Implementing an OCL Compiler for .NET

László Lengyel
Budapest University of Technology and Economics
Goldmann György tér 3.
Hungary 1111, Budapest
lengyel@aut.bme.hu

Tihamér Levendovszky
Budapest University of Technology and Economics
Goldmann György tér 3.
Hungary 1111, Budapest
tihamer@aut.bme.hu

Hassan Charaf
Budapest University of Technology and Economics
Goldmann György tér 3.
Hungary 1111, Budapest
hassan@aut.bme.hu

ABSTRACT

Model-Driven Architecture standardized by OMG facilitates separating the platform-independent part (PIM) and the platform-specific part (PSM) of a system model. The platform-independent artifacts are mainly UML models created with CASE tools. Due to this separation, PIM specified by the developers can be reused across several implementation platforms of the software. PSM is ideally generated automatically from PIM via model transformation steps performed by model compilers. Beyond the topology of the visual models additional constraints must be specified, which ensure the correctness of the attributes among others. Dealing with OCL constraints provides a solution for the unsolved issues, because topological and attribute transformation methods cannot perform and express the problems that can be addressed by constraint validation. This paper discusses the need for combining UML and OCL, it introduces the compilers in general, it shows the architecture of our OCL Compiler for .NET, and it presents the lexical and syntactic analysis as well as the semantic analysis and code generation techniques in detail. The OCL Compiler has been implemented as a module of our n-layer multipurpose modeling and metamodel-based transformation system called Visual Modeling and Transformation System (VMTS). The OCL Compiler module facilitates validating (i) constraints contained by the metamodels at the time of the model instantiation process, and (ii) constraints contained by the transformation steps during the metamodel-based graph transformation. An illustrative case study is also provided, which introduces how VMTS generates source code from a statechart diagram, and how it validates specific properties using the OCL Compiler.

Keywords

OCL Compiler, .NET, Constraints, Constraint Validation, UML, Metamodeling, VMTS

1. INTRODUCTION

Model transformation is a possible solution for realizing model compiler. Its methods are vital in several applications, for instance the Object Management Group's (OMG) Model-Driven Architecture (MDA) standard [OMG03a] strongly builds on model compilers, which automatically create a platform-specific model from the platform-independent models specified by the modelers. Software model transformation provides a basis for model compilers, which plays a central role in the MDA architecture.

.NET Technologies'2005 *conference proceedings,*
ISBN 80-86943-01-1

There are many CASE tools that support drawing UML diagrams and other features like code generation and reverse engineering. However, support for OCL attached to model transformation and mappings between models are rarely found in these tools. There are several tasks that a CASE tool should offer in order to provide support for OCL. For example, syntax analysis of OCL expressions and a precise mechanism for reporting syntactic errors help in writing syntactically correct OCL statements. An important feature is the semantic analyzer, which reports as many errors as possible in order to help the user develop solid OCL code.

Often we need to specify a model more precisely than a topology-oriented visual modeling language facilitates it. It is a prevalent case that we want to define expressions and constraints on our model. The

Object Constraint Language (OCL) [OCL03a] is a formal language for analysis and design of software systems. It is a subset of the industry-standard Unified Modeling Language [UML03a] that allows software developers to write constraints and queries over object models. A constraint is a restriction on one or more values of an object-oriented model or system. There are four types of constraints. (i) An invariant is a constraint that states a condition that must always be met by all instances of the class, type, or interface. (ii) A precondition to an operation is a restriction that must be true at the moment before the operation is executed. Obligations are specified by postconditions. (iii) A postcondition to an operation is a restriction that must be true at the moment that the operation has just ended its execution. (iv) A guard is a constraint that must be true before a state transition fires. Besides these, OCL can be used as a navigation language as well.

Our n-layer metamodel-based model storage and transformation software package is called Visual Modeling and Transformation System [Lev04a] [Vis03a]. VMTS is implemented using Microsoft .NET Framework [Mic03a] and illustrates an approach, where model storage and model transformation can be treated uniformly, and what links them together is the notion of the metamodel. Modeling environments built on metamodeling are highly configurable (visual) modeling tools allowing constraints to be specified in advance. VMTS uses graph rewriting for model transformation as a powerful tool with strong mathematical background [Lev04a]. The atoms of graph transformation are rewriting rules, where each rewriting rule consists of a left hand side graph (LHS) and a right hand side graph (RHS). Applying a graph rewriting rule means finding an isomorphic occurrence (match) of LHS in the graph to which the rule is being applied (host graph), and replacing this subgraph with RHS. Replacing means removing elements which are in LHS but not in RHS, and gluing elements which are in RHS but not in LHS. The graph transformation is defined as an ordered sequence of rewriting rules, in other words, we control the transformation process by sequencing the rewriting rules. Previous work [Lev04a] has introduced an approach, where LHS and RHS of the rules are built from metamodel elements. It means that an instantiation of LHS must be found in the host graph instead of the isomorphic subgraph of LHS. Hence LHS and RHS graphs are the metamodels of the graphs which we find and replace in the host graph.

Often it is not enough to match graphs using the topological information only. There are cases in which we want to restrict the desired match by other properties, e.g. we want to match a subgraph with a node which has a special property, or which has a unique relation between the properties of the matched nodes. The metamodel-based definition of the rewriting rules facilitates assigning OCL constraints to the pattern rule nodes contained by the transformation steps, and with OCL these conditions can be expressed easily. A *precondition* (*postcondition*) assigned to a rewriting rule is a Boolean expression that must be true at the moment when the rewriting rule is fired (after the completion of a rewriting rule). If a *precondition* of a rewriting rule is not true then the rewriting rule fails without being fired. If a *postcondition* of a rewriting rule is not true after the execution of the rewriting rule, then the rewriting rule fails. A direct corollary of this is that an OCL expression in LHS is a precondition to the rewriting rule, and an OCL expression in RHS is a postcondition to the rewriting rule. A rewriting rule can be fired if and only if all conditions enlisted in LHS are true. Also, if a rewriting rule finished successfully, then all the conditions enlisted in RHS must be true.

Constraints (pre- and postconditions) facilitate specifying precisely the execution of the steps contained by the transformation. Using constraints for each step, we can define the cases in detail, in which the step can be fired, and, of course, in which not.

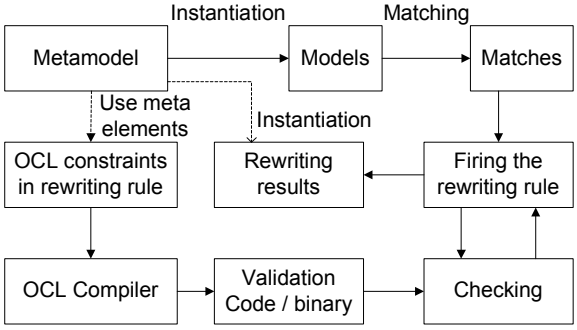

Figure 1. Block diagram for checking constraints during the rewriting process

Fig. 1 presents a block diagram to illustrate the method how VMTS checks the rewriting rule constraints during the rewriting process. It is possible in VMTS that LHS and RHS use different metamodels, but for the sake of simplicity in the block diagram they have a common metamodel. The rewriting rule contains OCL constraints. VMTS does not interpret the constraints during the rewriting, but an assembly is used that is generated by the OCL Compiler. The rewriting process uses the matches found by the matching process and the compiled assembly to validate the constraints on the matched parts of the host graph. The rewriting process generates the rewriting result if and only if a match satisfies the constraints (preconditions), and the step is successful if and only if the rewriting result

satisfies the postconditions. In Fig. 1 the rewriting result is also an instance model of the metamodel, because LHS and RHS use the same metamodel.

One of the most important parts of the constraint validation method is that our constraint checking approach does not interpret the constraints; OCL Compiler generates C# code and compiles it to an assembly, which validates the metamodel and the rewriting rule constraints. This method facilitates determining the complexity of the constraint validation method.

This paper introduces the steps necessary for the implementation of the OCL Compiler for .NET, which is capable of compiling OCL constraints into source code and a binary file that checks the OCL constraints on the rewriting rules of a transformation that realizes an MDA model compiler. Our example is a UML statechart model.

The rest of this paper is organized as follows: Section 2 introduces the concept of a compiler in general, it presents the architecture of our OCL Compiler and it discusses the lexical and syntactic analysis as well as semantic analysis along with the code generation in detail. In Section 3 we illustrate a case study how to design a C# form behavior using Visual Studio.NET Form editor, and how VMTS generates the user interface handler code based on the statechart model. In this way the programmer needs to write the application-specific parts of the code only. Finally, conclusions and future work are delineated in Section 4.

2. CONTRIBUTION

This section presents the general considerations related to compilers and their modules shortly and examines the VMTS OCL Compiler in detail.

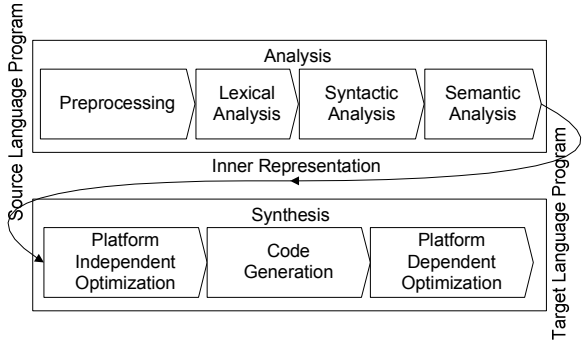

Figure 2. The steps of the compilation

Implementing a compiler is a complex task consisting of several well-defined subparts. The input of a compiler is a textual file written in the source language, and the output is a textual file or a binary in the target language. The source language and the target language can be the same or different. The two main parts of the compilation are: (i) the analysis of the source language input, and (ii) the generation (synthesis) of the target language output based on the retrieved semantic information. Fig. 2 introduces the steps of the compilation process.

Compiler Architecture

The OCL Compiler is a part of VMTS, therefore the generated code and the compiled assembly have to fit in this environment. The block diagram of VMTS and the place of the OCL Compiler in a metamodel-based model transformation system are depicted in Fig. 3. The user interfaces (Adaptive Modeler, Rule Editor) are functionally separated from the model storage unit (AGSI Core - Attributed Graph Architecture Supporting Inheritance), which uses an RDBMS (Microsoft SQL Server 2000) to store the model information. Besides this the AGSI Core exposes its interface to any other applications which may use other technique to process AGSI data.

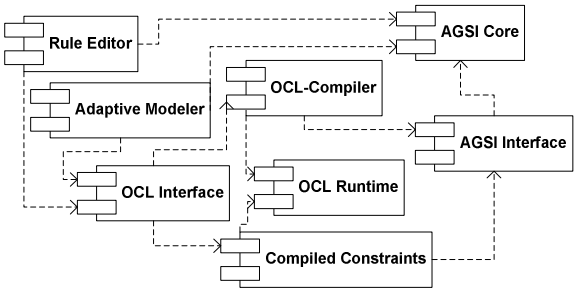

Figure 3. Block structure of VMTS

The OCL Interface provides a unified interface for the user interface modules to access constraint validation. If it is required, it uses the OCL Compiler and loads the compiled binary (Compiled Constraints). AGSI Core stores and handles the models as labeled graphs: it simply uses nodes and edges. In OCL constraints these nodes and edges appear with their names as types, instances and associations. The main purpose of the AGSI Interface is to provide a linkage between the OCL expressions and the model over which the expression should be evaluated. AGSI provides type information from the AGSI Core objects. During the compilation and the constraint validation process we run only select commands on the AGSI Core data, therefore AGSI Interface does not support operations modifying the model.

Lexical and Syntactic Analysis

Lexical and syntactic analyses are realized by code in the ANSI C language, it is generated by the tools Flex [Fle99a] and Bison [Bis98a]. We chose these tools because (i) the compiler is implemented using Microsoft Visual Studio and it was easy to integrate

the Flex and Bison tools into this environment, and (ii) the VMTS is executed also in the .NET environment.

The first step of the lexical analysis is the tokenization, which distinguishes between the identifiers (name) and the keywords of the language. Tokenization is achieved by a table, which contains the keywords. The result of this process is a sequence of tokens, which contains the meaning of the source program.

The task of the syntactic analysis is to find the deduction which generates the source code of the program, starting from the sentence symbol (S). The analysis is the same process but in the opposite direction. The analyzer reads the sequence of the tokens, and using the production rules it generates an Abstract Syntax Tree (AST), which is a model of the program that we want to compile. The AST is a direct association between the rules in the grammar and the nodes in the tree, and it is purely an abstract representation of the syntax, modeled as a tree [Ake03a] [Ham98a]. The inner nodes of the AST contain no terminal symbols, while the leaves contain the tokens.

Original rules	Reworked rules
`A -> b c? d`	`A -> b d	b c d`

`A -> b c* d`	`A -> b optionalC d`
	`optionalC -> /* empty */`
	`| optionalC c`

`A -> b c+ d`	`A -> b optionalC d` `optionalC -> c | optionalC c`

Table 1. Reworked EBNF rules for Bison

The UML specification [UML03a] uses EBNF notation [Ext96a] for the grammar specification, which we had to modify in certain places to be able to process it with Bison. We had to rework the `?` (optional element), the `*` (`0..*` multiplicity) and the `+` (`1..*` multiplicity) notations. Table 1 presents the original EBNF and the modified rules for Bison. The `/* empty */` notation means the empty symbol.

The generation of the AST is possible if and only if the program is syntactically correct [Loe03a].

Semantic Analysis

OCL allows certain abbreviations in numerous places and leaving out some identifiers if they do not cause misconceptions (e.g. the left `self` identifier). Before we can start the semantic analysis we must perform a syntax tree transformation, which inserts the missing identifiers into the AST.

In the OCL Compiler we cannot use the traditional symbol table, because the symbols are not in the code to be compiled, but it must be obtained from another place, namely, from the VMTS model database. The most important pieces information we need during the compilation are: (i) we have to decide about an identifier appearing in a type name position whether it is already defined, and whether it is visible for the context where it is appeared, (ii) during the OCL property selection we have to check the selected item of the class: whether it is an attribute, operation or association (and in this case whether it is navigable). For these tasks we implemented a class (`TypeHandler`) which hides the duality of the types from the other part of the OCL Compiler. We can consider this class as a dynamic symbol table of the types. The `TypeHandler` class contains the `typeOfCall` function:

```
String typeOfCall(String typeName, String propertyName, 'dot'|'arrow')
```

A type name is passed to the function along with a property name as a parameter, and the function returns a type name, which describes the type of the retrieved object when selecting the given property on an object of the given type. The third parameter is 'dot' or 'arrow' depending whether the function call refers to an `OclAny` or a `Collection` class. For the built-in types the function determines the result with the help of the System.Reflection namespace [Mic03a], and for the model types the AGSI Interface returns the answer.

In summary, the semantic analysis performs two activities: it maps string-based path names onto types, and maps OCL specific operations onto the appropriate semantic model constructs.

Code Generation

Code generation is realized using the System.CodeDom namespace of .NET Framework [Mic03a]. It means that the code generation is a syntax tree composition, from which the framework generates the source code. Using CodeDOM the generated source code will be syntactically correct in all cases; our task is only to deal with the appropriate semantic content.

The OCL Runtime (Fig. 3) contains C# language implementation for each predefined OCL types. Using these classes the operations contained by the constraints can easily be expressed in the C# language. While the current version of C# does not support class templates, the implementation of the collection types is more complex, than it would be with generics. The Set, Sequence and Bag classes are implemented as abstract classes in OCL Runtime, and when it is required, the compiler inherits from the adequate base class to create a new typed collection class. The task of the inherited collection classes is

the type conversion, while the fundamental operations are implemented by the base classes.

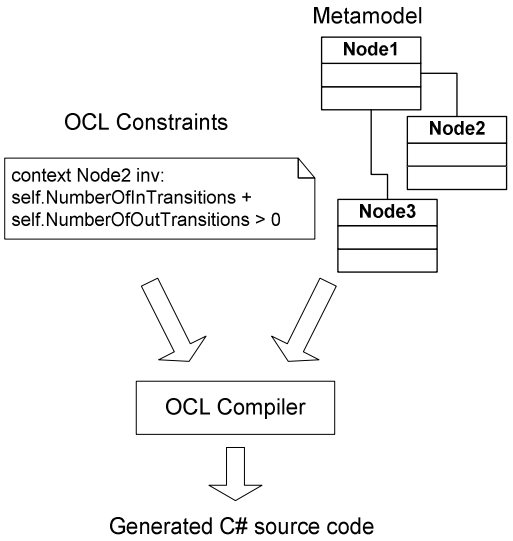

Figure 4. The input and the output of the OCL Compiler

Fig. 4 introduces the input and output of the OCL Compiler. In case of rewriting rules OCL Constraints are assigned to rule nodes; recall that rewriting rules are created from metamodel elements, therefore we also need the metamodel to access the properties of the meta types used in the rewriting rules.

The model data is stored in the database and the instantiation of the model elements, in fact, does not mean the creation of a .NET object, hence no .NET types exist in OCL Runtime. Type handling is realized with the `OclType` abstract class and its two descendants: `OclBasicType` and `OclModelType`.

In the CodeDOM tree there are well-defined nodes for certain syntax tree nodes. For each invariant, pre- and postcondition there is a public method with a `bool` return type. The methods of invariant constraints do not have parameters, while the methods of pre- and postconditions have the same parameters as the corresponding operation defined in UML. Finally, every OCL expression is an instance of the `OclExpression` class. It has an *evaluate* method, which returns the result of the expression. The evaluating method can be overridden in the descendant classes; it contains the code of the subtree starting from the `oclExpression` tree node.

3. A CASE STUDY

Using a case study we introduce how VMTS generates source code from a statechart diagram, applying graph-rewriting-based transformation methods. Furthermore we present how it validates specific properties using an assembly generated by an OCL Compiler during the transformation process with the help of constraints enlisted in the rewriting rules. The goal of this method is that if the statechart is specified in detail, then the generated code will handle the user interface of the system described by the statechart model.

The Cinema Ticket form is the main form of the application, which is used on mobile platform to order cinema tickets using a cellular phone.

In Fig. 5 a screenshot of the Cinema Ticket form is presented, and its operation is modeled with a statechart diagram (Fig. 6). The user interface edition of the "Cinema Ticket" form is accomplished with the form designer of the Visual Studio .NET, but the handler code is automatically generated from the statechart model.

When the form appears, the "Order" list is empty (*lbOrders*), the combo boxes (*cmbCinema*, *cmbFilmTitle* and *cmbDate*), the numeric up-down control (*nudTickets*) and the "Close" button (*btnClose*) are enabled, and the rest of the buttons are disabled. The user can create an order by selecting the desired "Cinema", "Film" and the exact date, and by specifying the number of the tickets. If a cinema is selected from the "Cinema" combo box, the Title of the "Film" combo box automatically refreshes its value, and similarly, if a film is selected, the "Date" combo box automatically loads the exact time when the movie starts. The "Add Order" and "Clear Fields" buttons (*btnAddOrder* and *btnClearFields*) become enabled when the value of the combo boxes or the numeric up-down control changes. Using the "Add Order" button, the user can add the actual values to the "Order list".

When the "Order" list contains at least one item, the "Order Tickets" button (*btnOrderTickets*) becomes enabled and naturally if an item is selected in the "Order" list, the "Remove" and "Edit" buttons (*btnRemove* and *btnEdit*) are also enabled. Using the "Order Tickets" button, the user can send the item of the "Order" list to the cinema as an SMS (or to cinemas if the list contains several cinemas). If the order was successful he gets a confirmation message.

The incomplete statechart diagram of the "Cinema Ticket" form is presented in Fig. 6, where only three events are modeled: *cmbCinema_SelectedIndexChanged*, *btnAddOrder_Click* and *lbOrders_SelectedIndexChanged*. The complete statechart diagram is too large to present here.

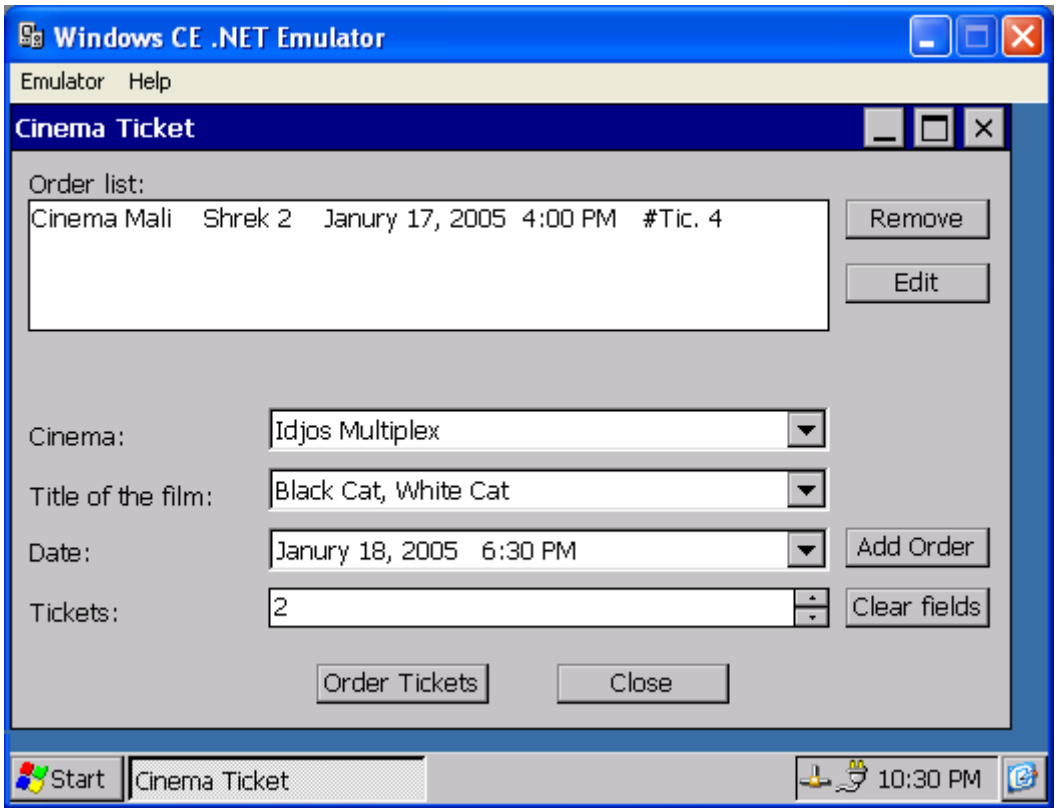

Figure 5. Cinema Ticket form for mobile platform

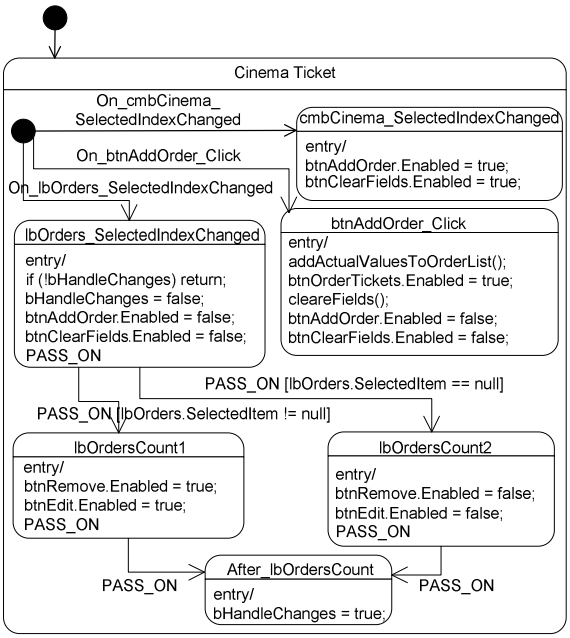

Figure 6. Statechart model of the Cinema Ticket form

In Fig. 6 one can see that each event has at least one handler state. E.g. if the *On_btnAddOrder_Click* event is fired, then the *btnAddOrder_Click* state handles it. The *On_lbOrders_SelectedIndexChanged* event is managed by four states: *lbOrders_SelectedIndexChanged, lbOrdersCount1, lbOrdersCount2,* and *After_lbOrdersCount*. This event handler is decomposed into sub-states, because the handling code depends on the value of the *lbOrders.SelectedItem* property.

The case study uses the statechart model (Fig. 6) as an input model and applies a rewriting rule (Fig. 7) to it. In the rewriting rule the LHS graph uses the meta-elements of the Statechart metamodel [UML03a] [Vis03a] and the RHS graph uses the meta-elements of the CodeDOM metamodel [Mic03a] [Vis03a]. On the left hand side of the rewriting rule there are two states which correspond to the statechart state, and there is a transition between them with a 0..* multiplicity on the side of the target state. It means that applying this rewriting rule exhaustively to a statechart model, it matches all the states with their target adjacent states. The rule has to match the accessible adjacent states, because we need them to generate the state-transitions into the source code. Of course, it is possible that a state has no outgoing transitions, and the reason why we enable the 0 in the multiplicity is that we want to match states having only incoming transitions in order to generate CodeDOM tree for them as well. On the right hand side of the rewriting rule the *CTypeDeclaration* represents a type declaration for a class, structure, interface or enumeration. *CMemberField* can be used to denote the declaration for a field of a type, and

CMemberMethod to phrase the declaration for a method. *CParameter* represents a parameter declaration for a method, property, or constructor, and *CSnippetStatement* means a statement using a literal code fragment. The code generation means a syntax tree generation (CodeDOM tree) from which the framework generates the C# source code.

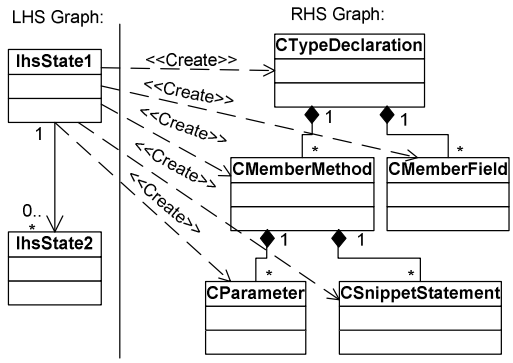

Figure 7. Rewriting rule of the case study

In a rewriting rule we can connect the LHS elements to the RHS elements, this relation between the LHS and RHS elements is called causality [Kar03a], which facilitates assigning an operation to this connection. Causalities can express modification or removal of an LHS element, and creation of an RHS element. In Fig. 7 the causalities are drawn as dotted lines. The *create* operation and attribute transformation, which is one of the most important parts of the rewriting process, are accomplished by XSL scripts. The XSL scripts can access the attributes of the object matched to the LHS elements, and they produce a set of attributes for the RHS element to which the causality point. VMTS stores models as labeled graphs, and each node and each edge have a property XML, which contains the attributes of the model element. In the current case study the VMTS rewriting engine concatenates the property XMLs of the matched states and transitions, and it uses the result as the input of the XSL script.

A part of the XSL script used by the case study to generate the rewriting result is presented in Fig. 8. The XSL selects the name of the actual state (method name) for the *methodName* variable. The first part of the script creates a NODE type *Element* with the following properties: the name of the new element should be the value of the *methodName* variable, the return type should be *void*, the modifier attribute should be *private*, the meta type should be *CodeMemberMethod*, the *RHSRuleNodeName* should be *CMemMethod*, the *ContainerName* should be *CinemaTicked* (this is the name of the class which contains the methods). Finally, the *CreatedProperties* part is also added.

```
<xsl:variable name="methodName" select="//Name"/>
<xsl:template match="/">
 <RewriteResult>
  <Element>
   <ElementType>NODE</ElementType>
   <Name><xsl:value-of select="$methodName"/></Name>
   <ReturnType>void</ReturnType>
   <Attributes>private</Attributes>
   <MetaTypeName>CodeMemberMethod</MetaTypeName>
   <RHSRuleNodeName>CMemMethod</RHSRuleNodeName>
   <ContainerName>CinemaTicket</ContainerName>
   <CreatedProperties>
    <CodeMemberMethod>
     <Name><xsl:value-of select="$methodName"/></Name>
     <ReturnType>void</ReturnType>
     <Attributes>private</Attributes>
    </CodeMemberMethod>
   </CreatedProperties>
  </Element>

  <Element>
   <ElementType>NODE</ElementType>
   <Name>sender</Name>
   <Type>object</Type>
   <MetaTypeName>CodeParameterDeclarationExpression
  </MetaTypeName>
   <RHSRuleNodeName>CParameter</RHSRuleNodeName>
   <ContainerName><xsl:value-of
   select="$methodName"/></ContainerName>
   <CreatedProperties>
    <CodeParameterDeclarationExpression>
     <Name>sender</Name>
     <Type>object</Type>
    </CodeParameterDeclarationExpression>
   </CreatedProperties>
  </Element>
...

  <xsl:for-each select="//InternalTransition/Statement">
   <xsl:call-template name="codeSnippetStatement"/>
  </xsl:for-each>
...

 </RewriteResult>
</xsl:template>

<xsl:template name="codeSnippetStatement">
 <Element>
  <ElementType>NODE</ElementType>
  <Name>Snippet</Name>
  <Statement><xsl:value-of select="Value"/></Statement>
  <MetaTypeName>CodeSnippetStatement</MetaTypeName>
  <RHSRuleNodeName>CSnipStat</RHSRuleNodeName>
  <ContainerName><xsl:value-of
  select="$methodName"/></ContainerName>
  <CreatedProperties>
   <CodeSnippetStatement>
    <Statement><xsl:value-of select="Value"/></Statement>
   </CodeSnippetStatement>
  </CreatedProperties>
 </Element>
 …
</xsl:template>
...
```

Figure 8. A part of the XSL script used by the case study to generate the rewriting result

The second presented XSL segment creates a parameter for the method, the third part selects the

Statements of the internal transitions, and it calls the *codeSnippetStatement* template for each *Statement*. Finally, a part of the *codeSnippetStatement* template is depicted.

Constraint Validation

We assign constraints to model elements and to the steps accomplished by generators to fully specify models and rewriting rules. With the help of these constraints we obtain a precise and consistent description of the transformation steps. In VMTS the main method to specify constraint validation is the relation between the pre- and postconditions and the OCL constraints assigned to the rewriting rules.

When we initialize the controls in .NET, e.g. change the *Text* value of a text box, then it a *TextChanged* event is raised, or the *SelectedIndex* property of a combo box is set, when it is sent a *SelectedIndexChanged* event. This behavior of the controls affects the operation of the form in an inappropriate way. There is an example for that in the case study, when the user selects an item in the "Orders list" and clicks on the "Edit" button, the form has to show the properties of the selected order. Hence it has to change the *SelectedIndex* value of the "Cinema" combo box, the *SelectedIndex* value of the "Film" combo box and so on. The effect of these operations is that the "Add Order" and "Clear Fields" buttons become enabled, but we do not want them so, because it is not a real property modification. We can eliminate this undesirable operation with a constraint (postcondition of the rewriting rule):

context CMemberMethod **inv** handle_changes:
if self.Type = 'EventHandler' **then** self.Statements.Count > 0 **and** self.Statements[0].Value = 'if (!m_bHandleChanges) return;'

This invariant constraint describes that if the type of an *CMemberMethod* object is *EventHandler,* then it should have more than zero *Statement*, and the value of the first statement should be '*if (!m_bHandleChanges) return;*'. A snippet statement is a code fragment, and this snippet guarantees that the event handler functions do not handle the events if the value of *m_bHandleChanges* variable is *false*.

In the "Cinema Ticket" order we require that the number of ordered tickets for a film to be at least 1 but maximum 12. Therefore if the user would like to add an order to the "Orders list", we have to validate that the value of the "Number of tickets" control is between 1 and 12. Therefore if the value of the *nudTickets.Value* is not proper, we have to prevent adding the actual values to the "Orders list", until the user does not modify the "Number of tickets" field. The constraint that describes this condition is the following (postcondition of the rewriting rule):

context CMemberMethod **inv** name_length:
if self.Name = 'btnAddOrder_Click' **then** self.Statements.Count > 1 **and** self.Statements[1].Value = 'if (nudTickets.Value < 1 || nudTickets.Value > 12) return;'

Using the following constraint (precondition of the rewriting rule), the rewriting rule validates that the states with the generated CodeDOM tree are not unreachable (isolated) states in the statechart diagram. It means that starting from the start state we can reach these states.

context state **inv** constraint_unreachable:
self.IsStartState or self.InTransitions->size() > 0

To validate the code which is generated by the OCL Compiler, please refer to [Vis03a].

When we generate source code from a statechart model, there is usually a function for each state in the generated source code, which implements the behavior of the state (the transitions and the internal transitions as well). In form-based, event-driven development the event handler methods of the controls provide the operation logic of the forms. Therefore the goal of the case study is to generate the skeleton of the user interface handler code; VMTS generates that part of the event handler methods for which it has enough information in the statechart diagram. E.g. based on the incoming and outgoing transitions and their conditions, the generator can produce a complete event handler function from several model states. An example in the case study is the *lbOrders_SelectedIndexChanged* event handler method, which is generated from four states, and its *if* branches are generated from the transition conditions. Furthermore the transformation generates the code fragments recommended by the constraints; a part of this code can be assertion code. An assertion checks a condition and displays a message if the condition is *false*. Assertions support the testing procedure and contribute to the correct operation.

Based on the presented principles, the whole process of the case study is the following: The OCL Compiler generates the constraint validation assembly, the matching process searches for topological matches in the statechart model (host graph). Then the Validation Module uses the validation assembly and checks the LHS graph containing constraints (preconditions) continuously at matching time or after the matching process on the found matches (this option if configurable in the system). If and only if a match satisfies the preconditions, the rewriting process generates the rewriting result with the help of a user defined XSL script. The Validation Module

checks the RHS graph containing constraints (postconditions) on the rewriting result. The rewriting rule is finished successfully if and only if the rewriting result satisfies the postconditions.

```
private void cmbCinema_SelectedIndexChanged(object sender,
System.EventArgs e)
{
  if (!bHandleChanges) return;
  bHandleChanges = false;
  btnAddOrder.Enabled = false;
  btnClearFields.Enabled = false;
  if (lbOrders.SelectedItem == null)
  {
    btnRemove.Enabled = true;
    btnEdit.Enabled = true;
  }
  if (lbOrders.SelectedItem != null)
  {
    btnRemove.Enabled = false;
    btnEdit.Enabled = false;
  }
  bHandleChanges = true;
}

private void lbOrders_SelectedIndexChanged(object sender,
System.EventArgs e)
{
  if (!bHandleChanges) return;
  btnAddOrder.Enabled = true;
  btnClearFields.Enabled = true;
}

private void btnAddOrder_Click(object sender, System.EventArgs
e)
{
  if (!bHandleChanges) return;
  if (nudTickets.Value < 1 || nudTickets.Value > 12) return;
  addActualValuesToOrderList();
  btnOrderTickets.Enabled = true;
  cleareFields();
  btnAddOrder.Enabled = false;
  btnClearFields.Enabled = false;
}
```

Figure 9. Generated event-handler source code

A part of the generated code is presented in Fig. 9. These C# functions form the generated *lbOrders_SelectedIndexChanged, cmbCinema_SelectedIndexChanged* and *btnAddOrder_Click* event handler methods based on the discussed statechart diagram (Fig. 6).

4. CONCLUSIONS AND FURTHER WORK

In this paper an OCL Compiler component of an n-layer multipurpose modeling and metamodel-based transformation system is presented. This work has introduced the need of combining UML and OCL during the modeling process, and discussed the steps (lexical and syntactic analysis, semantic analysis and code generation) of implementing a metamodel-based OCL Compiler module.

Based on the OCL Compiler and the possibilities provided by VMTS, a case study has been presented to show the applicability and the practical relevance of the presented tools. It has been shown that the metamodel-based graph rewriting method can be applied to transform statechart models to a syntax tree, generate source code from it, and to validate the rewriting rule constraints during the transformation

In statechart diagrams VMTS facilitates assigning function names as actions to the events. The event handler methods generated by the current version of the transformation are not fully specified ones; the user has to complete them on the source code level. As the next step of this method we will implement the feature to edit the event handler code at modeling time, and the transformation will use the specified event handler code snippets during the code generation. Furthermore, future work includes the design and implementation of branch conditions. With the help of branch conditions VMTS will support branch logic in the execution order of the rules during the transformation process, using RHS graphs containing constraints.

5. ACKNOWLEDGMENTS

The fund of „Mobile2004 Consortium" has supported, in part, the activities described in this paper.

6. REFERENCES

[Ake03a] David Akehurst, Octavian Patrascoiu: OCL 2.0 - Implementing the Standard for Multiple Metamodels, Workshop Proceedings, 6th International Conference on the UML and its Applications,<<UML>>2003, ENTCS, Oct. 2003

[Bis98a] Bison, Official Homepage, http://www.gnu.org/software/bison/bison.html

[Ext96a] Extended Backus-Naur Form (EBNF) ISO/IEC 14977:1996(E) standard

[Fle99a] Flex, Official Homepage, http://www.gnu.org/software/flex/

[Ham98a] Ali Hamie, John Howse, Stuart Kent: Interpreting the Object Constraint Language, Proceedings 5th Asia Pacific Software Engineering Conference (APSEC '98), December 2-4, 1998, Taipei, Taiwan, 1998

[Kar03a] Karsai G., Agrawal A., Shi F., Sprinkle J.: On the Use of Graph Transformations for the Formal Specification of Model Interpreters, Journal of Universal Computer Science, Special issue on Formal Specification of CBS, 2003

[Lev04a] Levendovszky T., Lengyel L., Mezei G., Charaf H.: A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS, Electronic Notes in Theoretical Computer Science, International Workshop on Graph-Based Tools (GraBaTs) Rome, 2004

[Loe03a] Sten Loecher, Stefan Ocke: A Metamodel-Based OCL-Compiler for UML and MOF. In OCL 2.0 - Industry standard or scientific playground, Workshop Proceedings, 6th International Conference on the UML and its Applications,<<UML>>2003, ENTCS, Oct. 2003

[Mic03a] Microsoft .NET Framework http://msdn.microsoft.com/netframework/

[OCL03a] Object Constraint Language Specification (OCL), www.omg.org

[OMG03a] OMG Model Driven Architecture homepage, www.omg.org/mda/

[UML03a] UML 2.0 Spec. http://www.omg.org/uml/

[Vis03a] VMTS Web Site http://avalon.aut.bme.hu/~tihamer/research/vmt