

3.9.2002

## **VeriTech project –smv to core**

This document describes the additions and changes that were made to the smv to core translation in summer 2002 by Efrat shabtai. The original smv to core translation was written by Eyal Sonsino in summer 2001.

### **XML Output**

The output of the original smv to core translation was in cdl format. In the improved translation the default output is in XML output when there is a way for the user to choose the former cdl output.

#### ***Implementation of the XML output***

I added a class to the Compiler package called XMLPublisher. This class implements IVisitor and is very similar to the class CDLPublisher which implements the cdl output in the original translation.

#### ***Command line flag***

Some code was added to the CmdLine class in package Compiler in order to allow the user to choose the cdl format for the translation output by adding `-c` in the command line. The default format for the translation format is XML format.

All the changes and additions to the code are marked with a comment that contains '@' in it.

### **Handling TRANS INVAR and INIT**

#### ***about TRANS INVAR and INIT***

Any module in smv can have TRANS, INVAR and INIT conditions.

TRANS defines a condition on the current state  $s$  and on the next state  $s'$ . If the condition is not true in the current state then  $s'$  is not created by the smv. It is a restriction condition on the "next states" and does not restrict the initial state.

INVAR defines a condition over all of the state – an invariant of the system. All states of the system should fulfill the condition.

INIT defines a condition on the initial states.

#### ***Data Structures***

I added new classes to represent TRANS, INVAR and INIT in the data structure.

The class for TRANS is called Trans and contains a private data member of type Expression\* holding the TRANS condition.

The class for INVAR is called Invar and contains a private data member of type Expression\* holding the INVAR condition.

The class for INIT is called Invar and contains a private data member of type Expression\* holding the INIT condition.

All of them are in the SMV package.

I added 3 private data members to the Module class, of type Trans\* Invar\* AND Init\*. If a certain module in the smv program contains more than one section of TRANS then the 2 conditions are combined into one with an 'and' between them. The same for INVAR and INIT.

These classes also implement an accept function as part of the Visitor Pattern.

### ***The translation of Eyal***

Let us consider an example of Eyal's translation:

The smv program:

```
MODULE main
VAR
    t : BOOLEAN;
    u : 1..4;
ASSIGN
    init(t) := TRUE;
    next(t) := !t;

    init(u) := 3;
    next(u) := u;
```

is translated into the core program:

```
HOLD_PREVIOUS

TYPE
    T#SYSTEM#u: 1..4;

MODULE SYSTEM()
{
VAR
    t: BOOLEAN;
    u: T#SYSTEM#u;

    (
        v#SYSTEM#t(t)
        || v#SYSTEM#u(u)
    )

}

MODULE v#SYSTEM#t(t: BOOLEAN)
{
VAR
    __init: BOOLEAN INITVAL TRUE;

TRANS T_t_0:
    enable: __init;
    relation: (t'=TRUE) /\ !__init';
```

```

TRANS T_t_1:
    enable:  !__init;
    relation: t'!=t;

}

MODULE v#SYSTEM#u(u: T#SYSTEM#u)
{
VAR
    __init: BOOLEAN INITVAL TRUE;

TRANS T_u_0:
    enable:  __init;
    relation: (u'=3) /\ !__init';

TRANS T_u_1:
    enable:  !__init;
    relation: u'=u;

}

```

Mind a few remarks about the translation:

- For each variable in the smv program a module is created in the core program, for example for the variable `t` in the smv program the module `v#SYSTEM#t` is created in the core program.
- All modules in core that represent variables in smv are combined synchronically in the module that represents the translation of the smv module that contained these variables in the smv program. In this example the synchronic combination of `v#SYSTEM#t` and `v#SYSTEM#u` is in module `SYSTEM` that represents the smv module `main`.

This combination will be referred here as “former module combination”.

- In the current translation the initial state of the smv program is in fact the state after the first transition in the core program. The reason for this is that in smv it is possible to define `init(X)` (the initial value of variable `X`) in a case structure and in core there is nothing parallel to that.

The translation creates a flag (a local boolean variable):

```
__init: BOOLEAN INITVAL TRUE;
```

in every module in which there is a need to determine the initial values of variables. This flag is initialized to true and in the first transition it gets the value false. In order to examine a correctness formula `f` on the initial state that is origin in smv, it has to be transformed into the formula `__init -> f`.

There is a global definition for “`__init`” called `INIT_FLAG` and declared in file `Global.h`.

I am not sure that this flag appears in every program but according to Eyal the fact that the initial state is after the first transition in core, is true for all translations.

### ***Changes to the current translation***

The current smv to core translation (Eyal's translation) doesn't deal with TRANS, INVAR and INIT.

The translation for TRANS *trans\_cond* appearing in module A in smv would be:

```
Module trans#A
{
VAR
    __init : boolean INITVAL true;

    TRANS first:
        enable : __init;
        relation : __init' = false;

    TRANS next:
        enable : !__init;
        relation : trans_cond;
}
```

This module will be combined synchronically with the “former module combination” This way the TRANS condition is not forced on the initial state, but every non initial state has to fulfill this condition.

The translation for INVAR *invar\_cond* appearing in module A in smv would be:

```
Module invar#A {
    TRANS next:
        enable : true;
        relation : invar_cond' ;
}
```

When *invar\_cond*' is the INVAR condition where each atom in the original condition appears here as a tagged atom (the next value of the atom).

This module will be combined synchronically with the “former module combination”. This way all the states including the initial state will fulfill the condition. States that do not fulfill it, won't be created.

The translation for INIT *init\_cond* appearing in module A in smv would be:

```
Module init#A {
VAR
    __init : boolean INITVAL true;

    TRANS first:
```

```
enable : __init;  
relation : (__init' = false) & init_cond' ;
```

TRANS next:

```
enable : !__init;  
relation : true;
```

```
}
```

When *init\_cond'* is the INVAR condition where each atom in the original condition appears here as a tagged atom (the next value of the atom).

The second transition is necessary since this module will be combined synchronically with the “former module combination” and therefore there should be at least one enabled transition in the module in all times.

### ***Main Implementation Notes***

The main changes to the existing code for handling TRANS, INVAR and INIT:

- XMLPublisher.cpp in function printSubModules – The TRANS INVAR and INIT modules are printed before all the other module combination. The isUpper() flag function is true for these modules.
- In order to create the transition for the TRANS, INVAR or INIT modules, 3 new classes were added – TransBuilder InvarBuilder and InitBuilder. The class CDLBuilder has a pointer to each one of these classes and when necessary it calls the function buildInvarTransitions() buildTransTransitions() or buildInitTransitions() in order to build the transitions.

All the changes and additions to the code are marked with a comment that contains '@' in it.

## **Additional Information**

Eyal did not save additional information of the translation.

### ***The additional information***

The additional information of the translation:

- **Type Replacement** – when a variable in smv is of one of the types of enumerate and range, there is a problem to translate the variable declaration directly because in core enumerate and range types can only be global types (to appear under the TYPE global section in the core program). The translation creates a new global type and the variable declaration contains this new type. For example – if the smv variable declaration was:

```
VAR  
  u : 1..4;
```

then the core code would look like:

```
TYPE
```

```
T#SYSTEM#u: 1..4;
```

```
VAR
```

```
u: T#SYSTEM#u;
```

- **Addition of arguments to a module** – In several cases there is an observation or changing of an internal field of a module that is passed as argument to the current module or is local to the current module. In core, in order to use this field we have to specifically pass it as an argument to the module (see Eyal’s documentation for more details).
- **Define inline** – There is a possibility to run the translation in a “define inline” mode. That means that if there was a definition in the smv program:  
DEFINE  
a := 3;  
then in all of the appearances of ‘a ‘ in the program will be replaced by 3. In this case there is a lose of information.
- **Index remaping** – In smv it is possible to define an array index that does not start with 0 or 1 for example: ARRAY –5..8 of boolean. In core only array indexes that start with 0 are allowed and index remaping is done if necessary.
- **Assignment/s - Module connection** – For each variable x in smv there is a curr assignment (x := ...) or an init and next assignments (init(x) := ... and next(x) :=...). In both cases a module is created in core that describes all the possible assignments to the variable according to the smv assignments.
- **TRANS, INVAR and INIT** – for each one of them (if exists) a module is created in the core (as explained before).
- **Addition of conditions** – if there were 2 or more sections of TRANS in the smv program, then all of these conditions are combined into one condition with an ‘and’ operator between them. This is a lose of information.

### ***Current Implementation***

In order to help maintain the additional information according to Yacov Estrin idea the following was made:

- I created a dtd file – smv.dtd which describes the smv grammar.
- In the existing code I added a call to ExtraInformation() function in the places where extra information should be saved into a file. This function is not yet implemented and should do the printing of the extra information in a xlink format. The definition of the function is:  
void ExtraInformation(void\* sources, void\* targets, char\* connection)  
Each call to this function in the code is in a remark and in that place there is a documentation of what should be saved as extra information at this point.
- ID tags were added to the XML output of the core program.

All of the code related to the additional information is marked with ‘@%’.

*Needs to be done*

- Add a code that will take the smv data structure and will create an xml format file out of it. This file should contain ID tags for every tag and should be consistent with the file smv.dtd.
- A dtd file that describes the additional information should be written according to the xlink notation.
- The function ExtraInformation() should be implemented. It should write the extra information to a file in an xml format that will be consistent with the dtd file that describes the additional information.

Efrat Shabtai