text{constdefs} & \hspace{1em} \text{cstage :: nat} \\
& \hspace{1em} \text{cstage} \equiv 1 \\
\text{constdefs} & \hspace{1em} \text{oe}_1 :: \text{ocelexpr} \\
& \hspace{1em} \text{oe}_1 \equiv \hspace{1em} \hspace{1em} \\
& \text{BOOLExpr} \left( \left( \text{VAR} \hspace{1em} \hspace{1em} \hspace{1em} cstage \right) \geq \left( \text{LITExpr} \left( \text{OCLInt} \hspace{1em} 15 \right) \right) \right) \\
\end{align*}
\]

\textbf{V. FUTURE WORK}

This encoding of the UML and OCL meta model in Isabelle is the first step in development of a framework for verification of specifications based on these two technologies. Two future directions exist along this path:

Currently, the syntax of this formalization, while being much simpler than previous works, is not feasible for complex specifications. The power of UML and OCL is the ability to specify systems using visual environment. Several tools exist which can export these visual specifications to an XML based standard – XML Metadata Interchange (XMI) [16]. An important future direction is the development of a tool which can convert these XMI sources to the syntax of this formalization.

Another important direction is encoding a complete real-life system specification in this formalization and identifying and verifying properties which can be useful for the industry. Since the intended audience of this formalization is the industry, this identification would require a thorough survey of the industry’s needs and practices along this line.