Abstract—The ability of UML to handle the modeling process of complex industrial software applications has increased its popularity to the extent of becoming the de-facto language in serving the design purpose. Although, its rich graphical notation naturally oriented towards the object-oriented concept, facilitates the understandability, it hardly succeeds to report all domain-specific aspects in a satisfactory way. OCL, as the standard language for expressing additional constraints on UML models, has great potential to help improve expressiveness. Unfortunately, it suffers from a weak formalism due to its poor semantic resulting in many obstacles towards the build of tools support and thus its application in the industry field. For this reason, many researches were established to formalize OCL expressions using a more rigorous approach. Our contribution joins this work in a complementary way since it focuses specifically on OCL predefined properties which constitute an important part in the construction of OCL expressions. Using formal methods, we mainly succeed in expressing rigorously OCL predefined functions.

Keywords—Formal methods, Z, OCL, predefined properties, metamodel types.

I. INTRODUCTION

THE Object Constraint Language (OCL) [1], as part of the Unified Modeling Language (UML) [2] standard, continues to attract interest due to its ability to present model constraints in a friendly way making it easier for non-programmer to understand the underlying logic. Even though OCL is based on logic notation and mathematical set theory, it is not completely devoid of ambiguity since it uses English text descriptions, a context-free grammar and many examples to illustrate the meaning of expressions. In addition, OCL suffers from the absence of well-formedness rules resulting in weak semantics and significant obstacles towards the build of proper tools support. As a consequence, there is a need for clarity to increase its adoption by practitioners and then its application at the industry level.

To meet this need, many attempts were elaborated to strengthen OCL semantics by expressing it formally using different formal approaches. Up to now, the most relevant one remains the work of Mark Ritchers et al. [7] that was incorporated into the OCL standard specification [1]. Based on a set-theoretic mathematical approach, this contribution contains the concept of object models which provide information used as context for OCL expressions and constraints; it also defines the type system of OCL and the set of standard operations. Although this approach gives more rigors to OCL language, it does not include the formalization of OCL predefined properties and it has the inconvenient to require a strong mathematical background which is not necessary the background for many software engineering stakeholders. Therefore, many other contributions exist nowadays with the common purpose to strengthen OCL syntax and semantics. Among them, Flake et al. [8, 9] extended the formal semantics given in the OMG specification by supplying descriptions of ordered sets, global OCL variables definitions, UML statechart states and OCL messages. Gergly et al. [10] created a new formalism for OCL based on the Abstract State Machines technique called OCLASM. While Kyas et al. [11] translated the UML and OCL constraints into the language of the theorem prover PVS. Also, Brucker et al. [12] established transformations rules of OCL constraints into B formal expressions. Unfortunately, none of the existing approach includes the OCL predefined properties.

This paper proposes an approach to formalize OCL predefined properties as a complementary step to the formalization of OCL expressions in a rigorous way that will overcome OCL limitations and help building tools for the evaluation of OCL expressions. We, first, propose a formalization of the OCL metamodel for types upon which the formalization of predefined properties will be established. The approach used in formalization is based on the Laurent Henocque work [5] on formalizing UML class structures concept.

The rest of this paper is organized as follows: In Section 2, we will explain the formal approach adopted, leading to Section 3 where we expose our formal description of OCL predefined properties over the formal OCL types metamodel. Finally, in Section 4, we will draw conclusions and expose some remarks about future work.

II. FORMAL APPROACH

This section describes the formal approach followed to express OCL predefined properties in a non-ambiguous and formal language. The use of formal methods, when applied wisely, brings many advantages mostly the elimination of ambiguity and the ability to be checked mechanically, allowing provability which brings significant benefits in terms of understandability and reliability compared to the use of natural or semi-formal languages [13]. However, formal methods, due to their heavy background requirements, are difficult to deal with directly and so, the main idea remains the introduction of formal methods indirectly in practice, acting in background and completely transparent in foreground.

The following formal approach consists of two points: the use of Z as the formal language to express the predefined properties definitions and the adoption of a formal model for structures concept of the metamodel types on which the predefined properties are expressed.

Z [3, 4], named after Zermelo-Fraenkel set theory, is a formal specification language that uses mathematical notation and is based on axiomatic set theory, lambda
provide a brief description of its elementary notion:

\[
\text{ClassDefClassifier} \equiv \text{ClassDefClassifier} \land [\text{ObjectDef} \\
\text{class} = \text{Classifier}]
\]

In the current contribution, this approach is used to formalize OCL types metamodel class structures consisting in inheritance and relationships as shown in Section 3.

III. FORMALIZING OCL PREDEFINED PROPERTIES

Similarly to Z, OCL is a typed language since each OCL expression has a type. To be considered as correct, all types used in the expression must follow the rules of type conformance. According to [1], the definition of “conformance” in OCL is as follows:

“TypeA conforms to typeB if an instance of typeA can be substituted at each place where an instance of typeB is expected.”

Also, the rules of type conformance used in OCL expressions are:

- Each type conforms to each of its supertypes.
- Type conformance is transitive: if type1 conforms to type2, and type2 conforms to type3, then type1 conforms to type3.

Table I summarizes all the type conformance rules from the OCL Standard Library as illustrated in [1]:

<table>
<thead>
<tr>
<th>Type</th>
<th>Conforms to/Is a subtype of</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set(T1)</td>
<td>Collection(T2)</td>
<td>If T1 conforms to T2</td>
</tr>
<tr>
<td>Sequence(T1)</td>
<td>Collection(T2)</td>
<td>If T1 conforms to T2</td>
</tr>
<tr>
<td>Bag(T1)</td>
<td>Collection(T2)</td>
<td>If T1 conforms to T2</td>
</tr>
<tr>
<td>OrderedSet(T1)</td>
<td>Collection(T2)</td>
<td>If T1 conforms to T2</td>
</tr>
<tr>
<td>Integer</td>
<td>Real</td>
<td></td>
</tr>
<tr>
<td>InlinedNatural</td>
<td>Integer</td>
<td>*is an invalid Integer</td>
</tr>
</tbody>
</table>

The conformance of different types is determined by a type hierarchy represented in the OCL metamodel types. Subsequently, before starting the formalization of the OCL predefined properties, we first proceed to the formalization of OCL types metamodel as a basis for the next step.

A. OCL Types metamodel Formalization

The presence of a metamodel for OCL types is very beneficial to the notion of conformance since OCL is a typed language. Figure 1 shows the OCL types metamodel extracted from the OCL standard; it has the advantage to be fully integrated with the UML metamodel. We contribute in...
this section to its formalization according to the approach described earlier in the previous section.

The following formalization is based on the basic notion already presented in the previous Section. It was entirely type-checked and verified using Z/Eves tool [14].

The actual name of the class is reserved to denote the corresponding type, whereas, the class names are obtained by adding the String “Class” before the actual name of the class:

\[
\text{CLASSNAME} ::= \text{ClassClassifier} \mid \text{ClassVoidType} \mid \ldots
\]

Each class of the model will be implemented using both constructs: the class definitions and the class specifications:

\[
\text{ClassDefClassifier}
\]

\[
\text{name: seq CHAR}
\]

The inheritance relationship is simply represented by calling the class definition of the inherited class (e.g. Classifier) into the class definition of the inheriting class (e.g. Class) as shown in the following example:

\[
\text{ClassDefClass}
\]

\[
\text{ClassDefClassifier}
\]

Class specifications allow an extension of the class definitions with class and object references:

\[
\text{ClassSpecClassifier} \equiv \text{ClassDefClassifier} \land \{\text{ObjectDef} \mid \text{class} = \text{ClassClassifier}\}
\]

\[
\text{ClassSpecClass} \equiv \text{ClassDefClass} \land \{\text{ObjectDef} \mid \text{class} = \text{ClassClass}\}
\]

\[
\text{ClassSpecDataType} \equiv \text{ClassDefDataType} \land \{\text{ObjectDef} \mid \text{class} = \text{ClassDataType}\}
\]

The detailed representation of each class type is given through an axiomatic definition where the declaration part contains the type sets corresponding to all the classes in the OCL metamodel types while several axioms constitute the properties:

\[
\text{Classifier, Class, VoidType,...: ReferenceSet}
\]

\[
\text{Classifier} = \text{instances ClassClassifier} \cup \text{VoidType} \cup \text{Class} \cup \text{AnyType}
\]

\[
\cup \text{MessageType} \cup \text{InvalidType} \cup \text{DataType} \cup \text{TemplateParameterType}
\]

\[
\text{VoidType} = \text{instances ClassVoidType}
\]

\[
\text{Class} = \text{instances ClassClass}
\]

\[
\ldots
\]

Each type is the union of all the corresponding class instances, and of the types of its subclasses (if existing).

\[
\text{instances ClassClassifier} = \{o: \text{ClassSpecClassifier} \mid o.\text{class} = \text{ClassClassifier} \land o.i\}
\]

\[
\text{instances ClassClass} = \{o: \text{ClassSpecClass} \mid o.\text{class} = \text{ClassClass} \land o.i\}
\]

\[
\ldots
\]

The same object reference cannot be used for two distinct objects in the same class.

B. OCL Predefined Properties Formalization

In OCL, a number of predefined properties are available to be instantly used in OCL expressions. These properties are textually defined in the OCL standard specification [1] and are not included in any formalization efforts of the OCL language even though they constitute a significant part of many OCL expressions. In this paper, we propose a precise and formal definition of the most relevant predefined properties as a complementary work to many other formalization efforts where these predefined functions are not taken into consideration but rather considered as given.

For the sake of formalization, we define the Boolean type:

\[
\text{Boolean ::= TRUE} \mid \text{FALSE}
\]

The formalized definition of each predefined property is expressed in Z by an axiomatic function that returns a Boolean (except for oclAsType which returns an ObjectDef) and takes the ObjectDef and a ReferenceSet as parameter.
oclIsTypeOf: Evaluates to true if the type of self and the type t given in parameter are the same. It deals with the direct type of an object and has the following signature:

\[ \text{oclIsTypeOf}(o: \text{ObjectDef}; t: \text{Classifier}) : \text{Boolean} \]

In the current formalization, we substitute the type of t by a ReferenceSet instead of Classifier. ReferenceSet in the formal method approach is a finite set of object references and is used to model object types.

\[ \forall o: \text{ObjectDef}; t: \text{ReferenceSet} \mid \text{instances } o.\text{class} = t \]

\[ \text{oclIsTypeOf}(o, t) = \text{TRUE} \]

\[ \forall o: \text{ObjectDef}; t: \text{ReferenceSet} \mid \text{instances } o.\text{class} \neq t \]

\[ \text{oclIsTypeOf}(o, t) = \text{FALSE} \]

When instances of o.class referring to object type is equal to the ReferenceSet given in parameter, the expression of oclIsTypeOf results in TRUE, otherwise it returns FALSE.

oclIsKindOf: Evaluates to true if the type t corresponds to the type of self or is one of its supertypes. In terms of conformance, we say that the type of self conforms to the type t. It has the following signature:

\[ \text{oclIsKindOf}(t: \text{Classifier}) : \text{Boolean} \]

\[ \forall o: \text{ObjectDef}; t: \text{ReferenceSet} \mid \text{instances } o.\text{class} = t \]

\[ \text{oclIsKindOf}(o, t) = \text{TRUE} \]

\[ \forall o: \text{ObjectDef}; t: \text{ReferenceSet} \mid \neg\text{instances } o.\text{class} \subseteq t \]

\[ \text{oclIsKindOf}(o, t) = \text{FALSE} \]

According to the formalization of the OCL metamodel types, each type is the union of all the corresponding class instances and of the types of its subclasses. Thus, if instances of o.class (referring the object type) are part of the type t given in parameter, then the expression oclIsKindOf returns TRUE, otherwise it means that the object type is not a subtype of t, resulting in FALSE.

oclAsType: Results to a re-typing or casting of the object self when the type t is one of its supertypes. It has the following signature:

\[ \text{oclAsType}(t: \text{Classifier}) : \text{instance of Classifier} \]

\[ \forall o: \text{ObjectDef}; t: \text{ReferenceSet} \mid \text{instances } o.\text{class} = t \]

\[ \text{oclAsType}(o, t) = o \]

\[ \forall o: \text{ObjectDef}; t: \text{ReferenceSet} \mid \neg\text{instances } o.\text{class} \subseteq t \]

\[ \text{oclAsType}(o, t) = \text{NIL} \]

\[ \forall o: \text{ObjectDef}; t: \text{ReferenceSet} \mid \text{instances } o.\text{class} \subset t \]

The type OclInvalid is an instance of the metatype InvalidType present in the OCL metamodel types. It is a type that conforms to all other types and it has one single instance, identified as invalid.

oclIsInvalid: Evaluates to true if self is equal to OclInvalid. It has the following signature:

\[ \text{oclIsInvalid}(o) : \text{Boolean} \]

\[ \forall o: \text{ObjectDef} \mid \text{instances } o.\text{class} = \text{InvalidType} \]

\[ \text{oclIsInvalid}(o) = \text{TRUE} \]

\[ \forall o: \text{ObjectDef} \mid \neg\text{instances } o.\text{class} = \text{InvalidType} \]

\[ \text{oclIsInvalid}(o) = \text{FALSE} \]

The result of OclIsInvalid is TRUE if the instances of o.class correspond to the InvalidType, FALSE otherwise.

oclIsUndefined: Evaluates to true if self is equal to invalid or equal to null. It has the following signature:

\[ \text{oclIsUndefined}(o) : \text{Boolean} \]

\[ \forall o: \text{ObjectDef} \mid \text{instances } o.\text{class} = \text{InvalidType} \cup o = \text{NIL} \]

\[ \text{oclIsUndefined}(o) = \text{TRUE} \]

\[ \forall o: \text{ObjectDef} \mid \neg\text{instances } o.\text{class} = \text{InvalidType} \cup o = \text{NIL} \]

\[ \text{oclIsUndefined}(o) = \text{FALSE} \]

TRUE is obtained when either instances of o.class correspond to the InvalidType or the object has a NIL value.

IV. CONCLUSION AND FUTURE WORK

This paper has described a formal approach to express OCL predefined properties in a clear and rigorous way. Based on Z notation, this approach describes first the formalization of OCL metamodel types that models the conformance notion between different types and then it proceeds in expressing formally the predefined properties. By this formalism concept, we overcome OCL limitations caused by its weak semantic and open rooms for provability by the ability to use theorem provers on Z specifications.
As future work, we plan to automate the process of evaluation and verification of OCL expressions by the means of a tool where the formal methodology adopted will work in background to preserve the friendly and easy-to-use syntax of OCL.

REFERENCES