Tool Paper: A Lightweight Formal Encoding of a Constraint Language for DSMLs

Arnaud Dieumegard, Marc Pantel, Guillaume Babin, and Martin Carton

IRIT ENSEEIHT
Université de Toulouse
France
first.last@enseeiht.fr

Abstract. Domain Specific Modeling Languages (DSMLs) plays a key role in the development of Safety Critical Systems to model system requirements and implementation. They often need to integrate property and query sub-languages. As a standardized modeling language, OCL can play a key role in their definition as they can rely both on its concepts and textual syntax which are well known in the Model Driven Engineering community. For example, most DSMLs are defined using MOF for their abstract syntax and OCL for their static semantics as a metamodeling DSML. OCLinEcore in the Eclipse platform is an example of such a metamodeling DSML integrating OCL as a language component in order to benefit from its property and query facilities. DSMLs for Safety Critical Systems usually provide formal model verification activities for checking models completeness or consistency, and implementation correctness with respect to requirements. This contribution describes a framework to ease the definition of such formal verification tools by relying on a common translation from a subset of OCL to the WHY3 verification toolset. This subset was selected to ease efficient automated verification. This framework is illustrated using a block specification language for data flow languages where a subset of OCL is used as a component language.

1 Introduction

Domain Specific Modeling Languages (DSMLs) are used in many domains where their capabilities were shown to be very useful for assistance in system and software engineering activities like requirements analysis, design, automated generation, validation or verification. The critical system and software industry is one of the domains where they have been used for many years. This has allowed to achieve better quality and safety of the products without having their development cost following exponentially the curve of the systems complexity.

Beside the obvious advantages of relying on DSML's for the formalization of requirements and design elements, and the automation or simplification of these activities, it remains difficult to convince at the certification level on their correctness. Formal methods have been used in this purpose and as such have been proven as being formidable allies, especially for the verification and validation tasks. In the recent upgrade to level C of aeronautics standards DO-178,

both technologies were introduced as DO-331 for Model Driven Engineering and DO-333 for Formal Methods. Both documents advocate DSMLs use but their combined use given in DO-333 looks more promising that the lone use of DSMLs in DO-331. Both documents advocate a precise, complete and non-ambiguous specification of DSMLs.

Their specification usually relies first on the model based definition of their concepts using a DSML such as the OMG Meta Object Facility (MOF). These metamodels are usually completed with static semantics expressed with constraint languages such as the OMG Object Constraint Language (OCL) that provides first order logic and model navigation construct. It allows specifying the static properties of the DSML itself and its concepts. Many formal specifications of both MOF and OCL have been conducted to assess their properties, validate the standards and participate in the verification of their implementation. These specifications may also provide formal verification tools for models expressed with these languages. However, this was usually not their main purpose and thus these tools do not usually behave well on the scalability side. This contribution targets a more scalable verification dedicated encoding of a subset of OCL in the WHY3 verification toolset that can be reused in different DSMLs. These DSMLs rely on this subset of OCL as a component language. One key change is that the data part of OCL must be adapted to fit the host language.

The WHY and WHYML languages are two complementary languages used for formal software verification. The first one is a high level specification language and the second one a powerful programming language (WHYML). In this setting, programs written using the latter are relying on the formal definitions for types, theories and functions prototypes defined with the former. The WHY3 toolset provides a harness for the manipulation of both specifications and programs and also for their translation in order to be verified either automatically using SMT solvers or manually using proof assistants like COQ, PVS or ISABELLE. The WHY3 toolset is very powerful in the sense that it allows to benefit from the combined strengths of each managed SMT solver and if necessary to rely on manual verification for proofs that may be too difficult to achieve automatically.

By providing a formalisation of a subset of the OCL in the WHY3 toolset, we target to build a framework allowing to ease the formal verification of DSMLs implementations and instances. In order to do so, we propose a formal specification and implementation of a lightweight version of the OCL language. DSMLs specification, expressed using for example MOF compliant metamodels and OCL constraints, or their instances can then be translated to the WHY language and be automatically and formally verified. Such verification can ensure the correctness of the DSML specification, assess the conformance of its instances, or verify complex properties on the instances. We have applied this approach for the verification of correctness properties over the instances of a data flow languages specification DSML [10].

This contribution first pictures our use case and its verification in Section 2. We then provide in Section 3 our lightweight formal encoding of the selected subset of OCL using the WHY and WHYML languages. We refer to it as lightweight

as it does not covers the whole OCL in order to have a better verification automation and scalability. We will additionally discuss the coverage of the OCL specification. We compare our approach to already existing ones in Section 4. We finally conclude and detail some of our perspectives in Section 5.

2 OCL as a DSML Component

As an Object Constraint Language, OCL has been mainly used to define properties to be verified on class instances in UML. It thus includes first order logic and query facilities over class instances and its data part is strongly related to the UML one. In the OMG MDA proposal, DSMLs were first defined as simplified class diagrams in the MOF subset of UML. OCL has then been used to model the static semantics of the DSML elements. Nowadays, most OMG standards rely both on MOF and OCL as specification languages.

These ones are thus now widely known in the software engineering domain. They can then be used as a language component to be included directly in the DSML [11] to allow the DSML user to model constraints and queries with a well known and standard notation. OMG already enforces the reuse of its languages like OCL in the QVT standard. Other DSMLs have integrated OCL as a language component like the use of TOCL for the specification of common behavioral requirements for EMF DSML models in[21] or for execution trace matching in the FEVEREL DSML [20], the Event Constraint Language (ECL) which allows to derive CCSL constraints from EMF DSML models [8], the Behavioral Coordination Operator Language which allows to specify executable model coordination operators[14]. These DSMLs metamodels combine several metamodel components including the OCL one that relies on identifier to connect to the elements in the other metamodel components. In the usual integration of OCL in UML or MOF, the OCL expressions are written in contexts that provides the identifiers from the UML/MOF part of the DSML and the data part of OCL is mapped to their data parts. The identifiers can then be used in the OCL expressions like the names of classes, attributes, methods and parameters. OCL itself can define identifiers that bind stronger than the ones from the context. A similar approach is used to connect OCL with other parts in the DSML metamodel. One key point is to connect also the OCL data part with the DSML one.

2.1 General approach for the verification of DSMLs

Formalisation of a DSML in order to conduct formal verification activities starts with the formalisation of the DSML structural elements and their properties. It is usually achieved by expressing the DSML structure, and its static semantics (the metamodel and its OCL constraints) with the formalism targeted for the verification. Two approaches are commonly used in that purpose: on the one hand, modeling the DSML in the formal verification platform including the model creation facilities and then building and verifying models directly in such a platform (a kind of denotational semantics); and on the other hand, providing a transformation from the DSML to the verification platform that is applied on each

model for verification (a kind of translational semantics). The first one builds the formal model at the language level, whereas the second one works at the instance level. In both cases, the writing (either being manual or automated) of the formal representation is parametrized by the DSML data specification (i.e. the data that are expressed in the models and verified by the constraints).

If the DSML relies on an additional language such as OCL as a component to express properties, it is then also required to translate those properties into the verification formalism. This work can be leveraged by first providing a formalisation for the OCL component language and then relying on this formalisation for the automatic translation of the other parts of the DSML.

Many DSML also include a definition for their execution semantics. This definition can be directly embedded in the DSML definition[5]; it can also be defined externally or can be provided using third party component action languages such as ALF[1]. The execution semantics should also be expressed using the verification formalism and the associated formal verification platform. This provides an execution semantics formal specification. This one can be manually or automatically translated into the formal verification platform.

Finally, the verification of a DSML is based on the expression of properties to be verified. These properties are either written directly using the verification formalism or automatically generated from the DSML instances. The second approach is preferred as it increase the efficiency of the verification and also eases the verification for the DSML user.

In the following, we illustrate the verification of a DSML through the example of the verification of the BLOCKLIBRARY DSML. This DSML allows using a limited subset of OCL as a component language. BLOCKLIBRARY DSML instances are translated as WHY3 theories. OCL expressions are translated to WHY3 expressions relying on the pre-existing formalisation of the restricted version of the language. Finally, high level properties are automatically generated from the DSML instances as WHY3 goals to be discharged through the WHY toolset.

2.2 The BlockLibrary DSML structural specification

The BLOCKLIBRARY DSML [9, 10] aims at the specification of the structure and semantics of data flow atomic blocks such as the ones at the core of the SIMULINK or SCADE languages. This DSML allows to specify the inherent variability of these blocks structure and semantics. The structural specification part of the DSML is inspired from the concepts of software product lines combined with the object oriented concept of inheritance in order to handle the highly variable nature of the blocks specification. Data types can be attached to blocks specification components (inputs, outputs, parameters and memories). The components are specified with constraints describing their allowed values (specified using a subset of OCL). The semantics of the blocks is expressed with a simple action language that relies on the same subset of OCL to express pre/post-conditions, variants and invariants.

[1] http://www.omg.org/spec/ALF/Current

The two main concerns in the DSML are the use of OCL as a language component for the specification of the DSML data and action constraints and the variability management that allows combining parts of the blocks specification into multiple block configurations.

2.3 Verification of BlockLibrary instances with Why3

A significant part of the effort in this case study has been focused on the formalisation of the DSML constructs as WHY theories to conduct formal proofs of the model completeness, consistency and correctness using the WHY3 platform.

A block specification using the BlockLibrary dsml We provide in Figure 1 a simplified specification for the `Delay` block written using the BLOCKLIBRARY DSML. The `Delay` block outputs the value provided as its first input and delays its value by a certain number of clock ticks. The number of clock ticks is either provided as a parameter of the block or as an additional input. The values of the output of the block for the first clock ticks cannot be computed from the inputs of the block and should thus be provided as either a parameter or as an additional input of the block. The block allows for scalar, vectors and matrix values. The specification provided here is limited to the handling of scalar values and the initial condition can only be provided as a parameter of the block.

A block specification is contained in a `blocktype` element bundled in a `Library` construct. The data types used in the specification are declared in the `Library`. The `variant` constructs contains the specification for a block structural elements: input (`in`) and output (`out`) ports, `parameters` and `memories`. For each declaration of a structural element, it is possible to specify its default value and some additional constraints using our subset of the OCL language. We provide such a constraint in line 20, where we specify that the `Delay parameter` can only have a positive value.

Each `variant` can extend, in an object oriented way, other `variant` constructs by relying on the `extends` construct. This composition is done through two n-ary operators that can be applied on `variant` constructs: `allof` and `oneof`. The former specifies mandatory compositions whereas the former specify alternative compositions. These operators are inspired from the software product line approach for the specification of features hierarchies.

The `mode` constructs allows for the specification of the blocks semantics variation points. They thus provide an implementation for one or more `variant` constructs. The previously presented composition language can be used in this purpose. Each `mode` construct must provide a definition for the `compute` semantics of the block variation point (lines 57 to 61). It may also provide the optional semantics for the `init` and `update` computation phases for the specified block. The blocks semantics are provided using a custom simple action language: the BLOCKLIBRARY action language (`bal`). A `mode` and its implemented `variant` make a set of block configurations. This set is extracted from the composition of `variant` elements through the `allof` and `oneof` constructs.

```
library Lib {

type signed realInt TInt32 of 32 bits
type realDouble TDouble
type array TArrayDouble of TDouble

blocktype DelayBlock {
  variant InputScalar {
    in data Input : TDouble
    out data Output : TDouble
  }
  variant ICScalar {
    parameter IC : TDouble default 0.0
  }
  variant ICVector {
    parameter IC : TArrayDouble
  }
  variant InternalDelay {
    parameter Delay : TInt32 {
      invariant ocl { Delay.value > 0 }
    }
  }
  variant ExternalDelay {
    in data Delay : TInt32 {
      invariant ocl { Delay.value > 0 }
    }
  }
  variant ListDelay_Scalar extends allof (
    oneof (InternalDelay, ExternalDelay),
    InputScalar, ICVector
  ) {
    invariant ocl { Delay.value > 1 }
    invariant ocl {
      IC.value.size() = Delay.value
    }
    memory Mem {
      datatype auto ocl {Input.value}
      length auto ocl {DelayUpperBound.value}
    }
  }
  variant Delay_Scalar extends allof (
    oneof (InternalDelay, ExternalDelay),
    InputScalar, ICScalar
  ) {
    invariant ocl { Delay.value = 1 }
    memory Mem {
      datatype auto ocl {Input.value}
      length auto ocl {1}
    }
  }
  mode DelayMode_Simple implements Delay_Scalar
      {
    init init_Delay_Simple bal {
      postcondition ocl {
        Mem.value = IC.value }
      Mem.value = IC.value;
    }
    compute compute_Delay_Simple bal {
      postcondition ocl {
        Output.value = Mem.value }
      Output.value = Mem.value;
    }
    update update_Delay_Simple bal {
      postcondition ocl {
        Mem.value = Input.value }
      Mem.value = Input.value;
    }
  }
  mode DelayMode_List implements
      ListDelay_Scalar{
    init init_Delay_List bal {
      postcondition ocl {
        Mem.value = IC.value }
      Mem.value = IC.value;
    }
    compute compute_Delay_List bal {
      postcondition ocl {
        Output.value = Mem.value->first() }
      Output.value = Mem.value[0];
    }
    update update_Delay_List bal {
      postcondition ocl { Mem.value =
        Mem.value
        ->excluding(Mem.value->first())
        ->append(Input.value)
      }
      for (var i=0; i < Mem.value->size()-1;
        i = i+1){
        Mem.value[i] = Mem.value[i+1];
      }
      Mem.value[Delay.value-1] = Input.value;
    }
  }
}}
```

Fig. 1. Delay block specification using the BlockLibrary DSML

The lack of space prevents us to detail further this DSML and we refer the reader to our papers[9, 10], the related thesis work [2], and the website[3] for additional informations.

BlockLibrary specification verification A transformation translates BlockLibrary instances to Why theories; OCL-like constraints to predicates relying

[2] http://www.dieumegard.net/thesis/Thesis-Arnaud_Dieumegard.pdf
[3] http://block-library.enseeiht.fr

on a lightweight formalisation of the subset of OCL detailed in Section 3; and semantics specification (specified using the action language) to WhyML functions.

In addition to this translation of the BlockLibrary specification, properties are generated to assess: the completeness (are all possible configurations of a block given in the specification ?) and the disjointness (are all possible configurations of a block expressed only once in the specification ?) of all the BlockLibrary DSML block specifications. The verification of these two properties is done automatically by SMT solvers in a few tenth of a second for rather simple blocks specification to up to a few seconds for mode complex ones containing up to a thousand different configurations for a block.

The most important lesson learned from this experiment was that as soon as the DSML constructs, their attached constraints, and the OCL component expressions are translated to Why, it is straightforward to write or generate high level properties to be verified using the Why and WhyML languages.

The following section summarizes the Why model of the specific subset of the OCL that was necessary for our DSML use case where a subset of the OCL is used as a DSML component.

3 OCL Formalisation Using the Why Toolset

OCL is a formal language for querying model elements and expressing model properties. It relies on first order logic and model traversal operators. It is guaranteed to be side-effect free and as such cannot modify the model it is applied to. The Why language is more expressive than OCL and can thus be used to encode any OCL construct.

We provide here some details on our formal specification of our specific subset of OCL in Why. It is important to keep in mind that the main objective of the BlockLibrary DSML is to allow for the formal specification of blocks. As such, we enforce the block specifier to provide strictly typed elements. The consequences of this limitation is impacting the subset of the OCL to be handled. It is indeed not required to handle type manipulation operations such as *oclIsKindOf* or *oclAsType* among others.

We will go through the support for some of the OCL constructs; the limitations we selected; and the translation strategies in order to go from OCL constructs to Why constructs. As we will not provide the complete details about our translation, we invite the interested reader to refer to our website[4] for additional informations.

3.1 OCL standard data types and collections

OCL is built on a simple set of types called primitive data types. The Why3 standard library provides a formalisation for primitive data types that largely covers the needs for the mapping of the OCL ones.

[4] http://block-library.enseeiht.fr/html

OCL primitive values can be gathered in collections that differs regarding their ability to handle multiple occurrences of the same value (*Bag*, *Sequence*) or not (*Set*, *OrderedSet*) and if these values are ordered (*Sequence*, *OrderedSet*) or not (*Bag*, *Set*).

In the WHY standard library, collections can be modeled as lists allowing multiple occurrences of the same unordered value. This makes them a direct translation for *Bag* collections. The standard library also provides the support for Arrays that would directly map to the *Sequence* collection and Sets for *Set* collection in OCL.

In our implementation of OCL, we do not take into account the type of the collections and only provide support for *Bag* collections as lists. This allows to simplify their management in our implementation as it removes the side effects on the collection management operations such as that elements are not automatically removed from the collections as multiple occurrences are allowed. We defined a WHY theory called *OCLCollectionOperation* containing the definition for some basic list accessors and operations. The other three kinds of OCL collections will be implemented in a similar way in a near future with no specific issues expected.

We decided not to support messaging related constructs and tuples constructs. Messaging is related to UML sequence and state machine diagrams which was out of the scope of our case study. Tuples are also missing in our implementation and could be implemented on a first approximation by relying on record types in WHY.

3.2 OCL operations translation strategy

OCL defines multiple operations that are to be applied either on primitive values – referred to as standard language operations – or on collection values – referred to as collection and iteration operations. Some of these operations are not supposed to be used on *Bag* collections and explicit conversions between collections types must be done. We do not enforce this currently in our implementation of OCL as we provide only one type of collection.

The translation of OCL expressions to the WHY language can be done using two strategies. First, operations can be translated to basic first order logic expressions and thus can directly be used in WHY. Second, the definition for the operations can be axiomatized using WHY function declarations and OCL constraints are translated as expressions using these functions. In our work, we rely on a combination of the two. The list getter is used for simple collection accesses, standard type operations have been defined as functions in WHY, simple collections operations are directly mapped to their logical expression equivalent.

This approach has the advantages of easing the translation work as the semantics of OCL standard types operations is already defined and thus avoid to generate too complex expressions. This has the pleasant side effect of easing the transformation verification activities as the translation itself is simpler.

In the following, we provide the mapping between the source OCL constructs and operations and the target WHY predicates, functions and expressions.

3.3 OCL standard library operations

In OCL expressions, operations can be applied on primitive OCL types. These operations are classical handling of primitive data types and are gathered in the OCL standard library. They have already been formalized in the WHY library. We thus rely on their formalisation for our translation.

We did not currently implement the transformation for the *div* and *mod* operations on OCL *Double* elements. The *div* operation is of particular interest as its behavior in OCL and in WHY are not the same. Indeed where the OCL implementation of *div* applied to any number and 0 (division by zero) returns the *null* value, the WHY version is simply not defined and rejected by the formal verification toolset. The management of specific values like *null* and *invalid* is not done and is a chosen limitation of our work to simplify the formal models and reduce the verification costs by avoiding three value logics (*True*, *False* and *invalid*). We give later more details regarding the related restrictions.

Boolean OCL expressions are implemented using boolean expressions in WHY. *xor* operator has been implemented with *and*, *or* and *not* operators. Numeric operations have been implemented using the standard WHY arithmetic theories and their operators. Finally relational operators are also based on standard WHY constructs.

3.4 Collection operations

As previously mentioned, there are four types of OCL collections. We only considered the use of the *bag* collections in our case study. In our handling of OCL collection operations, we do not handle OCL generic nature nor subtyping of elements. If the same operation is to be expressed on different types, it is then developed separately for each different type. Whereas this may seem to be a limited way of handling this problem, in practice, our supported restriction of OCL makes this easier as only a few operations needs to be encoded several times. Both kind of polymorphism could be handled by synthesizing sum types representing the allowed set of types for an operation and appropriate pattern matching that mimics late binding at a higher verification cost.

Our partial handling of OCL collections has an impact on the translation provided for some collection operations. The *append* and *including* operations have the same implementations and so are *subOrderedSet* and *subSequence*. According to the OCL specification, some collections operations are not allowed on *bag* collections: *append*, *at*, *first*, *indexOf*, *indexAt*, *last*, *prepend*, *subOrderedSet*, *subSequence*. We decided to allow their use in our implementation of OCL. This is a significant drift from the OCL standard but it has the advantage of greatly simplifying the translation mechanism without restraining the expressiveness of the language. This restriction could be enforced easily at the static semantics level. The formalisation of the usual OCL collections is of additional complexity as studied by Mentré et Al [16] but was not of primary interest for our current work so we decided not to address the related issues.

Each of these functions are defined through a set of axioms specifying their context of use: on which element type they are defined; what restrictions are required for their definitions; and the result of their computation according to the provided input values. The restrictions on their parameters are used to avoid the *invalid* value. The effort needed in order to achieve the complete verification remains important but is highly lightened by the use of SMT solvers to automate the verification.

3.5 Logical property assessment iteration operations

Iteration operations are the main operations used on OCL collections. They allow to assess a logical property verification on: every element of a list (forAll), at least one element of a list (exists), exactly one element of a list (one). They can also express the uniqueness of the result of the application of a function on every element of a list (isUnique). All these operations returns a boolean value.

In Table 1, we provide the translation for the *forAll*, *exists*, *one* and *isUnique* OCL operations on collections. Unlike the previous translations, we do not rely on predefined functions but we rather map these operations to simple first order logic expressions. Regarding the translation of the condition expression: *exp*, the defined OCL iterators: *it*, *it1* and *it2* are mapped to a call to the position of the element in the collection via the list getter operator. In practice, references to *it* or *it1* are replaced by *a[i]* and references to *it2* are replaced by *a[j]* in *exp*. This is expressed using the function application: $[a/b]c$ that substitute a to b in c.

ocl expression	**Target Why code**
a→**forAll**(it: DT | exp)	∀ i: int. 0 ≤ i < length a → [a[i]/it]exp
a→**forAll**(it1, it2: DT | exp)	∀ i j: int. 0 ≤ i < length a ∧ 0 ≤ j < length a → [a[i]/it1,a[j]/it2]exp
a→**exists**(it: DT | exp)	∃ i: int. 0 ≤ i < length a ∧ [a[i]/it]exp
a→**exists**(it1, it2: DT | exp)	∃ i j: int. 0 ≤ i < length a ∧ 0 ≤ j < length a ∧ [a[i]/it1,a[j]/it2]exp
a→**one**(it: DT | exp)	∃ i: int. 0 ≤ i < length a ∧ $[a[i]/it]exp$ ∧ (∀ j: int. 0 ≤ j < length a ∧ j <> i → $[a[i]/it]exp$ <> $[a[j]/it]exp$)
a→**isUnique**(it: DT | exp)	∀ i,j : int. 0 ≤ i < length a ∧ 0 ≤ j < length a ∧ i <> j → $[a[i]/it]exp$ <> $[a[j]/it]exp$

Table 1. OCL logical property verification operations mapping to WHY expressions

3.6 Value extraction iteration operations

Iteration operations also allow extracting values from a collection: according to the satisfaction (select) or not (reject) of a property; or by applying a treatment on every element of a list (collect). Value extraction operations are more complex to model as they do not only provide a single boolean value as output, they actually compute a list of elements.

Iteration operations semantics Iteration operations apply on collections and compute filtering of the collection values and/or mapping of functions on the

collection values. We decide thus to represent a collection as the $\langle c, p, f\rangle$ tuple. Its semantics is provided in (1).

$$[\![\langle c, p, f\rangle]\!] = \{f(v) | v \in c, p(v)\} \tag{1}$$

According to the previous notation, we define the initial value of a collection as in (2) where $\top$ is the predicate returning *true* and `id` the identity relation.

$$c = \{v_1, ..., v_n\} = \langle c, \top, \texttt{id}\rangle \tag{2}$$

The definition of iteration operations is hence specified as in (3). From these definitions, we extract the implementations for iteration operations using the WHY language. We provide in the following two possible implementations.

$$\begin{array}{rrr} \langle c, p, f\rangle \rightarrow & select(e|\varphi) = & \langle c, p \wedge [f/e]\varphi, f\rangle \\ \langle c, p, f\rangle \rightarrow & reject(e|\varphi) = & \langle c, p \wedge \neg[f/e]\varphi, f\rangle \\ \langle c, p, f\rangle \rightarrow & any(e|\varphi) = & \langle c, p, f\rangle \rightarrow select(e|\varphi) \rightarrow first() \\ \langle c, p, f\rangle \rightarrow & collect(g) = & \langle c, p, f \circ g\rangle \end{array} \tag{3}$$

Iteration operations as first order logic constructs Providing an implementation for iteration operations can be done using first order logic constructs. It is then required to generate a different function for each iteration operation call and then to write a call to the generated function in the translated OCL operation code.

Using first order logic to provide an implementation for iteration operations is quite straightforward. Nevertheless, the implementation and verification of the generation process might be quite complex. Indeed the generated code complexity might increase when complex expressions are used in the body of the iteration operation.

Iteration operations as higher order logic construct An alternative approach for the implementation of iteration operations is to rely on higher order logic to represent the operations in WHY. An example of such formalisation is provided in Listing 1.1 for the *select* iteration operation. The *select* function in WHY takes two arguments, the first one is the collection on which the operation is applied and the second one is the predicate that must be used in order to select the elements of the collection. In addition to the function definition, we provide a set of lemmas verified with the WHY3 platform generating proof obligations discharged using SMT solvers and proof assistants. In the case of the *select* function, part of the lemmas are verified using SMT solvers and others have been verified using the COQ proof assistant. We provide in Figure 2 a report generated with the WHY3 toolset for these verifications. The verification effort required for the verification of the iteration operations (writing of the specification and achieving the proof) is rather similar to the one needed for the verification of collection operations. Details regarding the formalisation and the proofs for the other OCL collection operations is provided in the first author PhD thesis work [5], and our website[6].

[5] http://www.dieumegard.net/thesis/Thesis-Arnaud_Dieumegard.pdf
[6] http://block-library.enseeiht.fr

In order to use the iteration expressions, one must provide a body containing the condition or the operation to apply on each element of the collection. The condition body of the *reject*, *select* and *anyAs* operations is translated as an inlined predicate whereas the function body of the *collect* operation is translated as an in-lined function. In-lined predicates and functions are provided as the second argument of their respective WhyML function call. Contrary to the first order operations, there is no need here to keep track of the variables used in the iteration operation as the in-lined nature of the function call makes their definition directly available.

```
function select (l: list oclType) (p: HO.pred oclType) : list oclType =
    match l with
    | Nil -> Nil
    | Cons hd tl -> if p hd then Cons hd (select tl p)
                            else select tl p
    end

lemma select_nil: forall p: HO.pred oclType.
  select Nil p = Nil
lemma select_cons_nil_verified: forall e: oclType, p: HO.pred oclType.
  p e -> select (Cons e Nil) p = Cons e Nil
lemma select_cons_nil_not_verified: forall e: oclType, p: HO.pred oclType.
  not (p e) -> select (Cons e Nil) p = Nil
lemma select_cons_verified: forall e: oclType, l: list oclType,
                                  p: HO.pred oclType.
  p e -> select (Cons e l) p = Cons e (select l p)
lemma select_cons_not_verified: forall e: oclType, l: list oclType,
                                     p: HO.pred oclType.
  not (p e) -> select (Cons e l) p = select l p
lemma select_mem_reduc: forall l: list oclType, b: oclType,
                               p: HO.pred oclType.
  mem b (select l p) -> mem b l
lemma select_mem: forall l: list oclType, b: oclType,
                         p: HO.pred oclType.
  (mem b l /\ p b) -> mem b (select l p)
lemma select_not_mem: forall l: list oclType, b: oclType,
                             p: HO.pred oclType.
  (mem b l /\ not (p b)) -> not (mem b (select l p))
```

Listing 1.1. Select iteration operation formalisation in Why using higher order logic

Proof obligations	**Alt-Ergo-Pro (1.0.0)**	**Coq (8.4pl3)**
lemma `select_nil`	0.03	
lemma `select_cons_nil_verified`	0.04	
lemma `select_cons_nil_not_verified`	0.05	
lemma `select_cons_verified`	0.03	
lemma `select_cons_not_verified`	0.04	
lemma `select_mem_reduc`		2.40
lemma `select_mem`		2.31
lemma `select_not_mem`		2.01

Fig. 2. Select verification through SMT solvers and proof assistants (time in seconds)

3.7 Additional restrictions on the support of the OCL language

In the previous sections, we gave the translation rules for a subset of the OCL language. OCL provides a well known syntax for software engineers. In addition, this subset of the language eases and secures the constraints writing process relying on first order logic by including only well known and standard functions and operations on classical data types.

A major restriction in our implementation of the OCL language is the absence of the specific *invalid* and *null* values. This is related to the binding to the data part of the DSML and to the formal verification introduced by the WHY3 toolset. The *invalid* value is used for the modeling the failure of an operation, we advocate that if such an operation is equipped with pre-conditions then the *invalid* value can never be an output of the operation and is thus not necessary. This value is also used when OCL is bound to UML or MOF to model optional attributes or references that are not defined in the model instances. We advocate that an empty collection is also a good model for this case. Regarding the *null* value, it is used in order to model an object that does not exists. Once again this should not happen in a formalisation of the language but it could be handled by relying on an *option* type. Adding the support for these specific values will have a strong impact on the WHY implementation of the OCL constructs. Indeed, these specific values will need to be taken into account in the implementation of all functions and on the related proofs. There are technical difficulties in implementing these values as shown by [2] but it is possible to overcome most of them.

The limitations we applied on the formalisation of OCL have the advantage of providing a simpler, well defined subset of the language enforcing the DSML developer to rely mostly on common knowledge OCL constructs and thus avoid the problems related to the use of more complex constructs such as the *closure* operator. These operators may cause some misunderstandings; make the specification more obscure; and make its verification more complex.

The subset of OCL we currently handle could be extended if required by the integration to other DSMLs as shown by [2]. We will try to keep it as limited as possible.

4 Related Works

Many works in the literature target the formal verification of DSMLs. We will only focus on those where the OCL language is used either in the DSML specification and/or as a language component part of the DSML; and is included in the verification process. Most of the formal DSML verification processes including OCL are related to its use in the context of UML specifications.

Our approach is close to the one proposed in the HOL/OCL project[3, 2]. Its authors target the formal verification of UML specifications including OCL constraints and contracts by providing a bridge to the HOL/OCL whose formal foundations relies on the ISABELLE proof environment. This work is specifically remarkable by its coverage of OCL and has highly contributed to the OCL community by providing formal specifications for the language that raised numerous

legitimate questions regarding the semantics of OCL operations and constructs leading to the clarification of the standard. Our work takes a different angle and aims at the verification of DSMLs in general whereas the HOL/OCL one is focused on UML. We also differ by our use of the WHY3 toolset instead of ISABELLE. Both toolsets allows for the use of SMT solvers for the automatic verification of properties but WHY3 can be used as a bridge to many more different solvers and proof assistants including ISABELLE. WHY3 also benefits from its simpler high level syntax. It is our belief that by relying on the WHY3 toolset, a wider range of users will be able to actually perform formal verifications of properties as the WHY and WHYML languages are simpler to apprehend than ISABELLE. Our work targets a significantly smaller coverage of the OCL standard that allows avoiding key semantics issues like the *invalid* and *null* values. This has not been a limiting issue for our use up to now. However, we are far from the completeness of the HOL/OCL project and we do not plan to reach it.

Clavel et al [4, 7] provides a translation from OCL to first order logic (FOL). Automated theorem provers and SMT solvers are then used to check the unsatisfiability of the generated FOL constructs. This work supports the *null* and *invalid* values. It focuses on the verification of OCL constraints expressed on data models. Our work is more focused on the verification of OCL as a language component for DSMLs. The same difference can be found in the work by Lano et al [15] where UML class diagrams and a subset notational variant of OCL (LOCA) are handled and verified through a transformation to the B language.

UML/OCL specifications are translated to SAT solving format by Soeken et al [18, 19] or integrated in the USE toolset by Kuhlmann et al [12]. Another formalism for the automated verification of specifications is the PVS theorem prover. Kyas et al in [13] provides a lightweight translation of OCL and UML state machines and class diagrams. Each of these approaches uses different formalism for the formal verification with efficient verification results but are tied up to the use of this specific formal verification technique. Our work benefits from the wider range of formal verification toolsets that the WHY3 platform provides including most of the ones used in these works.

Some other approaches have been used in order to formally model UML/OCL specifications as in the works from Kyriakos et al [1] and Cunha et Al[6] on the UML2Alloy tool. These provide means for the verification of UML/OCL specifications and put a specific emphasis on the feedback of the verification result to the DSML user. We did not focus on the feedback from solvers in our work but it would be possible as the WHY3 toolset provides access to SMT solvers with counter-example generation capabilities.

5 Conclusion and Future Works

We have described a lightweight formalisation of the OCL language in WHY3 where the limitations have been identified and discussed. We have briefly detailed an example of DSML where OCL is used as a component language. We rely on our formalisation of OCL to conduct the automated verification of high level properties on the DSML instances.

It is our belief that the DSML community would benefit from such a formalisation as it allows to improve the quality of model verifications by relying on formal verification techniques; it is based on the use of the WHY3 toolset that eases the writing of properties and is more accessible for industrial practitioners than other formal verification platforms (like proof assistants for example) yet still supporting their use. This accessibility is very likely to help on the adoption of formal methods as a natural and standard verification technique.

The work conducted here was mostly driven by the verification of a specific DSML: the BLOCKLIBRARY specification language. The next step is to ease the integration of OCL as a component language for other DSMLs. Specifically, we will focus on the adaptation of the translation of the DSML structure to WHY3 which is currently done manually. We plan on providing a translation from meta-models instances of MOF or ECORE to generically transform any DSML specification into a set of WHY theories. The translated theories will then be used in conjunction with our formalisation of OCL to conduct verification activities on the DSML instances.

We are investigating a larger subset of OCL that still allows for an efficient automatic verification of DSML properties. We will specifically focus on the handling of any type of collections; the constructs that still need handling (tuples); and collection operations we did not implement yet. Handling the special *invalid* and *null* values could be done by relying on an approach close to the one provided by Dania et al in [7].

In our BLOCKLIBRARY DSML the blocks semantics is specified using an action language. Such a semantics is then translated as a WHYML function. We could have used an already existing action language like FUML/ALF [17] for example. By relying on a custom language, we simplified the translation work by limiting it only to the constructs that were needed. We plan, as proposed for a subset of OCL, on providing a formalisation for a subset of such an action language component. This will then be used for the verification of the semantics definition for DSMLs and will thus complete the DSML verification approach.

References

1. Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to alloy. *Software & Systems Modeling*, 9(1):69–86, 2010.
2. Achim D. Brucker, Frédéric Tuong, and Burkhart Wolff. Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5. *Archive of Formal Proofs*, January 2014.
3. Achim D. Brucker and Burkhart Wolff. HOL-OCL: A formal proof environment for UML/OCL. In *FASE*, volume 4961 of *LNCS*, pages 97–100. Springer, 2008.
4. Manuel Clavel, Marina Egea, and Miguel Angel García de Dios. Checking unsatisfiability for OCL constraints. *ECEASST*, 24, 2009.
5. Benoit Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to Build Executable DSMLs and associated V&V tools. In *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*, Hong Kong, Hong Kong, December 2012. IEEE.

6. Alcino Cunha, Ana Garis, and Daniel Riesco. Translating between alloy specifications and UML class diagrams annotated with OCL. *Software & Systems Modeling*, 14(1):5–25, 2015.
7. Carolina Dania and Manuel Clavel. OCL2FOL+: coping with undefinedness. In *Proceedings of the MODELS 2013 OCL Workshop co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 30, 2013.*, pages 53–62, 2013.
8. Julien Deantoni and Frédéric Mallet. ECL: the Event Constraint Language, an Extension of OCL with Events. Research Report RR-8031, July 2012.
9. Arnaud Dieumegard, Andres Toom, and Marc Pantel. Model-based formal specification of a DSL library for a qualified code generator. In *Proceedings of the 12th Workshop on OCL and Textual Modelling*, pages 61–62, New York, NY, USA, 2012. ACM.
10. Arnaud Dieumegard, Andres Toom, and Marc Pantel. A software product line approach for semantic specification of block libraries in dataflow languages. In *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*, pages 217–226. ACM, 2014.
11. Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. Integrating OCL and textual modelling languages. In *Models in Software Engineering*, volume 6627 of *LNCS*, pages 349–363. Springer, 2011.
12. Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive validation of OCL models by integrating SAT solving into USE. In *Objects, Models, Components, Patterns*, volume 6705 of *LNCS*, pages 290–306. Springer Berlin Heidelberg, 2011.
13. Marcel Kyas, Harald Fecher, Frank S. de Boer, Joost Jacob, Jozef Hooman, Mark van der Zwaag, Tamarah Arons, and Hillel Kugler. Formalizing UML models and OCL constraints in PVS. *Electr. Notes Theor. Comput. Sci.*, 115:39–47, 2005.
14. Christian Laasch and MarcH. Scholl. A functional object database language. In *Database Programming Languages (DBPL-4)*, Workshops in Computing, pages 136–156. Springer London, 1994.
15. K. Lano, D. Clark, and K. Androutsopoulos. Uml to b: Formal verification of object-oriented models. In *Integrated Formal Methods*, volume 2999 of *LNCS*, pages 187–206. Springer Berlin Heidelberg, 2004.
16. David Mentré, Claude Marché, Jean-Christophe Filliâtre, and Masashi Asuka. Discharging proof obligations from Atelier B using multiple automated provers. In *ABZ'2012 - 3rd International Conference on Abstract State Machines, Alloy, B and Z*, volume 7316 of *LNCS*, pages 238–251, Pisa, Italy, June 2012. Springer.
17. Isabelle Perseil. ALF formal. *Innovations in Systems and Software Engineering*, 7(4):325–326, 2011.
18. M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL models using boolean satisfiability. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1341–1344, March 2010.
19. Mathias Soeken, Robert Wille, and Rolf Drechsler. Encoding OCL data types for sat-based verification of UML/OCL models. pages 152–170, 2011.
20. Faiez Zalila, Xavier Crégut, and Marc Pantel. Leveraging formal verification tools for DSML users: A process modeling case study. In *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, volume 7610 of *LNCS*, pages 329–343. Springer Berlin Heidelberg, 2012.
21. Faiez Zalila, Xavier Crégut, and Marc Pantel. Formal verification integration approach for DSML. In *Model-Driven Engineering Languages and Systems*, volume 8107 of *LNCS*, pages 336–351. Springer Berlin Heidelberg, 2013.