smv-morph: Working with Cadence SMV models in Moscow ML

Alma L. Juarez-Dominguez and Nancy A. Day

Cheriton School of Computer Science
University of Waterloo, Canada
{aljuarez,nday}@uwaterloo.ca

Technical Report CS-2012-24

December 10, 2012

Abstract

We describe *smv-morph*: a toolkit of functions to manipulate and print SMV models (both Cadence and NuSMV) in Moscow ML. Our toolkit includes the ability to read Cadence SMV models and counterexamples produced by both SMV model checkers. In this report, we concentrate our explanations on functionality related to Cadence SMV.

1 Introduction

This report serves as a tutorial for *smv-morph*, which allows us to manipulate SMV models, and counterexamples generated from the verification of such models using functions in Moscow ML [8].

Moscow ML is an implementation of Standard ML. ML stands for *metalanguage* and is a general-purpose functional programming language. SMV is a symbolic model checking tool with which one can verify temporal logic [5] properties of finite systems. If a model does not satisfy a property describing a desired behaviour, a model checker produces a *counterexample*, which is a path of the model's behaviour that fails the property. Our toolkit can manipulate both Cadence SMV [6] models and its counterexamples. Although *smv-morph* has the ability to print SMV models with NuSMV [1] syntax, this report concentrates on the functionality related to Cadence SMV because *smv-morph* cannot yet parse NuSMV models to be manipulated. Our toolkit supports a large subset of the SMV language, as shown in in Appendix A.

The files included in the package for manipulations are:

- `smv-lexer.lex`, `smv-lexer.sml` (Lexical analyzer)
- `smv-parser.grm`, `smv-parser.sml`, `smv-parser-main.sml` (Parser generator)
- `smv.sml`, `smv-manipulations.sml` (Manipulation functions)

These files, as well as the necessary libraries to use *smv-morph* are called in `setup-smv-morph.sml`.

The lexical analyzer and parser generator capture the structure of a Cadence SMV model or SMV counterexample, and create an abstract syntax tree (AST) from which an output can be generated, such as a modified SMV model. The AST can be modified by using manipulation functions that can *create*, *test* or *access* the structure of the corresponding SMV model or counterexample. The structure of an SMV model can contain modules, statements, expressions, types and properties. The structure of a counterexample is a series of steps, where each step is a collection of variable assignments at that point in the execution of the

model. Examples of `.smv` files (Cadence SMV models) and `.out` files (counterexamples) are contained in the package within the directory `parse-tests`. The Cadence SMV production rules are shown in Appendix A.

We have used the *smv-morph* package extensively during the automatic analysis of systems, where a model is modified given the information from a counterexample generated previously during the analysis [4]. We will briefly describe this process at the end of this report. Also, Faghih and Day have used the *smv-morph* in a translator from big-step modelling languages (BSML) to SMV [3].

2 Setting up *smv-morph*

Moscow ML (`mosml`) needs to be installed. The toolkit functions can be copy-pasted into a running `mosml` interactive session, or `mosml` can be run through emacs. To invoke the interactive system, type `mosml` at the shell prompt (within the *smv-morph* directory), where expressions and functions can be entered and evaluated:

```
$ mosml
Moscow ML version 2.01 (January 2004)
Enter 'quit();' to quit.
-
```

As indicated, the interactive system can be terminated with 'quit();'.

Once `mosml` is running, the following statements load the necessary libraries and functions to work with Cadence SMV models (as described in `test-parser.sml`, where comments are delimited by '`(*`' and '`*)`'):

```
(* Load necessary libraries and files to manipulate SMV models.
   We assume that mosml is run within the smv-morph directory.
   Otherwise, use loadPath to indicate the paths of the smv-morph
   directory. *)

use "setup-smv-morph.sml";
```

Then, to create the AST of an SMV model, call the function `parse_cad` (for Cadence SMV) with the name of an `.smv` file as parameter. The `mosml` keyword `val` binds a variable (in this example the Cadence SMV model `simple_model`) to a value (in this example the AST of an SMV model):

```
(* Parse file "parse-tests/simple.smv", loading structure simple_model *)
val (simple_model:smv.smv_model) = parse_cad "parse-tests/simple.smv";
```

Note that the elements of the AST structure, as well as the SMV manipulations functions related to this AST structure are all included in the `smv.sml` package file and they must be prefixed by '`smv.`', as it will be seen used throughout this report (unless one uses `open smv;` to access the SMV structure directly). The function `parse_cad` is part of the parser generator, which is not part of the manipulation functions related to the SMV AST structure. Therefore, `parse_cad` does not need to be prefixed by '`smv.`'. Also, the functions that are defined in this report to illustrate the toolkit's functionality use *smv-morph* functions but are not part of the toolkit.

To display an SMV model on the screen with Cadence SMV syntax, use the following commands (also included in `test-parser.sml`), where `simple_model` contains the AST representation of the SMV model:

```
(* Print parsed file "simple.smv" to screen - Cadence SMV syntax *)
smv.print_cadence simple_model;
```

The printed model is close to the original, except for some extra parenthesis to help illustrate relationships and priorities among elements in the model, spaces and tabs added for formatting, and potentially re-ordering of some model elements to preserve good modelling principles (for instance, gather and print together all variable declarations).

Finally, whether an SMV model has been modified or not, the function `print_to_file_cadence` or `print_to_file_nusmv` prints an SMV model, whose first parameter is the variable that contains the AST of an SMV model (in this example `simple_model`), and whose second parameter is a file name:

```
(* Print parsed file "simple.smv" to file - Cadence SMV syntax *)
smv.print_to_file_cadence simple_model "parse-tests/simple_cadence.smv";
```

The next sections describe how to use the manipulations functions to *create*, *test* or *access* the different parts of a Cadence SMV module, followed by the manipulation functions for a counterexample. To illustrate the parts that an SMV model contains, consider the design of a bit adder in Figure 1 (which is composed of two half-adders).

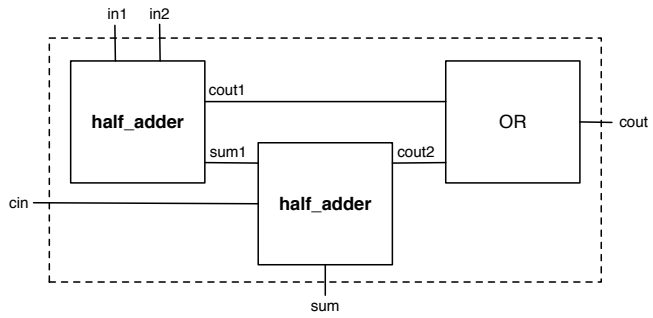

Figure 1: Design of a bit adder, composed of two half-adders

The Cadence SMV model of the adder is shown in Figure 2, which also labels the parts of the SMV model in terms of the elements that *smv-morph* can manipulate (*i.e.,* modules, statements, expressions, types and properties). An SMV model can contain multiple *modules* (*i.e.,* module `half_adder` and module `main` in `adder.smv` of Figure 2), but it should contain at least one module with the name `main`. An SMV module typically contains:

- **Declarations**. The variables are declared in lines 13-17 of Figure 2. Declarations are *statements*, and their creation involves *types*.
- **Assignments**. The functions for outputs `cout` and `sum` of the `main` module are defined in lines 20-21 of Figure 2. Assignments are *statements*, which in turn can be formed from other *statements* or *expressions*.
- **Properties**. The *properties* to prove are specified in lines 23-24 of Figure 2. Properties can be invariants, assertions and specifications. The later can be used in `assume` and `using`..`prove` declarations.

The following sections provide more information about manipulating different parts of the Cadence SMV model: Section 3 talks about modules, Section 4 and Section 5 discuss statements and expressions that form assignments, Section 6 describes types related to declarations, and Section 7 refers to properties. Finally, Section 8 explains manipulations of counterexamples, and Section 9 includes some frequently asked questions.

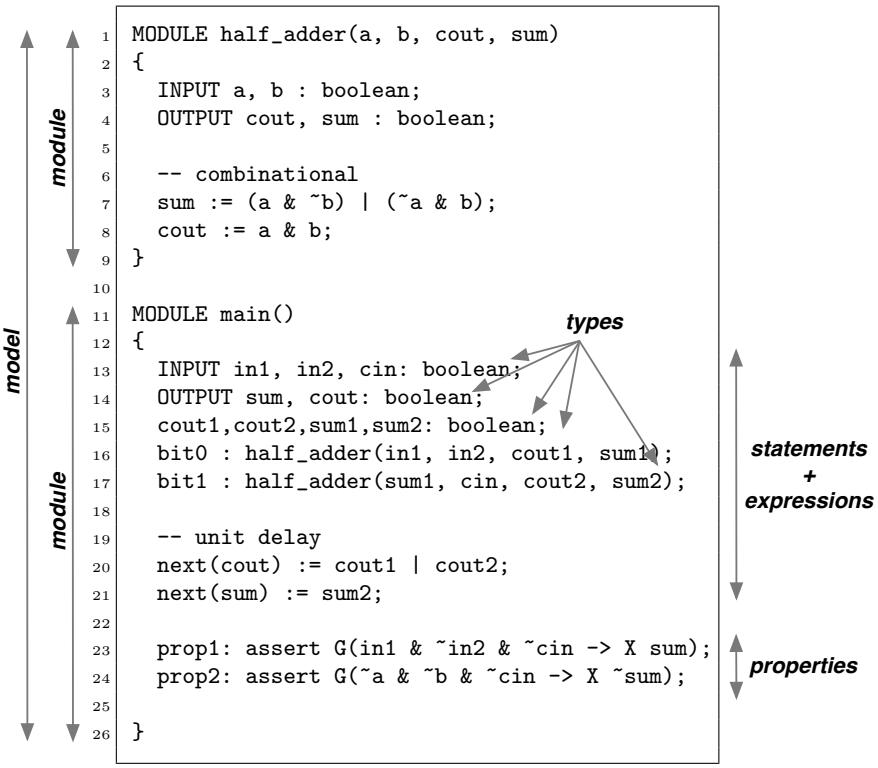

Figure 2: Cadence SMV model `adder.smv`[2] for the bit adder design shown in Figure 1

All the basic manipulation functions are included in `smv.sml`. To give a better initial intuition on the use of these manipulation functions, consider their naming convention:

- *test*: functions have as prefix "`is_`". For instance, function `smv.is_type_int` return `true` if the datatype has been declared as integer.
- *create*: functions have as prefix "`mk_`". For instance, function `smv.mk_type_bool` creates a Boolean datatype.
- *access*: functions' names refer to the type of information they access. For instance, the function `smv.vals_of_type_enum` returns the values of a enumerated datatype.

3 Manipulating Modules

To illustrate the manipulation of modules, consider the model `prio.smv` borrowed from the Cadence SMV Tutorial [7] and shown in Figure 3, which only contains a `main` module. All the manipulations to model `prio.smv` described in this report are also included in `test-parser.sml`.

```
1  module main(req1,req2,ack1,ack2)
2  {
3    input req1,req2 : boolean;
4    output ack1,ack2 : boolean;
5
6    ack1 := req1;
7    ack2 := req2 & ~req1;
8
9    mutex : assert ~(ack1 & ack2);
10   serve : assert (req1 | req2) -> (ack1 | ack2);
11   waste1 : assert ack1 -> req1;
12   waste2 : assert ack2 -> req2;
13 }
```

Figure 3: SMV model `prio.smv` (borrowed from the Cadence SMV Tutorial) [7]

As explained in the Cadence SMV Tutorial, the module `main` of model `prio.smv` contains:

- **Declarations**. The variables `req1`, `req2`, `ack1` and `ack2` are declared to be of type Boolean.
- **Assignments**. The functions for outputs `ack1` and `ack2` in terms of inputs `req1` and `req2`.
- **Properties**. The properties to be proved, in this case, assertions such as `mutex` and `serve`.

To manipulate the Cadence SMV model `prio.smv`, first we need to create the AST of the SMV model with `parse_cad` and assign it to a variable (for instance `m`). To do so, use:

```
(* Parse file "parse-tests/prio.smv", creating structure m *)
val (m:smv.smv_model) = parse_cad "parse-tests/prio.smv";
```

The first manipulation one can perform is to *access* the modules of an SMV model[1]. Because we are normally going to use the parts of the model that we are accessing, it is better to assign the accessed parts of the model to variables (using the `mosml` keyword `val`) as:

```
(* Access the modules of model prio.smv *)
val modules = smv.modules_of_model m;
```

Note that the result of this function returns a list of modules because, as explained before, an SMV model can contain multiple modules with the module `main` being one of them. Modules in this list can appear in any order, and the `main` module can be filtered out by its name. In this case, the list contains only the `main` module, which can be assigned to variable `mod_main` by the ML function `hd` for further manipulations:

```
(* Assign module main to variable mod_main *)
val mod_main = hd modules;
```

[1] In the rest of the report, we often use 'SMV model' to mean the AST of the corresponding SMV model.

Let's observe the module of model `prio.smv` in Figure 3: it has a name (`main`) and it also has four parameters: `req1`, `req2`, `ack1` and `ack2`. The first two parameters are inputs and the last two are outputs. We can access the name and parameters of the module using:

```
(* Access the name of module main (in variable mod_main) *)
val name_module = smv.name_of_module mod_main;

(* Access the parameters of module main (in variable mod_main) *)
val params_module = smv.params_of_module mod_main;
```

A more general way to access the name of module `main` would be to access the name of all the modules, using the `mosml` functions `map` and `fn` to go through all the elements of the list of modules (which has been assigned to the variable `modules`):

```
(* Access to all the names of all the modules (in variable modules) *)
val name_all_modules = map (fn x => smv.name_of_module x) modules;
```

To illustrate manipulation functions related to the *creation* of modules, consider the addition of a parameter, say `ack3`, to the `main` module of `prio.smv`. In *smv-morph*, a module is created with the function `smv.mk_module` by providing four inputs: name, parameters, statements and invariants. We have chosen these elements as input for module creation because they are the most common elements of an SMV module. Note that other elements, such as assertion properties, are added later, and the details will be discussed in Section 7. In our current example, we only want to modify the parameters by adding `ack3` to the list of parameter names. Therefore, to leave the name, statements, and invariants of the `main` module unchanged, we would basically access them and use the information returned to perform the creation of the new module as follows:

```
(* Access the statements of module main (in variable mod_main) *)
val stmts_module = smv.stmts_of_module mod_main;

(* Access the invariants of module main (in variable mod_main) *)
val invars_module = smv.inv_of_module mod_main;

(* Access the assertions of module main (in variable mod_main) *)
val asserts_module = smv.asserts_of_module mod_main;

(* OPTION 1: Creation of main module adding parameter ack3, *)
(* with parameters: name, parameters, statements and invariant, *)
(* chosen as these are the most common elements of an SMV module *)
(* Other elements, such as assertion properties, are added later *)
(* This is the more general method, using the toolkit's accessor *)
(* functions directly *)
val main_mod1 = smv.mk_module
                  (smv.name_of_module mod_main)
                  ((smv.params_of_module mod_main)@["ack3"])
                  (smv.stmts_of_module mod_main)
                  (smv.inv_of_module mod_main);
```

```
(* OPTION 2: Creation of main module adding parameter ack3, *)
(* with parameters: name, parameters, statements and invariant *)
(* This is a more specific method, using the variables assigned so far *)
val main_mod2 = smv.mk_module
                 name_module (params_module@["ack3"]) stmts_module invars_module;
```

Then, we can create the new model (using `smv.mk_model`) and display its content to the screen (with `smv.print_cadence`). In *smv-morph*, a model is created by providing two parameters: a list of modules (where `main` is one of them and in this example, it is the only module) and a list of SMV fairness constraints (although there are none in `prio.smv`, represented in the AST `m`). We remind the reader that the function `print_to_file_cadence` or `print_to_file_nusmv` would print an SMV model to a file.

```
(* Create NEW prio.smv model with the modified main module; *)
(* A model includes a list of modules and a list of fairness *)
(* Print modified "prio.smv" to screen - Cadence SMV syntax *)
smv.print_cadence (smv.mk_model ([main_mod1]) (smv.fairness_of_model m));
```

This simple example shows how to modify the elements of a module in a Cadence SMV model by using the manipulation functions that access and create the structure of the model represented in an AST.

4 Manipulating Statements

Now we turn to the manipulation of statements of a module, where the function `smv.stmts_of_module` returns a list of statements. The Cadence SMV statements supported by *smv-morph* are: variable declarations (regular, input and output variable declarations), conditional statements (*i.e.,* `if-then`, `if-then-else` and `switch` statements), `init` and `next` statements (to denote a sequence of values of a variable), combinational assignments and `macro` statements.

To be able to modify statements, we first need to *test* the statement being analyzed to be able to *access* its information depending on the type of statement. To illustrate this point, we create the function `print_stmt_type`[2] to test a statement given as input and print its type on the screen. Then, we call the function `print_stmt_type` with the first statement in the list `stmts_module`, which is the list of statements returned by `smv.stmts_of_module` (from module `main` in `prio.smv`).

```
(* Function that tests and prints the type of the statement given as input *)
fun print_stmt_type (stmt) =
  if (smv.is_var_decl (stmt)) then print "variable_declaration" else
  if (smv.is_init (stmt)) then print "init" else
  if (smv.is_stmt_next (stmt)) then print "next" else
  if (smv.is_macro (stmt)) then print "macro" else
  if (smv.is_comb_assign (stmt)) then print "combinational_assignment" else
  if (smv.is_stmt_if_then (stmt)) then print "if-then" else
  if (smv.is_stmt_ite (stmt)) then print "if-then-else" else
  if (smv.is_stmt_switch (stmt)) then print "switch" else print "other";

(* Test and print type of first statement of module main (in variable stmts_module) *)
print_stmt_type(hd stmts_module);
```

[2]Note again that the functions shown in this report to illustrate the functionality of the toolkit, such as `print_stmt_type` and `change_var_name_stmts` in this section, are not part of *smv-morph*.

To illustrate manipulation functions related to *access* and *creation* of statements, consider that we want to modify any assignment of `req1` to `ack1`, so that the value is assigned to `ack3` instead. The most general solution is to write a function that tests all statements in the list, and changes a variable name in the left-hand side of a statement to another name given as parameter. To perform this renaming, we need to use manipulation functions to access a variable's name (in the left-hand side of a statement), and if it is the one to be changed (in this case, the first parameter given to the function), then a modified statement is created by using the new name (given as a second parameter to the function). All other assignments (*e.g.,* conditional statements) are left unchanged. These manipulations are illustrated next by the function `change_var_name_stmts`.

```
(* Function that tests all statements in the list, and modifies the assignment
    whose left-hand-side's variable name is indicated as first parameter, giving
    it the name of the second parameter. Leave other statements unchanged. *)
fun change_var_name_stmts (var_name, new_name, stmts) =
  let
    fun mk_str_from_id  id =
      let
        val last_id = List.last id
        val all_but_last_id = List.take(id,List.length(id)-1)
      in
        (concat(map (fn x => x^".") all_but_last_id))^last_id
      end
    fun mk_id_from_str str =
      String.tokens Char.isPunct str
    fun str_eq(str1,str2) =
      if (String.compare(str1,str2)=EQUAL) then true
      else false
    fun change_var_name_stmt (var_name, new_name, stmt) =
      if (smv.is_var_decl (stmt)) andalso
        (str_eq(var_name,(mk_str_from_id(smv.name_of_var_decl stmt)))) then
           smv.mk_var_decl (mk_id_from_str(new_name)) (smv.type_of_var_decl stmt)
      else if (smv.is_comb_assign (stmt)) andalso
        (str_eq(var_name,(mk_str_from_id(smv.lhs_of stmt)))) then
           smv.mk_comb_assign (mk_id_from_str(new_name)) (smv.rhs_of stmt)
      else if (smv.is_init (stmt)) andalso
        (str_eq(var_name,(mk_str_from_id(smv.lhs_of stmt)))) then
           smv.mk_init (mk_id_from_str(new_name)) (smv.rhs_of stmt)
      else if (smv.is_stmt_next (stmt)) andalso
        (str_eq(var_name,(mk_str_from_id(smv.lhs_of stmt)))) then
           smv.mk_stmt_next (mk_id_from_str(new_name)) (smv.rhs_of stmt)
      else if (smv.is_macro (stmt)) andalso
        (str_eq(var_name,(mk_str_from_id(smv.lhs_of stmt)))) then
           smv.mk_macro (mk_id_from_str(new_name)) (smv.rhs_of stmt)
      (* conditional statements are unchanged (if-then, if-then-else, switch) *)
      else stmt
  in
    map (fn x => change_var_name_stmt (var_name, new_name, x)) stmts
  end;
```

```
(* Test, access and modify assignment to ack1, so it assigns to ack3 instead *)
val new_stmts = change_var_name_stmts ("ack1", "ack3", stmts_module);
```

As in the previous section, we create the model with the modified information to display its content to the screen (with `smv.print_cadence` using Cadence SMV syntax):

```
(* Creation of main module adding parameter ack3, and new list of statements *)
val main_mod3 = smv.mk_module
                 name_module (params_module@["ack3"]) new_stmts invars_module;

(* Create NEW prio.smv model with the modified main module; *)
(* A model includes a list of modules and a list of fairness *)
(* Print modified "prio.smv" to screen - Cadence SMV syntax *)
smv.print_cadence (smv.mk_model ([main_mod3]) (smv.fairness_of_model m));
```

5 Manipulating Expressions

Because *smv-morph* accessor functions decompose statements into other statements and expressions, this section discusses functions that manipulate expressions (test, access and creation). The expressions supported by *smv-morph* are: Booleans `true` and `false`, integer, identifier, range, enumerated set, integer set, boolean set, logical (and, or, not), equality, inequality, less than, greater than, less or equal than, greater or equal than, negation, addition, subtraction, multiplication, division, if-then, if-then-else, case, switch, and next.

To illustrate manipulation functions to access and creation of expressions, consider that we want to modify any assignment statement that has a conjunction on the right-hand side, creating a disjunction instead of a conjunction, leaving the expressions that were conjuncted unchanged. One way to perform this task would be writing a function similar to `change_var_name_stmts`, shown in the previous section, but modifying the right-hand side of the statements. Here we follow a different approach for illustration purposes.

First, we start by dividing the list of statements (from module `main` in `prio.smv`) into the ones that are variable declarations (which we do not want to modify as they are not assignments with conjunctions) and the ones that are not variables declarations.

```
(* Filter statements that are variable declarations into var_dec_stmts *)
val var_dec_stmts = List.filter (fn x => smv.is_var_decl x) new_stmts;

(* Filter statements that are NOT variable declarations into non_var_dec_stmts *)
val non_var_dec_stmts = List.filter (fn x => not(smv.is_var_decl x)) new_stmts;
```

The assignments that we want to modify are not variable declarations in `prio.smv`, which are the statements in the `mosml` variable `non_var_dec_stmts`. In particular, we want to modify the right-hand side of the assignments, so that if the expression is a conjunction, then we create a new expression that is the disjunction of the expressions that the conjunction had as operands. To do so, we write function `mk_conj_into_disj_stmts` that calls auxiliary functions to tests the type of statement and to perform the modification of conjunction into disjunction. Function `mk_conj_into_disj_stmts` and its auxiliary functions, use *test*, *access* and *creation* manipulation functions to generate a new set of statements when a conjunction is found, or leaving a statement the same otherwise (*e.g.,* conditional statements).

```
(* Function that creates new statements, making each statement that is an assignment
    to an expression that is a conjunction into an expression that is a disjunction
```

```
    by calling the auxiliary function mk_conj_to_disj_expr with the right-hand side
    of the statement. *)
fun mk_conj_into_disj_stmts (stmts) =
  let
    fun mk_conj_to_disj_expr (expr) =
      if smv.is_and expr then
        (smv.mk_or (smv.fst_arg_of expr) (smv.snd_arg_of expr))
      else expr
    fun mk_conj_into_disj_stmt (stmt) =
      if (smv.is_init (stmt)) then
        smv.mk_init (smv.lhs_of stmt) (mk_conj_to_disj_expr (smv.rhs_of stmt))
      else if (smv.is_stmt_next (stmt)) then
        smv.mk_stmt_next (smv.lhs_of stmt) (mk_conj_to_disj_expr (smv.rhs_of stmt))
      else if (smv.is_macro (stmt)) then
        smv.mk_macro (smv.lhs_of stmt) (mk_conj_to_disj_expr (smv.rhs_of stmt))
      else if (smv.is_comb_assign (stmt)) then
        smv.mk_comb_assign (smv.lhs_of stmt) (mk_conj_to_disj_expr (smv.rhs_of stmt))
      else stmt
  in
    map (fn x => mk_conj_into_disj_stmt x) stmts
  end;

(* Test, access and create new statements, making an expression that is a
    conjunction into an expression that is a disjunction. The function is
    called with the statements that are not variable declarations *)
val new_non_var_dec_stmts = mk_conj_into_disj_stmts (non_var_dec_stmts);
```

The new set of statements is the union of the original variable declarations (*i.e.,* `var_dec_stmt`) with the modified statements result from `mk_conj_into_disj_stmts` (*i.e.,* `new_non_var_dec_stmts`) since the order of statements in SMV does not matter. We create the model with all the modified information generated so far to display its content to the screen:

```
(* Creation of main module adding parameter ack3, and a newer list of statements
        from previously modified statements *)
val main_mod4 = smv.mk_module
                 name_module (params_module@["ack3"])
                  (var_dec_stmts@new_non_var_dec_stmts) invars_module;

(* Create NEW prio.smv model with the modified main module; *)
(* A model includes a list of modules and a list of fairness *)
(* Print modified "prio.smv" to screen - Cadence SMV syntax *)
smv.print_cadence (smv.mk_model ([main_mod4]) (smv.fairness_of_model m));
```

6 Manipulating Types

Let us consider the manipulations performed to variable declarations, which are statements that relate types to variables. The variable types supported by *smv-morph* are: Booleans, integers, ranges of integer values, enumerated lists, module identifier and module declaration.

To illustrate these manipulation functions, consider variable `ack1`, now declared as Boolean, to be made into an integer declaration. Function `change_type_to_int_stmts` follows a similar process to those described previously in this report: from the statements that are variable declarations, find the one whose variable name is `ack1` using accessor functions, and then, create its declaration to be of type integer. If variable name given as parameter to the function is already of type integer, nothing is done. Other variable declaration statements are left unchanged. Also, other variables declared along with the variable which is first parameter of the function, in this case `ack1`, are left as originally declared (*e.g.,* `ack2` of type Boolean).

```
(* Function that tests all variable declaration statements in the list,
    and modifies the type of the statement whose name is indicated as first
    parameter to make it an integer type (unless it is already of that type).
    Leave other statements unchanged, including other variables that
    were declared along with var_name (first function parameter). *)
fun change_type_to_int_stmts (var_name, stmts) =
  let
    fun str_eq(str1,str2) =
        if (String.compare(str1,str2)=EQUAL) then true
        else false
    fun is_var_in_list (var, l) =
        List.exists (fn x => str_eq(var,x)) l
    fun filter_var_from_list (var, l) =
        List.filter (fn x => not(str_eq(var,x))) l
    fun change_type_to_int (fnc, var_name, stmt) =
      let
        val other_vars = filter_var_from_list(var_name,(smv.name_of_var_decl stmt))
      in
         [fnc (other_vars) (smv.type_of_var_decl stmt),
          fnc ([var_name]) (smv.mk_type_int)]
      end
    fun change_type_to_int_stmt (var_name, stmt) =
      if (smv.is_type_int(smv.type_of_var_decl stmt)) then [stmt]
      else
        (if (smv.is_inp_var_decl (stmt)) andalso
           (is_var_in_list(var_name, (smv.name_of_var_decl stmt))) then
             change_type_to_int (smv.mk_inp_var_decl, var_name, stmt)
        else if (smv.is_out_var_decl (stmt)) andalso
           (is_var_in_list(var_name, (smv.name_of_var_decl stmt))) then
             change_type_to_int (smv.mk_out_var_decl, var_name, stmt)
        else if (smv.is_var_decl (stmt)) andalso
           (is_var_in_list(var_name, (smv.name_of_var_decl stmt))) then
             change_type_to_int (smv.mk_var_decl, var_name, stmt)
        else [stmt])
  in
  (* Need to flatten the list, as changing type for a variable passed as
     parameter creates more than one statement: one statement for the
     variable with type integer, and another statement for other variables
     with type of the original declaration *)
    flat(map (fn x => change_type_to_int_stmt (var_name, x)) stmts)
  end;
```

```
(* Test, access and create new variable declaration statements.
    The function is called with the statements that are variable
    declarations (in var_dec_stmts) and the name of the variable
    to modify *)
val new_var_dec_stmts = change_type_to_int_stmts ("ack1", var_dec_stmts);
```

As in the previous sections, we create the model with all the modified information generated so far to display its content to the screen (with `smv.print_cadence` using Cadence SMV syntax):

```
(* Creation of main module adding parameter ack3, and a newer list of
    statements from previously modified statements (both variable
    declarations and non variable declarations) *)
val main_mod5 = smv.mk_module
                 name_module (params_module@["ack3"])
                 (new_var_dec_stmts@new_non_var_dec_stmts) invars_module;

(* Create NEW prio.smv model with the modified main module; *)
(* A model includes a list of modules and a list of fairness *)
(* Print modified "prio.smv" to screen - Cadence SMV syntax *)
smv.print_cadence (smv.mk_model ([main_mod5]) (smv.fairness_of_model m));
```

7 Manipulating Properties

Properties are part of a module, and they can be proved or assumed. The kind of properties that can be proved and that are supported by *smv-morph* are: assertions (LTL properties) and specifications (CTL properties). When any these properties are part of a module, they can be accessed by the functions `smv.asserts_of_module` and `smv.specs_of_module` respectively. For instance, to access the assertions that are part of the `main` module of model `prio.smv`, use:

```
(* Access the assertions of module main (in variable mod_main) *)
val asserts_module = smv.asserts_of_module mod_main;
```

However, as described in Section 3, a module is created by providing the module name, module parameters, list of statements and a list of invariants. Then, properties can be added to a module that already exists, using the manipulation functions `smv.add_assert`, `smv.add_assert_with_comment`, `smv.add_spec`, and `smv.add_spec_with_comment`.

To illustrate the addition of properties to a module, consider the function `add_multiple_asserts` to add the properties in the variable `asserts_module` to the `main` module of model `prio.smv`, printing the result of this manipulation to the screen (using Cadence SMV syntax):

```
(* Function that adds multiple assertions to a module, where each assertion
   is composed of a name `n`, an LTL property `p`, and a comment `c` *)
fun add_multiple_asserts (mm, []) = mm
  | add_multiple_asserts (mm, (n,p,c)::rest) =
        add_multiple_asserts((smv.add_assert n p mm), rest);
```

```
(* Add assertion properties to main module "mod_main" of model prio.smv *)
val main_mod6 = add_multiple_asserts (mod_main,asserts_module);

(* Print modified "prio.smv" to screen - Cadence SMV syntax *)
smv.print_cadence (smv.mk_model ([main_mod6]) (smv.fairness_of_model m));
```

As another example, one can add the assertion properties in the variable `asserts_module` to a newly created module, which also requires a new parameter `ack3` to be added:

```
(* Optionally, creation of main module adding parameter ack3, *)
(* to which some assertion properties are added *)
val main_mod7 =
  let
     val mm = smv.mk_module
                name_module (params_module@["ack3"]) stmts_module invars_module;
  in
        add_multiple_asserts (mm,asserts_module)
  end;

(* Print modified "prio.smv" to screen - Cadence SMV syntax *)
smv.print_cadence (smv.mk_model ([main_mod7]) (smv.fairness_of_model m));
```

A similar process is followed to add specifications if they already exist. In the case that properties do not exist, but they need to be added to a module, *smv-morph* support functions for the creation of properties. To create assertions, the following functions can be used:

`ltl_atomic`	`ltl_not`
`ltl_and`	`ltl_multi_and`
`ltl_or`	`ltl_multi_or`
`ltl_implies`	`ltl_iff`
`ltl_X`	`ltl_U`
`ltl_F`	`ltl_G`

To create specifications, the following functions can be used:

`ctl_atomic`	`ctl_not`
`ctl_and`	`ctl_multi_and`
`ctl_or`	`ctl_multi_or`
`ctl_implies`	`ctl_iff`
`ctl_EX`	`ltl_EU`
`ctl_EF`	`ctl_EG`
`ctl_AX`	`ltl_AU`
`ctl_AF`	`ctl_AG`

To illustrate the process of property creation, let us add an assertion proposed by the Cadence SMV tutorial, which would check for avoidance of "starvation" as

```
no_starve:  assert G F (~req2 | ack2)
```

This property is created in *smv-morph* by creating each of the parts of the assertion incrementally as follows:

```
(* Creation of property no_starve: assert G F (~req2 | ack2); *)
val no_starve_prop = smv.ltl_G (smv.ltl_F
     (smv.ltl_or
          (smv.ltl_not(smv.ltl_atomic "req2"))
          (smv.ltl_atomic "ack2")));
```

Once the properties are created, one can test and access their elements, in an analogous way as it is done for other elements of a module. To continue with the example, we add property `no_starve_prop` to the `main` module of model `prio.smv` and display the resulting model to the screen using:

```
(*  Add new assertion property no_starve_prop main module *)
val main_mod8 = smv.add_assert "no_starve" no_starve_prop main_mod6;

(* Print modified "prio.smv" to screen - Cadence SMV syntax *)
smv.print_cadence (smv.mk_model ([main_mod8]) (smv.fairness_of_model m));
```

Finally, *smv-morph* also supports `assume` and `using`..`prove` declarations. To illustrate this point, consider assuming the property `no_starve`, and using it to prove `mutex`, which can be done using the following manipulation functions:

```
(*  Assume property no_starve in the main module of model prio.smv *)
val main_mod9 = smv.add_assume ["no_starve"] main_mod8;

(*  Use property no_starve to prove property mutex in the same main module *)
val new_main_mod = smv.add_using_prove (["no_starve"],["mutex"]) main_mod9;

(* Print modified "prio.smv" to screen - Cadence SMV syntax *)
smv.print_cadence (smv.mk_model ([new_main_mod]) (smv.fairness_of_model m));
```

8 Manipulating Counterexamples

Given an SMV model with extension `.smv`, the results of verifying the properties that are included in such model are in a file with the same name but with extension `.out`. This `.out` file is the one that can be parsed and manipulated by *smv-morph*, so that the information from a counterexample can be obtained, analyzed and used as described in this section.

Because SMV has the ability to verify all the properties that are part of a model, an `.out` file has a particular structure, with a block of information for each property verified.

- When a property is true, there is a block of information that includes the attribute 'truth value' equal to 1 (but most other information is empty). Therefore, if all the properties in a model are true, the `.out` file would have as many blocks of information as properties in the model.
- When a property is false, SMV produces a counterexample. Therefore, the `.out` file would have a block of information that includes the attribute 'truth value' equal to 0, as well as a path that shows how the property fails. Note that SMV stops the verification process when it finds the first property that is false, but the the `.out` file could still contain the blocks of information of properties that were satisfied along with the block information of the property that failed.

To illustrate the manipulation of counterexamples, consider the output from the verification of `prio2.smv`, which is similar to the model in Figure 3, except that variable `ack1` in line 6 has been changed to be:

```
ack1 := req1 & ~req2;
```

This change is suggested in the Cadence SMV tutorial, so that the property `serve` is false. Note that when verifying the properties of `prio.smv`, they are all true.

To manipulate a counterexample from the verification of an SMV model, first we need to create the AST of the `.out` file (which contains the output from the verification of the corresponding `.smv` file) and assign it to a variable (for instance `ce_out`).

```
(* Parse output file "parse-tests/prio2.out", creating structure ce_out *)
val (ce_out:smv.smv_ce_out) = parse_ce "parse-tests/prio2.out";
```

The first manipulation one can perform is to obtain the list of blocks of information from the verification output[3] contained in the `.out` file that was assigned to variable `ce_out`. To *access* this information, we use the manipulation function `smv.elems_of_ce_out` and assign it to the variable `ce_list`.

```
(* Obtain list of blocks of information from ce_out *)
val ce_list = smv.elems_of_ce_out ce_out;
```

As discussed above, the verification output might not contain a counterexample in the case that all the properties in the model are true. To check if a counterexample exists in the list of blocks of information contained in the `.out` file, we can create the function `exist_ce`, which checks that a block of information has attribute 'truth value' equal to 0, using the manipulation function `smv.val_of_ce` as follows:

```
(* Function that returns true if a counterexample exists, thus *)
(* checking if there is a block of information with attribute  *)
(* 'truth value' equal to 0 (from the list in ce_list) *)
fun exist_ce (l) =
  List.exists (fn x => (smv.val_of_int_ce(smv.val_of_ce(x))=0)) l;

(* Checking if a counterexample exists in the result of verification *)
(* of prio2.smv with function exist_ce, which returns true since the *)
(* property "serve" failed -- as described above *)
exist_ce (ce_list);
```

Once we are certain that a counterexample exists in the verification output (checked with function `exist_ce`), the next step is to filter out the block of information corresponding to the counterexample (*i.e.,* the one that contains the attribute 'truth value' equal to 0).

```
(* Filter out the block of information that contains the counterexample *)
val ce_fail = hd(List.filter (fn x => (smv.val_of_int_ce(smv.val_of_ce(x))=0)) ce_list);
```

Then, once a counterexample information has been filtered out, one can *access* the list of steps (or "states") of the counterexample using the function `smv.states_of_ce`. One can think of these steps as the path that shows, from the initial state, how the property failed. The variables and values in the counterexample obtained from the `.out` file usually contain the extra character '\', which we eliminate with the function `remove_extra_char`.

[3] In the rest of the report, we often use 'verification output' to mean the AST of its corresponding output information contained in the `.out` file.

```
(* Access the states of counterexample from prio2.out *)
val states_ce = remove_extra_char(smv.states_of_ce(ce_fail));
```

Each of the counterexample's steps or states is a list of equality expressions, with the left-hand side of the equality being a variable name and the right-hand side being its value at that particular point in the execution of the model. To *access* the operands of an equality expression, use the manipulation functions `lhs_of_eq_ce` and `rhs_of_eq_ce` to access the left-hand side and right-hand side respectively. Once the operands of the equality expression has been accessed, one can also access the information of the operands. However, we have to first *test* the kind of operand before accessing its information. However, there are only two kinds of expressions that the operands can have:

1. Integer expression: The manipulation function `is_int_ce` is used to *test*, the function `val_of_int_ce` is used to *access*, while the function `mk_int_ce` can *create* an integer expression.

2. Identifier expression: The manipulation function `is_id_ce` is used to *test*, and the function `name_of_id_ce` is used to *access*, while the function `mk_id_ce` can *create* an identifier expression.

To illustrate the use of these manipulation functions for expressions in a counterexample, consider the function `print_ce_info`, which first tests and then accesses the information in a counterexample to print to the screen. The information of the counterexample, as a list of states (each containing a list of equality expressions) is in the `mosml` variable `states_ce`.

```
(* Function takes as input a list of states (each state contains a list of *)
(* equality expressions), and test/access the information printing it to the *)
(* screen. Auxiliary functions: make integer to string and identifier to string *)
fun print_ce_info (states_list) =
  let
    fun mk_str_from_int i =
      if (i >= 0) then Int.toString i
      else ("-")^(Int.toString (abs i))
    fun mk_str_from_id  id =
      let
        val last_id = List.last id
        val all_but_last_id = List.take(id,List.length(id)-1)
      in
        (concat(map (fn x => x^".") all_but_last_id))^last_id
      end
    fun mk_str_from_expr_ce (expr_ce) =
      if (smv.is_int_ce(expr_ce)) then
         mk_str_from_int(smv.val_of_int_ce(expr_ce))
      else (* is_id_ce *)
         mk_str_from_id(smv.name_of_id_ce(expr_ce))
    fun print_expr_ce (expr_ce) =
      let
        val lhs_expr = smv.lhs_of_eq_ce expr_ce
        val rhs_expr = smv.rhs_of_eq_ce expr_ce
        val lhs_str = mk_str_from_expr_ce(lhs_expr)
        val rhs_str = mk_str_from_expr_ce(rhs_expr)
      in
```

```
        print (lhs_str^" = "^rhs_str^"\n")
      end
  in
    (* First map accesses all the states in states_list *)
    (* Second map accesses all the expressions in a state *)
    map (fn x => map (fn y => print_expr_ce y) x) states_list
  end;

(* Print information of counterexample from prio2.smv. Note that *)
(* the information of each state is not separated from the info *)
(* of other states, although c/e from prio2.smv has only one state *)
print_ce_info (states_ce);
```

In related work [4], we have used these manipulation functions to gather information from a counterexample just generated and then create a new property using this information. Then, we re-run the model checker Cadence SMV (with the new property and the added to the original model) from within a `mosml` session, repeating the process until no more counterexamples are generated. Section 9 briefly explains how to run Cadence SMV from within `mosml`.

9 Frequently Asked Questions

Q: **What functions are part of the toolkit that are not accessor functions?**
A: Some auxiliary functions are particularly relevant to the ability to print models in NuSMV syntax are:

- `create_nusmv_model`: Converts a Cadence SMV model into one with NuSMV syntax, particularly, performing if-lifting since NuSMV does not support if-then or if-then-else statements (only supports them as expressions). This function is included in `smv-manipulations.sml`.
- `smv.sort_stmts`: Performs sorting of statements for NuSMV printing, which allows the model to be printed with certain structure, thus promoting good form.

Other auxiliary functions that are included in `smv-manipulations.sml` for convenience of the user are described in the following list. These functions do not require the '`smv.`' prefix.

- **List manipulations**: Functions such as `flat` and `remove_duplicate`, both taking as input a list with elements of any kind.
- **String manipulations**: Functions such as `mk_str_from_id` and `remove_extra_char`, the later explained in Section 8.

Q: **How can one run SMV from `mosml`?**
A: Use the function `Process.system` with parameter a string, which contains the name of the command to run (in this case, "`smv`" for Cadence SMV), as well as the required flags, file name and directory where the `.smv` file exists (if not in the same directory where `mosml` is running). For instance, in our analysis in related work [4], we used the following function call:

```
Process.system ("smv -r -sift -f "^name_dir^"/"^name_file)
```

Q: **How can one catch the result of a process run by `mosml`?**
A: It is possible to catch and check the results of a process initiated by the function `Process.system`. The simplest way is to assign the result of the function call to a variable, such as `run_file`:

```
val run_file = Process.system ("smv -r -sift -f "^name_dir^"/"^name_file)
```

Then, one can check if the successful termination of a process by comparing variable `run_file` with the function `Process.success`, or check that an error occurred during the execution of a process with `Process.failure`. Note that a process spawned with `Process.system` can be wrapped in a shell script.

References

[1] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In *International Conference on Computer-Aided Verification*, number 1633 in LNCS, pages 495–499, 1999.

[2] N. A. Day. CS745/ECE725 Computer-Aided Verification (Course notes, Topic 4: Temporal Logic and Model Checking), Fall 2009.

[3] F. Faghih and N. Day. Mapping Big-Step Modeling Languages to SMV. Technical Report CS-2011-29, University of Waterloo, 2011.

[4] A. L. Juarez Dominguez. *Detection of Feature Interactions for Automatic Active Safety Features*. PhD Thesis, University of Waterloo, 2012.

[5] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1st edition, 1992.

[6] K. L. McMillan. *Symbolic model checking*. Kluwer Academic, 1993.

[7] K. L. McMillan. *Getting started with SMV (Tutorial)*. Cadence Berkeley Labs, 1998.

[8] S. Romanenko, C. Russo, and P. Sestoft. Moscow ML language overview. Version 2.0, June 2000.

A Appendix - Cadence SMV Production Rules

The supported subset of the SMV language is shown in regular font, while the unsupported elements are grayed out.

A.1 Identifiers

```
digit:
  one of 0 1 2 3 4 5 6 7 8 9

number:
    digit
  | number digit

character:
    one of A-Z a-z

simple_id:
    character
  | simple_id character
  | simple_id digit
  | simple_id _

id:
    simple_id
  | id . simple_id
  | id . [ expr ]
```

A.2 Types

```
typ_cad:
    boolean
  | int
  | expr .. expr
  | { expr_list }
  | array expr .. expr of typ_cad
  | simple_id ( expr_list ) -- module type with arguments
  | atom_list               -- module type without arguments

atom_list:
    simple_id
  | atom_list , simple_id
```

A.3 Expressions

```
expr_list:
    expr
  | expr_list , expr

expr:
```

```
    switch ( expr ) { cblk_expr }
  | case { cblk_expr }
  | number
  | { expr_list }
  | ( expr )
  | [ expr_list ]
  | [ expr_list : = expr .. ]
  | bin ( expr , expr )
  | - expr
  | ~ expr
  | + expr    -- unary expression
  | * expr    -- unary expression
  | & expr    -- unary expression
  | | expr    -- unary expression
  | ^ expr    -- unary expression
  | expr - expr
  | expr :: expr
  | expr ** expr
  | expr * expr
  | expr \  expr
  | expr << expr
  | expr >> expr
  | expr + expr
  | expr mod expr
  | expr in expr
  | expr union expr
  | expr = expr
  | expr ~= expr
  | expr < expr
  | expr <= expr
  | expr > expr
  | expr >= expr
  | expr & expr
  | expr | expr
  | expr ^ expr
  | expr <-> expr
  | expr -> expr
  | expr .. expr
  | next ( expr )
  | expr ? expr : expr -- if-then-else expression
  | expr ? expr        -- if-then expression
  | id
  | default

cblk_expr:
    expr : expr ;
  | cblk_expr expr : expr ;
```

A.4 Statements

```
bracket_block:
    { stmt_list }

stmt_list:
    stmt
  | stmt_list stmt

stmt:
    matched
  | unmatched

unmatched:
    if ( expr ) unmatched
  | if ( expr ) matched else unmatched
  | if ( expr ) bracket_block else unmatched

matched:
    if ( expr ) matched else matched
  | if ( expr ) bracket_block else matched
  | if ( expr ) matched else bracket_block
  | if ( expr ) bracket_block else bracket_block
  | other

other:
    input atom_list : typ_cad ;
  | output atom_list : typ_cad ;
  | atom_list : typ_cad  ;
  | define id := expr ;
  | id := expr  ;
  | next ( id ) := expr  ;
  | init ( id ) := expr  ;
  | id <- expr  ;
  | next ( id ) <- expr  ;
  | init ( id ) <- expr  ;
  | case { cblk }
  | switch ( expr ) { cblk }
  | for ( simple_id = expr ; expr ; simple_id = ) inside
  | chain ( simple_id = expr ; expr ; simple_id = ) inside

inside:
    stmt
  | bracket_block

cblk:
    expr : inside
  | cblk expr : inside
```

A.5 Modules

```
module_list_cad:
    module_cad
  | module_list_cad module_cad

module_cad:
    module simple_id ( ) { contents_list }
  | module simple_id ( param_list ) { contents_list }

param_list:
    simple_id
  | param_list , simple_id

bracket_block:
    { stmt_list }

contents_list:
contents_cad
  | contents_list contents_cad

contents_cad:
    stmt
  | invar expr ;
  | assume atom_list ;
  | using atom_list prove atom_list ;
  | atom_list : assert assert ;
  | atom_list : spec spec ;

assert:
    ( assert )
  | prop
  | F assert
  | G assert
  | X assert
  | ~ assert
  | assert U assert
  | assert & assert
  | assert | assert
  | assert -> assert
  | assert <-> assert

spec:
    ( spec )
  | prop
  | A F spec
  | A G spec
  | A X spec
  | E F spec
  | E G spec
```

```
  | E X spec
  | ~ spec
  | A spec U spec
  | E spec U spec
  | spec & spec
  | spec  | spec
  | spec -> spec
  | spec <-> spec

prop:
    id
  | number
  | - number
  | prop + prop
  | prop - prop
  | prop * prop
  | prop / prop
  | prop = prop
  | prop ~= prop
  | prop < prop
  | prop <= prop
  | prop > prop
  | prop >= prop
```

A.6 Model

```
smv_parser_cad: module_list_cad
```