PVS Metamodel

LUKMAN AB. RAHIM
PETRONAS University of Technology
Bandar Seri Iskandar, 31750 Tronoh, Perak
MALAYSIA
lukmanrahim@petronas.com.my

Abstract: - In translating Object Constraint Language (OCL) to Prototype Verification System (PVS) using model transformation technique, the problem is the absence of PVS metamodel. The PVS metamodel is crucial to the transformation because it defines the elements that may involves in it. This paper proposes a PVS metamodel. The metamodel is created based on the main language construct of PVS where each PVS language construct is represented by element in the metamodel. An example of how the metamodel is used in the transformation is also discussed.

Key-Words: - Metamodel, Object Constraint Language, Prototype Verification System

1 Introduction
Verifying models by translating to a formal language has been done in the past. Work such as [1], [2], and [3] translates UML to various formal languages such as Object-Z and B. In our research we translate OCL (Object Constraint Language) to PVS (Prototype Verification System). There are also other previous works that translate OCL to a formal language such as [4] and [5] but these previous works are translating OCL to a different language.

In translating OCL to PVS, model transformation is used to transform OCL to PVS. Model transformation is a technique that generates model (result model) from another model (source model) based on transformation scripts. When using model transformation, PVS metamodel is essential element and the problem is at the time of this research there are no PVS metamodel.

This paper only concentrates on the PVS metamodel that is created for the purpose of transforming OCL to PVS. The result of the transformation has been presented in [6] and the process of translating OCL to PVS is presented in [7].

In the next section works related to this paper will be presented. Section 3 will explain in detail about PVS specification language and Section 4 will present the PVS metamodel that we create. Section 5 show a small example of how the PVS metamodel is used in the transformation and Section 6 conclude the paper.

2 Related Works
In Model-Driven Architecture (MDA), there are four levels of model [8]. As shown in Figure 1, the four levels are model instance (level 0), model (level 1), metamodel (level 2) and meta-metamodel (level 3).

![Figure 1: Modeling Stack](image-url)

Model instances are the model that people create. Models are the language that is used to create the model instances. Metamodels is the model that explains about the modeling language. Meta-metamodels are models that explain about the metamodel. Currently there is only one level 3 model which is Meta-Object Facility (MOF) from OMG. The relationship between the models in the different levels is: models at level 0 must conform to models, models at level 1 must conform to its metamodel and the metamodels must conform to models in level 3.

To better understand the modeling stack, the analogy given in [8] is as follows: “At the lowest M0 level we find the real world, corresponding to a
given execution of say a Pascal program. At level M1 we find the models, corresponding for example to a given Pascal program. For one such Pascal program there is infinity of possible executions. At level M2 we find the meta-models, corresponding for example to a grammar for the Pascal language. For a given grammar, there is infinity of well formed programs. Finally level M3 is the meta-meta-model level. It may be compared to the self-defined, extended BNF formalism”.

Besides explaining the elements in a model, metamodels can also be used as source and target models in a model transformation. Model transformation is a technique introduced by OMG to automate the task of creating platform specific models from platform independent models or other platform specific models. To know more about model transformation and its approaches please read [8] and [9].

In translating OCL to PVS using model transformation, UML metamodel presented in [10] and OCL in [13] are being referred. Besides [13] OCL metamodel have also been presented in [14] and [15]. The metamodel being used in the transformation is being taken from [13] because [13] is the specification document for OCL, thus making the metamodel more acceptable and widely referred.


3 PVS

PVS is a verification system that is equip with a theorem prover and strongly typed, higher order logic specification language. The PVS is used for writing specifications and proving theorems [11]. This paper will concentrate on PVS specification language and the following sections and paragraphs PVS will mean PVS specification language.

PVS specification language is strongly typed which means all elements that is declared must be of specific type. When PVS theorem prover proves a PVS specification, one of the checks that it does is to check the type of each elements. Typing is also very important because the results of proving the theorems/axioms in a PVS specification are influence by the type of the elements inside the theorem. Theorem/axioms are sometimes true to elements from a particular type and false from another.

To support strongly type language, PVS has a very rich type system. Some of the primitive types are natural numbers, rational numbers and Booleans. PVS also considered functions, records and enumerations as types. All functions in PVS must be total because PVS is a higher-order logic based language. Total function means all elements in the domain of the function must have a relation to at least one element in the range of the function.

The following subsections will discuss in more detail about different parts of PVS specification language.

3.1 Theory

The main building construct of a PVS specification is the Theory. A theory represents the complex theorems that the prover will proof and a complex theorem can be structured into many theories.

In PVS, theories can be parameterized. Having parameterized theories increase the clarity of the specification by constraining the instantiation of the theory [11]. This benefit is similar to specifying arguments and return type for a function in a programming language. The parameters and return type act as a signature to the functions.

Inside a theory contains a set of declarations. These declarations can be for functions, types, theorems, axioms or other language elements that we will discuss in more detail in the following sections.

There are two parts of a theory which is the Assuming part and the Theory part. Both sections will have declarations that serve different purpose. Declarations in the Assuming part will be top-level entities that can be used in the Theory part and Assumptions. Assumptions are formula that is expected to hold in all instances of the theory and can be taken as something similar to invariants in Design by Contract (DbC).

Theories also have export and import statements. Export statements are used to specify parts of the theory that will be available to other theories that import them. Import statements are used for specifying external theories that will be used in the current theory.

3.2 Declarations

As mention earlier, PVS theories contain a set of declarations that can be written in either the Assuming part or the Theory part. Entities that can be declared in a theory are [12]:

- Type
- Variable
- Constant
- Formula
• Judgement
• Macro
• Conversion

Declaring a new type in PVS has two meanings. The first meaning is the user is creating a new type, either creating uninterpreted type or creating a subtype. Uninterpreted types are new types without any assumption about it. Subtypes are types that are created from other types, and usually values of a subtype are subsets of value in the parent type. Example of a subtype is integer is a subtype of a natural number. The second meaning of declaring a type is giving another name to a type to improve the readability of the specification. This is especially useful when we are declaring a function, records or tuple.

Declaring a function in PVS can be of two ways: as a type or as a constant. Declaring a function as a constant requires the function to have a value and in the case of functions the value is the function body. For function that is declared as a type, the function does not have a value. User can use function that is declared as a type to create constant function. Example of the declaration of functions in both ways is shown in Figure 2.

Declaring a function in PVS can be of two ways: as a type or as a constant. Declaring a function as a constant requires the function to have a value and in the case of functions the value is the function body. For function that is declared as a type, the function does not have a value. User can use function that is declared as a type to create constant function. Example of the declaration of functions in both ways is shown in Figure 2.

```plaintext
/* function as type */
Function1:TYPE = [int -> int]

/* function as constant */
Function2::[int -> int] = (lambda (x:int): x+1)

/* function as constant that use function type */
Function3:Function1 = (lambda (x:int): x+1)
```

Figure 2: Example of Function Declaration

Formulas are entities in the PVS specification that is going to be proof by PVS theorem prover. Generally there are four types of formulas: axioms, assumptions, theorems and obligations. All types of formula consist of Boolean expressions (expressions that return a Boolean value).

Variable declarations are used to create variables. In PVS variables are not attributes and have nothing to do with the states of the PVS theory. They are logical variables, variables used in formulas and binding expressions. For more elaborate information about declarations of formulas, variable, types, constants or other entities, please read [12].

3.3 Type and Type Expression
Type is a very important entity in PVS specification language. The proving of a theory depends greatly on the types use in the theory. A theory can be unprovable if the type is wrong.

Other than primitive types, there are subtypes, function type, enumerations, tuples and records. Users can also create their own type using type declarations or importing it from external theories.

Subtypes, function types, enumerations, tuples and records have Type Expression. Type expressions are expressions that are used when declaring a new type that is a subtype, function, enumerations, tuples or records. Type expression can be considered as a format in creating subtypes, function types, enumerations, tuples and records. Table 1 shows the format for subtypes, function types, enumeration, tuples and records.

<table>
<thead>
<tr>
<th>Type expression</th>
<th>General format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>[parameter type -&gt; return type]</td>
</tr>
<tr>
<td>Enumeration</td>
<td>{value1, value2, value3, …}</td>
</tr>
<tr>
<td>Subtype</td>
<td>{variable : type</td>
</tr>
<tr>
<td>Tuple</td>
<td>[element1, element2, element3, …]</td>
</tr>
<tr>
<td>Record</td>
<td>[# element1: type, element2: type, element3: type … #]</td>
</tr>
</tbody>
</table>

3.4 Expressions
PVS provide expressions that can be used in the declaration of subtypes or in the declaration of formulas. Some of the expressions are shown in Table 2.

<table>
<thead>
<tr>
<th>Expressions</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Expression that returns true or false</td>
</tr>
<tr>
<td>Numeric</td>
<td>Expression that involves numerals</td>
</tr>
<tr>
<td>Binding</td>
<td>Lambda, existential and universal</td>
</tr>
<tr>
<td>IF – THEN – ELSE</td>
<td>Selection expression similar in programming</td>
</tr>
<tr>
<td>LET-WHERE</td>
<td>Similar purpose as Binding express introduce to improve readability</td>
</tr>
<tr>
<td>Record</td>
<td>Assign value to records</td>
</tr>
<tr>
<td>Tuple</td>
<td>Assign value to tuples</td>
</tr>
</tbody>
</table>

More extensive information about the expressions can be obtained from [12].

4 PVS Metamodel
To create PVS metamodel, the main language
construct and the relationship among them are identified. The language constructs are later grouped into packages and the groupings follow the one used in [11]. There are five packages in the PVS metamodel as shown in Figure 3. PVSTheory package contains elements that make up a PVS theory, PVSTypeExpression package represents the general structure of PVS types, PVSType package contains representation of PVS types, PVSDeclaration package represents different types of declaration expression and PVSExpression package contains elements that represent different PVS expression.

Figure 4 shows the model that describes the concept of Theory. Basically what are shown in Figure 4 are the elements that exist in a theory. As mentioned in Section 2, a theory has two parts: theory and assumption. These two parts are represented in the model as Assuming and TheoryPart. Both of this part can have import clause that is represented by Importing. Theory can have parameters which can be a declaration of a new type or uninterpreted constant. In Figure 5, exporting clause is represented by Exporting. In PVS user can export the whole theory or certain elements in the theory.

ExportingTheory and ExportingNames represent the export of the whole theory and the export of certain elements respectively.

Figure 5 is the model that shows the different types of declarations in PVS. Hierarchy of TypeDecl represents the different type of declaring a new type. TypeRelatedDecl represent the declaration of language element that consist of a type expression such as variable, constant and judgement. Conversion inside the model represents the declaration of a conversion which a function that is used to convert between types. FormulaDecl represent the declaration of different kinds of formula.

Declarations usually consist of expressions and type expressions. Figure 7 shows the expressions in PVS and Figure 6 show the type expressions for function, enumeration, subtype, records and tuples. Function expression consists of parameter type and return type. Record expression consists of elements which are in the form of uninterpreted constant declaration and tuple expression consists of types of the tuple elements.
5 Using the PVS Metamodel

The PVS metamodel contains elements that represent the main language construct of the language. In translating OCL to PVS, PVS metamodel is used as the result model and the transformation will transform one OCL language construct to a PVS language construct.

In the transformation element in the metamodel that represent the OCL language construct and the element in the PVS metamodel that represent the PVS language construct will be used. As an example, Figure 8 shows the transformation of OCL Invariant to PVS Assumption.

Constraint is an element in OCL metamodel that represent OCL Invariant while AssumptionDecl represents PVS Assumption. The arrow shows the
direction of the transformation while the note contains description of how the transformation will be done. In the case of transforming OCL Invariant to PVS Assumption, the name of the assumption will be the name of the class that the invariant constraint and ended with ‘-invariant’. The assumption expression will be the expression of the OCL Invariant.

Class.invariant, class.name and class.invariant.expression show that the Class element in OCL metamodel has invariant and name properties; and invariant has expression properties.

\[
\text{Class.invariant} = \text{(aName:ASSUMPTION expression)}
\]

where

\[
aName = \text{class.name+"-invariant"}
\]

\[
\text{expression} = \text{class.invariant.expression}
\]

Figure 8: Transformation of Invariant to Assumption

6 Conclusion

In this paper, we have proposed a metamodel for PVS specification language. The metamodel is presented in UML Class diagram notation. This allows our metamodel to be easily readable by whomever that can read UML class diagram.

Detail explanation of PVS specification language is also presented in this paper. Although the explanation is not complete, it is sufficient for readers that are unfamiliar with the language to get the flavour of PVS.

The detail explanation of PVS specification language is also important in guiding the readers to understand the metamodel. The metamodel is also structured into packages that follow the elements of PVS specification language.

We have also presented our motivation in creating the PVS metamodel. We plan to use the metamodel for our future work in transforming various models into PVS theories in the hope of verifying and validating those models.

References: