Getting Started with OTK.

Version 2.5

Sergey Zelenov
zelenov@ispras.ru

1 Introduction

This document considers the process of test development with the help of OTK tool. As an example test generation for compiler modules transforming arithmetical expressions is considered.

Below by *target module* we mean a compiler module under test, and by *target language* we mean programming language processed by this compiler.

The document contains the following parts:

2 Target Module Description

In our example target module is a compiler module that perform some transformations of arithmetic expressions containing additions of integer constants, integer variables and subexpressions that are combinations of such additions.

3 Creation of a Project

The example project 'Expressions' considered in this document comes with OTK Tool. It is located in `examples/expressions` subdirectory, so you can skip the description of new project creation and open the existing one (select menu item **File** → **Open Project**, enter the directory mentioned above and open `Expressions.otk` file) and proceed with model development (see Model Development (see section 4)).

In order to create a project that will contain test generator for the target module, start OTK Tool (OTK Tool requires no installation, simply execute **otk.exe** in Windows or **otk** in Linux/Unix environment).

To create a new project select menu item **File** → **New Project**. In the appeared window **New project** (see figure 1) you should do the following:

- type in project identifier (**Project identifier** field),
- set project location (**Project base directory** field),
- set name of the file where different project settings will be stored (**Project file name** field),
- specify root package of the project (**Root package name** field), as for Java language.

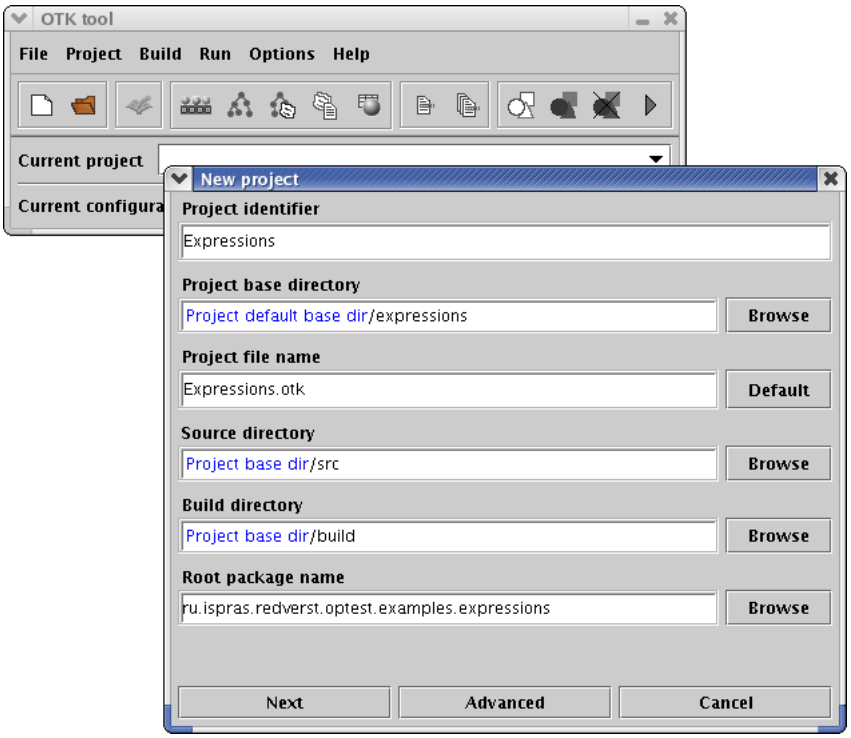

Figure 1: Creation of a new project

For directories with source files of the project (**Source directory** field) and and with compiled classes (**Build directory** field) OTK tool will propose default values (`src` and `build` subdirectories in the project location), you may use these settings or specify your own. When all is ready, press **Next**.

On the second step model name should be typed in (by default **Model** is used) and some additional information. In this dialog you may use default values for all fields. Press **Next** to proceed project creation.

In the third dialog you may use default values for all fields, too. Press **Finish** to finish project creation.

Now in the project directory the following artifacts are generated automatically:

- file with different project settings (with `.otk` extension),

- directory for project source files,
- directory for project compiled classes.

Furthermore, in the project source files directory the subdirectory for root java package will be created, and in this package the template for the formal description of abstract model will be generated (file with model name you've typed in earlier and with `.tdl` extension).

4 Model Development

OTK tool supports the *UniTESK technology*, that implies the clear and unambiguous description of input data of the target module. Such description of input data is called *abstract model of the input data* or simply *abstract model*.

In OTK abstract models are formally described using special language called TDL (Tree Description Language).

'Expressions' project already has abstract model (`Model.tdl` file in the root package of the project), so you can skip model development description and proceed with mapper development (see Mapper Development (see section 5)).

An abstract model is built on the basis of target module documentation. To create an abstract model one should analyze this documentation and build *summary diagram* of the model elements.

In our example documentation says: the target module erforms some transformations of arithmetic expressions containing additions of integer constants, integer variables and subexpressions that are combinations of such additions.

Thus our target module processes only arithmetic expressions (below we will simply call them "expressions") that are either addition or integer variable value or integer constant (see figure 2).

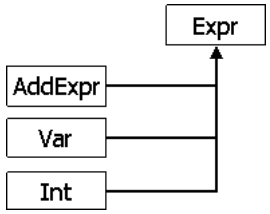

Figure 2: Kinds of expressions

An addition has two operands (left one and right one), each of them is also an expression (see figure 3).

Furthermore, one should specify the leading model element corresponding to the whole test (see figure 4).

Finally we get the following summary diagram (see figure 5):

On the basis of this diagram the formal description of abstract model is written using TDL. Open `Model.tdl` file located in the project root package. For opening files menu item

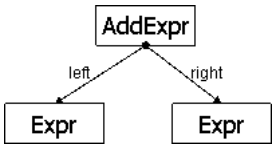

Figure 3: The structure of addition

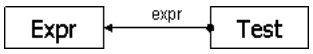

Figure 4: Leading model element

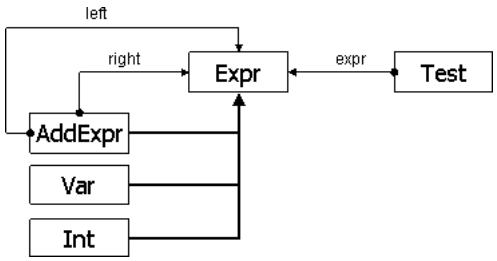

Figure 5: Summary diagram of the model

File → Open File can be used. By default the files are opened using a very simple built in editor, but you can specify your favorite editor using menu item **Options → File Editor**.

In a recently created project Model.tdl contains automatically generated template of model description:

```
[ translate.language = "java";
  visitor.name="TestVisitor";
]
tree ru.ispras.redverst.optest.examples.expressions.Model;

header
{
import ru.ispras.redverst.optest.OtkNode;
}
```

Here the model name is specified (using `tree` keyword), and there are some additional declarations (code enclosed with square brackets and a special `header` block).

After `header` block the definitions of the model elements should be inserted:

```
node Test : <OtkNode>
{
    child Expr expr;
}

abstract node Expr : <OtkNode>
{}

node AddExpr : Expr
{
    child Expr left;
    child Expr right;
}

node Var : Expr
{}

node Int : Expr
{}
```

Each definition of model element starts with keyword `node`, which can be preceeded by `abstract` modifier in case of generalized model element.

If the model element is an inheritor of the generalized model element then the name of the generalized element should be specified after colon. If the model element is not an inheritor of any other element, one should write '¡OtkNode¿' after colon.

References to other model elements are described as fields using `child` keyword.

Make sure that the model generated can be compiled without errors. Model compilation can be performed by **Build** → **Rebuild All** menu item.

5 Mapper Development

Our project contains formal description of model elements. Test generator will construct different structures from these elements. But in order to obtain tests for target optimizer, i.e. to generate tests in the target language (for example, in C) test generator should be provided with information about transforming abstract model elements to text in the target language. Such information is provided by a special component called *mapper*.

Mapper is typically a separate java class.

'Expressions' project already has the mapper called `DefaultMapper`, located in `ru.ispras.redverst.optest.examples.expressions.mapper` package, so you can skip the description of new mapper development and proceed with iterator development (see Iterator Development (see section 6)).

In order to create a new mapper select menu item **File → New File → New Mapper**, in the window appeared (**New mapper**) press **Add all non abstract nodes** to select all non abstract model elements (see figure 6). Then press **Next**.

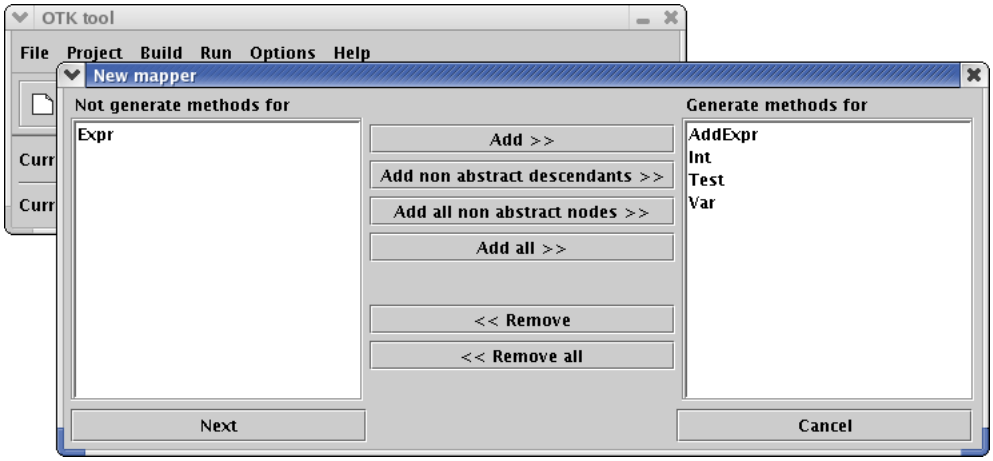

Figure 6: Selection of the model elements to be mapped

Nothing should be done for our mapper on the second step. Simply press **Next**.

On the third step package name and class name of the mapper should be specified (see figure 7). When all is ready, press **Finish**.

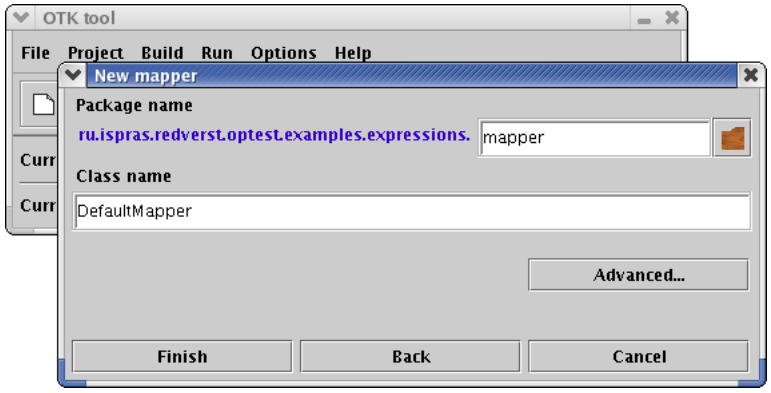

Figure 7: Specifying package name and class name for the mapper

Now the file containing the template for java class representing a mapper is generated. With comments thrown away, this template looks like following:

```
package ru.ispras.redverst.optest.examples.expressions.mapper;

import ru.ispras.redverst.optest.examples.expressions.Model;

public class DefaultMapper
    extends ru.ispras.redverst.optest.examples.expressions.EmptyMapper
{
    public void visitAddExpr( Model.AddExpr node ) {}
    public void visitInt    ( Model.Int     node ) {}
    public void visitTest   ( Model.Test    node ) {}
    public void visitVar    ( Model.Var     node ) {}
}
```

This class is an inhertor of `EmptyMapper` class, which is generated automaticaly for each project. For each model element selected during mapper creation the mapper class contains method, whose named is obtained from element's name by adding '`visit`' prefix. Such methods are responsible for mapping of corresponding model element into text.

Let's take a look at mapper method for **AddExpr** element, that describes addition expression:

```
public void visitAddExpr( Model.AddExpr node )
{
    txt( "( ${left} + ${right} )" );
}
```

In order to generate text the `txt` method from OTK Tool library is used. Our mapper inherits this method from `EmptyMapper` class. This method takes one argument – a string to be printed. This string can contain references to text that should be generating by processing the fields of the model element. Each such reference is a field's name enclosed in "{}" with leading "$": '`${...}`'.

Let's take a look at mapper methods for other model elements:

```
public void visitInt( Model.Int node )
{
    txt( "3" );
}

public void visitTest( Model.Test node )
{
    txt( "int main() {" ); nl();
    incIndent();
    txt( "int n = 2;" ); nl();
    txt( "int res = ${expr};" ); nl();
    txt( "printf( \"%d\\n\", res );" ); nl();
    popIndent();
    txt( "}" ); nl();
}

public void visitVar( Model.Var node )
{
    txt( "n" );
}
```

The mapper of **Test** element uses three more methods from OTK Tool library that can be used to indent the text generated:

- `nl` starts new line,
- `incIndent` increase the indention of text to be printed,
- `popIndent` restores the previous value of indention.

Mapper of **Test** element generates the text of function `main`.

Mapper of **Var** element generates simply prints the name of the variable - "n". In order to generate semantically correct programs, the definition of "n" variable is inserted in the **main** function.

In order to make the results of compiled test program work available for human, the call for **printf** function is inserted in the **main** function that will print the result of calculation of arithmetic expression from the test. In order to generate semantically correct programs, at the start of each program should be the appropriate include directive providing the definition of printf function. In order to specify the text that should be printed at the beginning of each file one should define `beginFile` method in the mapper:

```
public void beginFile()
{
    txt( "#include <stdio.h>" ); nl();
}
```

Make sure that the mapper developed can be compiled without errors. Mapper compilation can be performed by **Build** → **Rebuild All** menu item.

6 Iterator Development

Now our project contains formal description of abstract model elements and mapper that can transform different combinations of these model elements to the text in the target language. During test generation, test generator will construct different combinations of model elements. In general, one can create an infinite number of different combinations, but the size of test suite should not be infinite. In order to obtain a finite set of tests, one should limit the set of allowed model structures and provide generator with information on how to build model structures that satisfy these limitaions. In order to generate model structures from this limited set only, the special generator component is used, called *iterator*.

Iterator of model structures consists of the iterators of model elements. Iterator development involves the following:

- Development of iterators for model elements
- Creating of model structures iterator

6.1 Development of Iterators for Model Elements

For each model element an iterator should be developed. Normally, each iterator is a separated java class.

'Expressions' project already has iterators for all model elements, they are located in `ru.ispras.redverst.optest.examples.expressions.iterator` package. So you can skip the description of new iterator development process and proceed with specifying of iterator of model structures (see Creating a Model Structures Iterator (see subsection 6.2)).

Let's proceed with iterator development for model elements. To begin with, let's consider development of iterator for model elements without any fields: in our example these are **Var** and **Int** model elements. Let's start with iterator for **Var**.

In order to create a new iterator, select menu item **File** → **New File** → **New Iterator**, then in **New iterator** window choose iterator for non abstract model element – choose item **Non abstract node** (see figure 8) and press **Next**.

In the next dialog select model element for which iterator should be created from combobox (see figure 9). Nothing should be specified for **Var** element here, so just press **Next**.

Third, the package name and the class name of iterator should be specified (see figure 10).

Now press **Finish** and java class representing iterator for **Var** model element will be generated. Similarly iterator for **Int** element should be created.

Let's proceed with iterator development for model elements that have different fields and create an iterator for **AddExpr** element. The first dtep is the same as for **Var** iterator, and on the second step after selecting **AddExpr** in combobox one should specify iterators for **left**

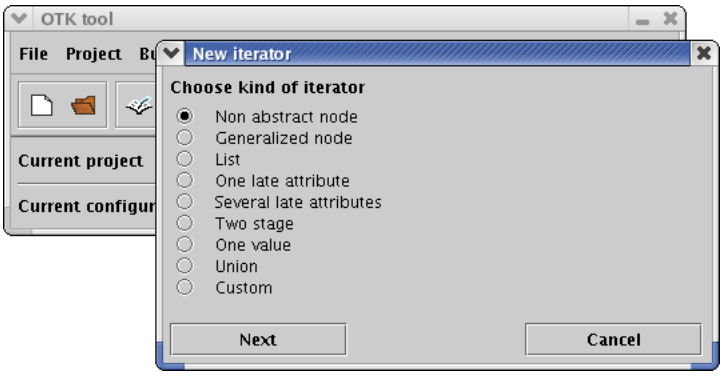

Figure 8: Choosing kind of iterator for non abstract model element

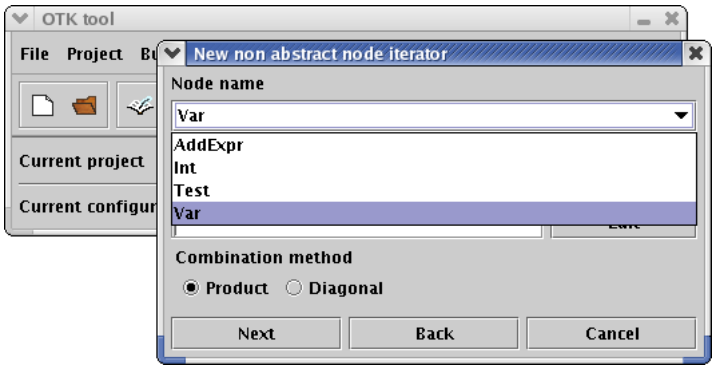

Figure 9: Choosing a model element to create iterator for

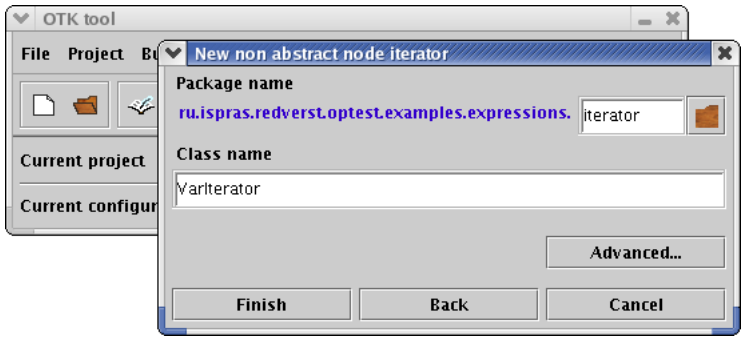

Figure 10: Specifying packege name and class name of the iterator

and **right** fields of these element (see figure 11). Both field are of **Expr** type, therefore the iterator of **Expr** model element should be used for them. Since the iterator for **Expr** model alement has not been created yet, one may choose iterator of one of its inheritors, for example, iterator for **Int** element. Select **left** field in dialog, press **Set iterator** and in the window appeared choose
`ru.ispras.redverst.optest.examples.expressions.iterator.IntIterator` java class from combobox (see figure 12). Similarly set the iterator for **right** field. Then press **Next**

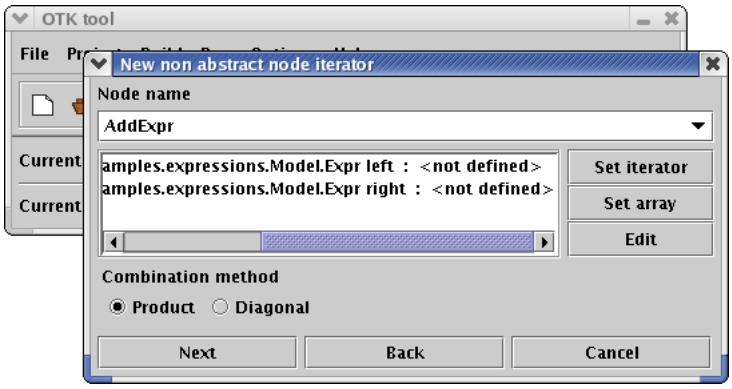

Figure 11: Iterator development for model element containing fields - the second step

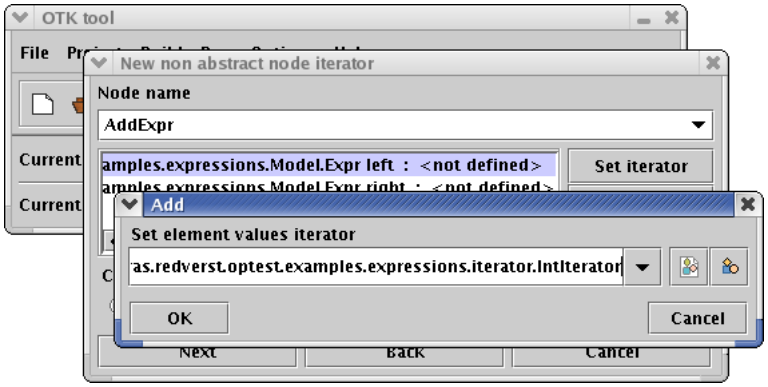

Figure 12: Choosing field iterator

and set packege name and class name of the iterator. When all is ready, press **Finish**.

The java class of iterator for **AddExpr** element is now generated. Since for element's fields we've chosen the iterator of **Int** element, the generated iterator will create only model structures corresponding to the addition of integer constants. In order to generate model structures corresponding to the addition of different expressions, one should specify the iterator of **Expr** element for **left** and **right** fields. First, this iterator should be created.

Expr model element is a generalized model element. In order to create a new iterator for it, select menu item **File** → **New File** → **New Iterator**, then in **New iterator** dialog select item **Generalized node** (see figure 13). When all is ready, press **Next**.

In the next dialog select model element for which iterator should be created – **Expr** element in our case. All inheritors of this generalized element will be shown (see figure 14). For each inheritor the appropriate iterator should be specified. In order to do this, select an inheritor and press the appeared **Add** button, then in the window appeared choose the appropriate iterator from combobox and press **OK** (see figure 15). When iterators of all inheritors are specified (see figure 16), press **Next**.

Third, the package name and the class name of iterator should be specified. When all is ready, press**Finish**.

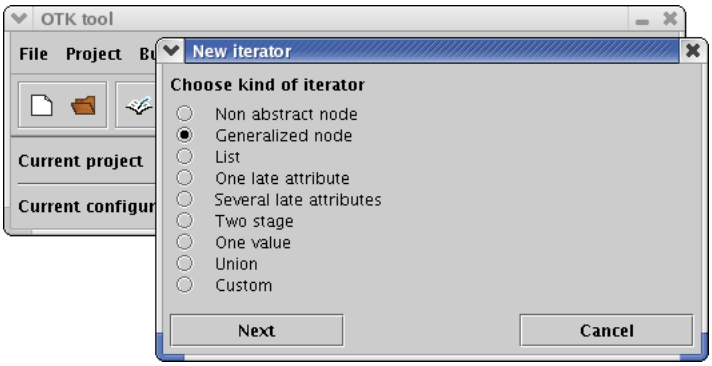

Figure 13: Choosing kind of iterator for generalized model element

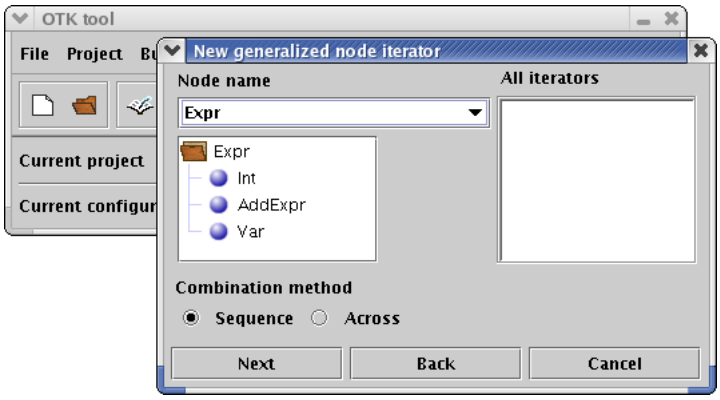

Figure 14: Setup window for iterators of generalized model element inheritors

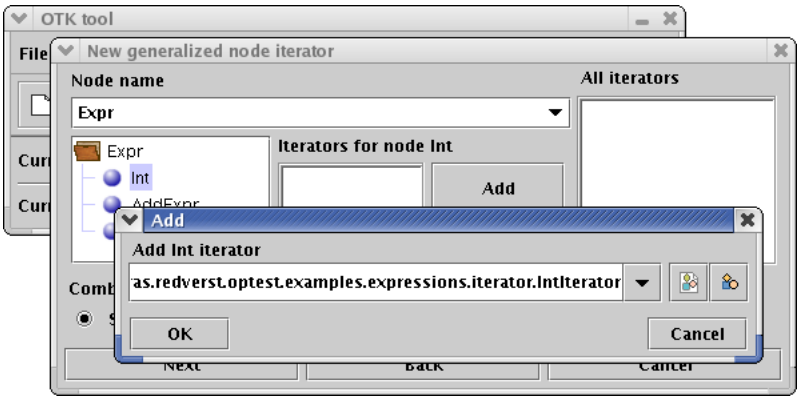

Figure 15: Choosing an iterator for one of the generalized model element inheritors

Now the java class representing iterator of **Expr** model element is generated.

Let's return to the iterator for **AddExpr** model element. Take a look at java code of this iterator's constructor. It contains `try` block, where the initialization of `main_iterator` is perform. With comments thrown away, the java code of `main_iterator` object initializations looks like following:

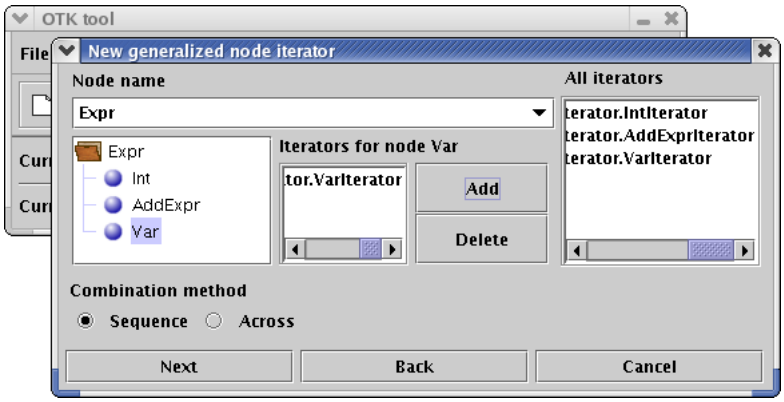

Figure 16: Specifying iterators for inheritors of generalized model element

```
main_iterator = IteratorFactory.createDetailsIterator
    ( OTK.getIteratorProperty( "iterator.AddExprIterator.combination" )
    , new ValueIterator[]
        { /* left */  new IntIterator()
        , /* right */ new IntIterator()
        }
    );
```

This code should be improved in order to create model structures corresponding not only to additions of integer constants, but to different addition expressions. In order to do this, one should specify the iterator of **Expr** element for the **right** and **left** fields.

But if we simply replace here `new IntIterator()` with `new ExprIterator()`, then during generation process the process of recursive constructor calls will be caught in an endless loop: in order to initialize **AddExpr** iterator it is necessary to create **Expr** iterator and vice versa. To avoid this problem each iterator provides **getDepth** method. If getDepth() ¿ 0 (i.e. we haven't reach the maximum allowed depth yet) then `new ExprIterator()` should be used, otherwise we should use iterator which don't have any recursive dependencies, for example, `new IntIterator()`. The improved java code of `main_iterator` object initialization looks like following:

```
main_iterator = IteratorFactory.createDetailsIterator
    ( OTK.getIteratorProperty( "iterator.AddExprIterator.combination" )
    , new ValueIterator[]
        { getDepth() > 0 ?
          (ValueIterator)new ExprIterator() :
          (ValueIterator)new IntIterator()
        , getDepth() > 0 ?
          (ValueIterator)new ExprIterator() :
          (ValueIterator)new IntIterator()
        }
    );
```

The iterator for **AddExpr** model element is ready.

The iterator for **Test** model element can be created in the same way as the iterator for **Var** model element, but on the second step the iterator of the **Expr** model element should be selected for **expr** field.

Make sure that the iterators developed can be compiled without errors. Compilation of the iterators can be perform by **Build** → **Rebuild All** menu item.

6.2 Creating a Model Structures Iterator

After iterators for all model elements are developed, the iterator of model structures should be created, that will be used by test generator.

'Expressions' project already has model structures iterator, so you can skip the description of creating new model structures iterator and proceed with test generation (see Test generation (see section 7)).

In order to create a new model structures iterator, select menu item **Project** → **Default iterator** and in the window appeared type in package name and class name for model structures iterator. Also in **Tree iterator class** field you should select an existing iterator which will be the leading iterator during test generation (in our case it is iterator of **Test** model element) (see figure 17). When all is ready, press **OK**.

Make sure that the project has no errors. Perform a project build using **Build** → **Rebuild All** menu item andd make sure that there are no compilation errors.

7 Test Generation

Now our project contains formal description of abstract model elements, mapper that can transform different combinations of these model elements to the text in the target language and the model structures iterator.

Before starting test generator, the project should be compiled. In order to create all necessary files, select menu item **Build** → **Rebuild All**.

Test generation involves the following actions:

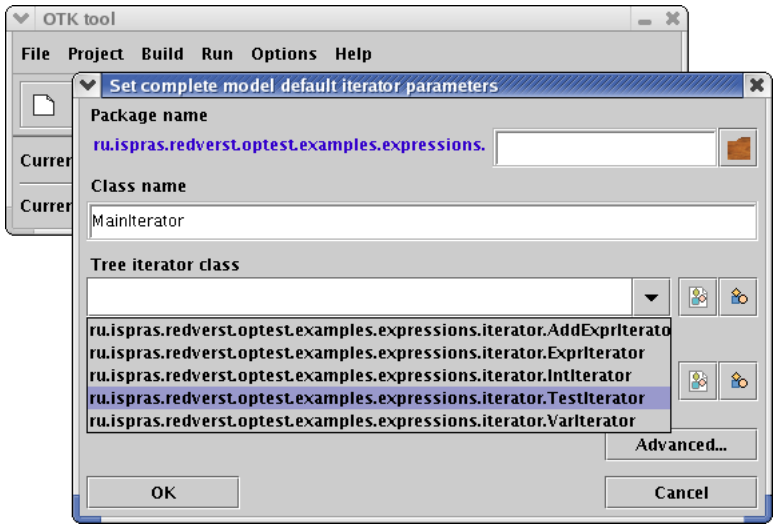

Figure 17: Creating model structures iterator

- Configuring test generator (see subsection 7.1)
- Starting test generation (see subsection 7.2)

7.1 Configuring Test Generator

During test generation process, the generator will generate different structures consisted of model elements one by one and map them into text in the target language. Before generator is started, it should be configured: one should specify the iterator of model structures and mapper.

'Expressions' project already has adjusted configuration, so you can skip description of new configuration creation and proceed with starting test generation process (see Starting test generation (see subsection 7.2)).

To create a new configuration select menu item **Run** → **New Configuration**, then in the appeared window **New configuration** specify the configuration identifier (see figure 18) and press **Next**. Now the configuration window will appear. The components of the test

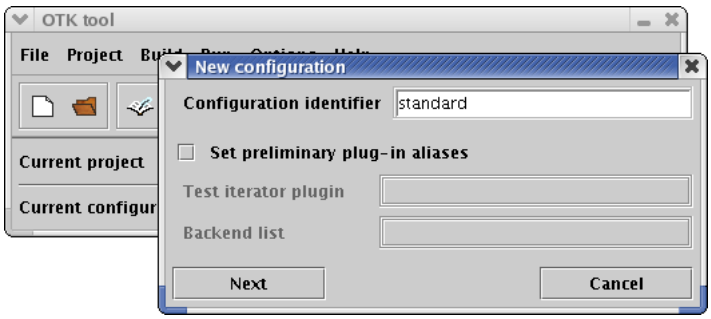

Figure 18: Creation of a new configuration

generator developed in our project should be set up in the configuration as *plugins*. First the iterator of model structures should be set up. Specify the name of iterator plugin in **Test iterator plugin** field and press **Default** button which is situated on the right from this field. The folder with information about iterator plugin should appear in the configuration window (see figure 19).

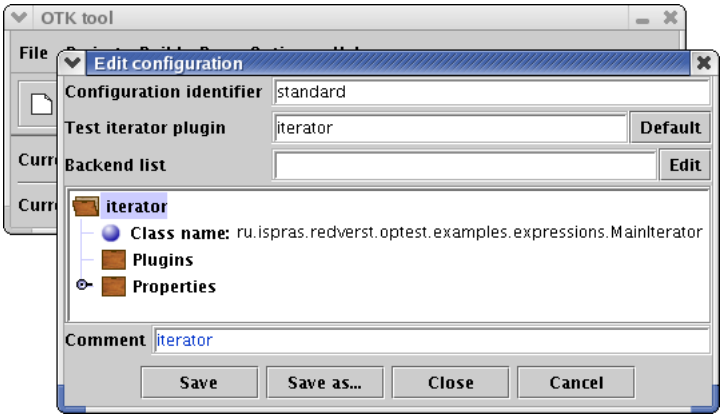

Figure 19: Setting up the iterator of model structures

In order to setup mapper one should open the window with the configuration of different model processors (backends) by pressing **Edit**, which is situated on the right from **Backend list** field (note that the **Backend list** checkbox should be checked). In the window appeared in the toolbar near the right border press the sixth button from top, (which has '**Insert new standard printer before selected item**' tip) (see figure 20). In the window appeared you

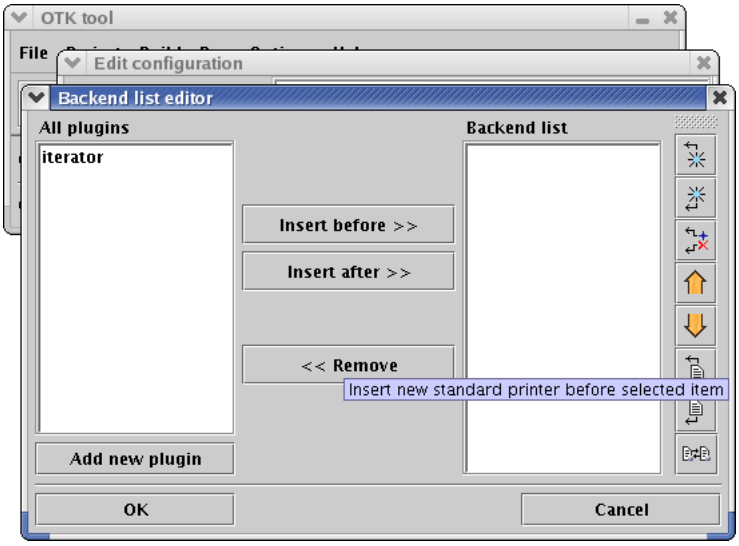

Figure 20: Adding printer in the backend list

should specify the printer plugin name (**Printer plugin name** field), select mapper java

class from combobox in **Mapper class name** field and type in the mapper plugin name (**Mapper plugin name** field) (see figure 21). Now you can press **OK** here. In the backend

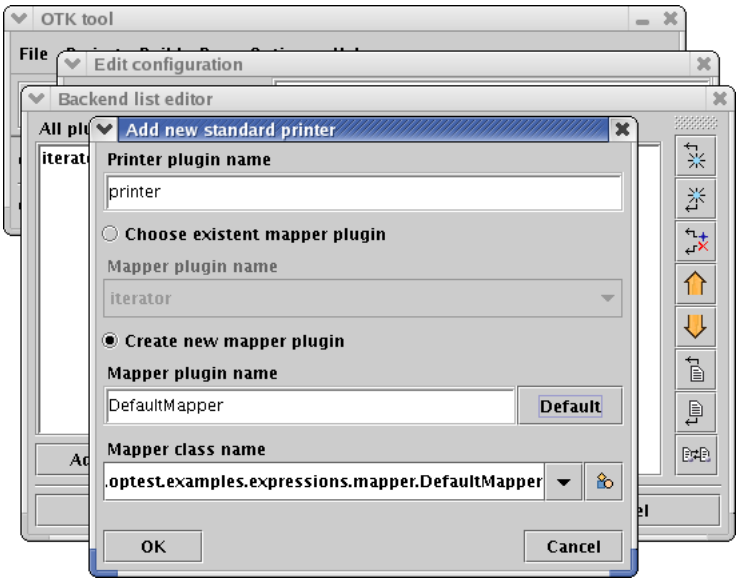

Figure 21: Specifying mapper class in the printer

editor press **OK**, too.

Now the configuration contains information about all necessary plugins. This information represented as a directory tree in the configuration window (see figure 22).

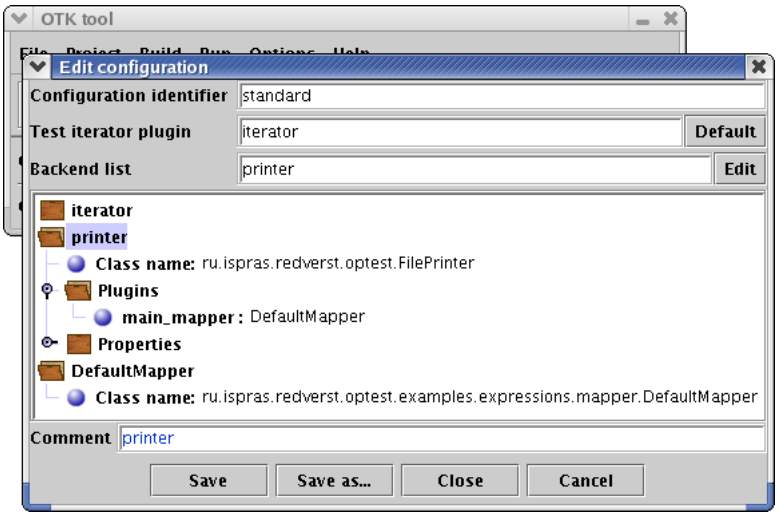

Figure 22: Configuration window after specifying all plugins settings

Now the printer should be configured. In configuration window in **printer** plugin folder open properties folder (**Properties**). Printer has four properties (see figure 23):

- `output.dir` – directory to hold generated files with tests;

- `file.prename` – prefix of the generated files names;
- `file.postname` – prefix of the generated files names;
- `file.size` – number of tests to be printed in one file.

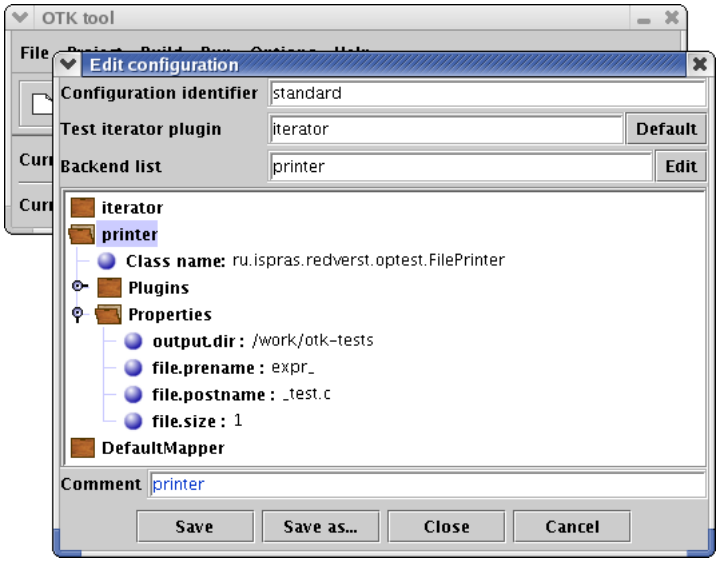

Figure 23: Configuring printer

Specify a directory where the generated files with tests will be situated. By left double click on `output.dir` property make it active (note that the GUI is quite slow regarding mouse double clicks; you should perform two clicks with some delay between them, or a fast "triple" click). In order to browse your file system, press the folder icon on the right of the property string and choose the directory where the test should be placed.

Specify the prefix of generated files names in `file.prename` field.

Specify the postfix of generated files names in `file.postname` field.

Set the number of tests printed in one file to 1.

Printer configuration is ready (see. figure 23).

Now the test generator configuration is ready. Press **Save** to save all changes, then press **Close** to close configuration window.

7.2 Starting Test Generation

Before starting generation make sure that the configuration contains valid directory name to place generated files into. Select menu item **Run** → **Edit Configuration**, in the appeared window **Edit configuration** open **printer** folder, then open its subfolder **Properties** (see figure 23) and by left double click on `output.dir` property make it active (note that the GUI is quite slow regarding mouse double clicks; you should perform two clicks with some delay between them, or a fast "triple" click). In order to browse your file system, press the folder

icon on the right of the property string and choose the directory where the test should be placed. When all is done, press **Save** to save all changes, and then press **Close**.

To start test generation, select menu item **Run** → **Start Generator**. The **Generate** window will appear reflecting generation process – for each generated file a dot will be shown. If generation finishes successfully, the total number of tests generated will be printed (see figure 24). Press **Close** to close the window.

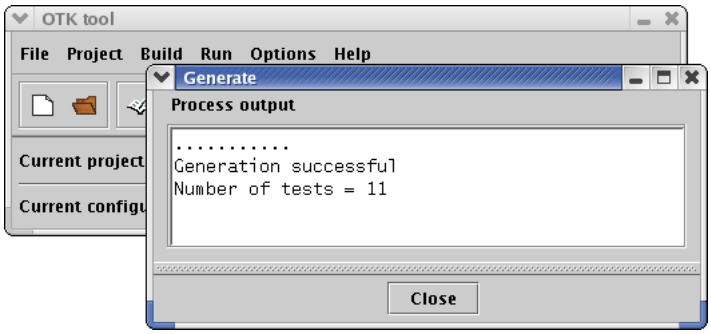

Figure 24: Test generation was finished successfully

By default test generator creates tests where the depth of subexpression nesting is limited by 2. In order to generate tests with greater depth of subexpressions nesting, one should specify the maximum allowed value of depth in the configuration. This could be done by selecting menu item **Run** → **Edit Configuration**, in the appeared window **Edit configuration** open **iterator** folder, and then its subfolder **Properties**. By left double click on **AddExpr.depth** make this property checked and change its value (see figure 25). When

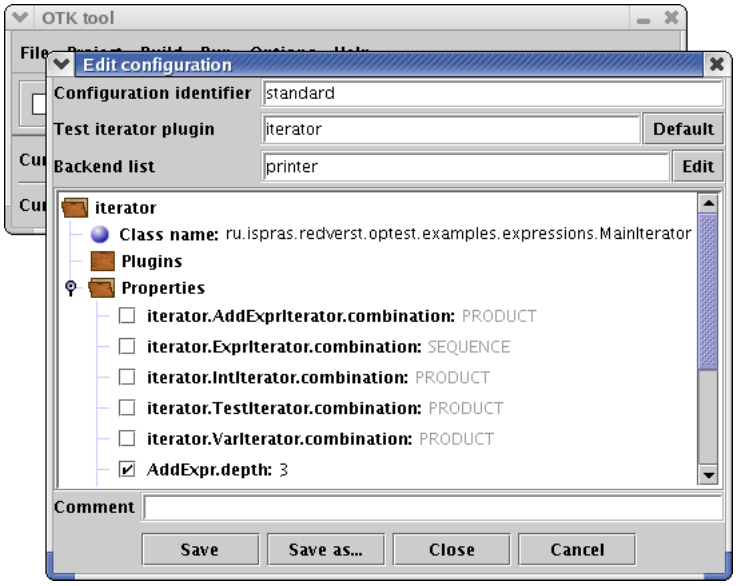

Figure 25: Changing the properties of the model structures iterator

all is done, press **Save** to save all changes and then **Close**. Now test generation can be

started by selecting menu item **Run** → **Start Generator**.