

OCL Tools: Status and Perspectives

Miguel Garcia, A. Jibrán Shidqie
Institute for Software Systems (STS)
Technische Universität Hamburg-Harburg (TUHH),
Hamburg, Germany

Abstract

With the availability of infrastructural support for OCL (Object Constraint Language) provided by MDT OCL, the next step consists in simplifying the authoring of OCL specifications with feature-rich tools, as part of the more general task of model authoring. The capabilities of the OCL Tools project are described (currently they comprise an OCL \rightarrow Java compiler and a text editor for OCL). A discussion of areas for improvement and related work follows (refactoring of OCL expressions, detecting *code smells*, analyzing and optimizing OCL expressions). These ideas for future work fall under the umbrella of *model compilers*, essential elements of Model-Driven Software Engineering.

1 OCL Tools: What's inside

OCL Tools is a recently added component to the Model Development Tools (MDT) Project aiming at providing first-class support to modelers working with specifications containing expressions written in OCL, the Object Constraint Language [23]. Such support includes editing, refactoring, code generation, execution, and interactive debugging of the OCL constraints given for some underlying (Ecore or UML2) class model. The functionality of OCL Tools builds upon the MDT OCL component, and has been seeded with two initial contributions:

1. an OCL \rightarrow Java compiler, that as of now takes as input an `.ecore` and a textual file containing OCL constraints (handling UML2 is in the To-do list). This compiler extends EMF code generation, producing Java 5 source code with side-effects-free methods for the given OCL invariants, pre- and postconditions, and derived attributes and operations.
2. an OCL text editor, supporting usability features such as AutoCompletion, navigation by means of hyperlinks, structured views (for the outline of a document and for the Abstract Syntax Tree of an OCL constraint), among others. Although MDT OCL itself is completely fluent in both Ecore and

UML2, only the Ecore binding is supported by the OCL text editor at this time.

After a quick overview in the next section of the OCL compiler, most of this article focuses on introducing the OCL text editor, mentioning ideas for improvement along the way. The last two sections (“Refactoring of OCL expressions” and “Conclusions and Future Work”) sketch areas where the OCL Tools implementation is lacking. References are provided to related work, so as to ease the learning curve to go from user to contributor of OCL Tools.

2 OCL Compiler

In a nutshell, the compiler translates expressions like that in Listing 1 into what is shown in Listing 2:

Listing 1: An OCL invariant

```
context LoyaltyAccount
inv invariant_points :
    self.points > 0
    implies self.transactions->exists( t | t.points > 0 )
```

Listing 2: LoyaltyAccount’s invariant_points translated into Java

```
public boolean invariant_points() {
    Boolean implies1 = (this.getPoints() > 0);
    if (!(implies1)) {
        implies1 = Boolean.TRUE;
    } else {
        Boolean exists2 = false;
        for (RandL.Transaction t :
            org.eclipse.oc1.util.CollectionUtil.asSet(
                this.getTransactions())) {
            if (!exists2) {
                exists2 = (t.getPoints() > 0);
            }
        }
        implies1 = exists2;
    }
    return implies1;
}
```

The examples in this section refer to the Royal & Loyal class model reproduced for ease of reference in Appendix A. Although the Java snippet above does not show it, code following the EMF Validation Framework can also be generated, as can be seen in the larger example at the end of this section.

The combined expressive power of Ecore + OCL allows capturing a sizable amount of development requirements in a declarative manner:

- data modeling aspects can be expressed as a class model further constrained by OCL invariants

- a number of functional requirements can be specified as operation pre- and postconditions, together with side-effects-free queries

In order to get the most out of these (laboriously prepared) specifications an automatic translation into Java is needed. The usual pros and cons of compilation vs. interpretation apply: interpretation is more flexible, in that ad-hoc expressions known only at runtime can be evaluated over the current object population. In contrast, compiled code is twice as fast on average over interpreted code (if you think you can do better, go ahead and join OCL Tools :) Actually, both interpreted and compiled code rely on the MDT OCL implementation of functions in the OCL Standard Library, thus explaining the relative good performance of interpretation.

The patterns exhibited by the resulting code are:

1. OCL preconditions and postconditions are translated into Java `assert` statements
2. OCL `define` and `derive` statements are translated into getters (for properties) and into method bodies (for operations)
3. OCL invariants are translated into dedicated Java methods. It is the responsibility of the user to invoke them at times deemed appropriate.

If you want to enjoy the best of both worlds, tutorials on how to interpret ad-hoc OCL constraints are available:

1. for Ecore, the an Eclipse Technical Article *Implementing Model Integrity in EMF with EMFT OCL* [7] by Christian W. Damus, with an exemplary realization using custom JET templates.
2. for UML2, at http://wiki.eclipse.org/index.php/MDT_1.0_New_and_Noteworthy#OCL_Integration

The internals of the OCL Compiler for EMF are described in a paper [16] available online here. The rest of this subsection will focus instead on the user-view, including a longer example. Getting up to speed with the compiler does not require knowledge about OCL, as two case studies are provided both in source (i.e., `.ecore` + `.ocl`) and compiled form: Royal & Loyal and JPQL.

1. Royal & Loyal is a model of a company managing loyalty programs for customers, it's the running case study in the OCL book by Warmer and Kleppe [23]
2. the second model falls in the Model-Driven Development category, it is the metamodel of the Java Persistence Query Language, the object-oriented query language standardized for Object-Relational Mapping (ORM) persistence by JSR-220. A discussion of why it's a *good thing* to have such language metamodels (in terms of the JPQL example) appears in [13]. A metamodel of BPEL 1.x (also encoding well-formedness rules in OCL) is described in [1].

Background information on the structure of OCL ASTs and tips on writing visitors for them are provided in the Eclipse Technical Article *How to Process OCL Abstract Syntax Trees*.

A summary of the internals of the OCL compiler follows. Please feel free to jump to the next subsection for a discussion on how to actually use the thing.

The OCL compiler follows the classical software architecture of a batch compiler [2]:

- a front-end component, shared among all “target architectures” (EMF now and UML2 soon). During the front-end phase, textual input is parsed into Concrete Syntax Trees (CSTs), which once validated are used to build the ASTs that compilation proper takes as input. These two activities involve a number of sub-steps, such as resolving mutual forward-references and collecting error markers. These sub-steps are realized with building blocks provided by MDT OCL, while bookkeeping and glue code has been added to hold everything together.
- the in-memory copy of the input model (our compiler does not modify the actual input) is decorated with **EAnnotations** that are taken later by EMF CodeGen (and in the future by UML2 CodeGen) as the source of method bodies and additional declarations: helper methods, helper types (e.g. those resulting from OCL tuples), and helper utilities (e.g. those to track all instances in a **ResourceSet**, realizing OCL’s **allInstances()**)

2.1 How to use it

The OCL Compiler is packaged as an Eclipse plugin with a basic UI (an action for an **IFile** to start compilation, a preferences page to choose compilation options). The input is assumed to be a single **.ecore** file accompanied in the same folder by an **.ocl** file with the same name.

The options **Suppress EMF Types** and **Suppress interfaces** are actually facades to options of EMF GenModel, and are included in the Preferences page because the OCL Compiler generates different Java code depending on their values. Their effect is (reproduced from [19]):

- **Suppress Interfaces:**
 - Generates implementation classes without separate interface declarations.
 - Improves performance since dispatch through an interface is slower and less likely to be in-lined by a JIT compiler
 - Reduces bytecode footprint somewhat
 - Can only be used for models without multiple inheritance
- **Suppress EMF Types:**

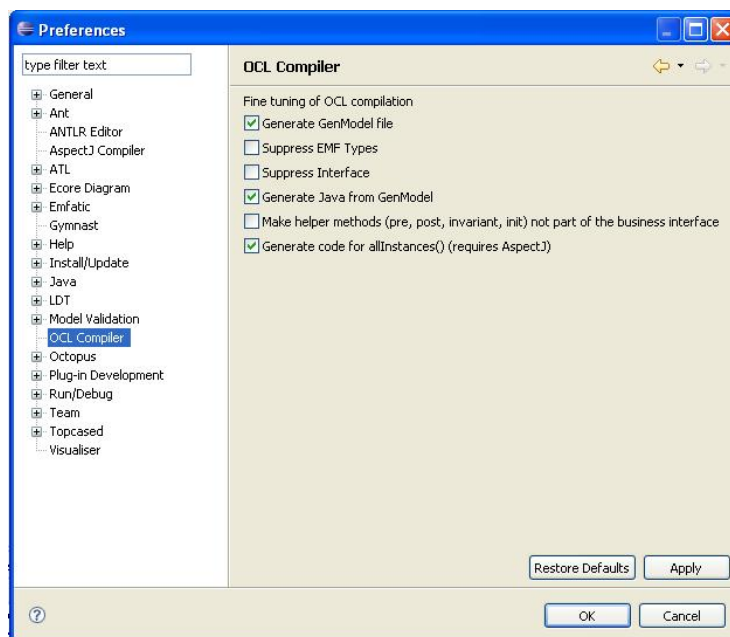


Fig. 1: Preferences page

- Generates methods for features and operations which use standard Java collection and object types instead of those from EMF (e.g., `EList`, `EMap`, `EObject`)
- Helps produce an API with no visible dependencies on EMF, i.e. a "pure" API.

Additional options of the EMF CodeGen are explained on pp. 52-ff. of the EclipseCON 2006 presentations *Advanced Features of the Eclipse Modeling Framework* [19].

The OCL Compiler options **Generate GenModel file** and **Generate Java from GenModel** control whether a `.genmodel` and Java files are generated. When Java generation is unticked, the user has the chance to touch values in the generated `.genmodel` file before (manually) invoking EMF CodeGen on it. The `.genmodel` file is written to a subfolder (named like the source `.ecore` file) in the same folder where the input `.ecore` and `.ocl` files can be found.

By ticking option **Make helper methods (pre, post, invariant, init) not part of the business interface** their Java counterparts are declared protected. Finally, the option **Generate code for allInstances() (requires AspectJ)** generates an aspect to track the instantiation of `EObjects`. The underlying technique has been described in [24] (so far we haven't found a way to inject our own code into the generation of a `protected new()` generated by

EMF CodeGen, in that case the dependency on AspectJ could be lifted). Moreover, currently `allInstances()` tracks all instances of each class in the same classloader, it should instead tracks those in the same `ResourceSet`.

2.2 Compilation Example

Referring to the class model depicted in Appendix A, the following OCL invariant (Listing 3) can be given as input to the OCL compiler to produce the output shown in (Listing 4).

Listing 3: An invariant involving OCL collection operations

```
context LoyaltyProgram
inv invariant_noAccounts :
-- when the LoyaltyProgram does not offer the possibility to earn
-- or burn points, the program members do not have LoyaltyAccounts
partners.deliveredServices->forall(pointsEarned = 0 and
    pointsBurned = 0)
implies memberships.account->isEmpty()
```

3 OCL Text Editor

There is a plethora of resources on how to manually build an Eclipse-based text editor. Although the user-interaction aspects are covered in detail, most tutorials stop short of addressing the “backstage” aspects of manipulating Concrete Syntax Trees (CSTs) and Abstract Syntax Trees (ASTs), because this processing used to be strongly intertwined with the particular parsing infrastructure. However, it is precisely those data structures that are essential to realizing use cases like hyperlinks navigation from usages to declarations, text hovers displaying the definition of the underlying identifier, etc. With the advent of metamodeling, efforts are underway to base the “parsing \rightarrow CST \rightarrow AST” pipeline on a common ground (e.g., expressing the well-formedness rules of a language in OCL itself). The sketched OCL integration is described in the context of a statechart DSL (Domain Specific Language) in the technical report *Generation of Eclipse-based IDEs for Custom DSLs* [15]. We mention all this because: (a) some of those ideas have found their way into the implementation of the OCL text editor, and (b) a more detailed how-to for the APIs to implement usability features are covered in Chapter 1 of that technical report.

The next subsection provides pointers to resources for those wanting to extend the OCL text editor (i.e. pointers to Eclipse’s JFace Text framework). After that, the following subsections discuss implementation aspects of the editor, including those with work-in-progress status. A section is devoted to the *Mark Occurrences* feature, as it is a good example of the way the underlying CST and AST representations simplify lookups.

3.1 Manual development of text editors for Eclipse

The following books contain chapters describing how to program text editors:

- Arthorne, J. and Laffra, C. 2004 *Official Eclipse 3.0 Faq (Eclipse Series)*. Addison-Wesley Professional. ISBN 0321268385.
- D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., and McCarthy, P. 2004 *The Java(TM) Developer's Guide to Eclipse, (2nd Edition)*. Addison-Wesley Professional. ISBN 0321305027.

Additionally, the on-line help provides concise and pertinent details about advanced features:

- Online help: Platform Plug-in Developer Guide > Programmer's Guide > Editors, for the topics listed in Figure 2.

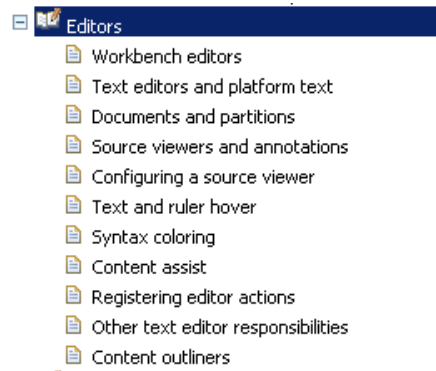


Fig. 2: Help on (mostly text) editors

Another account of the steps and APIs involved in programming a custom IDE is the three-part series *Create a commercial-quality Eclipse IDE* by Prashant Deva. Summary: *Part 1 introduces the architecture of an IDE and shows how to create the IDE's core layer. Part 2 shows how to implement the UI component of your IDE. Part 3 discusses additional UI elements Eclipse provides to enhance your editor* [11]

As to code examples exhibiting best practices, the JDT editor is of course the most advanced one, but it's overkill as a getting-started example. Rather, the Ant Editor bundled with JDT, and the TeXlipse editor [22] (an Eclipse-based editor for \LaTeX) are more manageable starting points. The technical documentation of TeXlipse additionally contains useful discussions for programmers of custom IDEs (in particular how to add your own builder). As of now the OCL Compiler is a batch-compiler (as opposed to an incremental one), and thus we made the decision to trigger compilation as a separate UI action, instead of letting a builder get activated upon changes to the input text files.

3.2 Usability features already implemented

The standard steps when developing a modern text editor using the JFace Text framework are documented in [25], describing for example how to achieve syntax highlighting, content assistance and content formatting. Underlying all those features, document partitioning is an aspect not directly visible at the UI level but affecting the implementation of the aforementioned functionality. Document partitions are contiguous, non-overlapping regions of text. Each such partition is associated with a specific content type, for which a different behaviour is relevant (for example, comments in a Java source file are one such partition, where AutoCompletion should behave differently as in non-comment regions). Coming back to the OCL text editor, Figure 3 depicts the different behaviours of syntax highlighting within `default` and `comment` partitions. The keyword `context` is highlighted in a `default`-content-type partition, but not in a `comment`-content-type partition.

```
*RoyalAndLoyal.ocd
pre: not self.participants->includes(c)
--context LoyaltyProgram::addTransaction(accNr: Integer, pName: String, servId: Integer,
--      amnt: Real, d: Date) : OclVoid
--post: let acc : LoyaltyAccount = self.memberships->collect( i_Membership : Membership | i_Me
context LoyaltyProgram::getServices() : Set(Service)
body: self.partners->collect( i_ProgramPartner : ProgramPartner | i_ProgramPartner.deliver
endpackage
```

Fig. 3: Different behavior in different partition definition

Another useful feature of a text editor is folding [10]. This capability is present in the OCL text editor, shown alongside other features in Figure 4.

If you're by now convinced of the advantages of textual notations, you might consider managing *class models* that way too. Emfatic is one such editor, based on the Gymnast (Generate Your Mega Nice AST) framework [6]. A newer version supports generics, hyperlinks navigation, and other useful features. Screenshots and summary of what's new can be found at <http://www.sts.tu-harburg.de/~mi.garcia/SoC2007/ImprovementsToTheEmfaticEditor.pdf>. The new version can be downloaded from Eclipse MDT EMFT (TODO URL).

The initial contribution of the OCL text editor supports in addition other usability features, as described below:

1. Syntax highlighting:

- OCL keywords receive a dark red in bold style (as in the Java JDT editor)
- String and number literals are highlighted blue. Other literals (e.g. tuple literals) remain in normal style

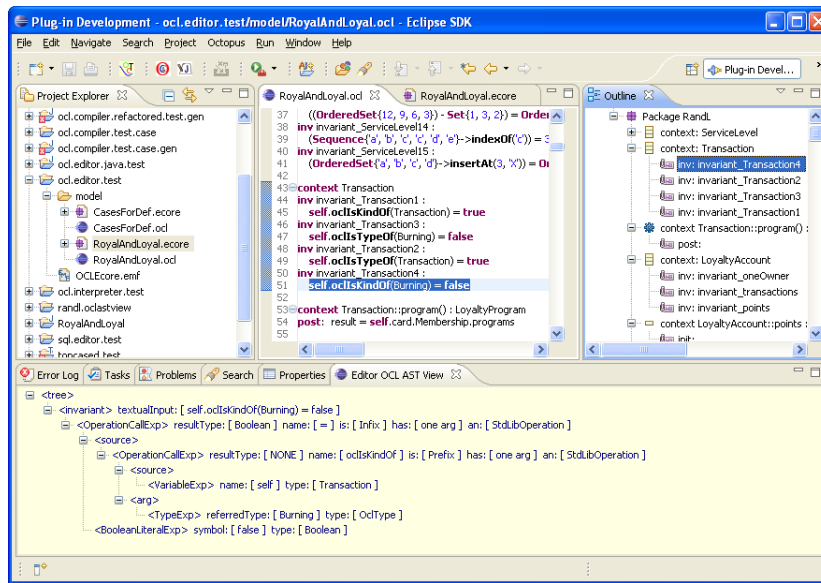


Fig. 4: OCL Editor Screenshots

- Usages of operations of the OCL Standard Library appear in black and bold
 - Comments appear in green
2. **AutoEdit, part 1:** Opening and closing punctuation marks. After typing a left-parenthesis, the corresponding right-parenthesis is added automatically (this feature is disabled inside the in `comment` content type). Other pairs exhibiting the same behavior: (1) `<` and `>` (2) `'` and `'` (3) `"` and `"` (4) `{` and `}`
 3. **AutoEdit, part 2:** In the Java editor, the SmartBrace feature places the cursor in a new, indented line after typing an opening curly brace and pressing `Enter`. Additionally, a closing curly brace is placed in yet another new line. The closest feature in the OCL editor triggers a simimilar behavior upon typing a colon (`:`) and pressing `Enter`. A new indented line will be added, letting the cursor ready for typing the body of the OCL expression.

In order to implement AutoEdit features, the subclass of `SourceViewerConfiguration` should include an override such as:

```
@Override
public IAutoEditStrategy[] getAutoEditStrategies(
    ISourceViewer sourceViewer, String contentType) {
    List<IAutoEditStrategy> s = new ArrayList<IAutoEditStrategy>();
    IAutoEditStrategy vonOben =
```

```

        super.getAutoEditStrategies(sourceViewer, contentType);
    for (IAutoEditStrategy autoEditStrategy : vonOben) {
        s.add(autoEditStrategy);
    }
    s.add(new DefaultIndentLineAutoEditStrategy());
    s.add(new EmfaticAutoEditStrategy(_editor));
    // partition type-specific auto edit strategies
    // could have been added
    IAutoEditStrategy[] res =
        s.toArray(new IAutoEditStrategy[0]);
    return res;
}

```

4. **Bracket matching:** The matching parenthesis is highlighted when the cursor is located either at an opening or closing bracket. Valid pairs: (1) { and } (2) (and) (3) < and > (4) [and].

An `org.eclipse.jface.text.source.DefaultCharacterPairMatcher` can be instantiated and registered with a document partition kind [15].

5. **Content assist** (see Figure 5). Displays a combo box with a list of choices for syntactically valid completions. Content assist kicks in after typing a dot separator or the `->` characters. It can also be manually activated by pressing `Ctrl + Space` as a shortcut.

- `.` triggers all the possible members of the source expression
- `->` triggers only all the possible collection operations for the source expression
- `Ctrl + Space` triggers all the possible completions given the current context

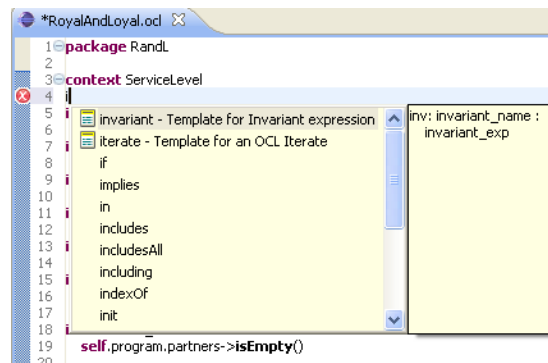


Fig. 5: Content Assist Example

It should be mentioned that the real work of computing the suggestions for completion is performed by MDT OCL, we just cherry-picked source

code from the OCL Interpreter [8] example, which can be seen in action in Figure 6.

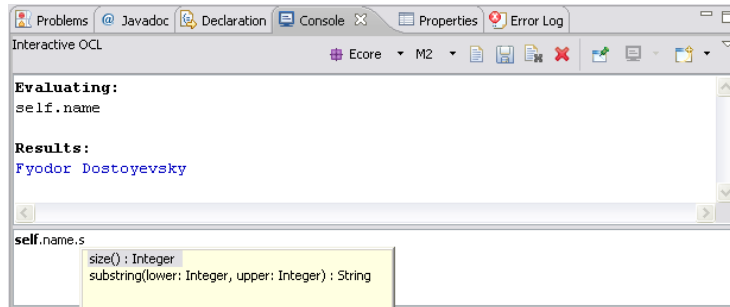


Fig. 6: OCLInterpreterExample

6. **Current line highlighting** and **Show range indicator**. In the Java editor, “Show range” displays a ruler bar to convey the extent of a method body. In the OCL case, we can distinguish two kinds of text regions: **package** and **context**. In Figure 4 it can be seen that the current line indicated by cursor position is highlighted, and on the left there is a ruler bar showing the range of the enclosing **context** in which the cursor is located. *Current line highlighting* and *Show range indicator* are controlled with the Preferences page.
7. **Text Folding**. Text part of the same conceptual unit can be folded. The foldable regions are (1) **package** and (2) **context**. The elements that can be folded might be extended in the future. In Figure 4, to the left of the text fragment “**context transaction**” a folding mark is available.
8. **Templates** (see right window in Figure 7). A time saving feature for frequently used constructs in OCL specifications, for example for the **iterate** construct. Template support is triggered by **Ctrl + Space** (same shortcut as for Content Assist). The out-of-the-box templates are:
 - **iterate** templates with the correct indentation and lines
 - **def** expression
 - **inv** expression

Additional templates can be added, and those available can easily be configured in the Preferences Page.

9. **Preference pages** (see Figure 7). Through these pages the configuration of Highlight Matching Brackets can be controlled, together with the colors for syntax highlighting and the text templates. The JFace Text framework provides an easy way to add preference pages. Furthermore,

by using `org.eclipse.ui.texteditor.ChainedPreferenceStore` we can combine several of them. This is required when the framework has relied on the configuration of the default preference page, e.g., show range indicator, highlight current line, etc. Figure 7 shows the current configurable features in the Preference Page.

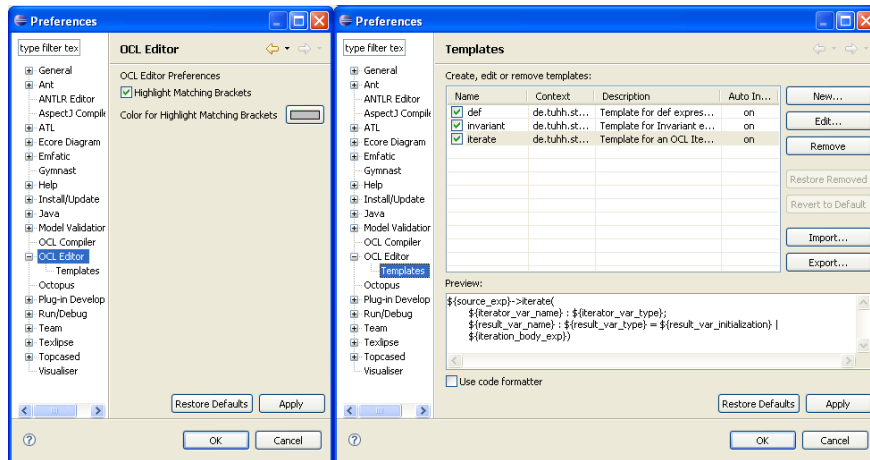


Fig. 7: Preference Pages

Historically, preferences have been implemented with classes from `org.eclipse.jface.preference`, as described in:

- Ch. 16 (Section “Contributing to the Preferences Dialog”) and Ch. 17 (Section “Storing Preferences Values for a Plug-in”) of [9]
- Ch. 12 (“Preference Pages”) of [3]

The `org.eclipse.jface.preference` API however shows its age, as it does not support the notion of scopes (per installation, per workspace, per project), and therefore also lacks the notion of overriding a setting in a more specific scope. The *User Settings* API solves that, alongside with providing support for the features already supported by the Preferences Store API (defaults, change notification). The User Settings API is described in Section “Scoped Value Management with User Settings” in Ch. 17 of [9].

10. **Problem and warning markers** (see Figure 8). Actually this feature displays at the UI level messages gathered as part of AST building. The document being edited is parsed whenever an update is made. Based on the error(s) reported by the parser (including start and offset), the editor displays them as problem markers (red squiggles on the text), thus also showing up in the Problems view, alongside a descriptive error message and the involved resource (i.e., the text file).

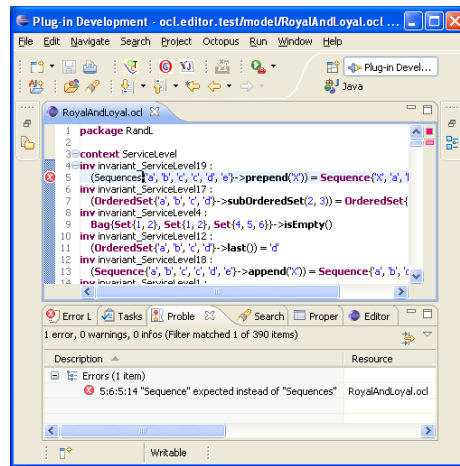
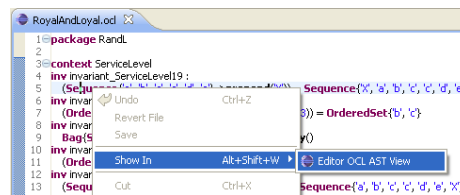


Fig. 8: Problem Markers example

11. **Outline view** (see Figure 4). This standard view shows the hierarchy of the defined OCL expressions for the whole document. The root package context is displayed on top. As children, it displays the **subPackages**, **context** (classifier context, operation context, property context) and for each of them the enclosed OCL expressions.
12. **OCL AST View** (see lower pane in Figure 4). This view was originally available as a separate plugin (as described in the Eclipse Technical Article [14]) to save time in visualizing the types of subexpressions of a (long) OCL expression. Instead of continuously tracking the cursor in the document, this view is displayed after an user action in the editor or in the outline view, with the *Show In* action provided both under the **Navigate** workbench menu and the context menu of the editor. After placing the cursor on the OCL expression of interest and invoking *Show In* (as depicted in Figure 9), the resulting AST will be displayed.

Fig. 9: Invoking the *Show In OCL AST View* action

3.3 A usability feature in focus: implementing Mark Occurrences

Figure 10 depicts the user view of this feature, which is realized by querying the CSTs and ASTs that have been built in the background. We review the main aspects of the implementation in this section.

```

56 context LoyaltyAccount
57 inv invariant_points :
58   (self.points > 0) implies self.transactions->exists( t : Transaction | t.points > 0 )
59 inv invariant_transactions :
60   self.transactions->collect( i_Transaction : Transaction | i_Transaction.points )->exists( p : Int
61   inv invariant_oneOwner :
62     (self.transactions->collect( i_Transaction : Transaction | i_Transaction.card )->collect( i_Custo
63
64 context LoyaltyAccount::points : Integer
65 init :
66   0
67
68 context LoyaltyAccount::totalPointsEarned : Integer
69 derive :
70   self.transactions->select( i_Transaction : Transaction | i_Transaction.isTypeOf( Earning ) )->
71
72 context LoyaltyAccount::usedServices : Set( Service )
73 derive :
74   self.transactions->collect( i_Transaction : Transaction | i_Transaction.generatedBy )->asSet()

```

Fig. 10: Mark Occurrences Example

Upon detecting a change in the cursor position, the innermost AST node (of the kinds listed below) that encloses such position is to be found.

1. **PropertyCallExp**. For example for `self.card`, each occurrence of `card` in the same context will be marked.
2. **OperationCallExp**. For example, if the cursor is placed on `isEmpty()` (as in `self.program.partners->isEmpty()`), then each occurrence of `isEmpty()` will be marked. Actually, the offset available for an **OperationCallExp** also encloses the *source* expression (in the example above, this source expression is `self.program.partners`). The exact offset of just the operation name has to be computed.
3. Clicking on the declaration of a **Variable** in a **LetExp**. Listing 5 shows the example of **LetExp**. Each occurrence of `popularTrans` will be marked in part.

Listing 5: Example of LetExp

```

let popularTrans : Set( Transaction ) =
  result->collect( deliveredServices )->collect( transactions )->asSet()
  in ( popularTrans->forall( date.isAfter( d ) )
    and ( popularTrans->select( amount > 500.00 )->size() ) > 20000

```

4. Clicking on the usage of a variable (an **VariableExp** AST node). For example, each occurrence of `self` can be marked.

Having found the innermost enclosing `ASTNode`, the counterparts to highlight can be found by querying or navigating the AST. Navigating from a usage to its declaration can be done in one step, as the AST directly supports it (e.g., a `Variable` has `getInitExpression()` and `getRepresentedParameter()`). As another example, having found a `PropertyCallExp` we can invoke the `getReferredProperty()` which returns the property (the `EStructuralFeature` in case of Ecore binding). In the Figure 11, the highlighted line shows the usage (`self.program`) of the occurrences that will be marked. However, once we have the `EStructuralFeature`, custom code has to be written to find its usages. This can be done in one of two ways:

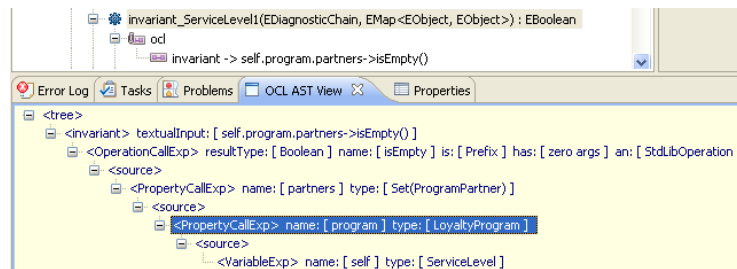


Fig. 11: an example of `PropertyCallExp` OCL AST View

- **Using a visitor.** `OCLEExpression` is a subtype of `Visitable` and thus may accept custom `Visitors`. A visitor can be created for each kind of `ASTNode` of interest, one such visitor will have just one handler checking whether the visited element is a usage of the declaration passed to its constructor. If so, the usage can be appended to a `List` declared as instance variable of the visitor.
- **Keeping a map.** The other approach involves keeping a one-to-many `Map` which has the declaration as key and its usages as values. In the case of `PropertyCallExp` with Ecore binding, the map is defined as `OneToManyMap<EStructuralFeature, PropertyCallExp>`.

The algorithm for *Mark Occurrences* comprises a high-level the following steps:

1. Find the CST node for the given cursor position.
2. Get the corresponding AST node for the selected CST Node.
3. Find the specific part of AST Node which is selected by the cursor (offset). The `getStartPosition()` method of `ASTNode` returns the start position of the full expression. The `getStartOffset()` method of `CSTNode` returns the start offset relative to editor starting point.

4. Find the occurrences of similar case of ASTNode if any
5. Get the corresponding CST Nodes
6. Finally mark the occurrences.

The current implementation of *Mark Occurrences* (shown in Figure 10) is limited and works only for `PropertyCallExp`.

3.4 Work in Progress

Beyond the “basics” discussed so far, additional features are thinkable to make the OCL text editor more useful. We review some of these candidate ideas here.

1. **Content formatting.** Format the document with correct indentation. For example, indentation for each clause of an `if-then-else-endif`. For a discussion of implementation issues see: (a) section “Content Formatting” in [25], i.e. at URL <http://www.realsolve.co.uk/site/tech/jface-text.php> or (b) Part 3 in the tutorial [11]
2. **Double click action.** A sensible reaction to double-click is selecting an enclosing fragment, as required later for cut&paste, for example. Moreover, repeated double-clicks should result in progressively larger fragments being selected, following the composition hierarchy for subexpressions in the OCL AST. The implementation of this functionality involves overriding `getDoubleClickStrategy(ISourceViewer sourceViewer, String contentType)` in the subclass of `SourceViewerConfiguration` [15]
3. **Hyperlinks for variables and types.** For this, besides querying the CST and AST for OCL expression it is also necessary to collect information about declarations available at the AST of the underlying class model (Ecore or UML2). With that, the usage of a variable or type can be traced back to its declaration. `getHyperlinkDetectors()` in `SourceViewerConfiguration` has to be overridden to achieve this functionality [15].
4. **Hover over text fragment.** To display hints of the underlying OCL construct. This requires overriding two methods in the subclass of `SourceViewerConfiguration`, both named `getTextHover()` [15].
5. **Show in (EMF or UML2) type hierarchy.** As in the Emfatic text editor, visualizing the Type Hierarchy (see Figures 12 and 13) is necessary when working with but the most basic class models (alternatively, one might rely on a separate graphical editor to display it). Method `getShowInContext()` of `IShowInSource` and method `getShowInTargetIds()` of `IShowInTargetsList` need to be overridden [15].
6. **Refactorings.** As with their Java counterpart, having tool support for refactorings (and for detecting *code smells*) increases productivity. Several refactorings for OCL have already been documented in the literature, to

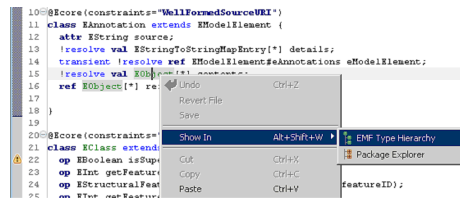


Fig. 12: The “Show In” way to display the EMF Type Hierarchy

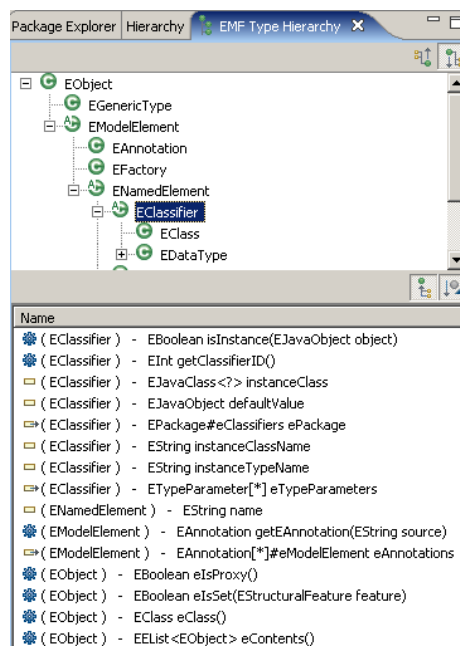


Fig. 13: EMF Type Hierarchy

which we offer pointers in the next section. Proposals run the gamut from renaming (a variable or a type), to more complex refactorings such as detecting and removing redundant expressions.

3.5 Relative complexity of usability features in text editors for DSLs

As a rough guide, we've classified text-editor customizations according to the layer on which they can be realized. For example, those in the first category are not really dependent on the grammar of the subject language, while those in the fourth category require the availability of ASTs to carry out its function.

A given functionality can usually be implemented at any of several levels. If given the choice, implementing it at a lower layer results in more solid code, as it is less prone to the failures of lower layers. For example, content formatting can be implemented at the AST level. In that case, only those regions of the document that parse flawlessly will have it enabled. A no-frills, text-level version of content formatting would instead be less brittle.

In general, API is available in JFace Text for functionality in lower layers, while for those in higher layers you need to write most of your own code (one could argue that an IDE generator [12] could also take part in generating such code). The classification has been excerpted from Ch. 1 of [15].

1. Document-independent functionality
 - Templates
 - Actions on the toolbar and menu bar, retargetable actions
 - New File wizard
 - Preferences
 - Session properties associated to the underlying `IFile`
 - Bracket matching (brace, parenthesis, etc.)
 - SmartBrace AutoEdit
2. Syntax-dependent, but otherwise standard customizations
 - Background Parsing
 - Document partitioning
 - Folding
 - Syntax Highlighting
 - Content Outline
 - Custom Problem Markers
 - Show range indicator
 - Double-click strategy
 - AutoIndent AutoEdit

3. Customizations requiring CST or AST

- Hover over text fragment
- Hyperlinks
- Mark occurrences, Select in Outline
- Show in
- Context menu, for example Go to Declaration
- Hovers over vertical ruler annotations
- Views, for example EMF Type Hierarchy
- Content formatting
- Content assist

4 Related Work: Refactoring of OCL Expressions

The presence of OCL constraints in an integrated Ecore + OCL model implies that refactorings at the class model level can no longer ignore their effect on the syntax, well-formedness, and semantics of the associated OCL constraints. A desirable property for refactorings is for them to be *semantics preserving*. What exactly is meant by that requires more precision. For example, every refactoring loses *some* semantics: a refactoring for speed results in programs having a different observable behavior from the original ones (which are slower). In the case of OCL, the authors of [21] cleverly define an OCL refactoring to be semantics preserving provided the following conditions are met:

- there is a one-to-one correspondence between old and new OCL constraints
- every possible evaluation of each new constraint results in the same value as its old version on the image of the system snapshot (how to compute the image function for each refactoring is not spelled out in detail)

The Eclipse-based tool RocLET (<http://www.roclet.org/>) aims at implementing the refactorings described in [21].

Sometimes, a simpler version of a verbose expression can be obtained without change in meaning, thus promoting readability. Listing 6 shows a sample verbose version, while Listing 7 displays its simplified counterpart. Rules for simplifying expressions (for both improved performance and readability) can be found in [18] and [17].

Listing 6: Example of Verbose Expression

```
context LoyaltyProgram
def verboseTransactions : Set(Transaction) =
  self.partners->collect ( i_ProgramPartner : ProgramPartner |
    i_ProgramPartner.deliveredServices )->collect (
    i_Service : Service | i_Service.transactions )->asSet()
```

Listing 7: Concise formulation of the fragment in Listing 6

```
context LoyaltyProgram
def conciseTransactions : Set(Transaction) =
  self.partners.deliveredServices.transactions->asSet()
```

Correa and Werner [4, 5] offer a catalog of *code smells* for OCL, i.e., patterns that go against established programming practice, for example by impairing readability or by complicating the evolution of the expressions. Typical smells include *magic numbers*, *and-chain* (several Boolean expressions anded together instead of broken apart into separate definitions), Law of Demeter [20], and duplicate code. To overcome their shortcomings, a number of refactorings are put forward. For example, *Implies Chain* denotes a chain of `implies` operators, that can be simplified by replacing the occurrences of `implies` with `and` except for the last one. Line 1 of Listing 8 shows the example of Implies Chain and the next line shows the result after refactoring.

Listing 8: Example of Implies Chain

```
a1 implies a2 implies .. implies aN implies b
(a1 and a2 and .. and aN) implies b
```

To our knowledge, no detailed algorithms have been presented in the literature for the following OCL refactorings:

- detect unused derived operations or attributes (not an error, but a symptom that a typo may have occurred)
- replace an arbitrary (complex) subexpression with the invocation to a defined operation. Notice this may involve passing as explicit arguments some bindings in the scope of the caller but not in the scope of the callee
- find unused explicit arguments
- compute the average complexity of an expression
- inline a `def` usage (also called macro expansion). This is the counterpart to factoring a subexpression into its own definition.

5 Conclusions and Future Work

Feedback from a large community of developers is critical to harvesting real-world and complete OCL specifications. Actually, harvesting such specifications reveals a chicken-and-egg problem: those specifying a system have a reduced incentive to invest effort in preparing OCL specs if they are to remain paper-only and thus not automatically enforceable. On the other hand, the developers of OCL tooling are reluctant to target a small audience. The OCL Tools component is in a good position to break this cycle, providing immediate benefit to the authors of OCL specifications, and accelerating the synergies of the Eclipse ecosystem.

A Appendix A: Class model of Royal & Loyal

The class model of the Royal & Loyal case study is shown for reference purposes below:

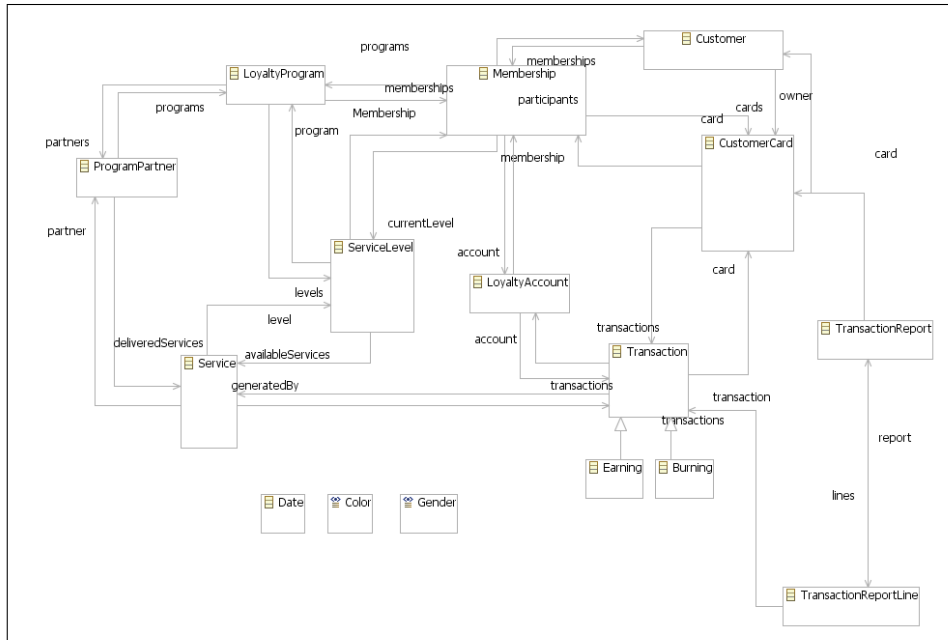


Fig. 14: Royal & Loyal class model

References

- [1] D. H. Akehurst. Validating BPEL Specifications using OCL. Technical Report 15-04, University of Kent at Canterbury, August 2004. <http://www.cs.kent.ac.uk/pubs/2004/2027>.
- [2] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2003. <http://www.cs.princeton.edu/~appel/modern/java/>.
- [3] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plugins (2nd Edition) (Eclipse Series)*. Addison-Wesley Professional, 2006. Book website: <http://www.qualityeclipse.com/>.
- [4] Alexandre Correa and Cláudia Werner. Refactoring Object Constraint Language Specifications. *Software and Systems Modeling*, 6:113–138, 2007.

- [5] Alexandre L. Correa and Cláudia Maria Lima Werner. Applying Refactoring Techniques to UML/OCL Models. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *UML*, volume 3273 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2004. <http://reuse.cos.ufrj.br/prometeus/publicacoes/Correa-UML2004.pdf>.
- [6] Christopher J. Daly. AST framework generation with Gymnast. In *Tech Exchange Panel: Language Toolkits*. EclipseCON 2005, 2005. Available at <http://www.sts.tu-harburg.de/~mi.garcia/SoC2007/GymnastSlides.pdf>.
- [7] C. W. Damus. Implementing Model Integrity in EMF with MDT OCL, 2006. Eclipse Technical Article, <http://www.eclipse.org/articles/Article-EMF-Codegen-with-OCL/article.html>.
- [8] Christian W. Damus. OCL Interpreter Example. <http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.ocl.doc/tutorials/oclInterpreterTutorial.html>.
- [9] Jim D’Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java(TM) Developer’s Guide to Eclipse (2nd Edition)*. Addison-Wesley Professional, 2004. ISBN 0321305027.
- [10] Prashant Deva. Folding in Eclipse Text Editors, March 2005. <http://www.eclipse.org/articles/Article-Folding-in-Eclipse-Text-Editor%2Fs/folding.html>.
- [11] Prashant Deva. Create a commercial-quality Eclipse IDE, 2006. <http://www-128.ibm.com/developerworks/edu/os-dw-os-ecl-commplgin1.html>.
- [12] Miguel Garcia. Generation of DSL Tools based on Language Definitions. Presentation at MDS Today 2007, <http://www.sts.tu-harburg.de/mi.garcia/SoC2007/GenDSLToolsFromLangDef.pdf>.
- [13] Miguel Garcia. Formalizing the Well-formedness Rules of EJB3QL in UML + OCL. In T. Kühne, editor, *Reports and Revised Selected Papers, Workshops and Symposia at MoDELS 2006, Genoa, Italy*, LNCS 4364, pages 66–75. Springer-Verlag, 2006. http://www.sts.tu-harburg.de/~mi.garcia/pubs/atem06/paper/Garcia_ATEM06_Formalizing_the_well-formedness_rules_of_EJB3QL_in_UML_OCL.pdf.
- [14] Miguel Garcia. How to process OCL Abstract Syntax Trees, Eclipse Technical Article, 2007. <http://www.eclipse.org/articles/article.php?file=Article-HowToProcessOCLAbstractSyntaxTrees/index.html>.
- [15] Miguel Garcia and Paul Sentosa. Generation of Eclipse-based IDEs for Custom DSLs. Technical report, Software Systems Institute (STS), Technische Universität Hamburg-Harburg, Germany, Sep 2007. <http://www.sts.tu-harburg.de/~mi.garcia/SoC2007/draftreport.pdf>.

-
- [16] Miguel Garcia and A. Jibrán Shidqie. OCL Compiler for EMF. In *Eclipse Modeling Symposium at Eclipse Summit Europe 2007, Stuttgart, Germany*, 2007. <http://www.sts.tu-harburg.de/~mi.garcia/pubs/2007/ese/oclcompiler.pdf>.
 - [17] Martin Giese, Reiner Hähnle, and Daniel Larsson. Rule-based simplification of OCL constraints. In Octavian Patrascoiu et al., editor, *Workshop on OCL and Model Driven Engineering at UML2004, Lisbon*, pages 84–98, 2004. <http://www.cs.chalmers.se/~danla/ocl04.pdf>.
 - [18] Martin Giese and Daniel Larsson. Simplifying transformations of OCL constraints. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *LNCS*, pages 309–323. Springer, 2005. <http://www.cs.chalmers.se/~danla/models2005.pdf>.
 - [19] Kenn Hussey and Marcelo Paternostro. Advanced Features of the Eclipse Modeling Framework. EclipseCON 2006, 2006. <http://eclipsezilla.eclipsecon.org/php/attachment.php?bugid=171>.
 - [20] K. J. Lienberherr. Formulations and Benefits of the Law of Demeter. *SIG-PLAN Not.*, 24(3):67–78, 1989.
 - [21] Slavisa Marković and Thomas Baar. *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems - MoDELS 2005 - Lecture Notes in Computer Science*, volume 3713, chapter Refactoring OCL Annotated UML Class Diagrams, pages 280–294. Springer Verlag, October 2005. <http://infoscience.epfl.ch/getfile.py?recid=54765&mode=best>.
 - [22] Oskar Ojala. Technical Specification TeXlipse Project, 2005. <http://texlipse.sourceforge.net/docs.html>.
 - [23] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, Boston, MA, USA, 2003. ISBN 0321179366.
 - [24] Darren Willis, David J. Pearce, and James Noble. Efficient Object Querying for Java. In Dave Thomas, editor, *ECOOP*, volume 4067 of *LNCS*, pages 28–49. Springer, 2006. http://www.mcs.vuw.ac.nz/~djp/files/WPN_ECOOP06.ps.
 - [25] Phil Zoio. Building an Eclipse Text Editor with JFace Text, April 2006. <http://www.realsolve.co.uk/site/tech/jface-text.php>.

Listing 4: The noAccounts invariant translated into Java

```

public boolean invariant_noAccounts(DiagnosticChain diagnostics,
    Map<Object, Object> context) {
    /*
    context LoyaltyProgram
    inv invariant_noAccounts :
    -- when the LoyaltyProgram does not offer the possibility to earn
    -- or burn points, the program members do not have LoyaltyAccounts
    partners.deliveredServices->forall(
        pointsEarned = 0 and pointsBurned = 0)
    implies memberships.account->isEmpty()
    */
    org.eclipse.oc1.util.Bag<RandL.Service> collect1 =
        org.eclipse.oc1.util.CollectionUtil.createNewBag();
    for (RandL.ProgramPartner i_ProgramPartner :
        org.eclipse.oc1.util.CollectionUtil.asSet(this.getPartners())) {
        collect1.addAll(org.eclipse.oc1.util.CollectionUtil.asSet(
            i_ProgramPartner.getDeliveredServices()));
    }
    Boolean forAll2 = true;
    for (RandL.Service i_Service : collect1) {
        if (forAll2) {
            Boolean equal3 =
                Boolean.valueOf(i_Service.getPointsEarned() == 0);
            Boolean and4 = equal3;
            if (and4) {
                Boolean equal5 =
                    Boolean.valueOf(i_Service.getPointsBurned() == 0);
                and4 = equal5;
            }
            forAll2 = and4;
        }
    }
    Boolean implies6 = forAll2;
    if (!(implies6)) {
        implies6 = Boolean.TRUE;
    } else {
        java.util.List<RandL.LoyaltyAccount> collect7 =
            org.eclipse.oc1.util.CollectionUtil.createNewSequence();
        for (RandL.Membership i_Membership :
            org.eclipse.oc1.util.CollectionUtil.asOrderedSet(
                this.getMemberships())) {
            collect7.add(i_Membership.getAccount());
        }
        implies6 = (new Boolean(collect7.isEmpty()));
    }
    if (!(implies6)) {
        if (diagnostics != null) {
            diagnostics.add(
                new BasicDiagnostic(Diagnostic.ERROR,
                    RLValidator.DIAGNOSTIC_SOURCE,
                    RLValidator.US_ADDRESS__HAS_US_STATE,
                    EcorePlugin.INSTANCE.getString(
                        "_UI_GenericInvariant_diagnostic",
                        new Object[] { "LoyaltyProgram_noAccounts",
                            EObjectValidator.getObjectLabel(this, context) },
                        new Object[] { this }));
        }
        return false;
    }
    return true;
}

```