On CSTs, ASTs and Model-Driven Editors

Edward D. Willink[1]**

Eclipse M2M/QVT Declarative Project ,
`ed__at__willink.me.uk`

Abstract. The use of multiple passes to compile a source language by converting text to an Abstract Syntax Tree is well known. The use of meta-models to define the intermediate models between passes is not uncommon. In this paper we show how these meta-models can be re-used to define the behavior of a semantically sensitive editor for the source language. We demonstrate this using OCL as an example, although the approach is language independent and was first applied to the QVT languages.

Key words: modeling, meta-modeling, model editing, editor definition, model parsing, OCL, QVT, CST, AST

1 Introduction

Development of textual programs traditionally used an

– edit
– save
– compile
– note down error line
– find error line

cycle.

Early development environments provided automatic save and automatic navigation to the source text by selecting a line in the compiler error display. This simplified development to an edit then compile cycle.

Modern Integrated Development Environments for languages such as Java integrate the full cycle so that error indicators appear automatically in the source editor after only a short delay.

The rapid turn around is possible because the compiler is integrated with the editor and because modern computers are fast enough to perform regular recompilation. With the compiler accessible by the editor, additional facilities, such as outlines, show-all-references, completion proposals and refactorings can be provided that exploit the semantic analysis that the compiler has performed. We therefore refer to this integrated capability as a semantically sensitive editor.

The significant productivity gains of a semantically sensitive editor become very evident when it is necessary once again to use a semantically ignorant editor.

** The author works for Thales Research and Technology (UK) Ltd, Reading, England.

We would therefore like to provide semantically sensitive editor capabilities for any language without the cost of developing a custom solution. In this paper we will show how such an editor can be provided at very little cost by re-using the compiler models and meta-models that drive the editor. We will occasionally refer to our Eclipse-based implementation that uses a common framework to support editors for OCL and the QVT languages.

Almost automatic production of a rich editing environment should provide further encouragement to the development of Domain Specific Language tooling so that production of the parser meta-models upon which our approach depends can also be assisted by better automation.

In Section 2 we will first provide the background to the contributions of this paper by following a very simple example through the traditional compilation stages. This should clarify the roles of the different compiler models, so that in Section 3 we can show how navigation between these models is used to solve problems of error message generation and to provide semantically sensitive editing behaviors. In Section 4 we will show how these models contribute to the editor displays and behavior. Then in Section 5 we introduce the language insensitive Editor Definition Meta-Model so that the editor behavior can be defined in terms of Abstract Syntax Tree (AST) and Concrete Syntax Tree (CST)[3] meta-classes. In Section 6 we discuss some practical aspects of the OCL[10] and QVT[11] editor tooling. Finally we discuss future and related work before concluding.

1.1 Working Example

Throughout this paper we will use a very simple example comprising an OCL definition constraint upon the meta-model shown as a UML Class Diagram in Figure 1. This meta-model comprises just the single class `Node` with an `age` property and any number of `child` `Node`s.

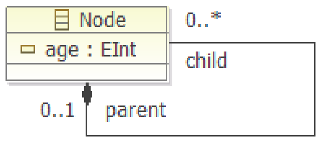


Fig. 1. Example Meta-Model for the `node` package

The example OCL constraint shown in Figure 2 `def`ines an `ages` property with the type `OrderedSet(Integer)` and a value computed by visiting each `child` to ascertain its `age` and then using the `asOrderedSet` built-in method to convert the resultant `Sequence(Integer)` to the required ordered set.

[3] the CST is called a parse tree in traditional texts[1]

```
context node::Node
def: ages : OrderedSet(Integer) = child.age->asOrderedSet()
```

Fig. 2. Example OCL Constraint

2 Parsing Models

Translation of source text to an AST typically involves the activities and representations shown in Figure 3. The sequence of characters for the source language

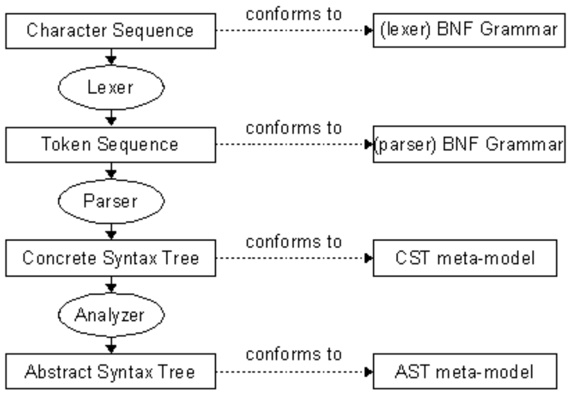


Fig. 3. Parsing Models and Activities

text are grouped into tokens defined by the lexer grammar. These tokens are in turn grouped into constructs that satisfy the parser grammar to form the CST. Semantic analysis of the CST yields the AST.

CST and AST comprise a tree of nodes that conform to their corresponding meta-models.

For simple languages, little or no semantic analysis may be necessary. It may then be convenient to merge semantic analyzer and parser avoiding a distinct CST stage.

In this paper we are primarily concerned with languages that have substantial semantic analyzers, although the techniques remain applicable to simple languages as well.

2.1 The Concrete Syntax Tree (CST)

The Concrete Syntax Tree for our running example is shown in Figure 4.

It has an unsurprising structure in which the root node is an `OCLDocumentCS` (for the document). The root has a `PackageDeclarationCS` child (for the default package), which has a `ClassifierContextDeclCS` child (for the `context`

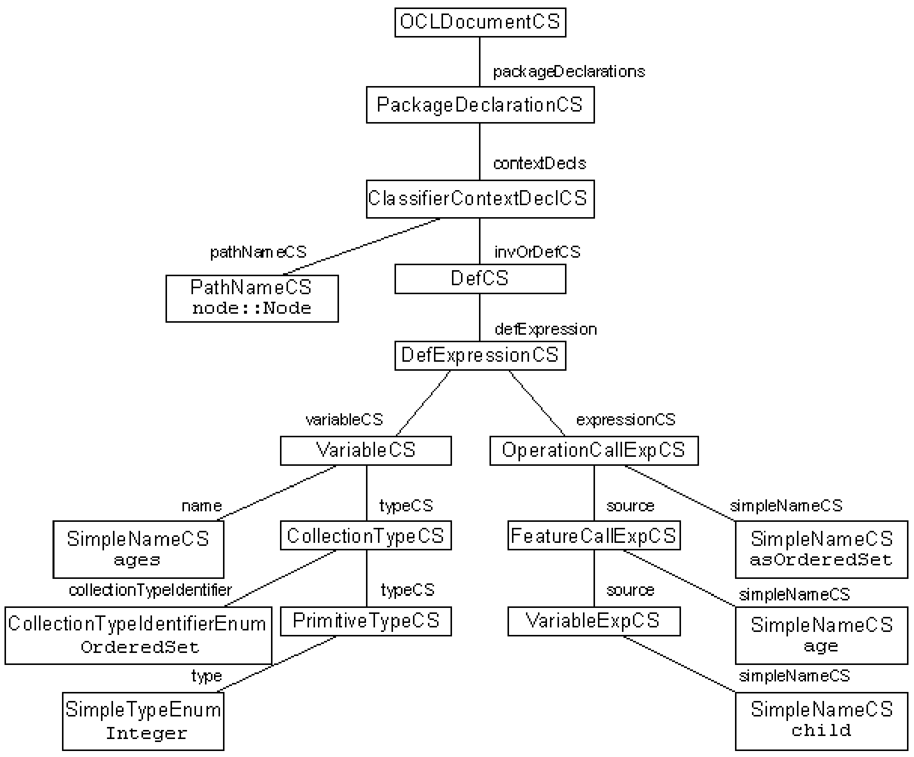


Fig. 4. Example Concrete Syntax Tree

`node::Node`). This in turn has a `DefCS` child (for the `def`inition constraint). Property values are conveyed by nodes such as `SimpleNameCS`, `PathNameCS` or `SimpleTypeEnum`.

Note that the tree structure prevents sharing of sub-trees. The variable definition is therefore provided by the `VariableCS` and its children which contain a definition of the `OrderedSet(Integer)` collection type.

2.2 The Abstract Syntax Tree (AST)

The CST is a direct representation of the source text. The CST does not therefore normalize syntactic sugar or resolve the intent or validity of the many text nodes (`SimpleNameCS` or `PathNameCS`). These problems are resolved by the semantic analyzer, so that the AST provides a simpler normalized and validated representation of the intent.

In common with many Object Oriented languages, OCL

– allows an implicit `self` variable to be omitted.
– uses the dot operator to navigate from an object to a named child or children

Unlike other Object Oriented languages, OCL

– uses the arrow operator to invoke a built-in operator upon a collection of objects
– uses the dot operator applied to a collection of objects as syntactic sugar for the built-in `collect` iteration operator.

Translation of the CST to an AST removes the syntactic sugar, resolves textual references to objects and creates an object structure complying with the AST defined by the OCL specification[10].

Removal of the syntactic sugar treats the source from Figure 2 as Figure 5.

```
package node

context Node
def: ages : OrderedSet(Integer) =
    self.child->collect(temp1 : Node | temp1.age)->asOrderedSet()

endpackage
```

Fig. 5. Sugar-free Example OCL Constraint

The scoped classifier declaration is repartitioned to place the scope in the package declaration. The implicit `self` variable is made explicit. The implicit `collect` iteration is replaced by an explicit iteration with an explicit iterator variable declaration.

The resulting AST shown in Figure 6 is significantly different to the CST. Indeed the AST is not a tree at all. Once the unresolved text strings of the CST are resolved to reference appropriate objects, the dashed lines in Figure 6 make the term graph more appropriate.

In the CST, the (`DefCS`) definition is a child of the (`ClassifierContextCS`) class which is a child of the (`PackageDeclarationCS`) package. The CST exists solely to support the pragmatics of a particular concrete syntax, so that potentially any convenient structure is possible[4].

The AST meta-model is rigidly defined so that it can be used by a variety of tools. OCL[10] defines constraints upon a MOF[9] (or UML) model, but OCL may not modify the MOF model. An OCL `Constraint` cannot therefore be a child of a MOF `Class`. Instead, an OCL AST comprises a set of `Constraints` that refer to the MOF model that they constrain. This inevitably requires a structural rearrangement between AST and CST representations.

In the CST, collection types are declared each time they are required. In the AST, a single variant of each collection type is defined within the `collections`

[4] Since the OCL specification refers to a specific CST, it is advisable to follow the specification quite closely

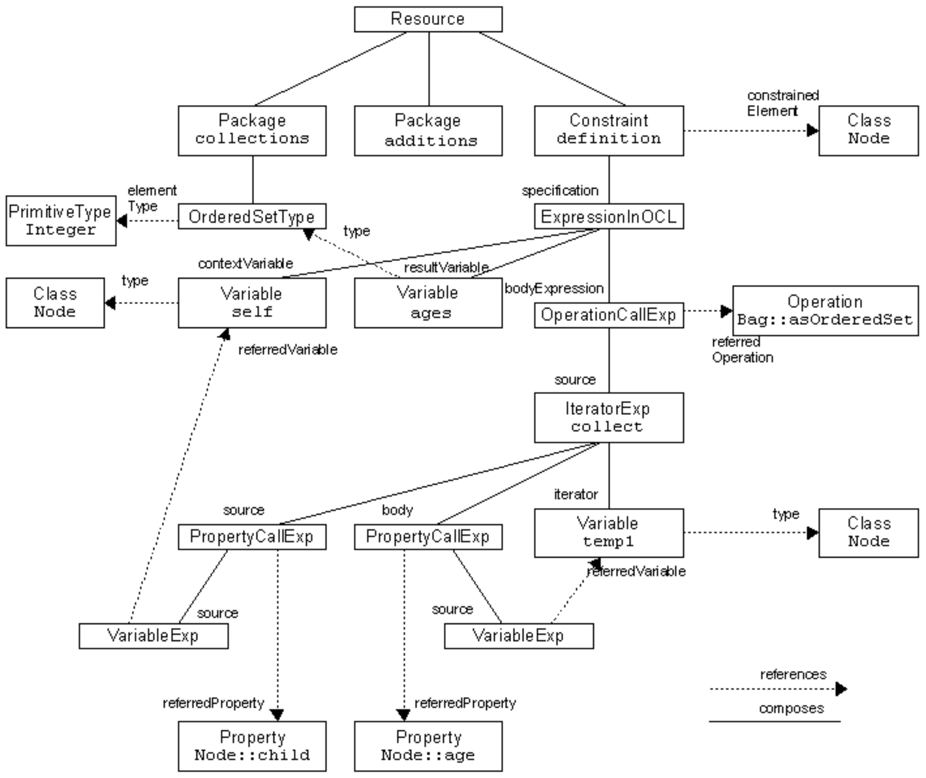


Fig. 6. Example Abstract Syntax Tree

package and then referenced each time it is required. This again requires a different arrangement of nodes in the AST and the CST.

In summary, significant differences between AST and CST arise from

- expansion of syntactic sugar
- resolution of name strings to object references
- structural incompatibility

3 Navigation

The previous section used our simple example to demonstrate the intermediate models that a typical compiler would use to translate source text to an AST.

The four levels of Character, Token, CST and AST representation from Figure 3 are shown again in four rows across Figure 7. The composite model (shown at the left) comprises a sequence of elements, or a tree of elements, (shown in the middle) that comply with either a grammar or with the meta-classes of a meta-model (shown on the right).

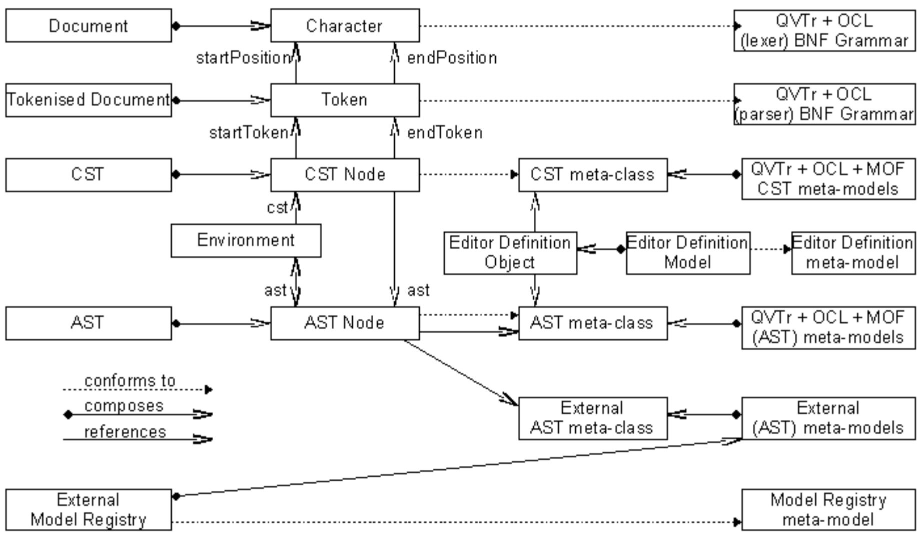


Fig. 7. Model-Driven Editing Models

The support for an extended language such as QVT Relations[11] is demonstrated by the use of extended grammars and meta-models. With appropriate lexer/parser tooling such as LPG[17], grammar extension can be achieved by import rather than a rewrite. Extended meta-models are naturally supported by tooling that supports multiple packages.

We will now introduce the remaining additional models that enable a variety of important problems to be resolved that require careful navigation from some context in one model to obtain information from a corresponding context in another.

3.1 Environments

Semantic references cannot in general be resolved by a single CST traversal, since the definition of a forward reference may not be encountered before the reference needs resolving. It is therefore convenient to use two passes for semantic analysis. A first pass places the definitions in a 'symbol table' where the second pass can resolve them. In practice, language-specific scoping rules make a global symbol table impractical. It is better to establish a hierarchy of semantic environments for important structural nodes such as MOF[9] `Class`es or OCL `LetExp`s so that symbol lookups respect the language semantics.

Since the AST meta-model is precisely defined by the OCL specification, it is not possible to put the symbol table information directly in the AST node. Therefore, when the first semantic analysis pass creates a partially defined AST node, a semantic `Environment` node may also be created to extend the AST node[5]. The second semantic analysis pass completes the definition of each AST node, using the hierarchy of environments created by the first pass to perform symbol name lookups.

3.2 External Models

Non-trivial applications re-use meta-models and so a mechanism is required to support location and use of those meta-models. The Model Registry[8] fulfills this role by providing a set of registrations between the names used to access a (meta-)model and the (meta-)model that is accessed. The semantic analysis therefore incorporates access to the registry as part of the semantic lookup.

A variety of different meta-model 'repositories' can be supported. For instance, models may be installed as plug-ins within Eclipse or defined as EMOF[9] or Ecore[13] files.

3.3 Backward Model Navigation

For compilation purposes, it is sufficient to convert source text files via tokens, CST nodes and Environments to the required AST nodes. The intermediate resources can be released promptly.

However, whenever errors arise it is helpful to display the errors in the context of the source program; possibly by enumerating the file name and line in a console error message or by highlighting the problem area in a source text editor. This is readily achieved by ensuring backward traceability between the models.

- each AST node identifies the CST node from which it was created
- each CST node identifies the first and last token from which it was constructed
- each token identifies the first and last source character it represents.

With this extra information available, error message code can navigate from AST node to CST node to first and last Tokens to first and last Characters. The location and/or content of the character range can then be used to make an error message much more relevant.

Unfortunately retaining the intermediate representations makes it difficult to release any memory resources until the translation and error message generation is complete.

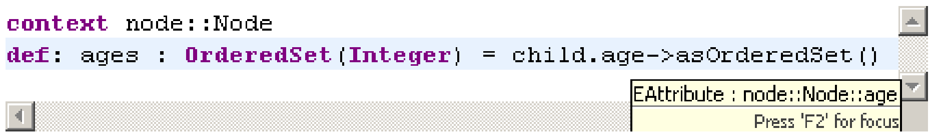
Helpful source context messages may also be useful in other applications such as a debugger, profiler or code analyzer.

3.4 Forward Model Navigation

The opposite, forward, navigation is required for advanced semantic editing facilities that answer some query at the source text level with information from the compiled AST.

Since semantic queries may be made at any time, it is unfortunately necessary to retain almost all intermediate resources for the duration of the editor session.

[5] In the Eclipse Modeling Framework[13] this `Environment` extension can be installed as an Adapter of the AST node.



```
context node::Node
def: ages : OrderedSet(Integer) = child.age->asOrderedSet()
```

Fig. 8. Example Hover Text

Hover Text In order to provide the hover text message shown in Figure 8 explaining the semantic interpretation of the `age` text string in our example, we must

– find the character position for the mouse pixel position
– find the narrowest CST node whose tokens span the character position
– find the AST node constructed from the CST node

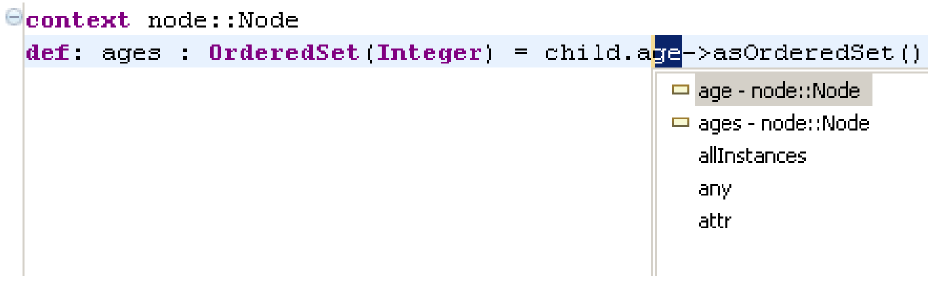
The example position corresponds to a `Property` AST node that contains the required semantic explanation that the mouse is hovering over the text string `age` that represents the `age` property[6] of the `Node` class of the `node` package.

The pixel to character conversion is performed by the Eclipse `StyledText` editor class.

The character to CST node conversion is performed by a top-down search of the CST that considers only those CST nodes whose first and last tokens span the required position. No child CST node covers a wider span than its parent, so the search can be bounded and the deepest CST node is the required result.

The CST to AST node conversion is resolved by an additional `ast` property of each CST node that is initialized when the AST node is created from the CST node.

Note that this conversion would be much less efficient without the CST, since the structural incompatibilities between AST and the source text confound an efficient search.



```
context node::Node
def: ages : OrderedSet(Integer) = child.age->asOrderedSet()
```

Fig. 9. Example Completion Proposals

[6] A MOF value `Property` is called an `EAttribute` in the Eclipse Modeling Framework's Ecore representation

Completion Proposals A more sophisticated forward navigation is required in order to provide the completion proposals shown in Figure 9, which is the response to a request for completion of the `a` in `age`.

A distinct algorithm is used for each kind of token; strings and numbers offer simple completion proposals based on other strings or numbers that appear in the source text.

The algorithm for an identifier such as `age` involves locating the AST node as above for a hover text explanation. The AST node is then used as a template for alternative proposals. All model usages of the selected node are examined to identify the meta-model relationships for which the selected node is a valid target. The target types of these meta-model relationships provide a type template for proposals. Each model node that satisfies the type template and shares a common name prefix with the required completion offers a language independent completion proposal. Language-specific rules may be used to expand the type template or to trim the list of proposals.

The above algorithm can yield good results provided the AST is able to provide a good type template. However, in the presence of incomplete or erroneous source text, it may not be possible to provide an accurate AST. These problems may be mitigated by ensuring that a plausible CST and AST are produced making use of 'invalid' place-holders for problem areas. Provided these place-holders have consistent meta-class types, the place-holders can still guide the completion proposals.

Non-trivial languages tend to provide a variety of similar syntaxes that produce distinct AST structures. Generation of proposals from one possible AST structure may therefore omit proposals for another possible structure. At present language-specific intervention is required to resolve these limitations. In the future, with the AST to CST semantic analysis defined by a declarative transformation, it may be possible to access the BNF grammar and automatically deduce similar structures in a language independent fashion.

3.5 Navigation Details

It is not always obvious how the CST to AST node mapping should be established.

In the case of an AST node such as a synthesized `CollectionType` declaration, which is shared by many declaring CST nodes, it is convenient for the AST node to identify the CST node that first caused the AST node to be created. This avoids unhelpful error messages that merely identify that there is an error somewhere.

In the presence of syntactic sugar, a CST node may be the source of multiple AST nodes. In the forward direction, the CST node is associated with the shallowest AST node to ensure that the selection covers the entire CST construct. In the reverse direction many AST nodes may indirectly reference the same CST node via their Environment extension.

In the case of a structural incompatibility, the shallowest CST and AST nodes in the incompatibility are associated.

4 Editor Views

The OCL Editor views shown in Figure 10 show the source text for our example in the main text editor view. Below this are two alternative views, one of the CST to the left, and the other of the AST to the right. At the bottom, a Properties View gives detailed information about a particular selection.

The text view may be the only view required by an OCL developer. However, the additional views may be useful to aid navigation or to acquire a more detailed insight into the CST or AST behavior. The particular layout of in Figure 10 is chosen to suit a portrait layout in this paper.

4.1 Text View

The text view shows the source text. The view exploits standard Eclipse editor tooling, so the basic editing functionality is very much as an Eclipse user would expect. Additional familiar capabilities are provided by the Eclipse Integrated Development Environment Meta-tooling Platform (IMP)[15].

Syntax coloring is used to highlight distinct syntax elements such as keywords, strings and comments.

The left and right borders of the editor, respectively, show a line-based and document-based error overview. Figure 11 shows some error markers.

A folding icon is shown at the left hand end edge of the `context` construct. Its body can be folded away behind its top line.

The cursor position in the text view causes a corresponding selection in the CST Outline, AST Outline and Properties views.

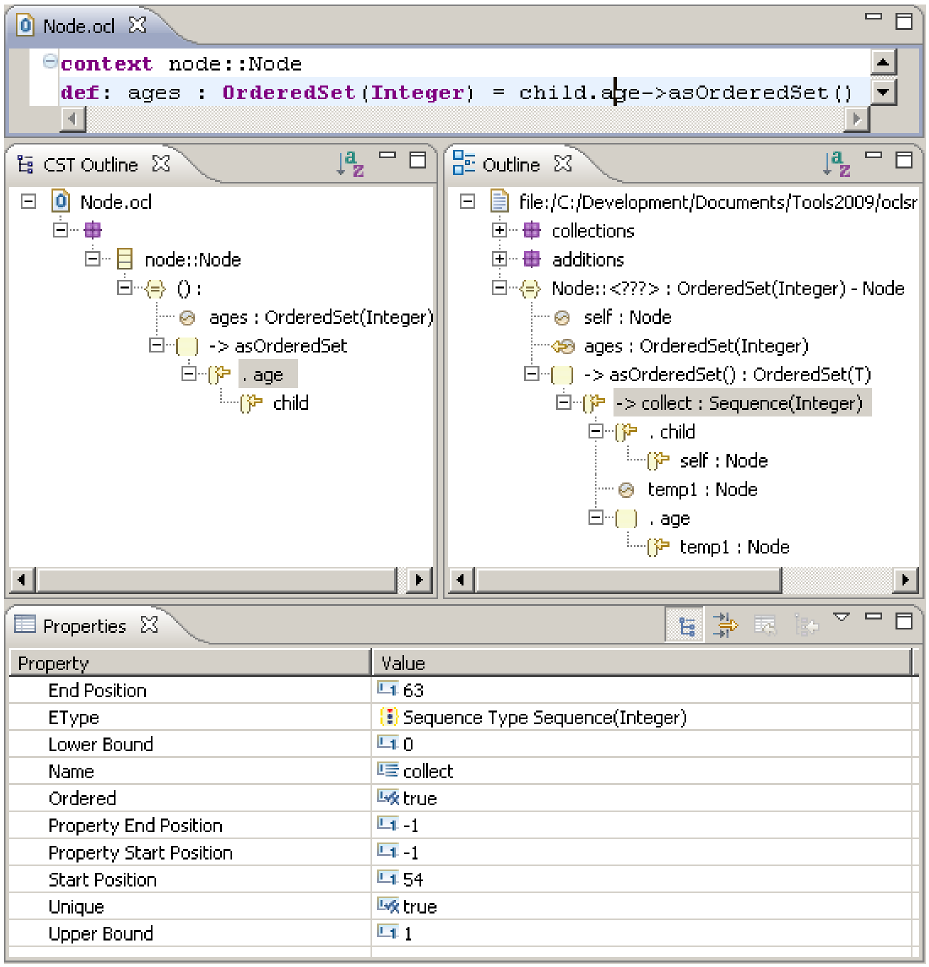
4.2 CST Outline View

The CST Outline provides a compressed tree-based view of the CST shown earlier in Figure 4. On each line:

– a folding icon controls whether child elements are shown indented below the parent.
– a context icon variously shows the CST node type and/or the CST node role with respect to its parent
– a text description identifies salient properties of the CST node.

Selecting an item in the CST Outline causes a corresponding selection in the text, AST Outline and Properties views.

The CST outline tends to be more useful than the AST outline for OCL, since the OCL AST has an unhelpfully flat set of Constraints for which key details are missing for OCL 2.0 property contexts.



```
context node::Node
def: ages : OrderedSet(Integer) = child.age->asOrderedSet()
```

```
Node.ocl
    node::Node
      () :
        ages : OrderedSet(Integer)
        -> asOrderedSet
          . age
            child
```

```
file:/C:/Development/Documents/Tools2009/oclsr
  collections
  additions
  Node::<???> : OrderedSet(Integer) - Node
    self : Node
    ages : OrderedSet(Integer)
    -> asOrderedSet() : OrderedSet(T)
      -> collect : Sequence(Integer)
        . child
          self : Node
        temp1 : Node
        . age
          temp1 : Node
```

Property	Value
End Position	63
EType	Sequence Type Sequence(Integer)
Lower Bound	0
Name	collect
Ordered	true
Property End Position	-1
Property Start Position	-1
Start Position	54
Unique	true
Upper Bound	1

Fig. 10. OCL Editor for the Example

4.3 AST Outline View

The main (AST) Outline provides a similarly compressed tree-based view of the AST shown earlier in Figure 6.

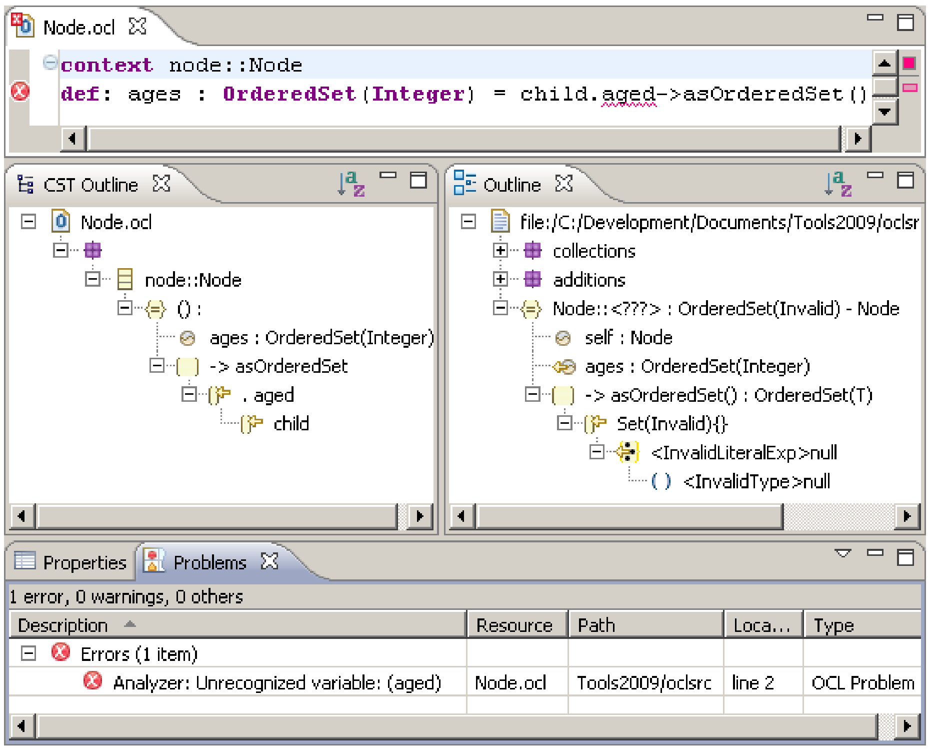
Selecting an item in the AST Outline causes a corresponding selection in the text, CST Outline and Properties views.

The AST outline tends to be more useful than the CST outline for the QVT languages, since they have a rich hierarchical structure that is similar in both CST and AST but the AST is more compact.

4.4 Properties View

The Properties View provides a detailed display of the values of a variety of named properties of a particular selection within the Text, CST or AST Outline views. The displayed properties are determined by the properties of the meta-class of the CST node selected in CST outline, or of the AST node directly selected in the AST outline or indirectly selected in the Text View.

4.5 Error Markers and the Problems View



```
context node::Node
def: ages : OrderedSet(Integer) = child.aged->asOrderedSet()
```

```
Node.ocl
    node::Node
      () :
        ages : OrderedSet(Integer)
        -> asOrderedSet
          . aged
            child
```

```
file:/C:/Development/Documents/Tools2009/oclsr
  collections
  additions
  Node::<???> : OrderedSet(Invalid) - Node
    self : Node
    ages : OrderedSet(Integer)
    -> asOrderedSet() : OrderedSet(T)
      Set(Invalid){}
        <InvalidLiteralExp>null
          ( ) <InvalidType>null
```

1 error, 0 warnings, 0 others

Description	Resource	Path	Loca...	Type
Errors (1 item)				
Analyzer: Unrecognized variable: (aged)	Node.ocl	Tools2009/oclsrc	line 2	OCL Problem

Fig. 11. OCL Editor for the Example With an Error

When the source text contains an error as in Figure 11, the editor shows the error in four ways.

- Marker at left hand side of line
- Marker at right hand side of document
- Underline of erroneous character range
- Problem summary in the Problems View

Hovering over any of the error markers causes a detailed error message to be displayed.

The error demonstrates an important difference between CST and AST outlines. The mis-spelling of `age` as `aged` has minimal impact on the CST since the CST precedes semantic validation. An accurate AST cannot be produced so a variety of invalid place-holder nodes are used to maximize the utility and validity of the AST. This ensures that as much of the AST-guided editor functionality as possible remains functional despite the semantic error.

5 Editor Definition

In order to solve language-specific problems without making our editor framework language dependent, we introduce the Editor Definition meta-model shown in Figure 12. The `EditorDefinition` for some language comprises customization of `Behavior`s for a variety of nodes identified by their Ecore[13] meta-class. Occasionally it is also useful to configure the extra-model Java nodes such as a `File` that form part of the host Platform User Interface.

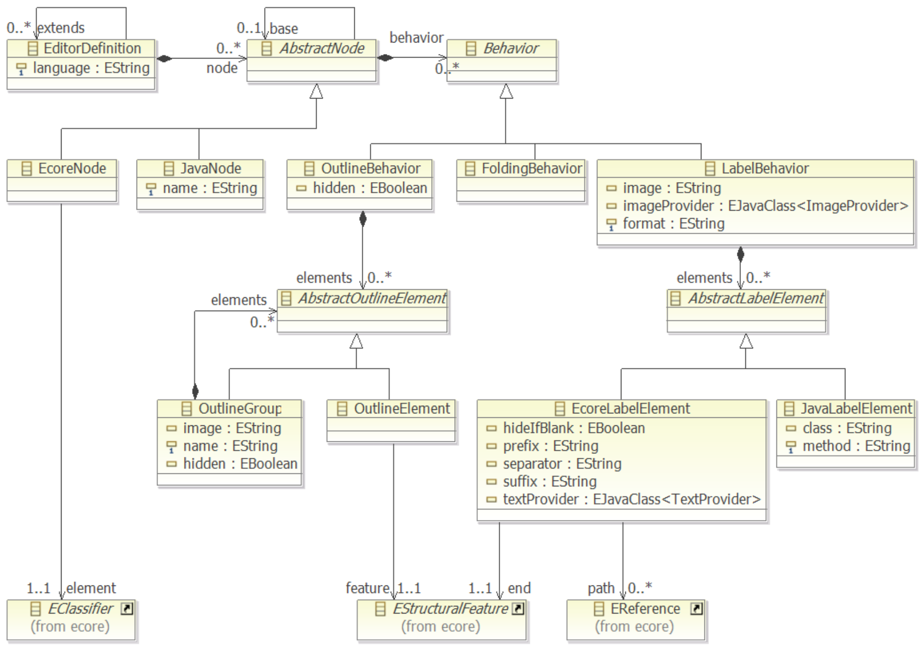


Fig. 12. Editor Definition Meta-Model

The Editor Definition Meta-Model exploits the CST and AST meta-models to isolate language-specific definition or customization of editing behavior in a single language-dependent model. This Editor Definition model is editable by its corresponding Editor Definition editor. The editor behavior can be changed while the editor is running by editing the Editor Definition Model.

Editor Definitions support extension, so that the definition of, for instance, a QVTr editor may import and so re-use the definition of an OCL editor.

The Editor Definition content is optional. Default behaviors are provided throughout. Editor Definitions provide customization by associating alternative behaviors with the selected language-specific AST or CST meta-classes. These behaviors are activated whenever a corresponding AST or CST node is processed by the editor framework.

Folding Constructs that offer folding are configured by a `FoldingBehavior` added to the `EcoreNode` referencing the relevant CST meta-class.

The example in Figure 10 shows that a `FoldingBehavior` has been associated with the `ClassifierContextDeclCS` meta-class.

Labels The default formatting of the text of an outline line comprises the meta-class name within angle brackets. A more readable presentation and an icon may be provided by a `LabelBehavior` that references the relevant AST or CST meta-class. The `LabelBehavior` may have a format control string and a variety of `LabelElement`s that navigate the model to obtain required contributions. These model navigations are well-defined, so it is comparatively straightforward to install the appropriate observers to ensure that the outline updates accurately in response to arbitrary complex model and meta-model changes. `LabelBehavior` and `LabelElement` properties support hiding unwanted text and supplying suitable prefixes, separators and suffixes for lists of contributions.

The `"ages : OrderedSet(Integer)"` label for the `VariableCS` in the Figure 10 CST Outline View shows the use of a `LabelBehavior` with a `"{0} : {1}"` format string and with the `VariableCS.name` and `VariableCS.type` `LabelElement` children to provide the format inserts.

Outlines The default presentation of a node in the outline comprises the label of the parent with an indented presentation of composed children below it. Children are displayed in model serialization order. A more appropriate presentation may be provided by an `OutlineBehavior`. This may omit children displayed in the parent's label, or group children by role, or impose a consistent ordering such as if/then/else.

The final outline in the Figure 10 (AST) Outline View is configured by defining an `OutlineBehavior` so that all instances of the `PropertyCallExp` meta-class show their `PropertyCallExp.source` as the sole child. The name of the `PropertyCallExp.referredProperty` is suppressed since it contributes to the `PropertyCallExp` label. The `PropertyCallExp.type` is also suppressed to avoid undue clutter. Suppressed properties can be viewed in the Properties View when required. The icon is also defined to suggest the use of a source to an expression.

6 Implementation

6.1 Language Independence

The running example in this paper uses OCL, since OCL may be familiar to many readers. The same editing framework has been used by the author to support semantically sensitive editors for the OCL, QVT Core, QVT Relations and KM3[7] languages. The framework has been re-used by Sánchez-Barbudo[7] to provide a similar editor for the QVT Operational Mappings language.

Since the editors exploit the CST and AST meta-models, the opportunities that the framework provides for language-specific overrides are not often required. For instance the syntax coloring is guided by the keyword and token definitions of the lexer. Therefore once the language-specific lexer and parser have been defined using LPG[17] and a semantic analyzer has been written, very little extra effort is required for the editor.

6.2 OCL Tooling Enhancements

In order to support the editing capabilities described above, the functionality of Eclipse MDT/OCL[14] has been enhanced to support the navigation from CST nodes forward to the AST node, and backward to the tokens. An option to use LPG's Backtracking Parser has been added to improve multiple error diagnosis and additional grammar rules were added to detect common errors. Support for grammar, parser and analyzer extension was provided so that the EssentialOCL[10] functionality could be re-used by the QVT languages.

6.3 IMP contributions

Use of the IMP tooling has provided many of the advanced editing facilities that were missing from the editors previously available from the Eclipse GMT/UMLX[16] project.

– incremental 'compilation' so that error markers update promptly
– folding
– hover text
– completion proposals

Use of the IMP tooling has also made implementation of some facilities much easier

– syntax coloring now uses the grammar definitions rather than independent code
– transient temporary files no longer need to be created

[7] A simple meta-model definition language[5]

6.4 Availability

The requisite enhancements to Eclipse MDT/OCL are present in the 1.3.0 development stream at M5. The editing functionality is therefore available to anyone with sufficient enthusiasm to obtain the latest versions of IMP, MDT/OCL and M2M/QVT Declarative. It is intended that a simple install will accompany the Eclipse Galileo release (3.5).

A complete download is also available by pointing the Eclipse Installer to the `http://download.eclipse.org/modeling/m2m/qvt_declarative/updates` update site.

6.5 Further Work

The information within the AST and CST provide potentially simple solutions to facilities that IMP does not yet support:

– Goto Definition
– Find Declarations
– Find References

Some other problems just require enhancement of the IMP support

– Distinction between warnings and errors

The Editor Definition currently supports customization of Folding, Label and Outline behaviors. It needs extension to cover all preferences and more advanced behaviors such as Quick Fixes and Refactoring.

7 Related Work

XText[2] provides auto-generation of lexer, parser and editor from a BNF grammar. This also operates in an Eclipse environment using Eclipse Modeling tools. For simple custom Domain Specific Languages for which no distinction between AST and CST is required, XText provides a very impressive capability. With AST similar to source text, generation of outlines and completions is straightforward. There is no need for an Editor Definition model. There is no ability to use an externally defined AST that differs from the automatically generated 'CST'.

TCS[6] also provides auto-generation of lexer, parser and a TGE editor. The source language is specified by text templates which are translated to a BNF grammar for parsing. The templates enable TCS to offer bidirectional capability since the templates include definitions of the pretty printing necessary for reverse parsing. However, it is not clear whether the template language or the project is sufficiently mature to compete with XText. As with XText, there is no ability to accommodate a significantly distinct or externally imposed AST. The TGE editor is driven by a model that defines the appearance of an outline.

Garcia[4] describes Gymnast and IDEalize, an Eclipse framework for building a text editor. Gymnast has the flexibility to handle distinct CSTs and ASTs for a custom DSL. IDEalize generates the corresponding IDE using an `.idegenmodel` file to control the generation process at compile-time, whereas we use an Editor Definition model that can be changed at run-time. The Gymnast framework is used to build the text editor for Emfatic[12].

An AST visualizer for Eclipse MDT/OCL is described by Garcia[3]. The visualization includes numerous keywords to try to make the AST readable. The Outlines shown in Figure 10 uses role icons to provide a more compact visualization. An alternate editor definition could be used to emulate Garcia's more verbose/readable form.

OCL LIB[18] is a subset OCL editor also based on Eclipse MDT/OCL, with perhaps more advanced features, but no overt model-driven capabilities. The primary development focus is on support of OCL libraries and support for regular patterns via text templates.

XEntrant[19] purports to be a model-driven XML editor, however it is not clear that this amounts to more than DTD/schema-awareness.

8 Acknowledgments

Many thanks to Christian Damus, the Eclipse MDT-OCL[14] project lead, for being so ready to accept enhancements to the OCL project, and for diligent revision of those enhancements.

Many thanks also to Adolfo Sánchez-Barbudo for useful feedback when re-using the framework to develop the QVT Operational Mappings editor[7].

And a final thank you to Victor Sanchez for helpful comments on a draft of this article.

9 Conclusions

We have reviewed the models and meta-models conventionally used for compilation in order to demonstrate how these models provide important information that may be re-used to support semantically sensitive editing in a language independent fashion.

Augmenting the models with backward navigation supports error indicators, and with forward navigation supports semantic queries.

Use of the CST and AST meta-models yields adequate language-independent solutions to semantically sensitive editing. Language-specific customization is available as an option rather than an onerous necessity.

A language-independent Editor Definition model has been introduced to support customization of editor behavior for language-specifics or personal preferences.

These contributions offer a significant reduction in the burden of providing a good quality editing environment for a Domain Specific Language.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques and Tools, Addison-Wesley, 1986
2. Friese, P., Efftinge, S., Köhnlein, J.: Build your own textual DSL with Tools from the Eclipse Modeling Project, Eclipse Corner Article, April 2008, URL: `http://www.eclipse.org/articles/article.php?file=Article-BuildYourOwnDSL/index.html`
3. Garcia, M.: How to process OCL Abstract Syntax Trees, Eclipse Corner Article, June 2007, URL: `http://www.eclipse.org/articles/article.php?file=Article-HowToProcessOCLAbstractSyntaxTrees/index.html`
4. Garcia, M., Sentosa, P.: Generation of Eclipse-based IDEs for Custom DSLs, Software Systems Institute (STS), Technische Universität Hamburg-Harburg, Germany, 2008 URL: `http://www.sts.tu-harburg.de/%7Emi.garcia/SoC2007/draftreport.pdf`
5. Jouault, F, and Bézivin, J: KM3: a DSL for Metamodel Specification. In: Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037, Bologna, Italy, pages 171-185. 2006. URL: `http://www.sciences.univ-nantes.fr/lina/atl/www/papers/KM3-FMOODS06.pdf`
6. Jouault, F, and Bézivin, J: On the Specification of Textual Syntaxes for Models. Eclipse Modeling Symposium at the first Eclipse Summit Europe, Esslingen, October 11-12, 2006. URL: `http://www.eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium4_TextualSyntaxesForModels.pdf`
7. Sánchez-Barbudo, A., Sánchez, E.V., Roldán, V., Estévez, A., Roda, y.J.L.: Providing an open Virtual-Machine-based QVT implementation, Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos, Vol. 2, No. 3, 2008, URL: `http://www.sistedes.es/TJISBD/Vol-2/No-3/articles/DSDM-08-sanchez-barbudo-pov.pdf`
8. Willink, E.: Proposal for and Prototype of a Model Registry, Eclipse GMT/UMLX Project Technical Report, 2007, URL: `http://www.eclipse.org/gmt/umlx/doc/MDD-TIF07/MDD-TIF07-ModelRegistry.pdf`
9. Meta Object Facility (MOF) Core Specification, OMG Available Specification, Version 2.0, OMG Document Number: formal/06-01-01, January 2006, URL: `http://www.omg.org/spec/MOF/2.0/PDF`
10. Object Constraint Language, OMG Available Specification, Version 2.0, formal/06-05-01, May 2006, URL: `http://www.omg.org/spec/OCL/2.0/PDF`
11. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0, OMG Document Number: formal/2008-04-03, April 2008, URL: `http://www.omg.org/spec/QVT/1.0/PDF`
12. Emfatic Project Wiki, URL: `http://wiki.eclipse.org/Emfatic`
13. Eclipse Modeling Framework Project (EMF) Home Page, URL: `http://www.eclipse.org/modeling/emf`
14. Eclipse Model Development Tools (MDT) Object Constraint Language (OCL) Component Home Page, URL: `http://www.eclipse.org/modeling/mdt/?project=ocl`
15. Eclipse IDE Meta-tooling Platform (IMP) Home Page, URL: `http://www.eclipse.org/imp`
16. Eclipse GMT/UMLX Home Page, URL: `http://www.eclipse.org/gmt/umlx`

17. LALR parser generator (LPG) Project Home Page, URL: `http://sourceforge.net/projects/lpg`
18. OCL LIB Editor, SQUAM, Quality Engineering Research Group, University of Innsbruck, URL: `http://squam.info/ocleditor`
19. XEntrant, TreeStages, URL: `http://www.treestages.com`