

Project in Software 236504
SSDL

Core to Murphi XML version

Roni Bar-Am
Royi Ronen

Supervisor:
Shahar Dag

Table of Contents:

1. General Project Description	3
2. User Guide	4
Sources:.....	4
Other files:	5
External libraries used in the code:	5
External websites used for our project:	5
Running the project:	6
Versions used in the project:.....	6
3. Algorithm Description	7
Introduction.....	7
Algorithm.....	7
Additional Information	9
4. Documentation	11
5. Code.....	14
6. Examples.....	15

1. General Project Description

This project is Java software that translates programs in the Core Description Language (CDL) to programs in Murphi (a verification language). The input is a CDL language that simulates verification tasks, and the output is a Murphi program that simulates the same tasks.

The input program is an XML document form. The document complies with `cdl.dtd`, a document type definition file. The output is a Murphi file, which complies with the Murphi definition grammar as appears in the Murphi manual and with the corresponding compiler that is submitted with this project. The translation algorithm translates first from the CDL XML document to an XML representation of the target Murphi program, and then creates a textual program from it, which can be an input to the Murphi compiler. The intermediate XML document, which represents the target Murphi program, complies with the `Murphi.dtd` file in this submission.

The translation process also produces an additional information file with information about the translation. This is a resource for information that was lost by the translation process.

The following illustrates our project's operation:

CDL program (`cdl.dtd`) →

Murphi XML (`murphi.dtd`) + Additional Info File →

Murphi Text + Additional Info File (unchanged from previous step).

This document contains:

User Guide;

Translation Algorithm Description;

Programmer's guide and code documentation (as HTML Java-Doc printouts);

Project Code;

Examples.

Roni Bar-Am: snm1@cs.technion.ac.il

Royi Ronen: ronenr@cs.technion.ac.il

Technion-IIT, Haifa, December 2004.

2. User Guide

The project consists of the following files:

Sources:

declaration/Constdecl.java
declaration/Decl.java
declaration/Typeddecl.java
declaration/VarType.java
declaration/Vardecl.java
expression/And.java
expression/ArrayElem.java
expression/Boolconst.java
expression/Designator.java
expression/Div.java
expression/EQ.java
expression/Expression.java
expression/FALSE.java
expression/GE.java
expression/GT.java
expression/IntegerConstant.java
expression/LE.java
expression/LT.java
expression/Minus.java
expression/Mod.java
expression/Not.java
expression/NotEQ.java
expression/Or.java
expression/Plus.java
expression/TRUE.java
expression/Times.java
rules/Alias.java
rules/AliasRule.java
rules/NonDeterAssign.java
rules/NonDeterRuleset.java
rules/Relation.java
rules/Rule.java
rules/Rules.java
rules/Ruleset.java
rules/SimpleRule.java
rules/StartState.java
rules/UnInitializedRuleset.java
rules/UnInitializedRulesetVar.java
statements/AssignStmt.java
statements/ForStmt.java
statements/IfStmt.java
statements/Quantifier.java

statements/Stmt.java
statements/Stmts.java
statements/Type.java
translation/Change.java
translation/Core2Murphi.java
translation/Murphi2Text.java
translation/MurphiNode.java
translation/OneToOneTranslation.java
translation/Translator.java
types/Array.java
types/Boolean.java
types/Enum.java
types/ID.java
types/Range.java
types/RangeInteger.java
types/StringName.java
types/TypeExpr.java
utils/ImprovedInteger.java

Other files:

cdl.dtd
murphi.dtd
Core2Murphi2 – Running Script
Core2MurphiXML.jar, core2MurphiText.jar -Executables
murphi – Murphi language compiler
Makefile
Example files.

External libraries used in the code:

We used the Document Object Model library (DOM) for all XML utilities.

We used Xalan for evaluating XPath queries used in the implementation, especially when transforming the XML Murphi program to a valid Murphi textual program.

External websites used for our project:

Murphi site: <http://verify.stanford.edu/dill/murphi.html>

Site for Xalan libraries: <http://xml.apache.org/xalan-j>

Installing the project:

The files can be run from every java enabled machine. Users must verify that the internal structure of the folder (i.e., sub-folders, dtd files etc.) is kept (before make). After make, users must ensure that files referred to in

the running (in examples for instance) are in the path used for the reference.

Make options are:

Make, make all: compiles the files and generates .class files.

Make clean: cleans the .class files and the executable jars.

Make release: compiles and packs the files in the two aforementioned jars.

Running the project:

The project is from the folder where the executable Core2Murphi file is.

In the command line, type:

```
>Core2Murphi <filename.xml> <murphi.dtd>
```

The project transforms flat CDL files. In order to translate non-flat CDL programs, it is required to first translate them to the flat form, using the existing Veri-Tech component (CDL to flat CDL). In this case, the command should look like this:

```
>Core2Murphi <filename.xml> <murphi.dtd> -flat
```

The translated file and the additional info file will be produced in the same folder the input was taken from.

Each of the two components can be run separately, if needed:

```
java -jar Core2MurphiXML.jar <inputCDLfile> <murphi.dtd>
```

```
java -jar Core2MurphiText.jar <inputMurphifile>
```

The target of the project can be compiled with the Murphi compiler,

```
>murphi <targetFile.m>
```

Versions used in the project:

Murphi: 3.1

Java: jdk1.41

Eclipse was used for development, version 3.0.

3. Algorithm Description

Introduction

The Core2Murphi program performs the following tasks:

1. translating between Core (flat Core to be exact) XML to Murphi XML
2. translating between Murphi XML to Murphi text.

The program uses the Java DOM (Document Object Model) in order to help us parse the XML file.

Algorithm

A flat Core program consists of a collection of *transitions*. A Core transition is based of the following three:

1. *Enable* expression – the transition's condition
2. *Body* – assignments statements to execute
3. *Relation* – a post condition to the transition.

In Murphi, the closest component which is related to a Core transition, is called a *simple rule*, which also has an enable expression, and a body (which can do much more then simple assignments like Core does, but we were only translating from Core to Murphi and not vice versa), but not a relation (will be explained later).

So basically, each Core transition is translated to a Murphi simple rule, and that is not a problematic issue. The challenges we encountered were:

1. Variables in flat Core program can be un-initialized – meaning that they can have any possible value from their range. In Murphi all variables must be initialized at the *start state*.
2. Core syntax supports non-deterministic assignments such as $a = \{1, 2\}$, and Murphi does not support those kind of assignments
3. If the flag *HOLD PREVIOUS* was not set in the Core program, in each transition, each variable that was not assigned in the body, does not keep its old value, but rather gets any value (same as non-initialization that was discussed at section 1)
4. Relation component does not exist in Murphi syntax.

To solve all of these problems, we use the Murphi *ruleset*.

A ruleset is basically a collection of rules, wrapped together. Each ruleset has a variable (or a few), which is unique to the ruleset, that can have any value from a range we specify at the beginning of the ruleset. Each rule can use this special variable at any stage.

There is a ruleset for example:

```
Ruleset i : MININT..MAXINT Do
Rule

// a regular rule that uses i
end;
end;
```

How that can help us? Let's observe the problems again:

1. For each non-deterministic assignment, we will create a global array (a new variable in the system) that will hold all of the possible values of the assignment. Also, we will create a ruleset, which its variable will be an integer that will get a value from the range of the array's indexes. Then we will assign the variable to the array at the index that was chosen.

For example, let's say we have the assignment $a = \{1, 2, 3\}$, then we will create an array that will have those three values, and we will create a ruleset:

```
Ruleset i : 0..2 Do
Rule
    ...
    a = array[i]
    ...
end;
```

2. For un-initialized variables, we can create a ruleset with a variable that receives values from the same range of the un-initialized variable, and assigns the un-initialized variable with the value of the ruleset variable. Therefore, we need to make sure that we will enter those rulesets of initialization before we start the 'main' rules (the rules that were created from the translation of the Core transitions).

3. In case of non-existing values after the end of transition, we will simply act exactly as in the first problem, but here we need to make sure that after the transition is over, we will go to the relevant rulesets (to initialize all the variables that their value was not kept), **before** we continue with the normal flow of the program.

For all of these solutions, we need to interfere in the normal flow of the program (we need some rules that will be executed before other rules, a request that Murphi does not support directly). That's why we need a special variable, which will keep the flow of the program the way we want it. This variable is called PC, and for each rule, or ruleset, its requested value will appear also in the rule condition.

For example, in order to initialize the first variable, we will demand in the rule that

PC = 0

And in the rule itself we will increment the PC by 1.

That is how we maintain the requested flow for the program.

4. As for relation, we implement those, by creating two simple rules. The condition of one of them is the relation condition, and the condition for the other one is the logical not (!) of the relation condition.

The simple rule that has the relation condition increments the PC, so the program can continue as usual. The simple rule that has the other condition, should stop (because if we go into the rule, that means that the relation condition was not satisfied), so we simply do not increment the PC variable, so the program is 'stuck'.

Additional Information

Additional information is added to the translation product in order to keep information regarding the source of translated items, since every translation implies loss of information.

The additional info mechanism that we have implemented ties Source elements to Target elements. They are both child elements of the every Change element in the additional info file. The type of connection between the two elements is the contents of the "title" attribute in the target and source elements. The possible connections are: Relation, Variable not initialized and Non deterministic assignment. The type of connection is the information about the origin of the translated item. Non deterministic assignment means that the translated target whose ID is element's Target ID came from the non-deterministic assignment that appeared in the Source item whose ID is in Sources' ID attribute.

Relation means that the translated feature was used. Each translation results in two rules, on holding the enable condition and the other – its negation. For the first rule, we continue normally in the program (by

updating the PC), and for the other one, we simply verify that we could not go to any other rule by setting a global failure flag. In the additional info file, there are two connections: a. the target ID is the ID of the first rule element, and in the source element is the ID of the element representing the relation in the cdl file. b. the target ID is the ID of the second rule element, and in the source element is the ID of the element representing the relation in the cdl file.

Variable not initialized means that the variable (either global or local to transition) is un-initialized. In Murphi, each variable has to be initialized, so we need to give it any value from the range it holds.

Therefore, we create for it a ruleset, such that the ruleset's variable is of the same type of the un-initialized variable. In the ruleset we assign the ruleset's variable value into the un-initialized variable. The ID of the target in the additional info is the ID of this ruleset. The target is the ID of the variable element in the cdl file that was not initialized.

Additional Info is easily expandable to as many types of connections as needed.

4. Documentation

Detailed Javadoc documentation is a part of the submission. Following are guidelines for the use of a prospective programmer who assigned to work on this code. This is an overview, and it is recommended to look at the Javadoc in the course of reading this overview.

Package translation consists of the main starting point of the code. Class `Core2Murphi` deals with the translation from Core in xml form to Murphi in xml form. Class `Murphi2Text` deals with the translation from Murphi xml to Text.

`Core2Murphi` creates the DOM objects for the source, target and additional info files, and constructs the translator, which is in charge of executing the translation itself. The central method is `translate()` which performs the translation in two parts:

1. Translating the declaration parts which include:
 - 1.1 - variable declaration
 - 1.2 - constants declaration
 - 1.3 - typedef declaration (alias in Murphi)
 - 1.4 - enum declaration
2. Translating the actual rules. The translation is based on information that is passed from the declaration part (like variable names, type definitions, etc.).

After the translation, the class serializes the files.

The translation itself is done using objects that are dedicated to the translation of each of the source elements. We take the Alias example:

```
public Alias(  
    Element coreNode,  
    ImprovedInteger counter,  
    Document targetDoc)
```

The constructor gets the source element as a parameter, together with the target document (Improved Integer will be discussed later). This class is responsible for the transformation of the XML node in the target document that has to be translated to Alias, and creates the corresponding nodes in the target docs. Note that in many cases it will have to use other classes for the translation, since the node to be translated may have many descendants that may be taken care of by other classes.

Each of the following is a package that consists of classes that perform the translation into the Murphi elements with names identical to the class names:

Declaration

Expressions

Rules

Statements

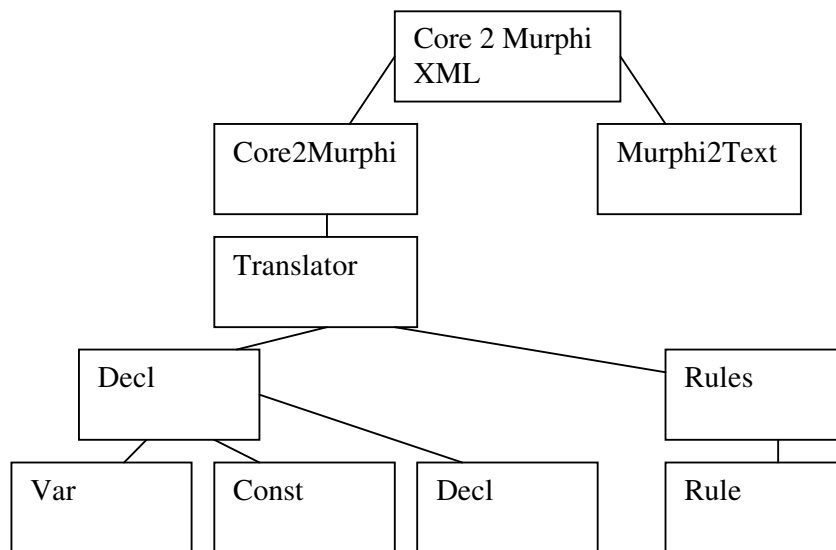
Types

Extensive explanation regarding the functionality of each class and documentation exist in the Javadoc files.

ImprovedInteger is our wrapping to Integer which allows increment, no other differences.

It is used for the counter object which is used in order to give unique ids to elements.

OneToOneTranslation is a class that is extended in any case where the translation is trivial (Ehad le-Ehad). There are only slight changes when translating these classes, therefore they are all treated in one class that gets parameters from its extensions.



The algorithm used in our project was taken from the previous C version of the translation, written buy Yaacov Esterin. We fixed a few bugs in order to be aligned with the algorithm (but the algorithm was not

changed). The only non-trivial bug was that the translation of one variable with more than one non deterministic assignment is not anymore translated to the same ruleset, which resulted in non-correct output. Now, every such assignment is translated to a dedicated ruleset. A limitation of Yaacov's algorithms was inherited: cdl variables cannot be reserved words in murphi (rare, uninteresting case).

Murphi does not have integer as a primitive type. Instead, it uses ranges. In cdl, integers are not bounded between -32K and 32K. This means that variables in Muprhi are limited to the 64K possible range integers. In practice this is not a real limitation.

Errors in examples: The following examples contain errors in the cdl file. semaphore_core (erroneous cdl), program_counters (erroneous cdl) and election core (variable names used but not defined).

5. Code

6. Examples

Each of the following examples consists of a translation source in flat CDL, and the translated Murphi code that our project produced from it. We include the additional info file as well. The additional info file can be used, on top of its regular use, to understand what can be learnt from every translation. The Connection element provides the type of translation used in every program. The examples have example.txt file in the directory, to point important points for the example.