Software through Pictures®

Millennium Edition

ACD Programming Guide

UD/UG/ST0000-10150/001

Software through Pictures
ACD Programming Guide
Millennium Edition
April, 2002

Aonix® reserves the right to make changes in the specifications and other information contained in this publication without prior notice. In case of doubt, the reader should consult Aonix to determine whether any such changes have been made. The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

World Headquarters
5040 Shoreham Place, Suite 100
San Diego, CA 92122
Phone: (800) 97-AONIX
Fax: (858) 824-0212
E-mail: info@aonix.com
http://www.aonix.com

Table of Contents

Chapter 3 Template Syntax and Output Handling

Chapter 4 TDL Libraries

Chapter 5 ACD and the Merge Utility

Appendix A Merge Utility Reference

Appendix B TDL Reference

1 About Architecture Component Development

About ACD

Architecture Component Development (ACD) provides you with the capability to automatically generate solutions from your UML models. ACD can generate code for up to 70% of your system. The basis for this approach is a set of templates written in the Transformation Description Language (TDL), a template-oriented approach that allows you to specify the architectural and technical infrastructure aspects of the system under construction.

ACD makes use of UML models to generate code for applications or systems. To help you get started quickly, StP/UML provides a library of standard templates supporting the following languages: Ada 95, C++, Java, and EJB. The libraries also contain unsupported templates for C, COM, CORBA IDL, and Visual Basic.

The ensuing sections of this document address the basic elements of ACD including templates, the ACD UML metamodel, TDL, and the merge utility. The information in these sections will allow you to either reuse or tailor the templates for specific application and implementation needs.

ACD Components

Templates

The implementation information for your system is captured in a set of templates that define your system's specific architecture and all of the standards, conventions, and common mechanisms associated with it. Once these templates have been developed, they can be reused with other projects having similar requirements.

A set of supported templates is supplied with ACD. These templates can be used "as is" or modified to meet your system requirements.

The StP-supplied templates are maintained, and may be upgraded in new product releases. However, Aonix is not responsible for maintaining and upgrading user-tailored templates.

Supported templates are:

- Ada 95
- C++
- Java
- EJB

Additional unsupported templates are:

- C
- COM
- CORBA IDL
- Visual Basic

Note: These templates can be activated by toggling the appropriate variables in the StP desktop's ToolInfo editor "advanced" dialog.

The ACD UML Metamodel

The ACD UML metamodel (henceforth referred to as the metamodel) is based on the standard Object Modeling Group (OMG) metamodel. For ACD, the metamodel comprises a set of rules for ACD on how to read

UML models and extract information from them to generate code. The metamodel diagrams also provide the paths and directions you need when creating a new template.

ACD UML Metamodel Diagrams

The ACD UML metamodel diagrams are located in the directory named *<StP>/Documentation/ACD_METAMODEL*.

Transformation Description Language

The domain-specific code generation templates are written in TDL, a template-oriented language that allows you to specify the architectural aspects of the system you are constructing. For reference information on how to use TDL, see Chapter B, “TDL Reference.”

The Templates

The Template Directory

The default template directory is *<StP>/templates/uml/qrl/code_gen/<language>*.

Opening TDL Templates for Editing

StP automatically opens a language-specific template directory when **Open ACD Template** is selected.

From the **Code** menu choose *<language>* **> Open ACD Template**.

The TDL Files

There is a set of TDL files associated with each supported language.

- Each TDL file consists of procedures and templates.
- Each set contains a main procedure that is typically located in the directory *<language>_main.tdl*.

- The main procedure is called first and then calls procedures and templates in other TDL files.

Modifying Templates

How To Modify Templates

Before you begin to modify a template, you must have a good understanding of at least one of the programming languages currently supported by ACD. In addition, you must also have a good working knowledge of TDL and the ACD UML metamodel.

To modify a template:

- Browse through the metamodel diagrams located in *<StP>/documentation/ACD_METAMODEL*. Use these diagrams as your road maps.
- Choose one supported template as your base template.
- Generate code from the base template so that you can compare the output code with its template.
- Open the base template. **Choose File > Save As**. Change the name to reflect your modified template so that you have two files to work with. Make changes only in the new file.
- Use the base template, its output code, the metamodel diagrams, TDL, and your knowledge of programming languages to guide you as you modify your template.

Note: When you are developing templates and do not change your model each time you run the template, you can turn off **Generate IMF File** in the **Generate** <*language*> **by ACD Template** dialog to speed up code generation. IMF needs to be generated only when the model is modified.

Templates are easily tailorable. You can use a third-party text editor, such as Notepad (for Windows), *vi* (for UNIX), etc., to modify templates. For Windows platforms, we recommend a third-party editor called UltraEdit. The tool is available at *www.ultraedit.com*. Aonix can provide a TDL language extension to use with UltraEdit for highlighting reserved words.

Additional References

The following white papers are available on request from your Aonix representative:

- *Jump Starting ACD Code Generators*

  This paper discusses how easy and quick it is to write a new code generator using ACD technology.
- *Combining Patterns with ACD*

  This paper shows how to write templates for debugging information, get()/set() methods, and a simple design pattern that allows templates to be shared among different projects.

2 The ACD UML Metamodel

Overview

The ACD UML metamodel, based on the OMG metamodel, is a roadmap that tells you how to extract information from a model system.

Based on the metamodel, TDL was created as a general mechanism for code generating. It is a high-level programming language.

Access to the Metamodel

Transformation of the object model into code is the central part of code generation. To transfer the object model into code, analysis information is dealt with, evaluated, and processed. This information is stored in the metamodel.

Basic Concept

Access to the metamodel begins with a loop statement. In connection with the navigation rules, the result quantity is decided upon. With metaattribute variables, information can be read within the block statements of a loop.

```
loop (navigation rules)// defines the result quantity
                        // within a block the result quantity
                       // can be read with the
                       // MetaAttribute variable...
```

```
end loop
```

Loop Statements

In order to access the metamodel, always use the keyword *Instances* inside the first loop statement. Once inside this loop block, you can navigate in the metamodel without using the Instances keyword. This is true until you exit the loop block. Each loop starts with an instance navigator. For example:

```
loop(Instances->MClass...)

loop(Instances->Package...)
```

As a result, all variants that are derived from the navigation rules are applicable. For example:

```
loop(Instances->MClass->MAttribute...)
loop(Instances->Package->MClass->MState->MNormalState->
                ToState->MTransition...)
```

Each metaclass that is mentioned in a navigation rule defines an instance. The instance is valid only within a loop block. The navigation rule also gives information about the type and the name of the instance.

The following example gives both instances the names MClass and MAttribute and the types the names MClass and MAttribute.

```
loop(Instances->MClass->MAttribute)

             statements ...
end loop
```

Such an instance is required to access the metamodel via a metaattribute variable.

For example:

```
loop(Instances->MClass)//Navigation Rule defines the
                   //instance
                   // MClass

   out = [MClass.name];// The instance MClass is a reference
                   // to be able to take hold of the
                   // attribute Name
```

```
end loop
```

Furthermore, the instance allows a nested loop statement. Each inner loop can relate to an instance defined in an external loop.

For example:

```
loop(Instances->MPackage->MClass)
                    // The instance MPackage
                    // and MClass are defined.

   statements ...

   loop(MPackage->SubPackage)// The loop refers to the
                    // instance package and navigates over
                    // all SubPackages. The loop
                    // itself defines the instance
                    // with the name SubPackage, which
                    // belongs to the package type.
      statements ...

   end loop

   statements ...

   loop(MClass->ReceiveEvent)// MClass is taken
                    // as a reference.
                    // Instance ReceiveEvent of the type
                    // MEvent is newly defined.

      statements ...

      loop(ReceiveEvent->MState)

         statements ...

      end loop
   end loop
end loop
```

Navigation Rules

The syntactic structure of the navigation rules is as follows:

> MetaModel-Symbol "->" MetaModel-Symbol ["->" MetaModel-Symbol]...

Any class, relation name, or role name can fulfill the metamodel symbol position. For example:

- MClass->MAttribute
- MClass->MOperation->MOperationPara

The navigation is divided into three basic navigators:

- Instance navigator
- Inheritance navigator
- Relational navigator

Instance Navigator

The instance navigator defines the repetition of all instances of a metaclass in the system. Each instance navigator begins with the keyword *Instances*, followed by a loop separator (->) and a metaclass.

For example, to define all classes in a system, you use:

> loop (Instances->MClass)

To define all states in a system, you use:

> loop (Instances->MState)

Inheritance Navigator

You can also navigate over inheritance trees in the metamodel. It is possible to navigate either from the parent to the child class or from the child to the parent class. When the direction of navigation is from parent to child class, it is understood as run-time type checking.

For example:

- MState->CreationState

  This navigation only gives a subset of the MStates, namely the one from the subtype CreationState.

Note: In the metamodel, child classes inherit relations and attributes from the parent class. Therefore, child classes have access to all navigations of the parent class.

For example:

- NormalClass->MAttribute

  NormalClass inherits all relations from MClass.

Relational Navigator

The relational navigator handles navigation over relations. Since the navigation over a relation is not automatically defined, the relational navigator is further subdivided into the following:

- Class-relational navigator
- Role-relational navigator
- Self-relational navigator
- Assoc-relational navigator

Class-relational navigator

Paths over relations leading from one class to another are defined by the class-relational navigator. The path is determined by the required result of a loop.

The class-relational navigator is characterized by metamodel class "->" metamodel class.

For example:

- loop (Instances->MClass->MAttribute)

  With this rule, all classes together with their attributes are collected.

Note: When you use the class-relational navigator, always use the keyword *Instances* inside the first loop statement. Once inside this loop block, you are can use the navigator without using the Instances keyword. This is true until you exit the loop block.

Role-relational navigator

When there are bidirectional paths (relations) between two classes in the metamodel, use the role name to define the direction of the path. It is important to always use the role name of the class you are navigating *toward*. The role-relational navigator must be used when a class in the metamodel has a role name.

For example, there are two paths (relations) between MClass and MEvent.

To obtain all events that a certain class receives:

- MClass->ReceiveEvent

  MClass is the class name, while ReceiveEvent is the role name of the class navigated toward.

To use the role-relational navigator to navigate from MState to MEvent:

- MState->IgnoreEvent

  In the metamodel, IgnoreEvent is the role name assigned to MEvent.

Self-relational navigator

Self-relating relations, as they are owned by MClass and MState, represent a direction-bound dependency between similar types of symbols.

Therefore, self-relating relations also require a clear navigation direction.

- MState->ToState

  Defines all subsequent states possible from a certain state.
- MState->FromState

  Defines all states that lead to the current one.
- MClass->SuperClass

  Defines all subsequent classes.

Note: If a self-relating relation has a name, both navigation directions can be registered with one rule.

The rule is structured as follows:

- Metamodel class "->" metamodel relation name

For example:

- MClass->Partner

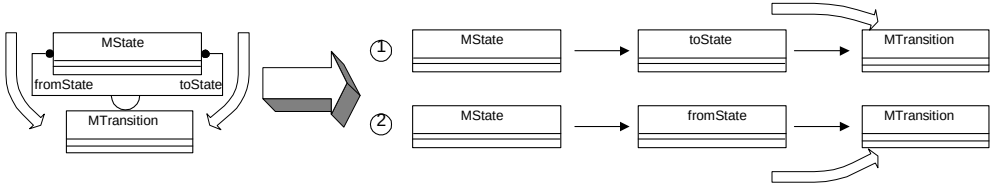
Association class - relation navigator

The last relational navigator deals with the navigation of associative metarelations. This navigation distinguishes between the navigation *toward* the associative class and the navigation *from* the associative class. In place of this kind of relation, the metaclass MTransition is described as an example.

1. Navigation toward the associative class

   The navigation of the associative class uses the following substitutional diagram:

Figure 1: Substitutional Diagram for Navigation Toward the Associative Class



   If all transitions lying between two states are to be registered, the state pair needs to be fixed beforehand as follows:

   - MState->ToState

   or

   - MState->FromState

   Afterward, the transition can be located. The complete rule is then:

   - MState->ToState->MTransition

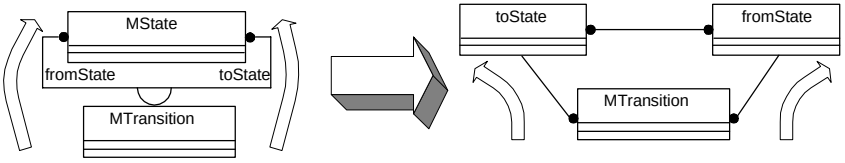
   or

- MState->FromState->MTransition.

2. Navigation from the associative class

   If the transition is known and the involved states are to be located, the navigation rule needs to be derived from the following substitutional diagram:

Figure 2: Substitutional Diagram for Navigation From the Associative Class



Thus, there is the choice of either locating the FromState:

- MTransition->FromState

or the ToState:

- MTransition->ToState

If there are no roles on the associations, the associations can still be accessed. For example:

In the association between MState->MAction there are no roles. To access MActionType, which is the association class, the navigation is:

- MState->MAction->MActionType

Alias Mechanisms

The alias mechanism allows you to change the name of an instance. With the key word *AS* ([aA][sS]) after the instance definition, an alias is allocated. The alias marker, as with local variables, consists of an initial text sign or an underline, followed by further text signs, numbers or underlines ([_a-zA-Z][_a-zA-Z0-9]*). Be aware that further reference of this instance is regulated by the alias marker.

```
loop(Instances->MClass->Role As FromRole->MAssociation as
MRel->MAssociationEnd As ToRole Where [FromRole.id] !=
[ToRole.id])
    out = [MRel.name]; // Access to the original
```

```
                    // Instance MRelationship now takes
                    // place over the name MRel.
   loop(MRel->AssocRel)// The alias term is also valid for
                   // the reference of the inner loop

      statement ...

   end loop
end proc

proc generateCode(MClass, MClass AS Partner)

   out = [MClass.name] ":" [Partner.name] "\n";
end proc
```

Where Conditions

Optionally, a loop can be regulated by a where condition. The use of the condition depends on the logical operators.

```
loop(Instances->MClass Where[MClass.stereotype]=="Segment")

      statement ...// Iterates over all classes that contain
                   // the stereotype segment.
end loop
```

Loop Procedures

Within a loop-statement procedure, calls might be found that are called either during the first loop round or with every round that follows. The procedures of the first loop round are listed according to *where* connections separated by a semicolon. Following that are all procedures that should be carried out during later loop rounds.

```
loop(MOperation->OpPara; setDelim(""); setDelim(","))
                   // The procedure setDelim is called up
                   // with a blank string during the first
                   // loop round, in any further round with
                   // a comma as argument.
[para] = [para] delim() [OpPara.type] [OpPara.name]
                   // This causes that only from the second
                   // round on, a comma of delim()
```

```
                    // is generated and an argument-list,
                    // separated by comma, arises.
                    //
end loop
```

ID-Set Mechanisms

A loop statement can be extended to include an ID-set mechanism. In this case, a list is defined that will deliver the result quantity of the loop. To each metamodel element, an unambiguous ID is assigned that can be used for it. They are listed within a pair of brackets in the loop after each instance definition.

```
loop(Instances->MClass(Id-List))

      statements ...

end loop
```

The ID list consists of one or several element IDs, separated by white spaces (tabs or blanks). It can be generated manually and/or by the ID-Set-Mechanism. The following code frame shows an example of how an ID list is generated and afterward can be used in a loop statement. For that reason, getIdListRec is employed.

```
loop(Instances->MClass)// Navigation of all
                       // classes in the system

   [ID_LIST] = getIdListRec(MClass->SuperClass);
                    // Generates an ID-list of all parent-
                    // classes

   loop(Instances->MClass([ID_LIST]))
                    // Navigation over all parent-classes

                    statements ...
   end loop
end loop
```

Metaattribute Variables

Metaattribute variables serve to read metamodel information. The name of the variable consists of two parts separated by a period and surrounded by square brackets, for example, [MClass.name]. The first part of the name is defined by the instance reference of the loop statement. The second part is characterized by the name of the attribute from which the value should be read. The choice of attributes is already given with the type of metamodel.

```
loop(Instances->MClass->Role As FromRole->MAssociation as
MRel->MAssociationEnd As ToRole->MClass As Partner Where [FromRole.id] !=
[ToRole.id])
out = [MClass.name] ":" [Partner.name] ":"
[MRel.name] "\n";
end loop
```

The metamodel also allows access to attributes of the parent class. This way, an instance of the type NormalClass can also access attributes of its parent class MClass.

```
loop(Instances->NormalClass)

   out = [NormalClass.name];
      // Name is an attribute of the
      // MetaClass MClass
end loop
```

Metaproperties

A met a property works like a user-defined attribute. At run time, a metaproperty is tied to a metaclass. A property's name consists of two parts separated by a colon. The first part corresponds to the instance reference of a loop statement. The second part characterizes the property's name. A property is defined as a left value and read as a right value.

```
loop(Instances->MClass)
   [MClass:AnyPropertyName] = [MClass.Id] ...;
             // property is defined and a value
             // is assigned.
end loop
```

```
loop(Instances->MClass)
   out = [MClass:AnyPropertyName];
      // property is used as right value
      //
end loop
```

To write a property:

```
loop (Instances ->MClass)
   [MClass:filename]=toLower ([MClass.name]);
end loop
```

To access a property from a parent metaelement:

```
loop (Instances->MNormalClass)
   output ( [MNormalClass:filename] ".cpp");
end loop
```

StP to Metamodel Mapping

This section describes the mapping between the metamodel objects and the objects in the StP editors. Most of the StP objects listed in this table display their type name in the lower right corner of the StP editor window if the mouse pointer is located on top of the object. Other objects in this table refer to annotation items, annotation notes, etc., as specified in the Notes column.

Table 1: StP to Metamodel Mapping

Metamodel Object	StP Object	Notes
MClass	Class	
MPackage	Package	
MOperation	Operation	
MAttribute	Attribute	

Table 1: StP to Metamodel Mapping

Metamodel Object	StP Object	Notes
MOperationParam	Operation parameter	
MAssociation	Association	Association link between two classes in a class diagram
MAssociationEnd	Role	Role specified at end of association link
MInheritance	Generalization	Generalization link between two classes in a class diagram
MTemplateClass	Parameterized class	
MTemplatePara	*<'Parameters' item>*	In the annotation editor, this is the Parameters item under the Class Definition note for a parameterized class. Also available through **Edit >Properties**.
MInstantiated-Class	Instantiated class	
MTemplate-Binding	Binds	The link between an instantiated class and its template class
MDataType	Class	A class with stereotype of DataType
MNormalClass	Class	Any class that is not a data type or an instantiated class or a template class
MInterface	Interface	
MDependency	Class	Connects MDependentElement instances MPackage, MClass, MAttribute, and MOperation are subclasses of MDependentElement.

Table 1: StP to Metamodel Mapping

Metamodel Object	StP Object	Notes
MNested	*<Nested link between 2 classes>*	No dierct StP Object type exists for MNested. This metamodel object refers to the nested link between two classes. Two classes in StP are nested when one contains the Class Definition annotation note with the Enclosing Scope item.
MObject	Object	
MPara	N / A	Exists in metamodel as a metaobject only. Supplies attributes to other metamodel objects.
MParaItem	N / A	Exists in metamodel as a metaobject only
MDependency	N / A	Exists in metamodel as a metaobject
MStateMachine	State machine	In state diagram editor
MTransition	State transition link	Link between two states. Note: Two separate MTransitions can be associated with the same MEvent (label)
MEvent	Name portion of label of the state transtition link	Portion of the label of the state transition link that represents the name of the event in StP. Format for label is: <name>[<guard>]/<actionlist>^<target>,<sendevent>(<args>) Note: the same MEvent can be associated with two separate MTransitions, whereas an MTransition is identified by the two states it connects

Table 1: StP to Metamodel Mapping

Metamodel Object	StP Object	Notes
MEventPara	Argument list on event	List of arguments of the send event on the label of the state transition link Format for the label is: <name>[<guard>]/<actionlist>^<target>, <sendevent>(<args>) Format for <args> is: <var>:<type>, <var>:<type>, ...
MState	State	State found in the state transition diagram
MAction	Action	In StP, actions can be specified in two ways: 1. Actions can be listed in the state table. 2. Actions can be listed as part of the label of the state transition diagram. Format for the label is: <name>[<guard>]/<actionlist>^<target>, <sendevent>(<args>) Format for <actionlist> is: <action>, <action>, <action>, etc.
MActionType	'Entry' or 'Exit'	In state tables, entry action has an MAction type of 'Entry' and exit action has an MActionType of 'Exit'
MCreationState	Initial state	
MFinalState	Final state	
MNormalState	State	Any state that is not a final state or an initial state
MCompositeState	State	

Table 1: StP to Metamodel Mapping

Metamodel Object	StP Object	Notes
MTaggedValue	*<Tagged value item>*	Tagged value from the property sheet or tagged value items under the Extensibility Definition annotation note
MAnnotNote	*<Annotation note>*	Any note in the annotation editor
MAnnotItem	*<Annotation item>*	Any item in the annotation editor.
MUseCase	Use case	
MExtend	Extends	Extends link between use cases
MActor	Actor	
MTarget	Target portion of the state transition link	Portion of the state transition link label that represents the target of the send event. Format for the label is: <name>[<guard>]/<action>^<target>,<sendevent>(<args>)
MPseudoState	Shallow history, deep history	Shallow history, deep history, junction point, and dynamic choice based on the [MPseudoState.kind] attribute

3 Template Syntax and Output Handling

General

This chapter provides information supplementary to ACD. Its main goal is to explain in detail how ACD handles templates, and to show possible solutions to problems that might occur during the template development process.

The Template Statement in TDL

All templates start with the keyword “template” followed by the template name and end with “end template”. For example:

```
template <template name>
....
....
end template
```

Note: The term “template” is typically used for both a TDL file (i.e., a Java template) and the template statement in TDL. In the scope of this chapter, the term refers to the TDL statement.

All lines between the template delimiters are written literally to the output. This means that anything you write is written directly to the output file. One exception is that procedural code statements can be inserted into the template enclosed in square brackets `[]`. This is used to include textual items and embed control structures in the output.

Using Variables in Templates

Variable values can be accessed in templates, but cannot be modified. Any attempt to modify the value of a variable will produce an error. To access the value of a variable, just enclose the variable name in square brackets. Here is a simple “Hello, World!” example:

```
proc main()
[var1]="Hello";
[var2]="World";
out = t1("END!");
end proc

template t1(var)
[var1], [var2]!
[p1([var])]
end template

proc p1(var)
    return [var];
end proc
```

Output:

```
Hello, World!
END!
```

Although you cannot directly modify variables in templates, you can do it indirectly. Because all variables are global by default, you can use a procedure to modify variables: The first parameter is the variable name; the second is the new value. A procedure for doing it already exists in *std.tdl* and is called setVar. Here is the definition of the procedure:

```
proc setVar(varName, varValue)
   [[varName]] = [varValue];
end proc
```

Adding Comments to Templates

When a TDL template is developed, it is useful to add comments to the code. Comments not only help make support easier, but also make the code more readable. In templates, comments are enclosed in square brackets; they start with two slashes and end at the end of the line. For example:

```
proc main()
out = t1();
end proc

template t1()
start
    [//comment line]
    some line between
    [// another comment line]
end
end template
```

Output:

```
start
    some line between
end
```

Calling Procedures or Templates

All procedure or template calls from within a template should be enclosed in square brackets. For example:

```
proc main()
out = main_template();
end proc

template t1(a)
template [a]
end template

proc p1(a)
    return "procedure "[a];
```

```
end proc

template main_template()
    This is just an example of calling procedures and templates
    from within a template :
        [p1("p1")]
        [t1("t1")]
end template
```

Output:

```
    This is just an example of calling procedures and templates
    from within a template :
        procedure p1
        template t1
```

Indentation

When a template is called from within another template, ACD preserves the current text indentation. For example:

```
proc main()
out = t1();
end proc

template t1()
    t1 - line 1
    t1 - line 2
    [t2()]
end template

template t2()
    t2 - line 1
    t2 - line 2
end template
```

Output:

```
    t1 - line 1
    t1 - line 2
        t2 - line 1
        t2 - line 2
```

Here, the text indentation of *t2* (4 spaces) in template *t1* is preserved and added to each line generated by *t2*.

Handling New Lines

Although new line characters are not visible while editing TDL templates, they have great impact on the code/text generated from the template.

In templates, every line ends with a new line character. ACD handles this properly and in cases when the template or procedure is called from within a template, the last new line symbol is stripped from the returned result. When a template or procedure is called from within a procedure, the last new line symbol is present in the returned result. For example:

```
proc main()
out = t1();
end proc

template t1()
    Start
    [t2()]
    End
end template

template t2()
    There is a new line character at the end
end template
```

Output with special new line handling:

```
    Start
           There is a new line character at the end
    End
```

Output without special new line handling:

```
    Start
        There is a new line character at the end

    End
```

In the second output box, the extra new line occurs because both the calling and the called template have an ending new line, whereas in the first output box (8.3 new line handling), the new line character of the called template is stripped off.

"Hard New Line" Characters

There is one special case in new line handling. When a template or procedure that returns nothing (an empty string) is called from within another template, and the call is alone on the line (no characters except spaces and tabs exist on that line), the line is removed from the output.

```
proc main()
    out = t1();
end proc

template t1()
    Start
    [empty()]
    End
end template

template empty()
end template
```

Output:

```
    Start
    End
```

There are some cases where blank lines need to be included in the output whether they are empty or not. In this case a "hard new line" character can be used. This special character is a caret ('^'), which is put last on the line (no additional tabs or spaces). This forces ACD to put the current line in the output whether it is empty or not. When template *t1()* is modified as follows, the output contains a blank line.

```
template t1()
    Start
    [empty()]^
    End
end template
```

Output:

```
    Start

    End
```

If ^ is the last character on the line, it has to be preceded by a backslash escape character (‘\’).

```
template t1()
    Start
    [empty()]\^
    End
end template
```

Output:

```
    Start
        ^
    End
```

Backslash Characters

The backslash character is used to notify ACD that no new line character should be generated for the current line. You can use it anywhere in a template as an ending line character. In this case, it works just like a line continuation. For example:

```
proc main()
out = t1();
end proc

template t1()
    Start
        line1\
        line2
        line3
    End
end template
```

Output:

```
   Start
       line1        line2
       line3
   End
```

Handling String Results Returned from TDL Statements

Often functions return a string that contains a new line character. Calling such functions within a template can cause unexpected results in some cases. For example:

```
proc main()
out = bad_indentation();
out = better_indentation();
end proc

template bad_indentation()
BAD INDENTATION
        ["   a\n   b\n   c\n"]\
end template

template better_indentation()
BETTER INDENTATION
["   a\n   b\n   c\n"]\
end template
```

Output:

```
BAD INDENTATION
           a
   b
   c
BETTER INDENTATION
   a
   b
   c
```

The first template causes unexpected results, while the second works fine. The reason this happens is that the string is expanded in the current template as is, without indentation. It is the responsibility of the TDL function or the TDL statement to prefix every line (except the first) with the current indentation in the calling template. Note that only results returned from templates have the correct indentation. This is because ACD does it automatically, as the template is executed and before the result is expanded in the calling template.

The second template above produces the expected result. This means that you can use such strings without modification only in cases where they stand alone on a line and start from the beginning of the line.

To solve the indentation problem, ACD's getIndentCount() function and TokenSet loop can be used. The createString() function from *std.tdl* can be used as well. For example:

```
USES std;

proc main()
out = indentation();
end proc

proc p()
    return "   a\n   b\n   c";
end proc

template indentation()
    [fixUp(p())]
       ------------------------
          [fixUp("   a\n   b\n   c")]
       ------------------------
    [fixUp(p())]
end template

proc fixUp(block)
    local res = "";
     loop (Instances->TokenSet([block]); setDelim("");
setDelim(createString(" ",getIndentCount())))
         [res] = [res] delim () [TokenSet.line]"\n";
    end loop
    return [res];
end proc
```

Note that each call to fixUp() in the template ends with a backslash character. There is also another limitation: to produce expected results, the fixUp() function call should be first on the line (and called only once). In other words, there should be no template / procedure calls or TDL statements before the fixUp() function call on that line.

Output:

```
       a
       b
       c
 ------------------------
              a
              b
              c
 ------------------------
a
b
c
```

Using Conditional Statements in Templates

As described previously in this document, any TDL statements can be used in templates, but they must be enclosed in square brackets. Very useful in TDL templates are control flow statements such as *if..then*, *loop()..end loop*, etc. Here is a simple example:

```
proc main()
out = t1("");
out = t1("b");
end proc

template t1(a)
start
    [if ([a] == "b")]
    [t2()]
    [end if]
end
end template
```

```
template t2()
template line 1
template line 2
end template
```

Output:

```
start
end
start
    template line 1
    template line 2
end
```

The example above shows that conditional statements act like guards - if the condition is met, the code block is included in the generated code. Note that the lines that the [if] and [end if] statements occupy are not included in the generated text. *This is a very important feature of ACD*. It is valid for every control flow or loop statement.

Because control flow statements are not included in the generated code / text, do not use one-line *if..else..endif* statements. The results may be unexpected. For example:

```
proc main()
out = t1("b");
out = t1("c");
end proc

template t1(a)
start
    [if ([a] == "b")][t2()][else][t2()][end if]
end
end template

template t2()
line1
line2
end template
```

Note: If template *t2* had been called from a procedure instead of a template, the last statement ("template line2") would have needed a backslash!

Output:

```
start
line1
                    line2
end
start
line1
                                line2
end
```

For good text formatting, use an *if...else...endif* construct instead:

```
[if ([a] == "b")]
[t2()]
[else]
   [t2()]
[end if]
```

Real World Example - Generating Comments

When code is generated, comments are generated as well. Often, comments are expected to look like a label:

```
/***************************/
/* Classes:                */
/*   Class_one             */
/*   Class_another         */
/***************************/
```

However, if you try to produce output similar to the above, using only templates, you will not succeed.

The best way to do it is to create a procedure to be called from within a template that returns a fixed number of characters. For example:

```
USES std;

proc main()
out = templ();
end proc
```

```
template templ()
    ##########################################
    # Operations defined:                    #
    [loop(Instances->MOperation)]
    #     [field([MOperation.name],36," ")]#
    [end loop]
    ##########################################
end template

proc field(str, width, fillchar)
    local len;
    [len] = getLength([str]);
    [len] = sub([width],[len]);
    return [str] createString([fillchar],[len]);
end proc
```

Possible output:

```
    ##########################################
    # Operations defined:                    #
    #     one_argument_funtion               #
    #     small_function                     #
    #     another_longer_function            #
    #     some_function                      #
    #     function                           #
    ##########################################
```

Data Driven Indentation

So far we have discussed static indentation, or indentation that is coded in the templates and is not sensitive to the data model being processed. In practice, there is often a need for data driven indentation, or indentation calculated at run time that depends on the data model being processed.

Example 1

Suppose you need to produce the following code:

```
class MyClass {
              float func1();
              int func2();
```

```
            char func3();
            }
class EmptyClass {}
```

In this case, the indentation of the functions in the class body depends on the length of the class name. So the length needs to be calculated dynamically and blank spaces appended to the next class body lines:

```
USES std;

proc main()
out = data_driven_indentation();
end proc

template data_driven_indentation()
[loop (Instances->MClass)]
////////// [MClass.name] declaration //////////
    [if (hasLoop(MClass->MOperation))]
    [setIndentation("class " [MClass.name] " ")]{
    [loop (MClass->MOperation)]
    [getIndentation()][MOperation.returnType]
[MOperation.name] ();
    [end loop]
    [getIndentation()]}

    [else]
    class [MClass.name] {}

    [end if]
[end loop]
end template

proc setIndentation(var)
    [__I_N_D_E_N_T_A_T_I_O_N__] = insertSpaces([var]);
    return [var];
end proc

proc getIndentation()
    return [__I_N_D_E_N_T_A_T_I_O_N__];
end proc

proc insertSpaces(tmplTxt)
    return replace([tmplTxt], ".", " ");
end proc
```

Output:

```
////////// MyClass declaration //////////
    class MyClass {
                  float some_function ();
                  void another_longer_function ();
                  int small_function ();
                  void one_argument_funtion ();
                  }

////////// AnotherClass declaration //////////
    class AnotherClass {
                       float function ();
                       }

////////// EmptyClass declaration //////////
    class EmptyClass {}
```

Example 2

Suppose you want all functions in your code to have the following format:

```
int func_name ( int arg1,
                float arg2 );
float another_func ();
```

At first it might appear that the approach should be the same as in Example 1. However, in this case the function arguments are formatted differently: there is a comma between every argument pair but no comma after the last argument. This requires a different solution. *Solving the problem by just using templates does not work; you must use a procedure.*

```
USES std;

proc main()
out = data_driven_indentation();
end proc

template data_driven_indentation()
[loop (Instances->MClass)]
class [MClass.name] {
    [loop (MClass->MOperation)]
    [fixUp(genOpParaList([MOperation]))]\
```

```
    [end loop]
}
[end loop]
end template

proc genOpParaList(MOperation)
    local operation_sig =
setIndentation([MOperation.returnType]" "[MOperation.name]"
(");
    local first_delim = "";
    local next_delim = ",\n" getIndentation();

    loop(MOperation-
>OpPara;setDelim([first_delim]);setDelim([next_delim]))
        [operation_sig] = [operation_sig] delim() " "
[OpPara.type] " " [OpPara.name];
    end loop

    [operation_sig] = [operation_sig] ");";
    return [operation_sig];
end proc

proc fixUp(block)
    local res = "";
    loop (Instances->TokenSet([block]); setDelim("");
setDelim(createString(" ",getIndentCount())))
         [res] = [res] delim () [TokenSet.line]"\n";
    end loop
    return [res];
end proc

proc setIndentation(var)
    [__I_N_D_E_N_T_A_T_I_O_N__] = insertSpaces([var]);
    return [var];
end proc

proc getIndentation()
    return [__I_N_D_E_N_T_A_T_I_O_N__];
end proc

proc insertSpaces(tmplTxt)
    return replace([tmplTxt], ".", " ");
end proc
```

Output:

```
class MyClass {
    float some_function ( int a,
                          char b,
                          float c);
    void another_longer_function ( int a,
                                   char b);
    int small_function ();
    void one_argument_funtion ( int a);
}
class AnotherClass {
    float function ();
}
class EmptyClass {
}
```

4 TDL Libraries

Along with the main functions of TDL, additional functions are kept in libraries. These can be added by the USES mechanism.

```
USES c_std;
proc genAnyThing()
...
end proc
```

Typically, libraries are organized into two parts, a language-independent library and one or more language-dependent libraries. The language-independent library is available in *templates/code_gen/tdl/std.tdl* and is usually included in all templates using a relative USES statement:
`USES ../tdl/std;`

It provides basic utility procedures and templates (see Table 1 on page 4-2 for an overview and the comments in this file for a detailed explanation).

Language-dependent utility procedures and templates are typically defined in *templates/code_gen/<language>/<language>_std.tdl*.

The Standard Library

The standard library (*std*) provides general, language-independent functions.

Table 1: Standard Library

Function	Parameter Type	Description
add(expression, expression)	1. Integer 2. Integer	Adds both arguments
sub(expression, expression)	1. Integer 2. Integer	Subtracts both arguments
mul(expression, expression)	1. Integer 2. Integer	Mulitiplies both arguments
div(expression, expression)	1. Integer 2. Integer	Divides both arguments
time()		Gives the current time
date()		Gives the current date
hostname()		Gives the host name.
setCount (expression)	Integer	Sets an initial value
addCount (expression)	Integer	Adds any number to the initial value
getCount()		Returns the current value
setDelim (expression)	String	Defines a delimiter
delim()		Returns the current delimiter back
setString (expression)	String	Defines any string

Table 1: Standard Library

Function	Parameter Type	Description
getString()		Returns the current string and then erases the string's contents
replace (expression, expression, expression)	1. String 2. Regular expression 3. String	Replaces in string (1) the occurrences defined in regular expression (2), with the string (3) in the third parameter. The result is given as the function's return.
toUpper (expression)	String	Converts a string into capital letters and returns the result
toLower (expression)	String	Converts a string into small letters and returns the result
regexp (expression, expression)	1. String 2. Regular expression	Supplies the part of string (1) covered by the regular expression (2)
split (expression, expression, expression)	1. String 2. Delimiter 3. Position	Divides string (1) by giving a delimiter (2) in numbered multiple parts. With position (3) a specific part can be accessed. Position can be: - 'T' , which supplies the last part - 'L', which supplies the first part - Number, which supplies the part connected with that number
createString (expression, expression)	1. String 2. Integer	Supplies n-occurences (2) of string (1)
getLength (expression)	String	Returns the length of the string
lsdir (expression)	String	Extracts the directory name from a file name
mkdir (expression)	String	Creates a directory

Table 1: Standard Library

Function	Parameter Type	Description
getDataType (expression)	1. Type	Supplies the type of data according to guidelines in “Type Mapping” below.
output (expression)	String	Sets the output channel on the file defined by the transfer parameter. Before that, the directory structure is created from this file if it does not already exist
getOutput()		Supplies the name of the current ouput file) works only in connection with output)

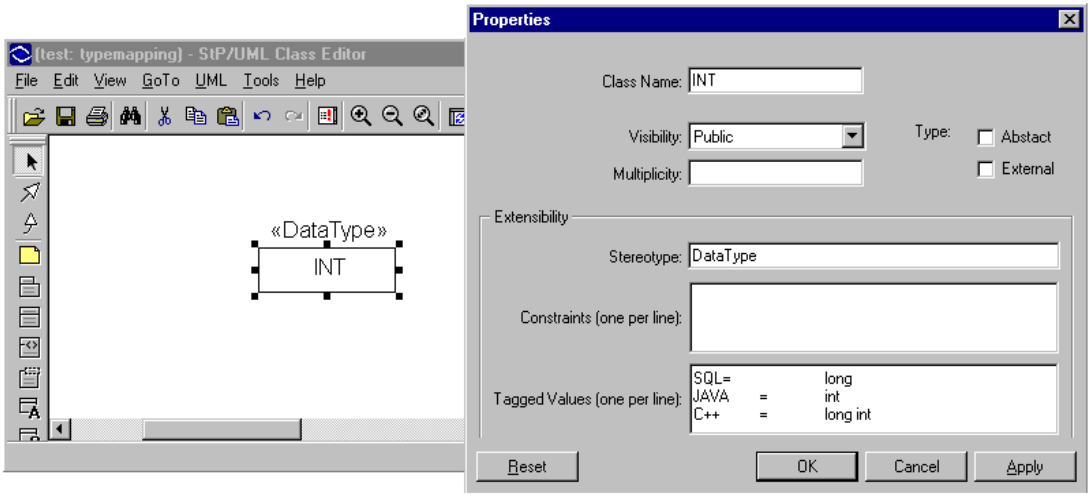
Type Mapping

One of the most powerful concepts of ACD is type mapping. Type mapping allows the decoupling of types used in analysis and design from the types needed in the implementation. Type mapping also helps to generate all necessary include, forward, and import statement; this allows the creation of 100% compilable code.

Furthermore, technology-specific type handling can be supported within the framework of type mapping. An example in the CORBA environment is the mapping of CORBA types to the respective implementation language types, and project-specific type conventions.

The type mapping is realized by UML classes. Therefore, a class will have a stereotype DataType. The class name corresponds to the analysis type. The different language-specific types are assigned by tagged values.

.



The respective type can be determined by the function getDataType (standard library *std*). Otherwise, direct access to the corresponding data type is also possible via the metamodel. Data type classes are depicted in the metamodel as MDataType

```
getDataType("INT","SQL");// returns "long"
getDataType("INT","JAVA");// returns "int"
getDataType("INT","CPP");// returns "long int"
```

5 ACD and the Merge Utility

This chapter describes the default merge configuration.

Configuration Files

The behavior of the Merge utility is based on entries in a configuration file. There is one configuration file for each language:

<StP> \ templates \ ct \ merge \ MergeConfig \ merge_config_<lang>.txt

Note: The default behavior of Merge covers most all needs. You can change its behavior to satisfy your needs by modifying these configuration files.

Comment Blocks

By default, ACD does not use code blocks for generated code. Instead, ACD creates comment blocks for the code to be added to source code files manually. This differs from the normal usage of Merge where the generated code should be included in code blocks and manual additions should be outside any code block. Advanced users can change it by modifying TDL files and the merge configuration file for the specific language.

Here is an excerpt of the code generated by ACD:

```
...
ClockBird::ClockBird()
{
```

```
   //#ACD# M(UDCC) Constructor Code <--comment tag(code
//block starts)
       //user's code to be manually added here...
   //#end ACD# <comment <--comment tag (code block ends)
}
...
```

After ACD generates code, Merge compares the previous version of a file with a newly generated file (same name with a leading underscore).

- If there are no differences for the code *outside* code blocks (i. e., generated code only), then it deletes the newly generated file and that is it.
- If there are differences (the model has changed), Merge extracts all pieces of code from code blocks (surrounded with comment tags) from the previous version and inserts them into the corresponding places in the newly generated file; then it removes the older file and renames the newly generated one (and removes the leading underscore).

Then ACD moves the previous version of the generated code to the bottom of the file and marks it as comments.

Syntax of Comment Tags

Comment tags are comment lines to let the merge utility know which lines are generated and which are manually changed / inserted.

Code lines between such comment tags are called code *blocks*.

A code block is started by a line of the following syntax:

```
   //#ACD#X(blockid)comment usermaintained=no
```

and ended by a line of the following syntax:

```
   //#end ACD#comment
```

where:

X: One character for distinguishing diverse merge strategies. Currently, **G** is used for a generated block; **M** for a manual block.

blockID: Unique ID for this merge block. This ID has to be unique in the generated file.

We recommend that the ID contain a sort of "blockType" (i.e., 'opimpl') for the implementation of an operation and theGUID of StP (or a scoped GUID like "classID::operationID").

Comment: Free string

Generated code blocks (X = **G**, see above) may not be changed manually (or these changes are overwritten the next time new code is generated).

In special cases you might want to change the generated code. This is done by setting the text "usermaintained=no" to "usermaintained=yes".

Note: You are free to choose whether to generate merge tag lines containing the `usermaintained=no` clause or not. In any case, you may add `usermaintained=yes` to any tag line of a generated code block to set it under manual maintenance.

File Types

In the default Merge configuration, all files are treated the same way:

- The entire content of the file can be changed *manually*.
- The comment tags for the generated code must take '**G**' as value for X as described in "Syntax of Comment Tags."

A block of generated code can contain another (nested) block of manual code.

Insert Strategy

When new generated code blocks appear in the new version of the generated file, the generated code is appended to the end of the file.

Note: New code blocks may appear because new elements, like a new operation, are added in the design.

GUIDsRemove Strategy

When generated code blocks disappear in the new version of the generated file, this generated code is set under C++ comments (‘//’) and an additional comment are added:

```
// DELETED BY INCREMENTAL CODE GENERATOR
```

Note: New code blocks may disappear because elements, like an operation, are removed in the design.

Global Object Rename

To get correct functionality in case of a global object rename, the merge tag must contain StP GUIDs at the proper place (see “blockID” at the syntax description of the comment tags).

Making Simple Changes in the Configuration

Change the Remove Strategy

There is another predefined remove strategy available, which causes code generated for removed elements to really be deleted from the file.

Change the configuration by changing in ***fileType*** rule the following line from:

```
{ removeStrategy "keepCCInactive" }
```

to:

```
{ removeStrategy "remove" }
```

Change the Style of the Comment Tags

You may change the style of the comment tags. For example:

- Change the start of these tags from ‘//#ACD#++’ to ‘//xyz START’: In the ***fileType*** rule, change the following line from:

```
{ generalPattern {^[ ]*//#ACD#}}
```

to:

```
{ generalPattern {^[ ]*//xyz BEGIN}}
```

In the diverse ***blockType*** rule change all pattern rules (***typePattern***, ***idPattern***) as shown in the following example:

```
{ typePattern {^[ ]*//#ACD#\+\+M\(}}
{ idPattern {^[ ]*(//#ACD#\+\+M\(.*\))} }
```

to:

```
{ typePattern {^[ ]*//xyz BEGIN M\(}}
{ idPattern {^[ ]*(//xyz BEGIN M\(.*\))} }
```

Note the escaped *plus* signs in the original version (‘\+’). The escapes are needed because the plus sign has a special meaning when used in regular expressions and therefore must be escaped when the plus sign is meant literally.

Change the Merge Strategy for a File Type

The merge configuration file contains one rule for another fileType, a C++ header file with suffix “.hpp”. When uncommenting these lines (remove the “#” signs at the beginning of the line), the rule is activated.

The new file type has a merge strategy which is the inverse of the other ones:

- The whole file contents is generated.
- All places for manual added code are provided by generating comment tags for the manual code.
- The comment tags for the manual code must take “M” as the value of X described as described in “Syntax of Comment Tags.”

A Merge Utility Reference

Low-level Functions

This section lists low-level functions to be used with user-defined areas.

- `udStart(expression)`

  Defines the marking of the beginning of the block as a regular expression.

- `udEnd(expression)`

  Defines the end of the block as a regular expression.

After the block markings are defined, user-defined blocks can be extracted.

- `udRead(expression)`

  udRead reads all user-defined blocks that fit in the definitions of udStart and udEnd.

- `udAutoRead(expression)`

If all user-defined blocks are saved with udRead, the saved blocks can be further processed.

- `udExists(expression, expression)`

  Two tag markings are transferred. udExists subsequently checks to see if a saved block fits to those two tag markings. The return value is either True or False.

- `udBlock(expression, expression)`

  This behaves like udExists, but returns the saved block as return value.

- `udSaveUnusedBlocks(expression, expression)`

  If not all blocks are written back with udBlock, the remaining blocks can be saved in a file. The first parameter defines the file name. The second allows you to define an additional password.

The following code excerpt shows what the operation with user-defined blocks looks like.

```
udStart("//// --UD.*-- *.* ////")// Defines a starting
                    // block
                    // marking

udEnd("//// --UD.*-- END ////");// Defines an End block
                    // Marking
...

udRead("file.txt"); // Saves all User-Defined
                    // Blocks from the file
                    //file.txt
...
setOutput("file.txt");// Defines the file file.txt
                    // as output and deletes
                    // at the same time the
                    // contents, which was saved
                    // with udRead beforehand
                    //
...

out = "//// --UDIF-- ////";// Generates a starting block-
                    // Marking, which is compatible
                    // with the
                    // definition in udStart


if(udExists("UDIF","")) // test if the block with
                    // the marking "UDIF" was saved
                    // beforehand.

   out = udBlock("UDIF","");// If yes, the
                    // saved block
                    // is being used again.
                    //else
   out = "\n\tuser code ... \n"// if no,
                    // Standard Text is generated.
```

```
end if

out = "//// --UDIF-- END ////\n";// Generates the End block-
                    // Marking, compatible to
                    // udEnd- Definition.

loop(Instances->MClass->Moperation AS Op)
                    // loop over all operation of
                    // the class

   out = "//// --UDCO--ó " [Op.name] " ////";
                    // Generates a starting block-
                    // Marking, which is compatible
                    // with the definition in
                    // udStart.
   if(udExists("UDCO",[Op.name])) // Test if the block with
                    //the
                    // marking "UDCO" and
                    // Operation name
                    // was saved beforehand.

      out = udBlock("UDCO",[Op.name]);
                    // If yes the saved block is
                    // being used again
                    //else
      out = "\n\tuser code ... \n"// If no, a Standard Text
/                   /is generated.
      out = "//// --UDCO-- END ////\n";
                    // Generates the End block-
                    // Marking, compatibel with the
                    // udEnd-Definition.
   end if

udSaveUnusedBlocks("Garbage.txt",date());
...                 // Saves all blocks, which cannot be
                    // ascribed anymore into the file
                    // Garbage.txt and generates to each
                    // block the date as additional
                    // information
```

Basic Merge Principles

The merge utility performs an intelligent two-way merge of generated files, which may be changed or extended manually at some defined places.

This chapter describes the basic principles of merging. Most aspects described here can be configured.

Files Involved in Merging

There are three files involved in merging:

- Generated file:

  The new version of the file as generated from the CASE tool
- Manual file:

  The previous generated version, changed manually (i.e., by implementing the operation bodies)
- Target file:

  The result of the merge activity

Tags and Blocks

The merge logic is based upon merge tags. Tags are special code lines (mostly comments) bearing information for the merge utility.

A group of code lines enclosed by a start tag and an end tag is called a block. A block is the primary unit used for merging files.

The entire file is logically treated as a block, but this block is not enclosed by comment tags. Merge distinguishes between FileType and BlockType.

When applying merge strategies, the file is treated exactly as any other block inside this file.

Blocks can be nested recursively. One tag can start or end more than one block simultaneously. See “BlockType” on page A-13.

Every block contains normal code lines directly or other (nested) blocks delimited by their comment tags. When merging, all subsequent normal code lines are treated internally as a single block.

Block ID

Blocks can be identified via a logical blockID. This enables finding corresponding blocks between the generated and the manually changed file, even if their order is not the same in both files.

Strategies

Merging is controlled by diverse merge strategies. Merge strategies can be overridden by the programmer in certain cases (see tag::overrideStrategy).

Merge Strategy

Merge strategies are applied when the same block exists in both files. The strategy defines the master for merging the contents of the blocks.

Note: The merge strategy applies to all contained blocks of the current block.

Insert/Remove Strategy

These strategies apply when blocks are inserted or removed in the new generated file.

Note: The insert and remove strategies apply to the current block, not to the contained blocks.

Mastership – Merge Strategy

We speak of *mastership* when merging. Mastership is a part of the merge strategy. It is defined for every block (or file) type by the merge strategy. We distinguish between Master Block and Slave Block.

Mastership can be set to:

- Generated (the block as contained in the generated file)
- Manual (the block as contained in the manual changed file)

The following properties are controlled by the master block:

- All directly contained code lines (including the insertion or deletion of these code lines)
- The order of the contained blocks and the code lines

The following property is always controlled by the generated block (regardless which block is the master block):

- The existence of contained blocks (that is: inserting and removing blocks)

Inserting New Generated Blocks – Insert Strategy

Blocks are inserted when a new element is inserted into the design model and the model is regenerated.

When the mastership of the block is “generated,” new subblocks are inserted at the position where they are generated.

But it is possible the generated file is the slave block. In this case, the order of contained blocks and directly contained code lines is determined by the manual file. When this occurs, the position of inserted generated blocks is not defined. The insertStrategy is used to define the place where these blocks are inserted.

Removing Blocks – Remove Strategy

Blocks are removed when elements are deleted from the design model and the model is regenerated.

Removing blocks need not result in removing code lines, as you perhaps would like to preserve its manual implementation for removed operations.

Therefore, remove strategies are supplied; they may do the following:

- Remove the code.
- Set it under comments.

It is also possible to control the way already removed (commented) blocks are treated in subsequent merges (i.e., when code is generated the next time).

The possibilities are:

- Leave commented code unchanged.
- Remove the commented code.

Example of Block Structures

Figure 1 below shows an example containing diverse block structures. In the example, all tag lines start with `// CG ##`.

Note: The whole file can be considered as a special block, associated with its own strategies.

Figure 1: Example of Block Structures

```
//Generated file header ...  - Block starts at file beginning; no tag line
// CG## fileheader end
normal text ...
normal text ...

   //CG## block1 ID01 name1 begin              - Normal block
      contents of block 1 ...

            //CG## block2 begin                   - Nested block
               inner contents of block 1 ...
            //CG## block2 end

   //CG## block1 ID01 name1 end

normal text ...

         //CG## blockDouble ID01 name1 begin
                  -              - Two blocks started simultaneously
            void op1 (...)
            //CG## blockDouble ID01 enddec1   - Inner block ends
      {
         manual code ...
      }
   // CG## blockDouble ID01 name1 endimp1      - Outer block ends
```

```
normal text ...

   // CG## filefooter begin          - Block ends at EOF; no tag line
   footer text ...
```

Calling the Merge Utility

The format for invoking the merge utility is:

```
tcl merge.tcl configfile manfile genfile targetfile [dumpfile]
```

where:

- *configfile* is the name of the configuration file.
- *manfile* is the name of the manually changed file. If this file does not exist, *genfile* is copied to *targetfile.*
- *genfile* is the name of the new generated file.
- *targetfile* is the name of the target file, i.e., the file which is created by the merge utility.

  If *targetfile* = *manfile*, *manfile* is overwritten.
- *dumpfile* (optional) is the name of the file that contains a dump after merging. The file contains a dump of the structured output of the block structure of the scanned files.

  See "Format of the Dump File" on page A-32.

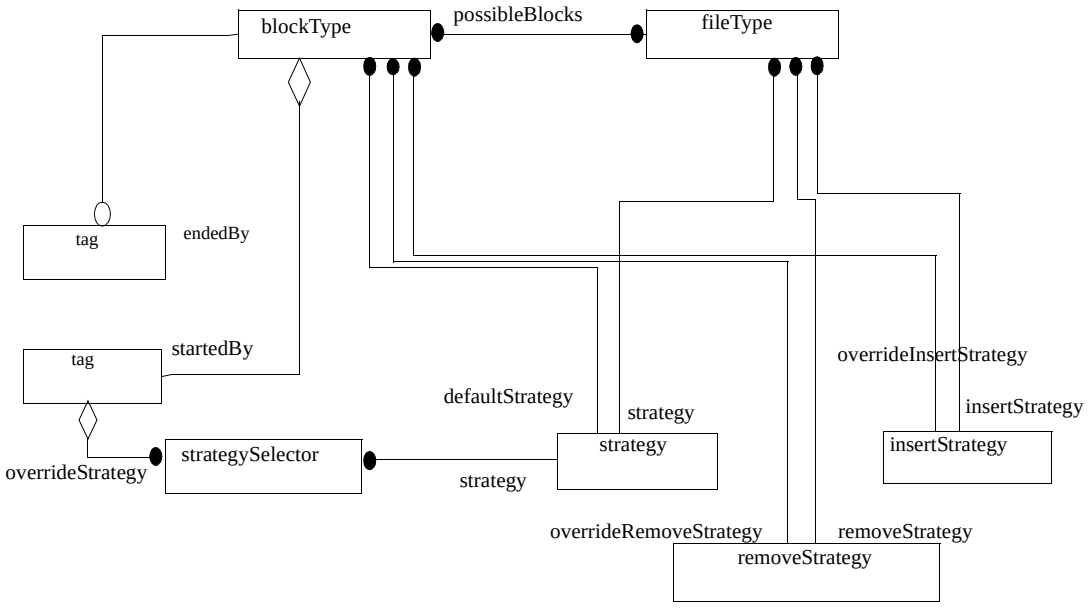
Configuration of the Merge Utility

Object Model

The following diagram (Figure 2 on page A-9) shows the structure of the configuration file for the merge utility.

- *Associations:* Reference to an object using its name
- *Aggregation:* Object is embedded directly

Figure 2: Structure of the Configuration File for the Merge Utility



Syntax of the Configuration File

Comments can be inserted by using the "#" sign at the beginning of the line.

- config-file-contents:
  config-object-list
- config-object-list:
  config-object-listopt config-object
- config-object:
  objectname { element-list }
- objectname:
  name of the object of the configuration
- element-list:
  element-listopt element-item

- element-item:

  { keyword { value } }

  { keyword “value-item“ } }

  { keyword {value-item} }
- keyword:

  Keyword as defined in the following description of the config-objects.
- value:

  value-list

  value-item
- value-list:

  value-listopt {value-item}

  value-listopt “value-item”
- value-item:

  Arbitrary string.

  If the value-item is enclosed in ‘{}’ braces, the whole string between the braces is considered (including blanks).

  If the value-item is enclosed in double quotes, all Tcl-relevant characters are evaluated and therefore have to be escaped with backslashes ‘\’.

  Tcl-relevant characters are: {} [ ] $

Data Types

- String

  Generally, any value-items are considered as string.
- Numeric

  Numeric value
- list <*string*>

  Zero or more elements of type *xxx* are given in a value list.
- Other config object

  Directly contained config object. In this case, the element list of the config object is contained directly.

Configuration Objects

This section describes all primary configuration objects. See Table 1 below. The name of the object is the keyword for defining it in the configuration file.

FileType

FileType defines the possible file types of the files to be merged.

The merge strategies for every file can be determined by the file type (which is derived from the file name by applying regular expressions).

Note: The file type can be seen as a special block type, which is not delimited by start and end tags and thus has its type defined using a pattern matching the file name.

Table 1: Configuration Objects

Keyword	Type	Description
name	string	Name of the file type. This name must be unique.
fileNamePattern	string	Regular expression, which matches for the file name. Every file, whose file name matches this pattern, is considered to be of this file type. Note, that the first file type, whose fileNamePattern matches, is used. In this case the order of appearance in the configuration file is relevant.
strategy	string	Name of the merge strategy for this file. This defines the strategy for all top / level text in this file. If this text contains blocks (enclosed by tags), the code lines inside these blocks are treated according to the block's merge strategy. Note that this strategy can be overridden by certain block types, using 'override strategy' inside their start tag ('startedBy') definitions.

Table 1: Configuration Objects (Continued)

Keyword	Type	Description
removeStrategy	string	Name of the remove strategy to be applied inside this file. (See also removeStrategy.) Note that this strategy can be overridden by certain block Types using 'overrideInsertStrategy'.
insert Strategy	string	Name of the insert strategy to be applied inside this file. See also insertStrategy. Note that this strategy can be overridden by blockTypes using 'overrideInsertStrategy'.
general Pattern	string	(optional, default: empty) General pattern matching for all start tags. This defines a general regular expression pattern, which matches for all possible start tags allowed for this file type. Providing this attribute may increase performance. If normal code lines (which are no start or end tags) match this pattern, the merge utility prints a warning during the merge. Note that end tags do not need to match this pattern. Generally, we recommend designing the comment tags (including the end tags) in a way that they all match a general pattern. This enhances readability of the code and minimizes errors when manually changing the generated files.
		Example: All comment tags start with '/ / CG ##'. The general pattern then could be defined as follows: {generalPattern {^ */ / CG##}}
generalOver-ride Pattern	string	(optional, default:empty) Regular expression pattern extracting the manual part of a comment tag (which defines overrides for the merge strategy). The manual part of the comment tag must occur in brackets '(',')' in the regular expression. The attribute must exist if the file can contain any tags containing overrideStrategies. As there is only one generalOverridePattern per file, all overideStrategies have to match this one pattern.

Table 1: Configuration Objects (Continued)

Keyword	Type	Description
		Example: Suppose comment tags look like this: / / CG## tttt iiii .... chgman=no The string 'chgman=no' can be changed to 'chgman=yes' changing the merge strategy to manual mastership (and thus not generating over the manual changed code inside this block). Then the generalOverridePattern would look like this: {generalOverridePattern { (chgman *=[^=]*) $ }}
possibleBlocks	list<string>	(optional, default: no blocks allowed) List of those blocks which may occur in files of this file type. The list contains the names of the block types. When processing the file and dividing it into blocks for merging, only these blocks are considered. All other lines (even if matching the 'generalPattern') are treated as normal code lines. Note that the order of blocks as defined in the 'possibleBlocks' item defines the order in which the block types are matched against the tag lines when merging a file.

BlockType

A block is the smallest unit treated by the merge utility. A block can consist of lines of code and other contained blocks. A block is delimited by tags.

Nested Blocks

Blocks can contain other blocks (nested blocks). Generally, a block type is not limited to a special level of nesting.

Starting Blocks Simultaneously by a Single Start Tag Line

A single start tag can start more than one block. This happens when the typePattern of more than one start tag matches the same line and the tag is allowed to start blocks simultaneously.

In this case the started blocks are considered to be nested. The order of nesting corresponds to the order the blocks are defined in the list ‘fileType::possibleBlocks’ in the configuration file.

Each of these blocks has its own strategies.

Note: The overrideStrategy of the start tag is only applied to the outermost block of the simultaneously started blocks.

Ending Blocks Simultaneously by a Single End Tag Line

In a similar way, a single end tag can complete more than one block. This happens if the typePattern of more than one end tag matches the same line and the tag is allowed to end blocks simultaneously (see ‘tag::allowSim’).

Starting Blocks at Beginning of File

Blocks can be started at the beginning of the file (BOF) without a start tag line. This is done by setting startedBy.typePattern to ““.

In this case no overrideStrategy or block ID is available.

Blocks started in this manner may not be inserted or deleted, either manually or by the code generator.

Ending Blocks at End of File

Blocks can be ended at the very end of the file (EOF) without an end tag line. This is done by setting endedBy.typePattern to ““.

In this case no block ID is available.

Blocks ended in this manner may not be inserted or deleted, either manually or by the code generator.

Table 2: Keywords for 'blockType':

Keyword	Type	Description
name	string	Name of the block type. This name has to be unique.
defaultStrategy	string	Name of the default merge strategy to be used in this block. The default merge strategy can be overridden by special definitions in the tag lines (see tag::overrideStrategy)
overrideRemove-Strategy	string	(Optional, default: no override) Name of the override insert strategy for this block type. This strategy overrides the strategy defined for the current file type.
overrideInsert-Strategy	string	(Optional, default: no override) Name of the override insert strategy for this block type. This strategy overrides the strategy defined for the current file type.
startedBy	tag	Tag definitions for the start tag of this block type. The start tag must match the generalPattern (see fileType::generalPattern)
endedBy	tag	Tag definitions for the end tag of this block type. After the start tag the next following line matching the end tag is considered to be the end tag. The end tag needs not to match the generalPattern (see, fileType::generalPattern)

Strategy

Merge strategies contain information about the merge strategies to be applied for all children of this block.

Table 3: Merge Strategies

Keyword	Type	Description
name	string	Unique name for this strategy.
master	string	Tells who the master is for the code block. Values: ***M*** - the manual changed file is master ***G*** - the generated file is master ***M***: - The order of the contained lines or subblocks in the code block is determined by the order found in the manually changed file. ***G***: - The order is determined by the new generated file.

InsertStrategy

InsertStrategy is the strategy you apply when inserting a block from the not-mastered block.

This strategy is valid for the block where it is defined (instead of the merge strategy, which only applies to its children).

The insert strategy is applied only if the inserted block is coming from the slave block (not from the master block). This is because only in this case is the position of the inserted block not defined.

Table 4: insertStrategy

Keyword	Type	Description
name	string	Unique name of this strategy.
location	string	Defines the place where the new block has to be inserted. Values: Bottom -currently the only possible value. The new block is inserted at the bottom of the containing block.
linesBefore	numeric	Defines how many empty lines shall be inserted before the inserted block.
strategy	numeric	Defines how many empty lines shall be inserted after the inserted block.

RemoveStrategy

RemoveStrategy is the strategy you apply when removing a block.

This strategy is valid for the block where it is defined (instead of the merge strategy, which only applies to its children).

The remove strategy is applied to all removed blocks, no matter how they are mastered (manually or generated).

Note: This rule applies to all other (recursively) contained blocks inside the removed block. This may be of special interest as the tag substitution patterns ('removedStart / EndPatternSubstitute') defined in the outmost removed block apply to all contained blocks, too.

The main possibilities of removing a block are:

- Remove it completely
- Change it (i.e., set it under comments)

Changing it can be defined by diverse substitution patterns and pre / post lines which can be printed around the block's start / start and end / end tags.

Changing the Tag Lines When Removing a Block

Changing the block also changes its tags (and those of the contained blocks). There are two ways to change the tag lines:

- *Deactivate* the tags, so that they are no longer recognized
- Change the tags to *another tag type*

If you change a tag to another tag type, this new tag type may affect the block in subsequent merges. If this is a concern, leave the tag unchanged or remove it.

Table 5: removeStrategy

Keyword	Type	Description
name	string	Unique name of this strategy.
keepRemoved	string	Defines how the removed block is treated in the target file. If the block is printed, the following attributes may define how the printed lines are changed (i.e., set under comment). Values: *Never* - do not print the removed block. *Always* - Print the removed block. *GeneratedMaster* - Print it only if the containing block is mastered by the generated file. *ManualMaster* - Print it only if the containing block is mastered by the manual file.
removedPre-Lines1	list<string>	(optional, default: no lines) Lines, which shall be printed before the start tag of the removed block.
removedPre-Lines2	list<string>	(optional, default: no lines) Lines which shall be printed after the start tag of the removed block
removedPost-Lines1	list<string>	(optional, default: no lines) Lines which shall be printed before the end tag of the removed block.

Table 5: removeStrategy (Continued)

Keyword	Type	Description
removedPost-Lines2	list<string>	(optional, default: no lines) Lines which shall be printed after the end tag of the removed block.
removedLine-Substitute	list<string>	(optional, default: no substitution) Two patterns define a substitution for all normal code lines of the removed block. This substitution is applied for all code lines of the removed block when it is printed. The first pattern may define portions of characters using brackets '()'. The second pattern may refer to these portions of the first pattern using '\n', where 'n' is a digit from 1 to 9 referring to the nth bracket of the first pattern. For details see Tcl 'regsub' command. Example: Add two slashes, but leave leading blanks unchanged: {removedLineSubstitute {{^(*)}{\1/ / }}}
removedStartPatternSubstitute	list<string>	(optional, default: no substitution) Two patterns define a substitution for all start tags inside this block. For substitution rules see 'removedLine Substitute'. This rule also applies for all tags of contained blocks inside this removed block. Example Change the tag beginning from 'CG##' to 'CGREM##', leaving leading blanks and the rest of the line unchanged:(removedStartPatternSubstitute{{^(*)/ /CG##}{\1/ /CGREM##}}} With this example you could define block types matching at'/ /CGREM##' and define special remove strategies for them. This allows definition of how to deal with removed blocks in subsequent merges.

Table 5: removeStrategy (Continued)

Keyword	Type	Description
removedEndPatternSubstitute	list<string>	(optional default: no substitution) Two patterns define a substitution for all end tags inside this block. Substitution rules see 'removedLineSubstitute' This rule also applies for all tags of contained blocks inside this removed block. Example: Change the tag beginning from 'CG$$' to 'CGREMEND##', leaving leading blanks and the rest of the line unchanged: {removedStartPatternSubstitute {{^(*)?? CG##}{\1/ / CGREMEND##}}}

Other Types

These types are used inside the configuration objects.

Tag

Merge tags enclose parts of code (called 'block'), which are treated as one entity when merging generated code files.

The merge tag (logically) contains information about the merge strategies to be applied when merging the enclosed code block. Additionally, the corresponding end tag for the code block can be determined.

Tags are specified using regular expression patterns matching these tags. It is important to check these patterns exactly when writing a configuration file, otherwise the merge utility does not work correctly or prints errors.

Important Rules:

- startPattern must not match any other start tags (except those that should be started simultaneously). See "BlockType" on page A-13.

- startedBy::idPattern must extract a string which is unique among all blocks contained in the same parent block (recommendation: idPattern should contain at least the pattern's type part)
- endedBy::idPattern (if it exists) must extract exactly the same substring as the startedBy::idPattern.

Tag Structure

When used as start tag, the tag should be structured so that the following information can be obtained from the comment tag line:

1. That it is a comment tag (and no normal code line). See also 'fileType::generalPattern'.
2. The block type, which is started by this tag (see 'typePattern').
3. The block id (see 'idPattern'). This id identifies a block uniquely inside its containing block. Suppose you use the CASE-tool's unique id or codegen-id, if available. If you do not, you would use the name of the generated element (attribute, operation, ...).
4. A comment (i.e., the name of the operation)
5. An override section, where you can change the default merge strategy. Normally you can use this to overtake generated code into manual maintenance and thusly prevent it from being overwritten by subsequent code generation.

These parts of a tag have to be matchable by regular expression patterns. It is not necessary that these parts of the tag consist of subsequent characters; they can be spread across the line (see "Block Type" in the following example).

Example for a start tag:

```
//CG## opimpl 123::15 Acct::balance begin chgman=no

//CG##              General pattern for
                    comment tags
       opimpl    beginBlock Type
       opimpl 123::15Block ID
          Accnt::balancecomment
              chgman=nooverride section
```

Table 6: Other Types of Structures

Keyword	Type	Description
typePattern	string	Regular expression pattern, which matches uniquely and exactly for this tag. *Start tags*: If this pattern is empty, the block is considered to start at the very beginning of the file. As the block does not have a start tag, no information given by the start tag is available for this block (overrideStrategy, block ID). *End tags*: If this pattern is empty, the block is considered to end at the very end of the file. As the block does not have an end tag, no information given by the end tag is available for this block (block ID)
idPattern	string	Regular expression which selects exactly the type+ID part of the patterns. This pattern has to extract the ID part of the pattern by using brackets '(...)'. *Start tags*: For non-empty start tags, the idPattern is mandatory. If the block type does not have an ID part (which is possible, if only one block of this type occurs inside the containing block), the idPattern has to extract the tag type part only. *End tags*: This attribute is optional when being used as an end tag. If provided for an end tag, the merger performs a check in order to see whether or not the ID of the start tag and the end tag are the same.

Table 6: Other Types of Structures (Continued)

Keyword	Type	Description
overrideStrategy	strategySelector	(for start tags only; optional default: no override) Definition for the override merge strategy. By providing this attribute, this block type strategy may be overridden by the programmer of the file by changing the override part of the start tag comment line. If one comment tag line starts more than one block simultaneously, the 'overrideStrategy' only applies to the outermost block and not to the other (nested) blocks started by the same tag line.
allowSim	String	(optional, default: 0) Tells whether this tag may start or end a block simultaneously to other blocks. Values: "0" - This tag may not start or end a block if the same tag line already started / ended another block. "1" - This tag may start or end a block if the same tag line already started / ended another block.

StrategySelector

StrategySelector selects a strategy by using a regular expression pattern. The strategySelector contains a pattern and a strategy name.

If the pattern matches the comment tag, the denoted strategy must be used.

Note: For all override strategies the tag part defining the override strategy must match the same general pattern (see fileType::generalOverridePattern).

When more than one override strategy is given for a block type, the first matching override strategy prevails.

Table 7: strategySelector

Keyword	Type	Description
pattern	string	Regular expression pattern
strategy	string	Name of the selected strategy

File Type

Table 8: File Type

fileType {	
{ name “CCImplementation”}	
{ fileNamePattern {\.C$}}	The file name ends with .c
{ strategy “man”}	For the immediate file contents, the master is the manual file
{ generalPattern {^ *// CG## }}	All start tags inside this file start with // CG##
{ generalOverridePattern {(chgman *=[^=]*)$}}	All override parts of tags look like chgman=xxx
{ removeStrategy “remove” }	Really remove deleted blocks
{ insertStrategy “bottom3” }	
{ possibleBlocks { blOpImpl }}}	For this example, only one block is allowed

The Strategies

Table 9: The Strategies

strategy {	
{ name “gen”}	
{ master “G” }}	Master is the generated file
strategy {	
{ name “man” }	
{ master “M” }}	Master is the manual file

Table 10: More of the Strategies

insertStrategy {	
{ name “bottom3” }	
{ location “Bottom” }	Insert new blocks at the bottom
{ linesBefore “3” }	Insert 3 empty lines before every new block
{ linesAfter “3” }}	Insert 3 empty lines after every new block

The second of the following removeStrategies (“keepCC_inactive”) does more than needed:

- The removed code block is set under both C comments (/* ... */) and C++ comments (//).

The start and end tags are deactivated by substituting them in a manner such that they are not recognized as tags in subsequent merges.

Note: This changes the former block into a simple series of code lines.

The question is, what happens with the removed (that is: the commented) code block the next time the file is generated and merged?

- If this removed block is contained in a “generated-mastered” block, the commented block is removed from the file at the next merge.
- If the removed block is contained in a “manual mastered” block, it remains unchanged.

Table 11: Use of Strategies

removeStrategy {	
{ name "remove" }	
{ keepRemoved {Never} }}	Do not keep removed block
removeStrategy {	
{ name "keepCC_inactive" }	
{ keepRemoved {Always} }	Keep removed block always
{ removedPreLines1 { {/*} \ {DELETED BY GENERATOR */} } }	Insert these two lines before the start tag
{ removedPreLines2 { {/*} } }	Insert this line after the start tag
{ removedPostLines1 { {DELETE END */} } }	Insert this line before the end tag
{ removedLineSubstitute { {^([ ]*)} {\1/ / } } }	Set lines inside removed block under C++ comment Deactivate the start and end tag by changing them so that typePattern no longer matches
{ removedStartPatternSubstitute { {^([ ]*)/ / CG##} {\1/ / CG--} } }	
{ removedEndPatternSubstitute { {^([ ]*)/ / CG##} {\1/ / CG--} } }}	

Block/Tag Definitions

In the following example, we define a block with the following start and end tag lines:

```
// CG## opimpl 123::15 Accnt::balance begin chgman=no
// CG## opimpl 123::15 end
```

The master for this block is the generated file.

The contents of the block are manually mastered when the tag is changed to "... chgman=yes". This means that subsequent code generation does not change the contents of this block.

This makes sense if the generator generates some default implementation which may be changed by the implementer.

The end tag also contains the block ID. This way the merge utility can check if the block IDs of the start and end tags are the same.

This block type overrides the file removeStrategy (which is "remove") by the strategy "keepCC_inactive". This sets the removed block under comments and leaves the block in the file.

Table 12: Block/Tag Definitions

blockType {	
{ name "blOpImpl"}	
{ defaultStrategy "gen"}	Generated file is master for this block
{ startedBy	Define the start tag
{ typePattern {^ *// CG## opimpl .* begin}}	
{ idPattern {^ *(// CG## opimpl *[^]*) .*begin} }	This pattern extracts type+block ID from the tag line. The type+block ID part is enclosed in brackets "()" in this pattern.
{ overrideRemoveStrategy "keepCC_inactive"}	Override the remove strategy

Table 12: Block/Tag Definitions (Continued)

{ overrideStrategy {{	
{ pattern { chgman *= *yes *$}}}}}}}	Changing the tag sets the master to 'manual'
{ endedBy	
{ { typePattern {^ *// CG## opimpl .*end *$}}	Pattern for the end tag
{ idPattern {^ *(// CG## opimpl *[^]*) .*end} }}}}}	By providing the idPattern for the end tag, the merger checks if the IDs of the corresponding start and end tag are the same

Developing and Testing a Configuration File

This section describes the steps for developing and testing a merger configuration.

When writing a configuration file, a thorough test activity is recommended.

Some test utilities are provided for:

- Normal testing
- Regression testing

Set Up Configuration

- Define the possible file types.
- Design the block structures for every file type.

  Note: Many file types can share the same block types.

- Define tag syntax.
- Define strategies.

All of these activities result in writing to the configuration file.

Set Up Test Cases

Depending on the complexity of your configuration, set up one or more test cases for every file type.

To work together with the test utilities, a test case should be written as follows:

Write Initial File

- Write an initial file containing all block types possible for this file and in a nesting structure, which is realistic for the generated files.
- If block types contain ID's, write at least two blocks of the same type.
- Fill arbitrary code lines into or between the blocks.
- Copy the initial file to two versions: *manual file* and *generated file.*

Change Manual File

Now change the manual file at all places in the file, meaning inside and between all contained blocks.

Do the following changes:

- Change mastership in some blocks (if provided through your defined tags) using the overrideStrategy.
- Add code at manual mastered places (or in blocks, where you overrode the strategy to 'manual').

  This code should contain the word __MANUAL__ in every changed line.
- Add code at generated mastered places (or in blocks, where you overrode the strategy to 'generated').

  Add the word __NOTOCCUR__ in every changed line.
- Deal with removed blocks.

  For every removed block, insert the word __NOTOCCUR__, if the remove strategy of this block shall really remove it.

- Change the order of the blocks inside blocks and files, which are mastered to ‘manual’.

  Note: Since the change of order cannot be checked automatically by the test utility, you must check manually after the test run.

Change Generated File

Now change the generated file at all places in the file, which means: inside and between all contained blocks.

Do the following changes:

- Add code at generated mastered places (or in blocks, where you overrode the strategy to ’generated‘ in the manual file).

  Add the word __GENERATED__ in every changed line.
- Add code at manual mastered places (or in blocks, where you overrode the strategy to ’manual‘ in the manual file).

  Add the word __NOTOCCUR__ in every changed line.
- Remove blocks (up to one of every type, where possible).

  Add the word __NOTOCCUR__ in every changed line.

  Note: Do not forget to remove some nested blocks, if possible.

  Note: Not all block types can be removed, i.e., blocks at the beginning or end of the file.
- Insert blocks (up to one of every type, where possible).

  Add the word __GENERATED__ in every changed line.

  Do not forget to insert some nested blocks, if possible.

  Note: Not all block types can be inserted, i.e., blocks at the beginning or end of the file or blocks with no ID (they would not be unique if you insert a second block of the same type).
- Change the order of the blocks inside blocks and files, which are mastered to ‘generated’.

  Note: Since the change of order cannot be checked automatically by the test utility, you must check manually after the test run.

Run the Test Cases

Start the test utility on the previously written files:

```
tcl mergetest.tcl configfile manfile genfile
targetfile checktarget checkdump
```

where:

- *configfile* is the name of the configuration file.
- *manfile* is the name of the manually changed file.
- *genfile* is the name of the newly generated file.
- *targetfile* is the name of the target file, that is, the file which is created by the merge.

  When testing, this file should be different from *manfile*.
- *checktarget* (optional) is the name of the file containing a verified result of the merge activity.

  Using this parameter allows regression testing.
- *checkdump* (optional) is the name of the file containing a verified dump of the block structure of the parsed files.

  This allows regression testing of the internally parsed file representation.

This utility performs a merge and checks the following things:

- Every __MANUAL__ word from manual files must occur in the target file
- Every __GENERATED__ word from generated files must occur in the target file
- No __NOTOCCUR__ word must occur in the target file.

If checktarget and checkdump are provided, they are compared to the result of the merge run. Any mismatches are reported.

How to Get Initial Checktarget

The more secure way:
Write it on your own by manually 'merging' both the manual and the generated file.

The easier way:
Run the merger, validate the result and take it as the checktarget.

How to Get Initial Checkdump

Start the *merge.tcl* script with the parameter dumpfile. This creates an initial dump file. We also recommend checking this dump file.

Format of the Dump File

The dump file contains the complete text of the file with added annotations for every block.

The dump file contains the generated file followed by the manual file.

Note: In the dump, normal code lines are enclosed in so-called 'line blocks' for manipulation reasons. Line blocks do not have any properties nor can they be configured in the configuration file.

Dump of the start line of a block:

-----BLOCK *index blockType strategy isFirst*----------

- *index* is the internal number of the block.
- *blockType* is the name of the block type as given in the configuration file
- *strategy* is the name of the actual strategy applied for this block. This value may be influenced by the overrideStrategy.
- *isFirst* is the first block started by a tag line

  The tag line starting this block started another block already simultaneously.

Dump of the start line of a line block:

-----BLOCK *index* ----------

- *index* is the internal number of the block

A line block is a block only containing subsequent lines of the code file.

All normal code lines are enclosed in such line blocks.

Dump of the end line of a block:

----- *index* ----------

- *index* is the internal number of the block

The block with number ***index*** ends here.

B TDL Reference

Overview

This chapter explains the basics of the Transformation Description Language (TDL).

TDL supports the following coding styles:

- Templates

  Text written in templates is copied to the output, allowing it to include metamodel references, variable contents, other templates, and general TDL expressions.
- Procedures

  Procedures define the sequence of code generation, or, more generally, the flow of control. They are also used to encapsulate sophisticated text pattern generations and other general-purpose functions. Procedures can be called with parameters and can return a string value. Their return value can be embedded in a template. Procedures may write to the output, though this feature is not normally used.
- Tcl procedures

  Tcl procedures are used as an extension mechanism for TDL. Tcl (tool command language) scripts can be included in TDL to do things that are not supported in standard TDL procedures. Tcl can be used for operating system access and sophisticated string handling. Tcl procedures can be called with any number of parameters and can return a string that may be assigned to a TDL variable or directly embedded in template output. Tcl procedures can call TDL procedures and vice versa.

- Virtual templates and procedures

  Normal templates and procedures can also be virtual.

With these mechanisms, text generators can be written completely in TDL without the need for additional configuration files. Using TDL, a typical system might include:

- Templates to define the layout of the generated text, e.g., code, HTML, documents
- A main procedure and any associated utility procedures to control text production
- Tcl procedures to support non-standard features

Templates

Template Syntax

All templates start with the keyword “template” followed by the template name and end with “end template”. For example:

```
template <template name>
....
....
end template
```

All lines between these template delimiters are written literally to the output. This means that anything you write is written directly to the output file, except that procedural code statements can be inserted into the template enclosed in square brackets `[]`. The latter is used to include textual items and embed control structures in the output.

Examples of textual items are:

- Metamodel references like [MClass.name]
- Variable contents like [[delimiter]]

  To make template code more readable, simple variable names with no enclosed blanks can be written in single brackets like [delimiter].

- Return values of TDL and Tcl procedures like [udOut("UDIF", "", "Include Files")]

  Control structures support loops and conditions. These must be written on a separate line that is not written to the output.

Examples:

- Loops iterate on sets of metamodel items via a loop statement like this:

```
[loop(Instances->MClass)]
class [MClass.name];
[end loop]
```

- Conditional output is realized using the *if* statement like this:

```
if([MClass.active])
/// State Machine Code ////////////////////////////////
.........
[end if]
```

- The switch statement chooses from many possibilities:

```
[switch([cardinality])]
[case "Many" :]
       return " *_" [ToRoleName] "[100]";
       [break]
[case "One" :]
       return " *_" [ToRoleName];
       [break]
[default :]
       return " *_" [ToRoleName] "[" [cardinality] "]";
       [break]
[end switch]
```

Templates allow escape sequences to write square brackets to the output and to suppress newline characters. Escape sequences use the '/' character. As a consequence, backslashes must be escaped, too.

Examples:

- Create a C++ array state name that uses a TDL variable stateIndex as an array index and records the state names from the metamodel:

```
statenames\[[stateIndex]\] =[Mstate.name];
```

- Construct a DOS file path from a TDL variable pathBase and the UML property FileName of MClass:

```
   [pathBase]\\[MClass:FileName]
```

The following example is intended to show the most important features of TDL templates. For reasons of readability this segment does not show a complete C++ header file generator.

```
template CxxHead(MClass)
#ifndef _[MClass.name]_h
#define _[MClass.name]_h
#include "classes.h"
[loop(MClass->SuperClass)]
#include "[SuperClass:FileName].h"
[end loop]

[udOut("UDIF","","include files and other stuff")]\

class [MClass.name] : public [baseClassList([MClass])]

[udOut("UDBC","","Base Classes")]
{
public:
   [MClass.name]();
   ~[MClass.name]();
//--Attribute Accessors --------------------------------

private:
[loop(MClass->MAttribute as Att)]
   [[BINDING_[Att.binding]]] [Att.type] [Att.name];
[end loop]

public:
[loop(MClass->MAttribute as Att)]
   [[BINDING_[Att.binding]]][Att.type]get[Att.name]()
const;
[end loop]

//   and so on ........

end template
```

Loop Statements

The basic function to access the metamodel is the loop statement. With it, you can navigate through the metamodel by supplying a navigation rule and then iterate the results in the subsequent block (which is terminated by the end loop statement). The syntax for a loop statement is:

```
loop(Navigatorrules;firstProc1(),firstProc2()
,.;nextProc1(),.)
   statement_11;
   statement_12;
   ...
   statement_1n;
end loop
```

Further processing of the result list is described in "Access to the Metamodel" on page 2-1. Additional loop procedures can be used.

Optionally, a loop statement can include a list of additional procedures. The list of procedures is iterated in parallel to the loop iteration. In other words, the first procedure is called for the first iteration (-> the first metamodel element returned by the navigation rule), the second procedure for the second iteration and so on.

A common example for this is a loop which creates delimiters for all results, except the first:

```
local result;
loop(MOperatione->OpPara; setDelim(""), setDelim(", "))
[result] = [result] delim() [OpPara.type] " " [OpPara.name];
end loop
```

For the first operation parameter, the delimiter is set to an empty string; for all others to the ", " value.

Note: The last procedure is "sticky", if the loop returns more results than there are procedures defined, it always calls the last procedure.

Break Inside Loops

To set up a break inside of a loop block, use 'if'.

For example:

```
loop(Instances->MClass)
   if([MClass.name]==”cuckoo”)
      break;
   end if
end loop
```

Note: A break is supported in both loop and switch statements.

Indent Mechanism

When a template procedure is called within another template procedure, the position of the procedure calls also defines the indention. Therefore, output lines of the called template procedure do not begin before that position.

The call-up line is ignored if the line is blank and the procedure does not return a value.

Procedures

A procedure is defined by placing the string “proc” in front of the procedure name. The procedure name corresponds to the definition [_a-zA-Z][_a-zA-Z0-9]*.

```
proc func()

   statements ...

   return "return of func";

end proc
```

Function Parameters

Each procedure can contain a function parameter. Permissible types are string and metadata types. Strings, in this case, behave like local variables. Metadata types are defined inside loop statements. They are formed out of metaclasses. Access to the metaclasses within a function is similar to access within a loop statement.

```
proc func1()
   loop(Instances->MClass->MAttribute)
      func2([MClass], [MAttribute], "call func1");
   end loop
end proc

proc func2(MClass, MAttribute As Att,str)

   out = [MClass.name] ":" [Att.name] ":" [str];
end proc
```

Return Values

Each procedure can return a string value by using “return <*expression*>”.

```
proc func1()
   statements ...
   return "return of func1";
end proc
```

Except for procedures, templates implicitly return the entire content as defined in the body of the template. Templates also return a string value automatically, while procedures have to do that explicitly. Typically a procedure then assigns the return value of the template call to the “out” function in order to write this into a file.

```
template hello_world()
   hello world
end template

proc main()
   out = hello_world();// returns "hello world"
end proc
```

Note: In order to generate this output into a file you need to have opened a file by using the setoutput(<FILE_NAME>) function.

In addition, templates can explicitly return by using a [return] statement. In this case, the return value holds the content of the template up to the return statement.

```
template hello_world()
   hello world
   [return]
   hello world 2
end template

proc main()
   out = hello_world(); // returns "hello world"
end proc
```

Prototypes

Function prototypes do not need to be defined in TDL. Therefore, a function command can also come before a function implementation.

Tcl Procedures

The base TDL language has only a small set of TDL procedures. Additional extensions that cannot be covered by TDL are covered by Tcl procedures. A Tcl procedure can contain a transfer parameter and deliver return values. Transfer parameters and return values are explicitly of type *string*. The definition of a Tcl procedure starts with “tcl_proc” and ends with “end proc”. In between is the Tcl code.

This is demonstrated by the example of the function *regexp*.

```
tcl_proc regexp(str,reg)// implementing of the
                    //procedure regexp
   regexp $reg $str ret;
   return $ret;
end proc

proc main()
   out = regexp("hello world","h.*o");// calling the
                // procedure regexp
end proc
```

Calling TDL Procedures from Tcl Procedures

To call a TDL procedure from a Tcl procedure, prefix the name of the TDL procedure with “tdl_”.

For example:

```
proc main()

   tclProcedure()

end proc

tcl_proc tclProcedure()
       set test [tdl_backInTdl 1];
end proc

proc backInTdl(arg)
       return ”arg:”[arg]
end proc
```

Virtual Procedures and Templates

Procedures can be tied to metaclasses, such that a virtual calling mechanism is made possible. The declaration of a virtual procedure is similar to that of a normal procedure or template. The exception is that the name of the metaclass to which the procedure is to be bound precedes the procedure name; the metaclass and procedure are separated by a period. This naming allows the procedure to refer to the appropriate metaelement.

```
proc MClass.generate()
   return [MClass.name]
end proc

proc MNormalClass.generate()
   loop(MNormalClass->MAttribute)
   ...
   end loop
end proc

template MInterface.generate()
...
end template
```

Calling a virtual procedure has to be done via a metaelement.

```
loop(Instances->MClass)
   [MClass].generate()
          // dependent on MClass-Type it calls up
          // NormalClass.generate, MInterface.generate
          // or when MClass does not correspond to the
          // previously mentioned types, the procedure
          // MClass.generate
end loop
```

Comments

The characters '/*' start a comment; the characters '*/' end a comment. The character sequence '//' can be used to mark the start of a comment that ends on the same line. Notice that inside templates, a comment must be enclosed with square brackets. Example:

```
[loop(MClass->MAttribute)] [/* This text is an example of a
comment in a template extending over more than one line. */]

[loop(MClass->MAttribute)] [// single line comment]
```

Escape Sequences

An escape sequence always begins with a '\' (backslash) followed by another sign. The following escape sequences are available in TDL

Table 13: Escape Sequences

Name	Escape Sequence
Line feed	\n
Horizontal tab	\t
Backslash	\\
Quote	\"

Table 13: Escape Sequences

Name	Escape Sequence
Octal	\nnn
Hex	\xnn
Decimal	\dnnn

Strings

A string is a sequence of signs that is surrounded by quotes (“...”). A string can contain escape sequences.

- String handling:
  - Concatenation

```
[var]=”1234”;
[var]= “1” “2” “3” “4”;
```

  - Escape sequences

```
“\n”,
“\x02”,
“\t”...
```

Data Types and Variables

TDL knows the string data types and the metaclass types that are derived from the classes of the metamodel. However, metadata types are only important in connection with function parameters. Therefore, this chapter concentrates on the data type string. The data type string definition field falls in either the global or local field.

Global Variables

The scope of global variables extends over the entire program. A global variable is not declared. It is solely applied inside functions. Each variable is of the data type *string*. A variable begins and ends with square brackets `[]`. In between, one or more terms form the name of the variable according to the operation. Possible terms are:

- A name (alpha- numerical symbols)

  `[var], [name], ...`

- A string

  `[“var”], ["name"], ...`

- A variable

  `[[var]], ...`

- A metaattribute variable

  `[[MClass.name]], [[MRelationship.name]], ...`

- A combination of the above

If variable *var1* has value Name, the following three statements always refer to “varName” variables:

```
[var[var1]] = "Any String";// variable meant: varName
["var"[var1]] = "Any String"; // variable meant: varName
[varName] = "Any String"; // variable meant: varName
```

Furthermore, metaattribute variables can also form the name of a variable.

```
[[MClass.name]"_"[ToMClass.name]] = "Any String";
```

Local Variables

Local variables are processed on the stack. The scope ends with the end of a function. Unlike global variables, local variables require a special declaration. This local declaration can be located anywhere in a function. The name of a local variable always begins with a text symbol or an

underline, followed by other text symbols, figures, or underlines ([_a-zA-Z][_a-zA-Z0-9]*). Function parameters are also treated like local variables.

Only one local variable can be declared after “local”.

```
proc example
local var1;
local var2;
          // local variable var1 and var2

[var3] = "Any Content";
          // global variable var3
local var3;
          // Definition of the local
          // variable var3

[var3] = "Any Content";
          // from now on local variable var3
["var3"] = "Any Content";
          // global variable var3, because the
          // term of a local variable only
          // consists
          // of text signs and numerical
          // signs
end proc
```

Metaattribute Variables

This type of variable is for reading metamodel information. The name of the variable consists of two parts separated by a period and surrounded by square brackets (e.g., [MClass.name]). Metaattribute variables can only be applied in a loop statement.

Metaproperties

Metaproperties roughly correspond to metaattribute variables. However, the property name is separated from the metaclass name by a colon. In addition, creation of metaproperties is possible only during run time.

USES Mechanism

With the USES statement, additional TDL scripts can be added to the translation.

```
USES java_std;
```

The USES statement has to be used outside procedures and templates. Here the listed TDL script is given without an ending. Multiple USES connections of the same script are ignored.

Statements and Expressions

In TDL, an expression is formed by concatenation of strings, variables, or functions. Order and quantity are not limited. Designating a variable of a function parameter can look like this:

```
[var] = "String1" [variable2] function1() "String2"
function2();

func("String1","String2" function1(),[variable1]);
```

Logical Operators

Logical data types are depicted as True or False. The following table gives an overview of all operators:

Table 14: Logical Operators

Operator	Function	Application
!	logical NOT	!expression
<	smaller	expression1 < expression2
<=	smaller than / equal	expression1 <= expression2

Table 14: Logical Operators

Operator	Function	Application
>	larger	expression1 > expression2
>=	larger than / equal	expression1 >= expression2
==	equality	expression1 == expression2 Comparison is case sensitive (as all other string compares in ACD)
!=	inequality	expression1 != expression2
&&	logical AND	expression1 && expression2
||	logical OR	expression1 || expression2
In	logical IN	expression1 In expression2

Each logical operation is itself an expression that can serve as an operand for further logical operators. Furthermore, the precedence can be laid down with bracketing of single expressions.

Program Flow

The program flow of statements is generally sequential. Each TDL program begins with the first statement in procedure main(). Subsequent statements are carried out one after the other until the program is finished after the last statement. If the sequence is not applicable for certain problems, it can be branched with the statement in the program code. With the exception of the *if*, *switch*, and *loop* statements, all statements are closed with a semicolon.

If Statement

An if statement examines a given condition. If the condition is met, several statements can be carried out. The syntactic form of the if statements is:

```
if (condition)
   statement_1;
   statement_2;
```

```
   ...
   statement_n;
end if
```

Another form of an if statement is the “if else” statement. If the condition is not met, statements could be defined that should be carried out in this case. The syntactic form of the if-else statement is:

```
if (condition)
   statement_11;
   statement_12;
   ...
   statement_1n;
else
   statement_21;
   statement_22;
   ...
   statement_2n;
end if
```

Switch Statements

The switch statement consists of several *case marks* and an optional default mark. Such a mark can contain several statements. The switch statement evaluates the expressions and tries to find, one after the other, a correspondence with a case mark. If no case mark is accurate, the statements of the default mark are carried out. A case mark ends with a break, because otherwise the statements of the following case mark will be carried out until either a break appears or the switch statement is finished.

```
switch(expression)
case expression_1 :
   case_statement_1;
   break;
case expression_2 :
   case_statement_2;
   break;
case expression_3 :
   case_statement_3;
   break;
default :
   default_statement;
```

```
end switch
```

TokenSet Loops

Use the TokenSet loop to deal with multiline text. It works much like the ID-set mechanism. Note that TokenSet is not defined in the metamodel. To access values, TokenSet has the attribute line shown below.

```
loop(Instances->MClass->MOperation)
   out = [MOperation.name] ":\n";
   loop(Instances->TokenSet([MOperation.actionBlock]))
   out = "\t" [TokenSet.line] "\n";
   end loop
end loop
```

Predefined Functions

Output

TDL distinguishes between two different output facilities, line-based and stream-based output. Both variations can write in two output channels each (out and info). The output channel *out* can be set with the function setOutput.

- setOutput(*expression*)

  Sets the output channel to the file defined in the function parameter. The default output channel of *out* is standard output.
- setOutput() is a low-level function call. Use output() from the standard library to get additional functionality, e.g., to add the generated file to the list of files to be merged.

Line-Based Output

- `out = expression`

  Writes the output on the file defined with setOutput.

- `info = expression`

  Always writes the output to standard output.

An example of dealing with *out* and *info* can look like this:

```
out = "hello world";// output in the standard output

setOutput("dat1.txt");
out = "hello world";// output in the file dat1.txt
out = ...
.
.
.
setOutput("dat2.txt");
out = "hello world";// output in the file dat2.txt
out = ...
.
.
info = "hello world";// output in the standard output
.
.
.
```

Stream-Based Output

As in the template, stream-based output allows a "what you see is what you get" (WYSIWYG) syntax within a *proc* procedure. The stream starts with the line

```
>>
```

and ends with one of the following lines

```
>> out              // write to the output
>> info             // write to the terminal (stdout)
>> [variable]       // store output in a TDL variable
>> return           // return output as return value of
                    // current procedure
```

Stream-based output is normally used to store fractions of WYSIWYG code in a variable or return it from a procedure. Writing directly to *out* from TDL procedures is not recommended. Writing to *info* may be used for embedding messages or tracing information.

Some indentation control is supported with stream-based output. Leading white space characters to the left of the >> operator are removed from the beginning of all output lines. This allows syntactically correct indentation in the TDL procedural code. The following example produces different code depending on the number of function parameters. For a function with two parameters or less, all parameters are written on a single line, while longer parameter lists are formatted one parameter per line.

```
proc functionParams(MOperation)
   local plist;
   if(loopCount(MOperation->OpPara) < 3)

      loop(MOperation->OpPara;setDelim("");setDelim(","))
         [plist] = [plist] delim() [OpPara.type]
         [OpPara.name];
      end loop
      return "(" [plist] ")";
      else
         >>
         (
            [loop(MOperation->OpPara;setDelim("");
               setDelim(","))]
            [delim()] [OpPara.type] [OpPara.name]
            [end loop]
         )
         >> return
   end if
end proc
```

The resulting code using this TDL procedure might look like this:

```
bool foo(int bar);
long multiParamFunc
   (
        int param1
      , long param2
      , char* param3// etc.
   );
```

User-Defined Block

Manually generated code parts can be protected by user-defined blocks. If you generate code repeatedly, these parts are saved beforehand and are put in again later. For this reason, special markings in the code are necessary.

The templates shipped with StP use a generic TDL-template (typically defined in *<language>_std.tdl*) called "mergeOut" to create these user-defined blocks. The template is used like this:

```
[mergeOut("UDCO::"[MNormalClass.guid], "---- additional constructors")]
```

The template is typically called with (1) a logical identifier (UDCO for user-defined constructor in this case), (2) the object's GUID (which is mandatory to ensure data consistency if elements are globally renamed in StP), and (3) an additional string which is inserted as comment into the code.

To use this functionality, you must initialize the merger by calling:

```
delMergeFile();
appendMergeFile("merge_config_<language>.txt");
```

in your template's main procedure, where <language> is set to the value of your language.

When creating a template for a new language you must create a new *merge_config_<language>.txt* file (based on an existing file in *templates \ ct \ merge \ MergeConfig*) and make sure that the start and end tags used in the mergeOut template match those which are defined in the new MergeConfig file.

See also *Chapter 5, "ACD and the Merge Utility"*.

Other Functions

Loop Functions

Loop functions use navigation rules as parameters. They check certain characteristics of the result quantity. Alias mechanisms, *where* conditions, and the ID-set mechanism can also be used here.

- `hasLoop(navigation rules )`

  Returns True if the loop quantity contains at least one element, otherwise False.
- `loopCount(navigation rules )`

  The number of elements in the loop quantity

ID-Set Functions

The ID-set mechanism can be used to generate lists of metamodel element IDs. The functions getIdList and getIdListRec are available to populate these lists. Both functions are called by supplying common navigation rules, including conditions.

- `getIdList(navigation rules )`

  Returns a list of all metamodel element IDs defined by (navigation rules)
- `getIdListRec(navigation rules )`

  This is like getIdList, except that the navigation rule is used recursively.

  Note: Because of the difference between getIdList and getIdListRec, you cannot use getIdListRec like this:

  ```
  getIdListRec (Instances->MClass->MAssociation)
  ```

  You cannot use it because it is recursive and requires a role name after MClass that is MClass and not another metamodel class. In this case, use:

  ```
  getIdList (Instances->MClass->MAssociation)
  ```

  For example:

```
   loop(Instances->MClass)
   [ID_LIST] = getIdList(MClass->SuperClass);
                  // Id-List of all direct parent-
                  // classes.

   loop(Instances->MClass([ID_LIST]) AS SuperClass)
                  // Loop over all direct parent-
                  // classes.
      statements ...

   end loop

   [ID_LIST] = getIdListRec(MClass->SuperClass);
                  // Id-List of all parent classes. Also
                  // over several levels

   loop(Instances->MClass([ID_LIST] AS SuperClass)
                  // Loop over all parent classes of the
                  // inheritance tree.
      statements...
   end loop
end loop
```

Insert Mark Functions

The insert / mark functions provide a mechanism that makes it possible to mark a part of the code (mark) whose content is defined (inserted) at another place (before or after). It works roughly in a way that *insert* makes it possible to fill any number of containers and *mark* defines the place where the container are emptied.

- `insert (marker, expression)`

  Insert enables generated text to be inserted to a marked place. With *insert*, any text can be captured.

  The first argument defines an unambiguous indicator. In the second argument the main text is captured.

- `mark (marker, procName)`

  Marks a place in the output where the code captured by *insert* should be placed.

  The first argument defines an unambiguous indicator. This indicator connects *mark* and *insert*. The second argument defines a procedure that formats the captured text-pieces.

There are three cases that are applied successively until one of them is satisfied:

1. The procName argument contains a metaclass procedure / template name and insert() is called with element ID. The procedure / template must have one of the following definitions:

   ```
   proc <MetaClass>.<Procedure-Name>()
   ...
   end proc
   ```

   or

   ```
   template <MetaClass>.<Template-Name>()
   ...
   end template
   ```

   In this case, the procedure is called for every insert() statement of a metamodel element ID. If there is no such procedure defined for the element's metaclass (or its superclass), the next case is tested.

   For example:

   ```
   template genClassHeader(MClass)
       ...
       [mark("include", "inc_proc")]
       ...
       loop(MClass->SuperClass)
           ...
           // some class declarations...
           ...
           [insert("include", [SuperClass.id])]
           ...
       end loop
       ...
   ```

```
end template

template MClass.inc_proc()
#include "[MClass.name]"
end template
```

This example will call MClass.inc_proc() for every superclass of MClass (the template parameter).

2. The procName argument contains a TokenSet procedure name. The procedure needs to have the following form:

```
proc TokenSet.<Procedure-Name>()
...
end proc
```

This procedure is called for every line of every insert() statement.

```
template genClassHeader(MClass)
    ...
    [mark("include", "inc_proc")]
    ...
    loop(MClass->SuperClass)
        ...
        // some class declarations...
        ...
        [insert("include", [SuperClass.name])]
        ...
    end loop
    ...
end template

template TokenSet.inc_proc()
#include "[TokenSet.line]"
end template
```

3. The procName argument contains a blank string or there is no TokenSet procedure with the specified name defined. In this case the code, added by *insert*, is left unformatted into the output.

```
template genClassHeader(MClass)
    ...
    [mark("include", "")]
    ...
    loop(MClass->SuperClass)
        ...
        // some class declarations...
        ...
        [insert("include", "#include \""
[SuperClass.name] "\"")]
        ...
    end loop
    ...
end template
```

Property Functions

Property functions regulate global settings.

- `autoIndent(expression)`

  Defines the setting for the indent mechanism. The default is True.

- `delLeadWS(expression)`

  In a template procedure, the leading white spaces should be erased using delLeadWS(True). The default is False.

- `outDiff(expression)`

  If expression is True, a file, opened by setOutput is only modified (-> touched) if the output to be written has actually changed. The default is False.

- `udAutoRead(expression)`

  Automatically executes the function udRead before setOutput. The default is False.

Miscellaneous Functions

Besides the functions previously introduced, TDL offers more help functions:

- `eval_tcl(expression)`

  Evaluates the given parameter as a Tcl command.

- `close()`

  Closes the actual opened output file. Further output is directed to the standard output.