

Description of Murphi Abstracter

I will outline how the abstraction of the Murphi model is done and the changes that were made to Murphi in order to accomplish this task. As a general overview, the program takes in three filenames as arguments. The first file is the big model that will be the base for the abstraction that takes place. The second file is one that contains the abstracted declarations and the new startstate. The main point here is to have the same declarations as the big model, except some are removed/abstracted. The declarations are those that declare types, constants, and variables at the global scope – those that are usually at the top of the model file. The third file is the output file for the abstracted model.

In its native form, the Murphi compiler (mu) takes a Murphi model file and generates C++ code that can be compiled and executed to perform model checking. Thus, the initial focus of this project was to modify this output, so produced Murphi code instead of C++ code. Therefore, once it produced Murphi code, the internal data structure could be modified so it produced the current output with the abstracted variables gone. The internal data structures are split into separate classes for each type, constant, variable, statement, expression, etc. Each of these classes have a “generate_code()” code function that originally outputted the C++ code, of which I have changed to instead output Murphi code.

In the internal Murphi data structure, the type declarations are split so there cannot be a complex type inside a complex type. What this means is that a record entry can have a simple type like a Boolean, but it cannot have a type such as an Array. In order to allow an Array to be a type for a record entry, the array is declared to be a new type, and then in the record entry, it uses that type for the variable instead of inserting the array declaration directly in. For example, “Type rec : Record ShrSet : Array[NODE] Of boolean; Endrecord;” becomes, “Type newType : Array[NODE] Of boolean; Type rec: Record ShrSet : newType; Endrecord;”. This isn’t only done with records, but types such as arrays that can hold complex types also. The abstraction program renames these types to “<type_name>_for_internal_use_<internal_type_number>”, so the type name is relatively unique and not something that would be used in an ordinary program and cause name conflicts.

The program compares all of the type, variable, and constant declarations in the big model with those from the abstract model. For each declaration in the big model, it searches all of the declarations in the abstract model for a match by name. If a match is not found then it doesn’t necessarily mean it isn’t in the abstract model, because of the reasoning for the *_for_internal_use_* types that will not have matching names. If it doesn’t find the type by name, it searches the abstract model for variables that are contained within the abstract type. For an array this is the array index and data field values, and for records this is for the multiple variable names contained within. If a match isn’t found this way, then the variable/type/constant is marked as abstracted (a member variable of the object). I’m not sure if multiple records can have the same variable names, but if one record was removed in the abstracted model and it shares variable names with a record that is not abstracted, I can see that this approach might not detect that the entire record is to be abstracted. However, this is very specific, and I don’t see it to cause a problem unless something was specifically crafted to behave this way. Once this is completed for all of the variables/types/constants in the big model, all of the variables that are to be removed are marked as so.

For types such as record types, if it is found in the abstracted type list and should not be abstracted, further processing is done. For each of the entries in the record, it is matched against an entry in the matching abstract record. Therefore, if the entire record isn’t abstracted, but only a few entries in the

record are, these entries too are marked as being abstracted in the main data structure. Also, in the main model and abstract models, constants are sometimes used in type declarations such as subrange bounds, or scalarset declarations. The internal data structure does not keep a reference to the constants that were used, so these are converted to ordinary integers in the outputted model. The constant declarations are still kept, however, because there could still be a constant reference that was not converted to an integer later on in the model.

Once all of the constants, types, and variables that were not abstracted away are printed, the startstate is outputted from the abstracted model input file. It comes from the abstracted model file because there could be different starting routines that need to be used for the abstracted model. After the startstate, the abstracted model input file is not used further. It could be a file with just declarations and a startstate and everything would work. If there is extra information such as additional rules, these are ignored, as all of the rest comes from modified output of the main model file.

Next, the rules are printed. This is the most complex process because the rule contains rulesets, aliases, the guard, and the body. Calling all of the functions in the rule `generate_code()` function isn't very complicated, but it requires the output of all of the classes' `generate_code()` for the entire rule to be the correct abstracted output. Everything is done recursively because the linked lists of the rules, parameters, etc. are all backwards. For example, there is a linked list of all the rules, but the first rule is the one at the end of the linked list. I call functions in order to output the code for the rulesets and the aliases for each rule.

The ruleset and alias values are stored in a member variable called "enclosures." The content of the enclosures linked list was learned through some experimentation. It seems like the linked list contains the values for all rules, and one has to check a scope member variable to see if it is a parameter for the current rule. Each of these parameters have a type associated with it, so if the type is abstracted away at the global scope, the local parameter is marked as abstracted away also, so the body output won't use variables that are to be removed.

The rule guard uses an `expr` (expression) object and its inheritors that handle things such as Boolean, arithmetic, and conditional expressions. `Expr` also consist on function calls, `forall` statements, "isundefined" statements, and other Murphi constructs. If an expression is abstracted, the expression will print "true." If the expression is a Boolean expression, there are tests that simplify the expressions. For example, if the original expression is "AbstractedVar & NotAbstractedVar", the expression will be abstracted to "true & NotAbstractedVar". The abstracter is programmed to know that that statement is equivalent to "NotAbstractedVar", so this will be printed instead. The same sort of simplification is also done with OR and IMPLIES Boolean expressions.

The rule body is a linked list of statements that are classes for each of the Murphi statements (for example: `if`, `switch`, `assignment`, `for`, etc.). The rule essentially walks the linked list, calling `generate_code()` for each of the statements which outputs the correct Murphi code for each of the statements. There are however three statements that require extra work instead of the `generate_code()` function. The three statements that require additional work are assignments, `if` statements, and `switch` statements. The rest of the statements just check to see if the variable used was abstracted away, and if it was it just doesn't print anything in its `generate_code()` function.

If the left hand side of an assignment is not abstracted, but the right hand side is to be abstracted, this introduces a nondeterministic assignment. To fix this, a ruleset parameter is added to the rule which adds a variable that iterates through all of the values that are possible for the type of the abstracted variable on the right side of the assignment. These assignment statements are located before the rule is printed, because in Murphi the ruleset declaration needs to be before the rule declaration. The additional ruleset parameters aren't added to the official rule's ruleset parameters in the enclosures member variable, so the rule isn't modified with this operation.

The most challenging and complicated is the *if* statement. If the *if* condition is to be abstracted and removed, then the *if* statement needs to be split into multiple rules. To split the rule for *if* statements, I had to make a new `generate_code()` function. Arguments to the new function are a pointer to the current rule object, the statement before the *if* statement, and the statement after the *if* statement. The new code generate function returns a linked list of new rules that were generated as a result of splitting the *if* statement. It is handled this way because I am unable to insert the new rules into the original rules linked list, as the list is in reverse order and traversed via recursion. Therefore, if I insert the rules into the original linked list, the new rules would never be seen. Instead, a complete linked list is generated containing all of the new rules for the complete body, and at the end of the current rule output, it calls the output functions for each of the new rules before returning.

If the *if* statement condition is abstracted and removed, then only the if body would be executed because the abstraction will make the if condition true, and the else code would never be executed. Therefore, the *if* statement is split into only the if body, and then in an additional rule, only the else code. Therefore, all of the cases that could have originally occurred in the not abstracted if will occur here, because both branches have a chance to execute. To add the additional rule, a shallow copy is made of the original rule, and then a deep copy is made of the body statements because these will be changed. A shallow copy is fine for the rest of the member variables, because they are not changed. Before the copy is made, the else code is spliced into where the *if* statement resides in the linked list of statements for the rule body. The previous and next statements are required for the new `generate_code` function because I need to change the next pointer of the previous statement and set the end of the else code to the next statement. With the else code spliced in, a copy of the rule is made and added to a linked list to be returned.

Next, the if body is spliced in to where the else code just was. This will not change the additional rule in the linked list, because a deep copy was made. For the current rule, the if body is printed to the output file. The else code will then be printed in the next rule when the copied rule is printed. The if body is kept in the rule and is not restored with the original *if* statement. The reasoning for this is because additional *if* statement splits after the current *if* statement will need the if body code. For example, if there is a second *if* statement that will be split, another copy of the rule will be made. If the original *if* statement was restored for the first *if*, then on the copied rule, the first *if* statement would then be split again, which would be wasteful, because this would have already been covered in another rule, so duplicate rules would be created.

If the *if* statement doesn't have an else clause, then instead of defaulting to replace the removed expressions to "true", the rule is again copied and in the copied rule, the removed expressions get replaced with false. In Murphi, there is a concept of "elsif", but in the data structure, this is replaced with an "else" and then another "if" statement. Therefore one has to walk the entire *if* statement, to see if the

every *if* statement in the else code is an “*elsif*” statement, and find the last “else” to find if the *if* statement has an “else” clause.

The last statement that requires additional work is the switch statement. If the switch expression is abstracted, then the case statements need to be split like the if code and else code does for the *if* statements. To do this, the code for the first case statement outputted in the current rule. In the copied rule, the first case statement is removed, and the second case statement takes the place of the original first case statement. This will ultimately split the switch statement into each of the corresponding case statements in multiple rules. Like the *if* statement, if the switch statement does not have an else clause, the abstracted variables in the condition are replaced with “false” instead of “true”. However, in the case of the switch statement, it is easier to determine if there is an else clause, because there is an `elsecode` member variable in which you don’t have to worry about *elsif*’s.

When testing this implementation, I found that there were problems with if statements that were the first statements in the body of another if statement. This problem was from the arguments I passed in for the previous and next statements. Since the *if* statement was the first statement in the body, the previous statement was NULL. If I used the actual *if* statement as the previous statement, then it would modify the next statement to the if statement, which was after the *if* statement had printed. If I used the statement before the *if* statement, then the surrounding if would be overwritten, and would not be printed. The solution to this was to add null statements to the front and back of the linked lists in those statements that encapsulated other statements such as *if*, *switch*, and *for*. Therefore, I could use the null statement as the previous statement, and the splicing in of the if and else code worked correctly.

That is essentially everything done to produce the output of the rules. The final item are the invariants, which are much simpler. An invariant has a name and an expression. If the expression was completely abstracted away, then the invariant is removed. If the invariant wasn’t completely abstracted, then the invariant is printed with its name and expression.

The final feature of the Murphi abstracter program is that the output is indented correctly. Originally, all output of the program was made with `fprintfs`. I made a wrapper function for `fprintf` called `fprintf2` that reads a global variable called `indentationAmount`. The `fprintf2` function outputs to a file the code from the arguments to the function, but for every newline it first prints `2*indentationAmount` spaces, by doing a find and replace on the string before it is printed. Therefore, the indentation is kept by incrementing the global variable when there is a block to be printed that should be indented, and then the global is decremented after that block of code is printed.

This description is essentially how everything works in the abstracter program. Now it should be relatively simple to understand what all of the code means. The essential file is `mu_code.C` that has all of the functions that implement what was described above. The only other files that were changed were some of the header files to add additional member variables, and `mu.C` so it will accept three file arguments as input to the program.